

Julio César López Dávila



# CURSO DE JAVA

---

Desde cero hasta conexiones con bases de datos MySQL

---



# CURSO DE JAVA

## ***Desde cero hasta conexiones con bases de datos MySQL***

Este curso de Java trata de ser una *guía paso a paso*, desde cero, para crear una aplicación de escritorio que se conecte al servidor de MySQL; está dirigido a quienes dominan los fundamentos de programación en algún otro lenguaje, que además tienen los rudimentos de las consultas a las bases de datos relacionales. A lo largo del curso se usan diversas librerías de la *Java Estandar Edition* tales como *java.lang*, *java.awt*, *javax.swing* y *java.sql* de las que se estudian, ejemplifican y aplican las clases esenciales.

La estructura del texto sigue el modelo de enseñanza de los tres primeros módulos de Java impartidos por el Ing. Jorge Sánchez Barraeta, profesor de [3CT](#).

Julio César López Dávila pertenece a la plantilla de profesores de Java SE y EE en [3CT](#); además, imparte allí mismo los cursos de Fundamentos de programación y XHTML, de los servidores de bases de datos MySQL y SQL Server, así como de los lenguajes de programación PHP y .NET.

# Presentación

## *¿Qué y para qué es Java?*

Al hablar de Java, nos estamos refiriendo a tres cosas asociadas con la programación de *software*: un lenguaje, una plataforma y un fenómeno. La eficacia y la flexibilidad del *lenguaje* permitieron crear una *plataforma* tan extensa que tiene alcance lo mismo para aplicaciones de propósito general en computadoras personales, para el funcionamiento de dispositivos móviles y aparatos electrónicos, y hasta para sitios *web*; este alcance ha creado un verdadero *fenómeno* tecnológico; tanto, que hoy por hoy hay más de 4 500 millones de equipos que tienen instalado Java. Entonces, en estricto sentido, esta tecnología sirve para hacer aplicaciones, virtualmente, para cualquier componente que tenga un procesador de *software*.

La plataforma para el desarrollo de Java está dividida en tres ediciones: la estándar (JSE), la empresarial (JEE) y la de dispositivos móviles (JME). La primera contiene, entre muchas otras cosas, **los elementos del lenguaje, los objetos para las interfaces gráficas y los mecanismos de conexión a base de datos**, que son lo primero que debe saberse para desarrollar aplicaciones Java básicas.

## ***Objetivos y metodología del curso***

Antes de seguir, es necesario dar noticia de que el éxito de la plataforma de Java se debe a que cumple plenamente con las exigencias de la programación orientada a objetos (POO); esto obliga a que todo curso de Java tenga una introducción a los fundamentos

de este modelo de diseño de *software* o a que se asuma su conocimiento. Sin embargo, hay una ambigüedad en la iniciación en Java porque sus aplicaciones *deben* orientarse a objetos pero para explicar lo esencial no es necesario. De tal manera que se pueden crear programas sencillos sin seguir ningún patrón, pero esto impide la comprensión de la parte no trivial del lenguaje. Por eso, en este curso, presentaremos una introducción a la POO y los programas de la aplicación de ejemplo se apegarán a ella, incluso los de la introducción a los elementos, aunque, para facilitar el aprendizaje, a menudo haremos demostraciones sencillas sin orientación alguna.

Ahora bien, es del conocimiento general que la curva de aprendizaje de Java es ardua. Esto se debe principalmente a que los desarrolladores no trabajan con el lenguaje en sí sino con la plataforma. En el momento que iniciamos una aplicación debemos usar algunos objetos preexistentes que son parte de la edición estándar y que realizan las labores comunes; además, es necesario elegir de entre varios cientos de otros objetos aquéllos que son útiles para alcanzar las metas del sistema. Así que en la edición estándar hay un amplio número de librerías que contienen una multitud de objetos que se deben conocer suficientemente para comenzar a programar.

Así que hay que aprender mucho antes de lograr una aplicación robusta; la tarea puede ser agotadora y confusa, y los resultados, muy pobres. Ante esto, la metodología de este curso será explicar los elementos y la sintaxis básica con un grado suficiente de profundidad; y, después, crear una aplicación de muestra, un “paso a paso” con lo mínimo necesario para hacer un programa Java funcional. Entonces, los objetivos serán dos: 1) Explicar lo esencial del lenguaje, y 2) Hacer una aplicación de ejemplo que sea una interfaz gráfica que haga consultas a una base de datos de MySQL y muestre los resultados en tablas.

# Primera parte. Iniciación al lenguaje Java

## ***Instalación***

Java funciona mediante un *software* conocido como la máquina virtual (JVM por sus siglas en inglés), que es el corazón del entorno de ejecución y que debe estar instalado en el sistema operativo para que las aplicaciones Java se ejecuten. En Windows está en una ruta semejante a esta:

C:\Program Files\Java\jre[versión]

En la que la versión es una sucesión de números como 1.4.5.6 , 1.6.0\_07, etc.

Para que los programas puedan ejecutarse hay que asegurarse que el JRE (*java run enviroment*) esté instalado (que es lo más probable); si no, se descarga de [www.java.com](http://www.java.com) que es un sitio dedicado exclusivamente a la disponibilidad de la máquina virtual de Java.

Para programar es necesario el *kit* de desarrollo (JDK) que sirve para crear y probar las aplicaciones y que ya incluye el JRE. En el momento de la escritura de este curso la *URL* de descarga es:

<http://java.sun.com/javase/downloads/index.jsp>

En la que debe elegirse la presentación sencilla que dice:

Java SE Development Kit (JDK)

JDK Update [versión]

En la que, de la misma manera, versión es un número como 12, 14, 15, etc.

Hay otras descargas que tienen JFX, JEE o NetBeans pero son más grandes y los complementos que ofrecen no son necesarios para los objetivos de este curso, aunque cualquiera de ellas sirve igual.

Una vez instalado el JDK estará en un subdirectorio semejante a este al que se le conoce como [JAVA-HOME]:

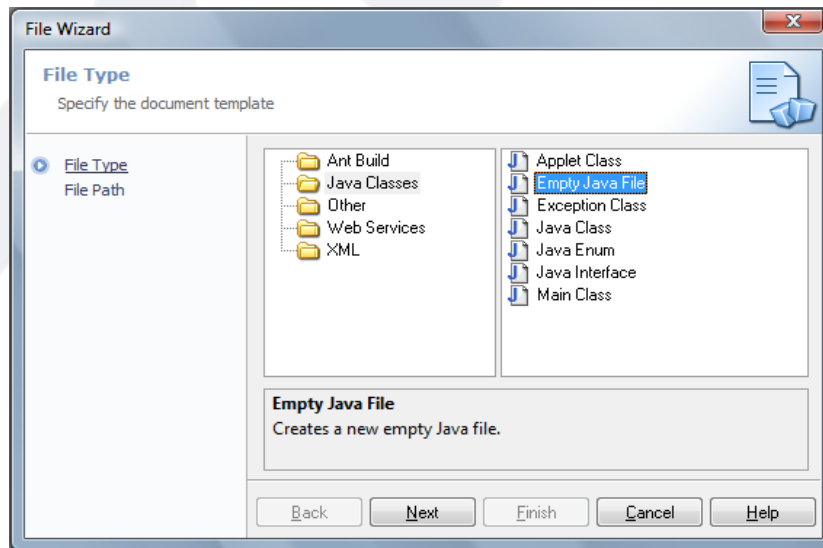
C:\Program Files\Java\jdk[versión]

En cuanto al desarrollo de aplicaciones, aunque es posible trabajar los archivos de Java con cualquier editor de texto plano, aquí lo haremos con la versión LE de JCreator cuya página de descarga es:

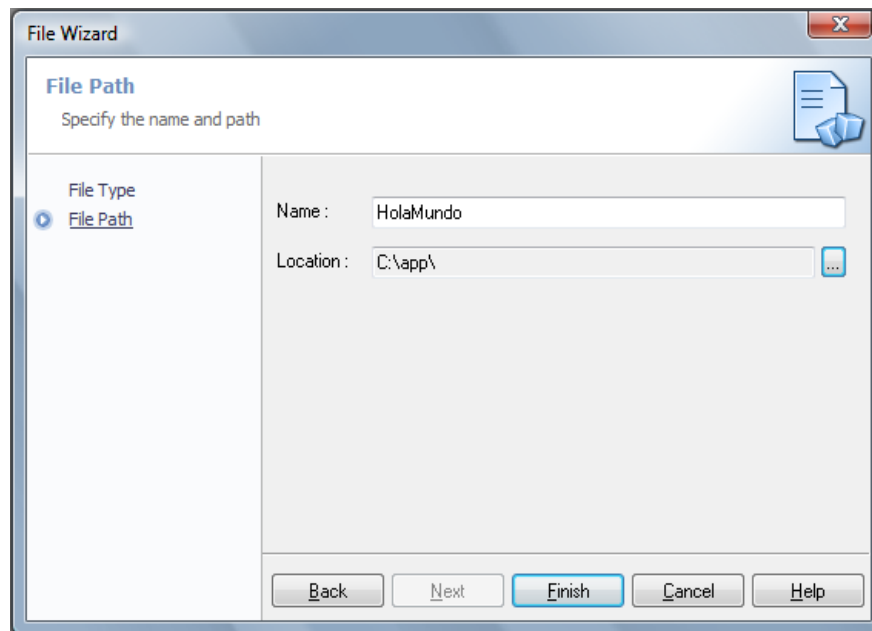
<http://www.jcreator.com/download.htm>

JCreator no debe instalarse antes del JDK porque no funcionará.

Para probar que todo está como se espera debemos ejecutar JCreator y crear un archivo nuevo vacío:



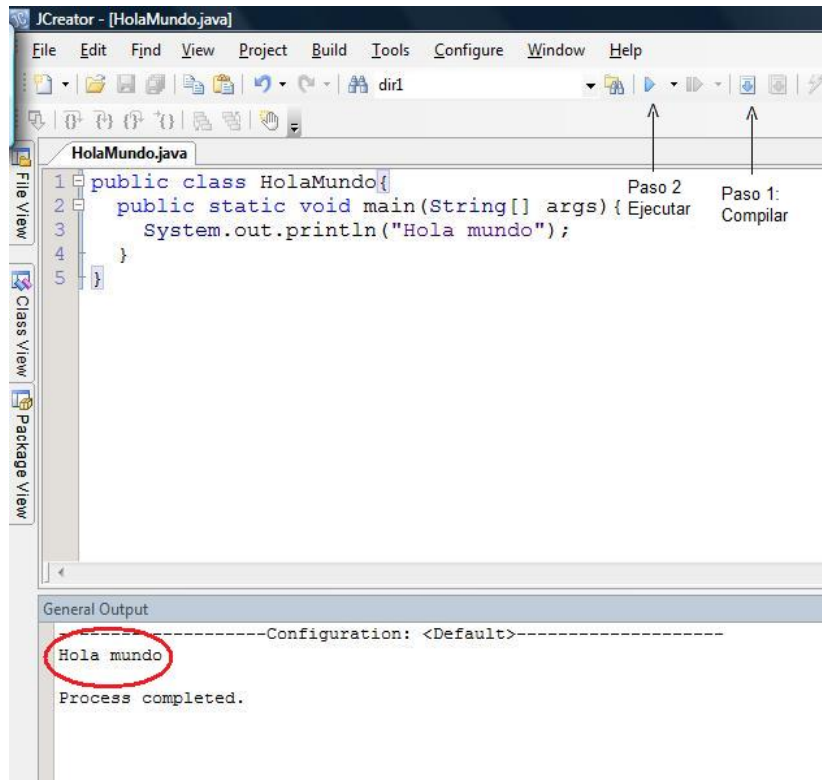
Oprimimos <Next> y en el *Name* escribimos *HolaMundo*, así, todo junto y respetando las mayúsculas. Además, elegimos el subdirectorio donde se debe guardar el archivo y oprimimos <Finish>:



En seguida, copiamos y pegamos el siguiente programa:

```
public class HolaMundo{
    public static void main(String[] args){
        System.out.println("Hola mundo");
    }
}
```

Obsérvese que el nombre de la clase y el del archivo son el mismo. Ejecutamos y compilamos con los botones que indica la figura siguiente y debe aparecer el mensaje “Hola mundo” en la ventana *General Output*:



Si no se obtiene este resultado, es inútil seguir adelante. Los errores los marcará JCreator y pueden deberse a que no está instalado el JDK o a que el programa se escribió a mano y tiene errores. El compilador crea un archivo llamado HolaMundo.class que es el que contiene el programa y el que la máquina virtual puede interpretar.



## **Definición breve de la Programación Orientada a Objetos (POO)**

Antes de establecer los elementos del lenguaje, es necesario tener presentes los conceptos básicos de la programación orientada a objetos porque la sintaxis y el formato de Java están plenamente apegados a ellos.

Para empezar, todo parte del hecho de que el desarrollo de la programación de computadoras entró en crisis en los años 60 y 70 del s. XX porque las aplicaciones a menudo hacían cosas raras. Un caso es el del Centro de Cómputo Noruego en Oslo en el que desarrollaban simuladores de vuelo; sucedía, cuando los ejecutaban, que las naves colisionaban. Un análisis del problema probó que la aplicación confundía las características entre uno y otro objeto simulado; es decir, que la cantidad de combustible, la posición en el espacio o la velocidad de una nave eran atribuidas a otra. Se concluyó que esto se debía al *modo* como programaban y que los lenguajes de entonces eran incapaces de resolver el problema. Ante esto, los expertos del Centro Noruego desarrollaron Simula 67 que fue el primer lenguaje orientado a objetos.

Así que la POO es una manera de diseñar y desarrollar *software* que trata de imitar la realidad tomando algunos conceptos esenciales de ella; el primero de éstos es, precisamente, el de *objeto*, cuyos rasgos son **la identidad, el estado y el comportamiento**. No debemos dejarnos intimidar por estas expresiones, son muy simples:

La identidad es el nombre que distingue a un objeto de otro.

El estado son las características que lo describen.

El comportamiento es lo que puede hacer.

Se debe tener presente que los objetos, sean reales o su proyección en *software*, se abstraen en clases. Por ejemplo: de la clase perro pueden existir dos objetos Fido y Firuláis (esta es su identidad). Fido es un san bernardo enorme, pinto, de 5 años de edad; mientras que Firuláis es un labrador, negro, de 3 años (este es su estado). Ambos perros ladran, merodean, juegan, comen y duermen (este es su comportamiento).

Si nos pidieran que hiciéramos un programa orientado a objetos que simulara lo anterior haríamos la **clase Perro** que tendría las **variables** *raza*, *color* y *edad*, y los **métodos** *ladrar()*, *merodear()*, *jugar()*, *comer()* y *dormir()*. Firuláis y Fido son los **identificadores** que podríamos usar en una aplicación que pretenda mostrar dos objetos de la clase Perro.

## **El programa Hola Mundo y los conceptos de abstracción y encapsulamiento**

Otros conceptos de la POO son el de **abstracción** y el de **encapsulamiento**, que están muy ligados y tienen que ver con el diseño de programas. Ambos se refieren a que los objetos deben hacer tareas que les son propias y no de otros. Por lo común, los objetos de la realidad no dan problemas porque ya existen. No fue difícil abstraer la clase Perro ni encapsular su comportamiento porque existe en la realidad. Para llevar esto al ámbito del *software* analicemos el caso del programa Hola Mundo. Hagamos el proceso de abstracción para encapsular sus características y su comportamiento.

El primer Hola Mundo lo popularizó Brian Kernighan en los años 70 del siglo XX, en un libro que causó mucho interés en su tiempo y que escribió junto a Dennis Ritchie: *The C Programming Language*. Hoy en día, es una tradición presentar los lenguajes con un programa de este tipo, que lo que debe hacer es mostrar la frase “Hola mundo” en la pantalla, y sirve para probar que el lenguaje está debidamente instalado y funcionando.

Entonces, abstrayendo esto, podemos decir que el comportamiento de los objetos del tipo Hola Mundo es **mostrar un mensaje** y su característica, el **mensaje** mismo.

Lo que sigue es mostrar cómo los elementos de lenguaje Java nos permiten apegarnos a la orientación a objetos; para eso es necesario conocer dichos elementos, pero antes, para finalizar, un breve resumen.

Hemos introducido cinco conceptos de la POO:

1. La identidad, que es el nombre que distingue a los objetos
2. El estado, que se refiere a sus características o atributos
3. El comportamiento, que indica los métodos que se deben programar para que los objetos realicen acciones
4. La abstracción, que es el mecanismo mental para aislar su naturaleza
5. El encapsulamiento, que exige que sus características y métodos estén bien definidos y no se confundan con los de otros

Faltan dos conceptos muy importantes: la herencia y el polimorfismo, que veremos más adelante, cuando el conocimiento del lenguaje facilite su comprensión.

## ***Elementos del lenguaje Java***

El siguiente programa presenta la primera versión del Hola Mundo orientado a objetos:

HolaMundoOO.java

```
/*Aunque el compilador importa la librería java.lang completa
 *es conveniente importarla explícitamente por razones didácticas*/
import java.lang.*;
public class HolaMundoOO{
    String saludo; //La clase String la importamos de java.lang
    public void mostrarSaludo(){
        saludo="Hola mundo";
        System.out.println(saludo); //La clase System la importamos de java.lang
    }
}
```

Y está conformado por los siguientes elementos:

- Identificadores
- Sentencias
- Bloques de código
- Comentarios
- Expresiones
- Operadores
- Metacaracteres
- Palabras reservadas

Explicaremos cuál es el uso de cada uno de ellos y, después, como funcionan en el HolaMundo.java.

## Identificadores

Son los nombres que pueden tener las clases, los métodos y las variables y no pueden contener espacios ni caracteres especiales. Estos nombres deben respetar ciertas convenciones según la siguiente tabla:

Tipo de identificador	Convención	Ejemplo
Clase	Comienza con mayúscula	HolaMundoOO
Método	Comienza con minúscula	mostrarSaludo ()
Variable	Comienza con minúscula	saludo

Si el identificador está formado por más de un vocablo, a partir del segundo las iniciales deben ser mayúsculas. Además, se recomienda que los nombres de las clases sean sustantivos, los de los métodos verbos y que las variables expresen con claridad su contenido.

## Sentencias

Son las órdenes que se deben ejecutar en el programa y terminan siempre con un punto y coma:

;

Por ejemplo:

```
String saludo;
```

## Bloques de código

Son el principal mecanismo de encapsulamiento y se forman con un grupo de sentencias y de otros bloques de código delimitados por una llave de apertura y una de cierre (considerados metacaracteres en java, como veremos más adelante):

{ }

Por ejemplo:

```
{
    saludo="Hola mundo";
    System.out.println(saludo);//La clase System la importamos de java.lang
}
```

## Comentarios

Son líneas de texto insertas en el programa para documentarlo y facilitar su lectura. Los tipos de comentarios más usados son:

Tipo	Caracteres que los identifican	Ejemplo
De una sola línea	//	//La clase Sring la importamos de java.lang
De varias líneas	/* */	/*Aunque el compilador importa la librería java.lang completa es conveniente importarla explícitamente por razones didácticas*/

## Expresiones

Las expresiones son entidades formadas por dos o más miembros separados entre sí por operadores que los evalúan y los relacionan.

Por ejemplo;

saludo="Hola Mundo";

## Operadores

Los operadores son signos especiales para hacer acciones específicas y son el mecanismo con el cual los objetos interactúan relacionando los datos y devolviendo nuevos valores; los mostraremos conforme los necesitemos. Se clasifican así:

- Aritméticos
- De comparación y lógicos
- De asignación

## Metacaracteres

Existen otro tipo de caracteres particulares que sirven para el control y la significación puntual en las sentencias y los bloques de código:

`( [ { \ ^ - $ | ] ) ? * +`

## Palabras reservadas

Hay un grupo de palabras en Java con las cuales, entre otras cosas, se realizan las tareas principales, se delimitan los alcances de los objetos, sus datos y sus métodos, etc. Se pueden clasificar así y las mostraremos también conforme avancemos:

- Tipos de datos

- Sentencias condicionales
- Sentencias iterativas
- Tratamiento de las excepciones
- Estructura de datos
- Modificadores y control de acceso

A continuación, una imagen que esquematiza los elementos en la clase *HolaMundoOO*

```

/*Aunque el compilador importa la librería java.lang completa
es conveniente importarla explícitamente por razones didácticas*/
import java.lang.*;
public class HolaMundoOO{
    String saludo; //La clase String la importamos de java.lang
    public void mostrarSaludo(){
        saludo="Hola mundo";
        System.out.println(saludo); //La clase System la importamos de java.lang
    }
}

```

Identificadores:	
Sentencias:	
Bloques de código	
de la clase:	
del método	
Comentarios:	
Operadores:	
Metacarecteres	
Palabras reservadas	

(Obsérvese que la línea `saludo= "Hola mundo";` es, además, una expresión porque utiliza el operador de igual (=))

La imagen anterior muestra que, a pesar de la sencillez del programa, son muchos los elementos que están involucrados. Las reglas de organización de dichos elementos conforman la sintaxis cuya comprensión nos obligará a desmenuzar el programa *HolaMundoOO* en el siguiente apartado.



## **Sintaxis**

Para comprender mejor la sintaxis del programa debemos pensar en términos de ámbito o alcance. Primero hay que saber que los comentarios están fuera del programa, no están dentro de su ámbito, el compilador no los interpreta, son señales que el programador usa para facilitar la comprensión del código.

El ámbito más externo es donde importamos los recursos que se requerirán para el programa y donde declaramos el programa mismo. Aquí está el principio de todo. En Java **siempre** desarrollaremos clases, y **siempre** usaremos clases ya hechas que importaremos. Los recursos mínimos para programar están en el paquete *lang* de la librería *java* que el compilador importa por omisión, aunque aquí (ya lo decíamos en un comentario de la clase *HolaMundoOO*) lo haremos explícitamente por razones didácticas.

Así que al comenzar a desarrollar un programa debemos primero determinar **las clases externas** necesarias para ayudar a la clase que crearemos nosotros, y después crear ésta. Para la primera labor usamos la palabra reservada *import* y todas las clases invocadas así podrán ser utilizadas en cualquier lugar del bloque de código de la clase; para la segunda, empleamos las palabras reservadas *public class* seguidas del nombre que deseemos asignarle; debe ser único, preferentemente un sustantivo, iniciar con mayúscula y expresar claramente su función porque será el identificador de la clase; después van dos llaves que contendrán el bloque de código, como muestra la figura siguiente.

```
import java.lang.*;
public class HolaMundo{

}
}
```

El rectángulo representa el ámbito o alcance. Los objetos del paquete *lang* están disponibles en todos lados.

Dentro del bloque de la clase está el lugar donde deben crearse las propiedades o atributos y declararse los métodos. Las propiedades deben escribirse primero, fuera de cualquier método y su alcance será toda la clase, que es lo que significa el rectángulo interno en la siguiente figura.

```
import java.lang.*;
public class HolaMundo{
    String saludo;
}
}
```

Aquí hace falta hablar de la sintaxis de la creación e inicialización de las variables. En Java, toda variable se crea estableciendo su tipo seguido de un nombre, que deberá ser único en el ámbito donde se le declara; se inicializan siempre con una expresión. *Crear* significa asignarles un espacio en memoria, mientras que *inicializar* es darles un valor:

String saludo; → Crea una variable llamada saludo.

saludo= "Hola mundo"; → La inicializa

Cuando las variables son atributos deben crearse únicamente y debe hacerse fuera de los métodos; por otro lado, serán inicializadas dentro de alguno de ellos en el que convenga. Es posible hacer ambas labores en una sola línea pero será sólo aquellas que sirven a las tareas de los métodos:

```
String muestra= "Esto es una muestra";
```

Los métodos, por su parte, se declaran estableciendo primero el nivel de acceso. En este caso *mostrarSaludo()* tiene acceso público (para eso usamos la palabra reservada *public*), que significa que cualquier objeto externo puede invocar la tarea encapsulada en el método. Hay otras posibilidades, por ejemplo *private*, que es también una palabra reservada, y que significa que el método sólo puede ser usado al interior de la clase en la que es declarado. Después del nivel de acceso, está el tipo de datos que el método devuelve; *mostrarSaludo()* no regresa ningún valor; para indicar esto se usa otra palabra reservada: *void*, que, como veremos después, podrá ser sustituida por cualquier tipo de datos. Todos los métodos deben tener su propio bloque de código, en el que, como ya dijimos, está encapsulada su tarea. El método en cuestión inicializa *saludo*, e invoca la clase *System* (que importamos del paquete *lang* al principio) cuyo atributo *out* es un objeto que tiene el método *println(cadena)* cuya tarea es imprimir una línea con el texto que recibe como parámetro en la salida estándar del sistema. El método *mostrarSaludo()* y su alcance están representados en el rectángulo más interno de la siguiente figura.

```
import java.lang.*;
public class HolaMundo{
    String saludo;
    public void mostrarSaludo(){
        saludo="Hola mundo";
        System.out.println(saludo);
    }
}
```

En cuanto a la sintaxis, falta decir que otra de las responsabilidades de cualquier clase es autoconstruirse por lo que tendrá un método que se llamará igual que la clase misma, deberá ser público, no tiene modificador del tipo que devuelve y lo llamamos **el constructor**, con lo anterior la clase queda como sigue (obsérvese que en este método se inicializa el atributo):

```

import java.lang.*;
public class HolaMundoOO{
    String saludo; //La clase String la importamos de java.lang
    //Creamos el método constructor con el mismo nombre de la clase
    public HolaMundoOO(){
        saludo="Hola mundo";//En el constructor se inicializan las propiedades
    }
    public void mostrarSaludo(){
        System.out.println(saludo);//La clase System la importamos de java.lang
    }
}

```

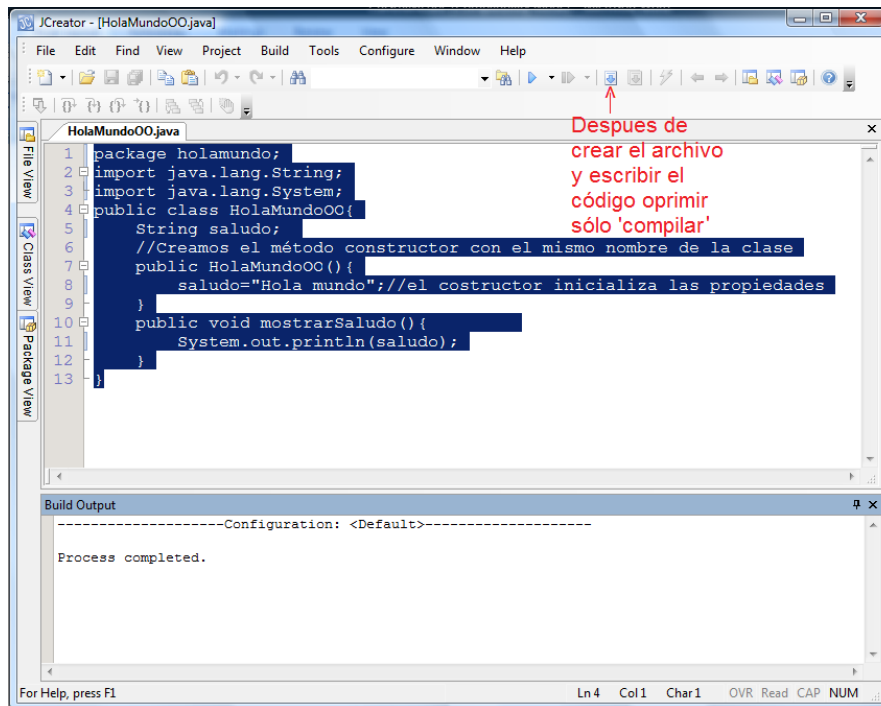
Finalmente, otro mecanismo de encapsulamiento es crear paquetes para guardar en ellos las clases que hacen labores afines. Esto se logra con la palabra reservada *package* seguida del nombre que identificará al paquete; éstos se escriben sólo con minúsculas. La versión final de la clase *HolaMundoOO* que queda encapsulada en el paquete *holamundo*, que importa de *java.lang* las clases *String* y *System*, que tiene el atributo *saludo* de tipo *String*, que se autoconstruye e inicializa la propiedad, que tiene un método que muestra *saludo* usando la clase *System* queda así (Atiéndanse los comentarios):

```

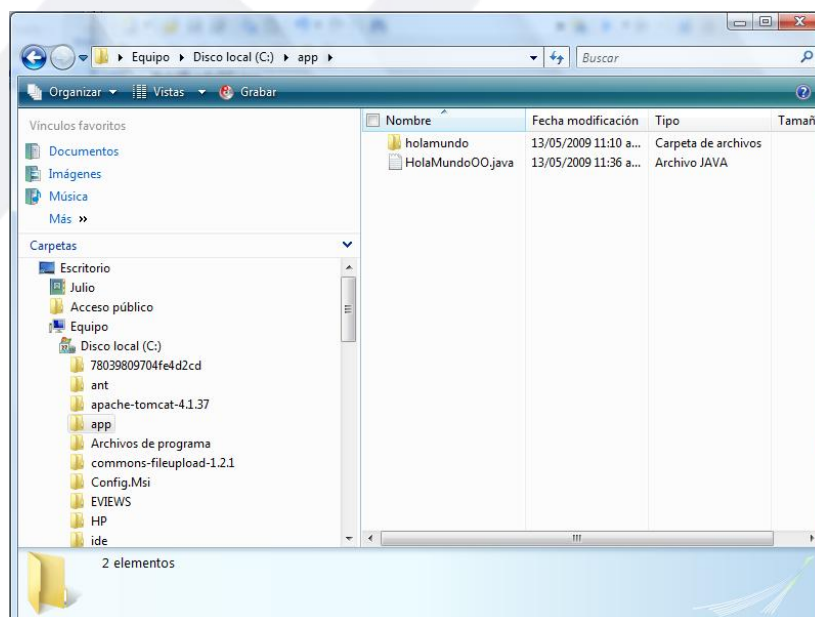
package holamundo;//Los paquetes son subdirectorios
import java.lang.String;
import java.lang.System;
public class HolaMundoOO{
    String saludo;
    //Creamos el método constructor con el mismo nombre de la clase
    public HolaMundoOO(){
        saludo="Hola mundo";//el constructor inicializa las propiedades
    }
    public void mostrarSaludo(){
        System.out.println(saludo);
    }
}

```

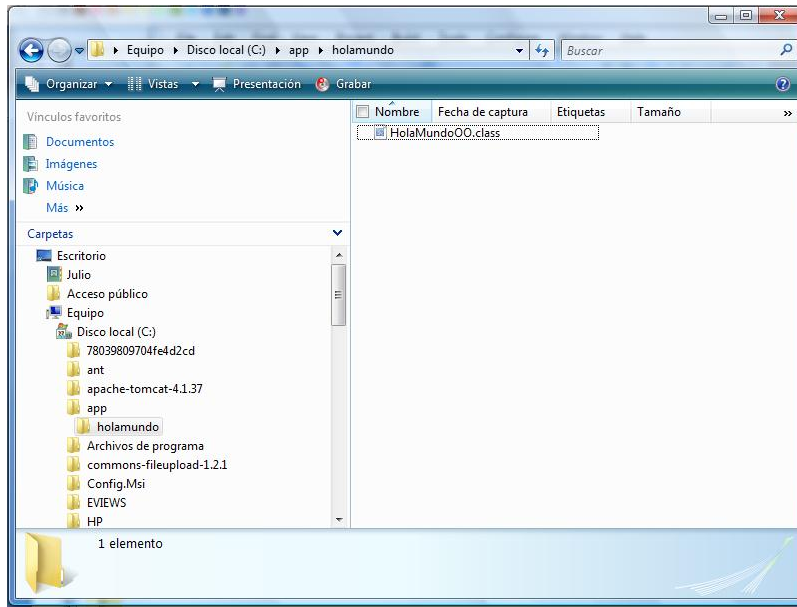
Si aplicamos al programa anterior los mismos pasos del apartado de instalación para probar JCreator, excepto el de ejecución; es decir, crear un nuevo documento vacío con el mismo nombre de la clase, copiar el código anterior y oprimir el botón de compilación debemos obtener algo parecido a esto:



El efecto de compilar es que se crea un archivo .java y el paquete es un subdirectorio:



Dentro del paquete queda el archivo .class, que es el que la máquina virtual puede interpretar:



## ***Ejecución y flujo de datos***

### **Ejecución**

Falta decir que en toda aplicación Java debe haber una clase que tenga un método *main* que es el primero que se ejecuta. La clase que hemos creado no lo tiene porque no es su responsabilidad, si nos apegamos al proceso de abstracción según el cual la diseñamos. Para eso haremos otra clase llamada *EjecutorHolaMundoOO* cuya responsabilidad será correr el programa y que tendrá como propiedad un objeto de la clase *HolaMundoOO*. Esto nos servirá como práctica de sintaxis y para analizar el flujo de datos en Java.

Hagámoslo por pasos, ámbito por ámbito:

1. Empaquetamos en *util*, importamos *HolaMundoOO* y declaramos la clase:

```
package util;
import holamundo.HolaMundoOO;
public class EjecutorHolaMundoOO {

}
```

2. Creamos una propiedad del tipo o clase *HolaMundoOO* con el identificador *hola*:

```
package util;
import holamundo.HolaMundoOO;
public class EjecutorHolaMundoOO {
    HolaMundoOO hola;

}
```

### 3. Añadimos los métodos:

3.1 Agregamos el constructor en el que inicializamos la propiedad *hola* e invocamos el método *mostrarSaludo()* de dicho objeto, utilizando el punto(.), que es el metacaracter que nos permite invocar los atributos y los métodos de los objetos:

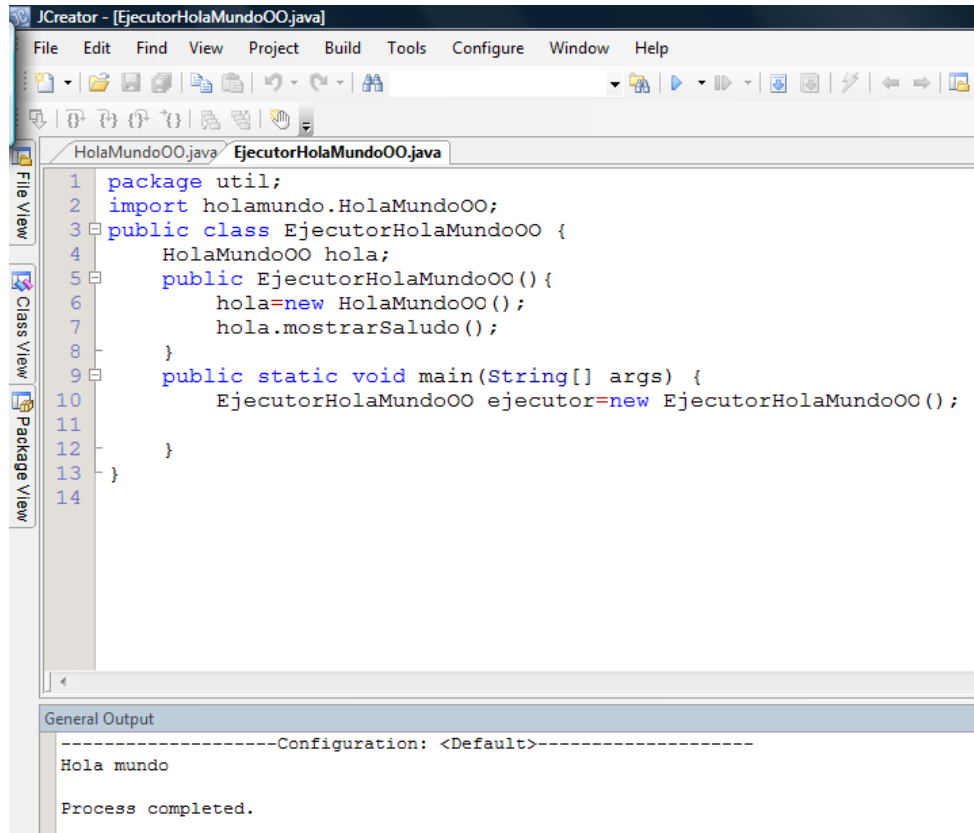
```
package util;
import holamundo.HolaMundoOO;
public class EjecutorHolaMundoOO {
    HolaMundoOO hola;
    public EjecutorHolaMundoOO(){
        hola=new HolaMundoOO();
        hola.mostrarSaludo();
    }
}
```

3.2 Agregamos el método *main*. Ni la palabra reservada *static* ni lo escrito en los paréntesis del método debe inquietarnos. Por el momento, basta con saber que la sintaxis es rigurosa. El método que ejecuta la aplicación debe escribirse así, lo único que cambia es el contenido del bloque de código:

```
package util;
import holamundo.HolaMundoOO;
public class EjecutorHolaMundoOO {
    HolaMundoOO hola;
    public EjecutorHolaMundoOO(){
        hola=new HolaMundoOO();
        hola.mostrarSaludo();
    }
    public static void main(String[] args) {
        EjecutorHolaMundoOO ejecutor=new EjecutorHolaMundoOO();
    }
}
```



En JCreator, en un archivo vacío llamado igual que la clase (*EjecutorHolaMundoOO*) y que debe estar en el mismo subdirectorio que el *HolaMundoOO.java*, escribimos el código; Compilamos y ejecutamos el programa, con lo que debemos obtener el siguiente resultado:



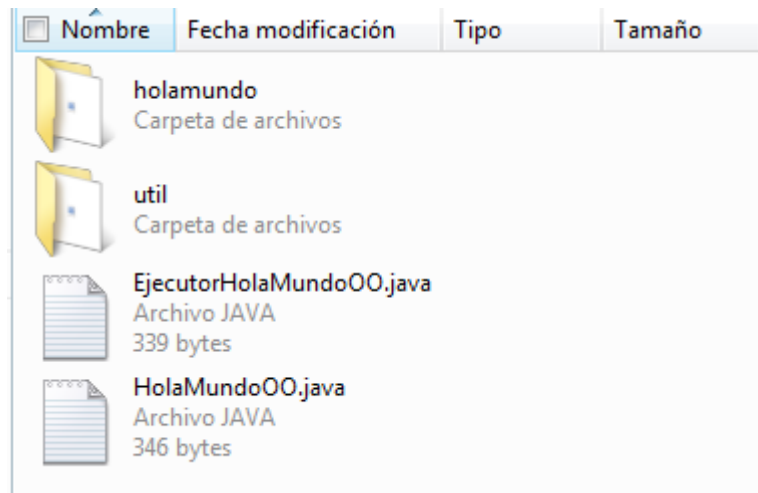
The screenshot shows the JCreator IDE with the following code in the main editor:

```
1 package util;
2 import holamundo.HolaMundoOO;
3 public class EjecutorHolaMundoOO {
4     HolaMundoOO hola;
5     public EjecutorHolaMundoOO() {
6         hola=new HolaMundoOO();
7         hola.mostrarSaludo();
8     }
9     public static void main(String[] args) {
10        EjecutorHolaMundoOO ejecutor=new EjecutorHolaMundoOO();
11    }
12 }
13 }
14 }
```

The General Output window at the bottom shows the following output:

```
-----Configuration: <Default>-----
Hola mundo
Process completed.
```

La estructura de directorios debe presentar los archivos *.java* y los dos subdirectorios de los paquetes dentro de los cuales debe estar cada uno de los archivos *.class*:



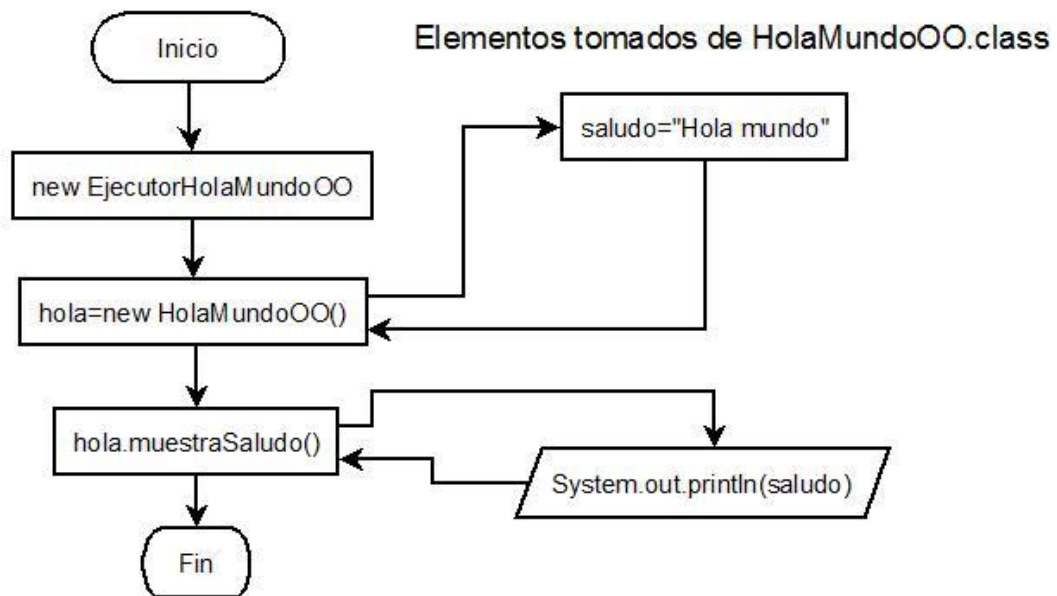
## Flujo de datos

Lo que sigue es explicar cómo funciona lo anterior. Cuando oprimimos el botón ejecutar, el compilador de Java busca el método *main*, si no lo encuentra, mostrará un error. En nuestro caso, esta es la sucesión de pasos:

1. Comienza a recorrer el bloque de código del método *main* que contiene un objeto de la clase *EjecutorHolaMundoOO* que llamamos *ejecutor* y que inicializamos usando la palabra reservada *new* y el método constructor de dicha clase.
2. Esto inicializa el objeto *hola* con el constructor de la clase *HolaMundoOO*,
  - a. Que llena el objeto *saludo* con la cadena “Hola mundo”
3. Después, en el constructor de la clase *EjecutorHolaMundoOO*, se invoca al método *muestraSaludo()* del objeto *hola*
  - a. Que envía, en una línea, la variable *saludo* a la salida estándar del sistema, que en JCreator es la ventana *General Output*.

El siguiente diagrama muestra el proceso anterior:

## Flujo principal EjecutorHolaMundoOO.class



## Tipos de datos

### Generalidades

En Java, las variables podrán ser de dos tipos: objetos y tipos primitivos. Hay diferencias entre unos y otros. En realidad las primeras clases que conforman la plataforma están hechas sólo de tipos primitivos. Estos se obtienen con las palabras reservadas: *byte*, *short*, *int*, *long*, *float*, *double*, *char* y *boolean*. En este curso usaremos solamente *int* para enteros, *double* para números con punto decimal, y *boolean* para los valores lógicos. Por su parte, los objetos siempre son tomados de una clase que lo mismo puede estar en alguna librería del JDK o en algún archivo creado por nosotros.

Siempre que se necesite una variable **debe** ser declarada especificando su tipo seguido de su identificador (ya habíamos mostrado esto):

`int valor;` → Declara una variable para números enteros

`valor=12;` → Le asigna un valor

`String cadena;` → Declara una variable que es un objeto de tipo String

cadena= "palabras" → Le asigna un valor

Ya antes dijimos que ambas tareas se pueden hacer en la misma línea:

```
int valor=12
```

La diferencia central entre los tipos primitivos y los objetos es que éstos disponen de los atributos y los métodos que les confieren las clases a las que pertenecen; como también ya dijimos, para acceder a ellos se usa el metacaracter (.):

```
cadena.length()
```

Aquí debemos hablar de las *Java Foundation Classes* (JFC) y de su *Application Programming Interface* (API); es decir, de las **clases base de Java** y su **interfaz de programación de aplicaciones**. La descripción de los métodos, de los atributos y de las propias clases base con las que programaremos están en la API, que está publicada en:

<http://java.sun.com/javase/api/>

Las JFC en sí están en el JDK y por eso lo descargamos e instalamos. Hemos usado dos de ellas: *String* y *System*; sólo hemos invocado la propiedad *out* de ésta; no obstante, cada una tiene muchos métodos y muchas propiedades que están descritas en la API; de cuyo conocimiento dependen las alternativas de programación de las que disponemos. Cada método o propiedad mencionados aquí serán explicados sencillamente. La profundización en todas las posibilidades de cada clase es responsabilidad del destinatario de este curso. Visitar el *site* de la API no es una elección para el programador de Java, es una exigencia obligatoria. Se puede decir que del mayor conocimiento de la API depende la capacidad de programación en el lenguaje.

## Los tipos de datos y los métodos

Ya dijimos que los métodos pueden devolver cualquier tipo de datos; falta agregar que también pueden recibir parámetros. Esto hace que existan cuatro tipos de métodos:

Sintaxis	Devuelve valor	Recibe parámetros
void tarea()	No	No
Tipo tarea()	Sí	No
void tarea(Tipo parametro)	No	Sí
Tipo metodo(Tipo parametro)	Sí	Sí

El primero de ellos lo usamos en la clase HolaMundoOO y es muestraSaludo(). Veamos un demo para cada una de los otros casos.

### Caso de método que sí devuelve un tipo y no recibe parámetros

Estúdiese el siguiente programa:

DemoMetodoSiTipoNoParametros.java

```
public class DemoMetodoSiTipoNoParametros {
    String saludo;
    public DemoMetodoSiTipoNoParametros() {
        /*En la expresión siguiente el atributo
        *saludo es llenado con lo que devuelve el
        *método
        */
        saludo=devuelveHolaMundo();
    }
}
```

```

        muestraSaludo();
    }
    //Obsérvese que el método devuelve un objeto de la clase String
    public String devuelveHolaMundo(){
        String hola="Hola mundo";
        /*return es obligatorio si el método devuelve
        *un valor. El modificador de return debe ser del mismo
        *tipo que el valor devuelto
        */
        return hola;
    }
    public void muestraSaludo(){
        System.out.println(saludo);
    }
    public static void main(String[] args) {
        new DemoMetodoSiTipoNoParametros();
    }
}

```

En este código, la propiedad *saludo* es inicializada no con una cadena sino con el valor que regresa el método *devuelveHolaMundo()* cuyo tipo es *String*. En dicho método hemos creado otra variable del mismo tipo, y la hemos inicializado con la cadena “Hola mundo”. La palabra reservada *return* sirve para indicar cuál es el valor que devolverán los métodos.

### **Caso de método que no devuelve un tipo y sí recibe parámetros**

En el programa siguiente, que es una variante del Hola Mundo, en el constructor se inicializa la variable ‘saludo’ y es mandada como parámetro al método *muestraSaludo()*, que copia el contenido a la variable ‘recibido’.

DemoMetodoNoTipoSiParametros.java

```

public class DemoMetodoNoTipoSiParametros {
    String saludo;
    public DemoMetodoNoTipoSiParametros() {
        saludo="Hola mundo";
        //Se envía la variable 'saludo' para que el método la use como parámetro
        muestraSaludo(saludo);
    }
    /*El contenido de 'saludo' es escrito en la variable
    'recibido' del parámetro del método muestraSaludo()*/
    public void muestraSaludo(String recibido){
        System.out.println(recibido);
    }
    public static void main(String[] args) {
        new DemoMetodoNoTipoSiParametros();
    }
}

```

## Caso de método que sí devuelve un tipo y sí recibe parámetros

Es fácil deducir de los ejemplos anteriores el último caso. No obstante, falta decir que mientras que los métodos pueden devolver sólo un tipo, el número de parámetros puede ser múltiple. Véase el ejemplo que calcula el área de un triángulo con la base y la altura como argumentos:

DemoMetodoSiTipoSiParametros.java

```

public class DemoMetodoSiTipoSiParametros {
    double resultado;
    public DemoMetodoSiTipoSiParametros() {
        /*resultado se llena con el valor que devuelve
        *el método obtieneAreaTriangulo(), que a su vez
        *recibe los dos parámetros que usa en una fórmula.
        */
        resultado=obtieneAreaTriangulo(2.5,6.3);//Los parámetros se separan con comas
        System.out.println("El resultado es: "+resultado);
    }
    public double obtieneAreaTriangulo(double base, double altura){

```

```
        double area =(base*altura)/2;
        return area;
    }
    public static void main(String[] args) {
        new DemoMetodoSiTipoSiParametros();
    }
}
```

Falta decir solamente que los parámetros no deben ser necesariamente del mismo tipo y que cuando son más de uno se usa el metacaracter coma (,).

Un comentario final sobre la API. Cuando buscamos los atributos y los métodos de una clase, en la URL en cuestión se mostrará el tipo (aun si es *void*); y en el caso particular de los métodos se indicarán, también, los parámetros con su tipo. A partir de ahora, siempre que citemos una clase nueva de las JFC, será con un enlace a la API.

## ***Estructuras de control***

### **Introducción**

Para terminar la sección de iniciación al lenguaje es necesario explicar cómo se controla el flujo de datos. Para empezar, es útil decir que en todos los lenguajes de programación existen tres tipos de sentencias: las secuenciales, las selectivas y las repetitivas. Hasta ahora, sólo hemos usado el primer tipo, en el que las posibilidades del flujo de datos se reducen a una secuencia de pasos; para hacer que exista más de una alternativa o que cierta tarea se repita varias veces, en Java usa un grupo de palabras reservadas para controlar el flujo y todas ellas usan un bloque de código.

Para ver los ejemplos de cómo funcionan dichas estructuras usaremos la clase [\*JOptionPane\*](#) que sirve para mostrar cuadros de diálogo y tiene los métodos *showInputDialog()* y *showMessageDialog()* que se usan para pedir y dar información al



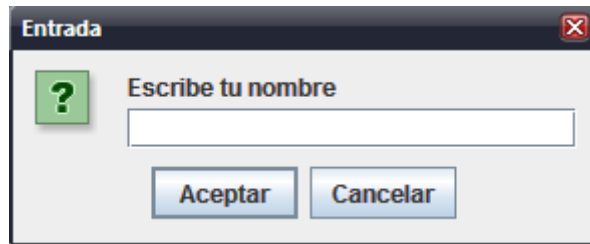
usuario, respectivamente; para ver cómo operan haremos una clase que será otra variante de nuestro tan traído HolaMundo, que primero pedirá el nombre del usuario y luego lo saludará.

Antes de seguir, hagamos un poco de abstracción. Obsérvese que lo que queremos hacer es una demostración de cómo la clase JOptionPane sirve como mecanismo de entrada y salida de datos, por eso la llamaremos DemoIOJOptionPane (IO de Input/Output); sus responsabilidades serán pedirle su nombre al usuario, y saludarlo en pantalla; dicho nombre será una propiedad. Cabe aclarar que como el sentido de la clase será dar una demostración, el método *main* sí entra dentro de sus capacidades.

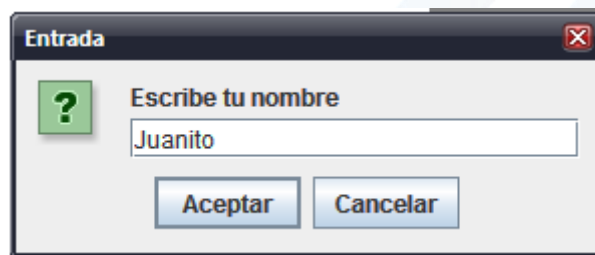
#### DemoIOJOptionPane.java

```
package cursojava.demos; //La encapsulamos en un lugar para todos los demos
import javax.swing.JOptionPane; //Importamos la clase JOptionPane
public class DemoIOJOptionPane {
    String nombre; //Este será la información para el I/O
    public DemoIOJOptionPane() {
        //El constructor llama a sus métodos
        pideNombre();
        muestraSaludo();
    }
    /*La palabra reservada null del primer atributo en los dos métodos
    *de JOptionPane es porque, por lo común, esta clase es llamada desde una ventana
    * que es su propietario como no tenemos tal ventana, lo indicamos así.
    * El segundo parámetro es el mensaje que aparecerá en el cuadro de diálogo.
    */
    private void pideNombre(){
        nombre=JOptionPane.showInputDialog(null,"Escribe tu nombre");//Pide el nombre
    }
    private void muestraSaludo(){
        JOptionPane.showMessageDialog(null,"Hola "+nombre);//Saluda en pantalla
    }
    public static void main(String[] args) {
        new DemoIOJOptionPane();
    }
}
```

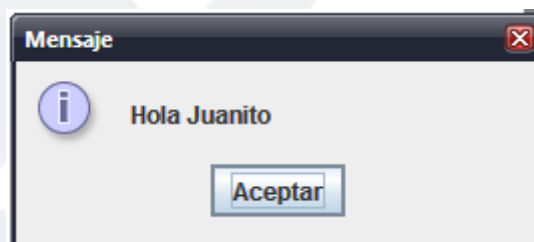
Al ejecutar el programa, aparece un cuadro de diálogo con el mensaje que le mandamos como segundo parámetro al método `showInputDialog()`:



Dentro del cuadro de texto se debe escribir un nombre y oprimir el botón <aceptar>:



Luego de lo cual aparecerá otro cuadro de diálogo con la cadena “Hola ” concatenada (usando el operador +) con el nombre escrito por el usuario; el valor obtenido de esta concatenación es lo que mandamos como segundo parámetro del método `showMessageDialog()`:



Con esto podemos empezar a trabajar con el flujo de datos. Si se ejecuta el programa y en el primer cuadro de diálogo se oprime <cancelar> en lugar de <aceptar>, la cadena que mostrará después será *Hola null*, que no es el resultado esperado, si se quisiera controlar esto, se tendría que controlar el flujo. Existen varias posibilidades para esto y las veremos en seguida.

## El bloque *if...else*

Analice y ejecute el siguiente programa (que es una variante del anterior) oprimiendo el botón cancelar:

```
package cursojava.demos; //La encapsulamos en un lugar para todos los demos
import javax.swing.JOptionPane; //Importamos la clase JOptionPane
public class DemoIfElse {
    String nombre; //Este será la información para el I/O
    public DemoIfElse () {
        //El constructor llama a sus métodos
        pideNombre();
        muestraSaludo();
    }
    private void pideNombre(){
        nombre=JOptionPane.showInputDialog(null,"Escribe tu nombre");//Pide el nombre
    }
    private void muestraSaludo(){
        //Inicio de la estructura de control if...else
        if(nombre==null)  { //Obsérvese que el operador de comparación es == (doble igual)
            JOptionPane.showMessageDialog(null,"Oprimiste cancelar");//Mensaje por la cancelación
        }else {
            JOptionPane.showMessageDialog(null,"Hola "+nombre);//Saluda en pantalla
        }
        //Fin de la estructura
    }
    public static void main(String[] args) {
        new DemoIfElse ();
    }
}
```

En este caso, si el usuario decide no escribir el nombre y cancelar, el programa lo muestra.

## if.. else anidado

Aunque hemos avanzado, el programa sigue teniendo problemas. Si el usuario no escribe el nombre pero oprime el botón aceptar, la cadena mostrada será *Hola* solamente; si queremos que aparezca otro mensaje para este caso, hay más de una solución; una de ellas es anidar un *if...else* dentro del que ya hicimos:

### DemoIfElseAnidado.java

```
package cursojava.demos; //La encapsulamos en un lugar para todos los demos
import javax.swing.JOptionPane; //Importamos la clase JOptionPane
public class DemoIfElseAnidado{
    String nombre; //Este será la información para el I/O
    public DemoIfElseAnidado() {
        //El constructor llama a sus métodos
        pideNombre();
        muestraSaludo();
    }
    /*La palabra reservada null del primer atributo en los dos métodos
    *de JOptionPane es porque, por lo común, esta clase es llamada desde una ventana
    * que es su propietario como no tenemos tal ventana, lo indicamos así
    */
    private void pideNombre(){
        nombre=JOptionPane.showInputDialog(null,"Escribe tu nombre");//Pide el nombre
    }
    private void muestraSaludo(){
        //Inicio de la estructura de control if...else
        if(nombre==null) {
            JOptionPane.showMessageDialog(null,"Oprimiste cancelar");//Mensaje por la cancelación
        }else
            if(nombre.equals("")){//Las cadenas no se comparan con == sino con el método equals()

```

```

        JOptionPane.showMessageDialog(null,"Oprimiste aceptar sin escribir tu nombre");
    }else{
        {
            JOptionPane.showMessageDialog(null,"Hola "+nombre);//Saluda en pantalla
        }
    }
}
}
public static void main(String[] args) {
    new DemoIfElseAnidado();
}
}

```

En este caso, el programa podrá enviar tres respuestas distintas:

1. Mandar el saludo
2. Indicar que se oprimió cancelar
3. Indicar que no se escribió el nombre

## Operadores lógicos

Se puede simplificar el programa anterior utilizando operaciones lógicas. Esto se refiere a que los valores de las variables, al compararse, siguen ciertas normas. En el programa que hemos estado trabajando ya mostramos que el argumento que recibe la palabra reservada *if* es un valor booleano. Es decir, una expresión que devuelve verdadero o falso (para las que se usa las palabras reservadas *true* y *false*, respectivamente). En este tipo de expresiones se usan operadores especiales (ya usamos el de asignación), que son binarios, es decir que tienen un operando a cada lado, y que se muestran en la siguiente tabla:

Operador	Significado	Es <i>true</i> cuando:	Es <i>false</i> cuando:
!	Negación. Cambia el valor de verdad del objeto que modifica	El objeto que modifica es falso	El objeto que modifica es verdadero
==	Compara si los miembros que están a ambos lados son iguales	Ambos objetos son iguales	Los objetos son distintos
!=	Compara si los miembros que están a ambos lados son distintos	Los objetos son distintos	Ambos objetos son iguales
>	Compara si el objeto de la izquierda es mayor que el de la derecha	Si el objeto de la izquierda es mayor que el de la derecha	Si el objeto de la izquierda no es mayor que el de la derecha
<	Compara si el objeto de la izquierda es menor que el de la derecha	Si el objeto de la izquierda es menor que el de la derecha	Si el objeto de la izquierda no es menor que el de la derecha
>=	Compara si el objeto de la izquierda es mayor o igual que el de la derecha	Si el objeto de la izquierda es mayor o igual que el de la derecha	Si el objeto de la izquierda no es mayor ni igual que el de la derecha
<=	Compara si el objeto de la izquierda es menor o igual que el de la derecha	Si el objeto de la izquierda es menor o igual que el de la derecha	Si el objeto de la izquierda no es menor ni igual que el de la derecha

Junto con estos, existen otros que sirven para hacer operaciones entre objetos boléanos.

Los más importantes se presentan a continuación

## AND

El operador *AND* se representa en Java con un doble *ampersand* (&&) y su funcionamiento es de acuerdo con la siguiente tabla:

Operando 1	Operador	Operando 2	Resultado
True	&&	True	True
False	&&	True	False
True	&&	False	False
False	&&	False	False

## OR

El operador *OR* se representa en Java con un doble *pipe* (`||`). Este caracter normalmente está en la misma tecla del número 1. Es esa línea vertical cortada en el centro. Su código ASCII es 124); funciona de acuerdo con la siguiente tabla:

Operando 1	Operador	Operando 2	Resultado
True		True	True
False		True	True
True		False	True
False		False	False

Con lo anterior podemos hacer una nueva versión del programa en el que, en lugar de anidar las estructuras de control, usaremos el operador *OR* (`||`):

DemoIfElseOr.java

```
package cursojava.demos; //La encapsulamos en un lugar para todos los demos
import javax.swing.JOptionPane; //Importamos la clase JOptionPane
public class DemoIfElseOr{
    String nombre;
    public DemoIfElseOr() {
        pideNombre();
        muestraSaludo();
    }
    private void pideNombre(){
        nombre=JOptionPane.showInputDialog(null,"Escribe tu nombre");//Pide el nombre
    }
    private void muestraSaludo(){
        if(nombre==null || nombre.equals("")){//Uso del operador OR
            JOptionPane.showMessageDialog(null,"No diste tu nombre");//Mensaje de error
        }else{
            JOptionPane.showMessageDialog(null,"Hola "+nombre);//Saluda en pantalla
        }
    }
}
```

```

    }
    public static void main(String[] args) {
        new DemoIfElseOr();
    }
}

```

## La estructura de control *while*

Si suponemos que en el programa anterior el usuario *debe* escribir su nombre, podemos hacer que el flujo de datos repita la operación de solicitarlo *mientras* el usuario no proporcione la información. Para eso usamos *while*:

DemoWhile.java

```

package cursojava.demos; //La encapsulamos en un lugar para todos los demos
import javax.swing.JOptionPane; //Importamos la clase JOptionPane
public class DemoWhile{
    String nombre;
    public DemoWhile() {
        pideNombre();
        muestraSaludo();
    }
    private void pideNombre(){
        nombre=JOptionPane.showInputDialog(null,"Escribe tu nombre");//Pide el nombre
    }
    private void muestraSaludo(){
        //Inicio del bucle while
        while(nombre==null||nombre.equals("")){
            nombre=JOptionPane.showInputDialog(null,"Debes escribir tu nombre");
        }
        JOptionPane.showMessageDialog(null,"Hola "+nombre);//Saluda en pantalla
    }
    public static void main(String[] args) {
        new DemoWhile();
    }
}

```



El programa anterior mostrará reiterativamente el cuadro de diálogo que solicita el nombre, mientras el usuario no escriba una cadena.

## La estructura de control *try...catch*

El bloque *try...catch* es un poco más complejo que el *for* y el *switch*, que veremos después, pero el uso combinado con *try catch* facilita su comprensión por lo que veremos primero éste.

La estructura de control en cuestión se usa para capturar el flujo de datos cuando, por la intervención del usuario o de algún dispositivo externo, sucede un error. Por ejemplo, supongamos una aplicación de fórmulas matemáticas en la que el usuario debe ingresar valores numéricos, lo que normalmente sucede es que el sistema recibirá cadenas que deberán ser convertidas a un tipo útil para las operaciones; pero si el usuario ingresa algo distinto a un dígito o a un operador, el programa fallará. Veamos un ejemplo:

DemoErrorNumero.java

```
import javax.swing.JOptionPane;
public class DemoErrorNumero {
    String mensaje;
    public DemoErrorNumero() {
        String valorCadena=JOptionPane.showInputDialog(null,"Escribe un entero");
        /*El método parseInt() de la clase Integer de la librería lang
        *convierte un String en un int
        */
        int valorNumero=Integer.parseInt(valorCadena);
        JOptionPane.showMessageDialog(null,"El valor es "+valorCadena);
    }
    public static void main(String[] args) {
        new DemoErrorNumero();
    }
}
```

Al ejecutar este programa, si en el primer cuadro de diálogo escribimos un entero, aparecerá un segundo cuadro de diálogo indicando el valor; si, por el contrario, escribimos otra cosa, el programa se detiene y en la ventana de *General Output* de JCreator se mostrará el siguiente mensaje:

```
-----Configuration: <Default>-----  
Exception in thread "main" java.lang.NumberFormatException: For input string: "Juanito"  
  at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)  
  at java.lang.Integer.parseInt(Integer.java:447)  
  at java.lang.Integer.parseInt(Integer.java:497)  
  at DemoErrorNumero.<init>(DemoErrorNumero.java:6)  
  at DemoErrorNumero.main(DemoErrorNumero.java:10)
```

La línea que esta subrayada indica cuál fue el error: la cadena 'Juanito' (que fue lo que recibió la variable *valorCadena*) tiene un formato numérico incorrecto; *java.lang* es la librería en la que está la clase *NumberFormatException* que es capaz capturar esos errores y se usa con el *try...catch* como se muestra en el siguiente ejemplo:

#### DemoTryCatch.java

```
import javax.swing.JOptionPane;  
public class DemoTryCatch {  
    String mensaje;  
    public DemoTryCatch() {  
        String valorCadena=JOptionPane.showInputDialog(null,"Escribe un entero");  
        try {  
            int valorNumero=Integer.parseInt(valorCadena);  
            /*Si lo escrito no es un entero la línea que sigue no se ejecuta,  
            *el programa busca el bloque catch y ejecuta su contenido  
            */  
            mensaje="Gracias";  
        }  
        catch (NumberFormatException ex) { //El bloque catch indica el error que captura.  
            mensaje="No escribiste un Entero";  
        }  
        JOptionPane.showMessageDialog(null,mensaje);//El mensaje enviado según el caso
```

```

    }
    public static void main(String[] args) {
        new DemoTryCatch();
    }
}

```

Se debe saber que existe la clase [Exception](#) que captura todos los errores, y que derivadas de ella hay muchas otras para atender faltas específicas, como el caso que acabamos de probar. Más adelante en este curso, veremos algunas otras excepciones.

## El bloque *for*

El bloque *for* sirve para hacer ciclos controlados por un contador. La sintaxis que exige en sus parámetros es más compleja que las anteriores:

Parámetro	Utilidad
Primero	Inicializa el contador y debe ser un entero
Segundo	Establece mediante un operador boléano (> , <, <=, etc) el límite final del contador
Tercero	Establece el ritmo de incremento

```

    Inicia el contador          En cada ciclo
                               se incrementará en uno
    {
    for(int i=0; i<=10; i++){
    }
    }
    Se ejecutará mientras
    sea menor o igual a 10

```

La expresión de *i++* significa que al valor de *i* debe agregársele 1. ++ es el operador de incremento; también existe el de decremento (--) que resta en uno al valor que modifica.

Para ver cómo funciona el bucle *for* hagamos un ejemplo combinado con un *try...catch* que nos muestre la tabla de multiplicar solicitada:

```
import javax.swing.JOptionPane;
public class DemoFor {
    String mensaje;
    public DemoFor() {
        String valorCadena=JOptionPane.showInputDialog(null,"Qué tabla de multiplicar que desea");
        try{
            int valorEntero=Integer.parseInt(valorCadena);//Si esto falla, entra al catch
            mensaje= "Tabla del "+valorCadena+"\n";// \n agrega un fin de línea a las cadenas
            for(int i=1;i<=10;i++){
                /*La siguiente línea se repetirá diez veces y concatenará
                *cadenas del tipo "5x4=20" a la cabecera puesta antes del bucle
                */
                mensaje=mensaje+i+"x"+valorCadena+"="+i*valorEntero+"\n";
            }
        }catch(NumberFormatException ex){
            mensaje="No es un entero";
        }
        JOptionPane.showMessageDialog(null,mensaje);
    }
    public static void main(String[] args) {
        new DemoFor();
    }
}
```

Este programa muestra la tabla de multiplicar solicitada en un cuadro de diálogo, a menos que el usuario no haya escrito un número, ante lo cual mostrará el mensaje “No es un entero”

## El bloque *switch*

Esta es una estructura de control selectiva que elige de entre un grupo de alternativas en virtud de una variable entera. Esta aplicación solicita un valor entre cero y diez; en virtud de lo obtenido, elige el *case*; si se da un número distinto a los requeridos, se elige

la opción *default*; gracias al bloque *try...catch*, si el usuario no ingresa un número, aparece un mensaje indicándoselo. Pruébese varias veces el siguiente programa ingresando diferentes valores para ver su funcionamiento pleno:

#### DemoSwitch.java

```
import javax.swing.JOptionPane;
public class DemoSwitch {
    String mensaje;
    public DemoSwitch() {
        String califCadena=JOptionPane.showInputDialog(null, "Escriba la calificación con número");
        try{
            int califNum=Integer.parseInt(califCadena);//Si esta línea falla entra al catch
            String calif;
            switch(califNum){
                /*Los casos se eligen en virtud del valor
                *de la variable califNum
                */
                case 0: calif="NA";break;
                case 1: calif="NA";break;
                case 2: calif="NA";break;
                case 3: calif="NA";break;
                case 4: calif="NA";break;
                case 5: calif="NA";break;
                case 6: calif="S";break;
                case 7: calif="S";break;
                case 8: calif="B";break;
                case 9: calif="MB";break;
                case 10: calif="MB";break;
                default: calif="Inválida";break;
                /*El default sucede en el caso de que se de un número
                *distinto de los casos indicados. Es decir, un número
                *entre 0 y 10
                */
            }
            mensaje="La calificación es: "+calif;
        }catch(NumberFormatException ex){
            mensaje="No escribió un número";
        }
    }
}
```

```
        JOptionPane.showMessageDialog(null,mensaje);
    }
    public static void main(String[] args) {
        new DemoSwitch();
    }
}
```

Con esto terminamos el tema de las estructuras de control. Téngase en cuenta que en todos los casos el flujo de datos puede cambiar en virtud de las posibles respuestas del usuario; y que todas las estructuras vistas aquí son *anidables* prácticamente sin restricciones.

# Segunda parte. Aplicación de ejemplo

Como hasta ahora, cuando mencionemos una clase de la API, aparecerá en una liga que nos remitirá a la página *web* que la describe. No obstante, pondremos en el código una descripción de lo que hace cada uno de los métodos usados aun cuando en muchos de ellos es obvio. Además, antes de cada clase haremos una descripción de su funcionamiento. Sin embargo, antes de comenzar a mostrar la aplicación hay ciertos temas que es necesario agregar para comprender mejor su funcionamiento.

## ***Consideraciones preliminares***

### **Herencia y polimorfismo**

Para facilitar el diseño de una aplicación, se debe usar la herencia de la programación orientada a objetos. Para explicar este término, haremos otro Hola Mundo que usará una ventana, que contendrá una etiqueta; esto requiere tres clases de la librería *swing*: [\*JFrame\*](#), [\*JPanel\*](#) y [\*JLabel\*](#). Veamos la primera versión, sin orientación a objetos ni herencia, en la que las clases colaboran para lograr los objetivos:

DemoClasesColaborando.java

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JLabel;
public class DemoClasesColaborando {
    public DemoClasesColaborando() {
        JFrame ventana= new JFrame();
        JPanel panel=new JPanel();
        JLabel etiqueta= new JLabel();
        //Asigna un texto a la etiqueta
```

```

    etiqueta.setText("Hola Mundo");
    //Agrega la etiqueta al panel
    panel.add(etiqueta);
    //Agrega el panel a la ventana
    ventana.add(panel);
    //Ajusta el tamaño de la ventana al mínimo necesario
    ventana.pack();
    //Asigna el estado de visibilidad de la ventana a verdadero
    ventana.setVisible(true);
}
public static void main(String[] args) {
    new DemoClasesColaborando();
}
}

```

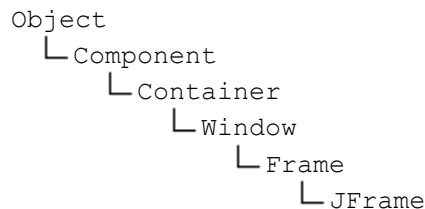
Después de compilar y ejecutar el código anterior, se presenta la siguiente ventana en la esquina superior derecha de la pantalla:



Cada una de las clases que estamos usando pertenece a tres ámbitos distintos: *JFrame* es una ventana que sirve como contenedor principal; *JPanel* es un contenedor de propósito general que se usará para albergar a los componentes como *JLabel*, o los botones, los campos de texto, etc. Todas están colaborando con la clase que acabamos de compilar (de ahí su nombre) para lograr el objetivo de mostrar en la etiqueta la cadena “Hola mundo”. Es posible continuar en este camino y agregar otros componentes al panel, más paneles a la ventana y, por supuesto, más componentes a cada uno de éstos nuevos contenedores. No obstante, esto es una mala práctica. Muy frecuentemente, es conveniente crear clases que al momento de declararlas obtengan todos los métodos y las propiedades de otra; a esto se le conoce como ‘herencia’ en la POO. De hecho, todas las clases (aun las que creamos nosotros) heredan de la clase



Object si no se indica otra cosa; dicha clase tiene casi una docena de métodos que están disponibles incluso antes de que escribamos miembros con nuestra propia programación. Una de las ventajas de la herencia es que las clases se pueden ir especializando. Como ejemplo, presentamos enseguida todo el linaje de la clase JFrame:



De tal manera que esta clase tiene todos los métodos de las que le anteceden, además de los propios (que, en total, resultan ser cientos). Este beneficio lo tienen todas las instancias de JFrame (incluida la que colabora en el programa anterior). Pero las ventajas de la herencia van más lejos. Sucede que si yo quiero tratar al objeto *ventana* como *Object* lo puede hacer, como se muestra en el siguiente código:

DemoVentanaComoObjeto.java

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JLabel;
public class DemoVentanaComoObjeto {
    public DemoVentanaComoObjeto() {
        JFrame ventana= new JFrame();//ventana es un JFrame
        JPanel panel=new JPanel();
        JLabel etiqueta= new JLabel();
        //Tratamos a la ventana como Object
        Object objetoVentana=ventana;//ventana también es un Object
        /*Usamos el método getClass() de la clase Object
        *y el getString () de Class para obtener
        *el nombre de la clase a la que pertenece ventana
        */
    }
}
```

```

String claseObjeto=objetoVentana.getClass().toString();
etiqueta.setText(claseObjeto);
panel.add(etiqueta);
ventana.add(panel);
ventana.pack();
ventana.setVisible(true);
}
public static void main(String[] args) {
    new DemoVentanaComoObjeto();
}
}

```

Este programa usa los métodos de la clase *Object* para obtener el nombre de la clase a la que pertenece el objeto *ventana*. Lo anterior significa que la herencia vuelve polimórficas a las clases: se les puede tratar como si fueran cualquiera de aquellas que están en su linaje (y de hecho *son* cada una de ellas).

Muy frecuentemente será conveniente hacer clases que hereden de otra. En un buen diseño, si necesitamos usar una ventana, conviene que la clase sea un *JFrame*. La herencia se logra con la palabra reservada *extends* escrita a reglón seguido después del nombre de la clase y sólo se puede heredar de otra. El código siguiente es nuestro tan traído Hola Mundo pero ahora usando los conceptos que estamos explicando.

#### DemoHeredaJFrame.java

```

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JLabel;
//Obsérvese la sintaxis de extends
public class DemoHeredaJFrame extends JFrame {
    public DemoHeredaJFrame() {
        /*Obsérvese que a estos tres métodos los usamos igual
        *aunque solo organizaComponentes()está escrito en
        *el bloque de código de la clase.
        *Los otros dos los heredamos de alguna de las clases

```

```

        *del linaje de JFrame
        */
        organizaComponentes();
        pack();
        setVisible(true);
    }
    private void organizaComponentes(){
        JPanel panel=new JPanel();
        JLabel etiqueta= new JLabel();
        etiqueta.setText("Hola mundo");
        add(panel);
        panel.add(etiqueta);
    }

    public static void main(String[] args) {
        new DemoHeredaJFrame();
    }
}

```

En este caso hemos encapsulado la tarea de agregar los componentes de la ventana en un método para mostrar que, en términos de sintaxis, los métodos propios de las clases se portan igual que los heredados. Por otro lado, al haber extendido los alcances de nuestra clase para heredar los de JFrame, se convierte en parte su linaje; ahora es polimórfica porque *es* cada una de las que presentamos a continuación:

```

Object
├── Component
│   ├── Container
│   │   ├── Window
│   │   │   ├── Frame
│   │   │   │   ├── JFrame
│   │   │   │   │   └── DemoHeredaJFrame

```

## Captura de eventos

Desde el apartado anterior hemos estado tratando conceptos que son del entorno de las interfaces gráficas: ventanas, componentes, etc., que usan privilegiadamente los

sistemas operativos comunes de la actualidad. Junto con estos conceptos, que se relacionan con objetos visuales, está el tema de los eventos que se desencadenan por las acciones del usuario sobre éstos con los dispositivos de entrada tales como el teclado y el *Mouse* entre otros.

De dichos eventos el más usado es, precisamente, el clic del *Mouse*, de cuya programación se pueden obtener muchos beneficios; de hecho, en nuestra aplicación de ejemplo sólo usaremos este caso. Para hacer que una clase capture los eventos es necesario usar otra palabra reservada: *implements*, que sirve para agregar interfaces a las clases, y que se usa también en la línea de la declaración de éstas, después de la sentencia *extends* (si existe); es útil decir que se puede implementar más de una interfaz. Todos los programas que requieran responder al clic del *Mouse* usarán la interfaz [ActionListener](#) que nos obliga a agregar el método *actionPerformed()* a nuestra clase (Las interfaces son clases especiales que es inútil explicar aquí y que casi siempre exigen que se agreguen métodos específicos). Los clics capturados remitirán el flujo del programa al método *actionPerformed()* que recibe como parámetro objetos de la clase [ActionEvent](#) .

Para mostrar esto haremos una clase que sea un *JFrame*, que contenga un [JButton](#) y que al recibir un clic mande un mensaje:

DemoEventoCapturado.java

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JButton;
import javax.swing.JOptionPane;
/*Las clases ActionListener y ActionEvent
 *son de la librería awt (Abstract Windows Toolkit)*/
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
/*Las interfaces se implementan
```

```

*después de la definición de extends */
public class DemoEventoCapturado extends JFrame implements ActionListener{
    JButton elBoton;//Es atributo porque se usa en organizaComponentes() y en el actionPerformed()
    public DemoEventoCapturado() {
        organizaComponentes();
        pack();
        setVisible(true);
    }
    private void organizaComponentes(){
        JPanel panel=new JPanel();
        elBoton=new JButton("Haz clic");
        /*A los objetos que pueden recibir el clic
        *debe agregárseles un ActionListener
        *que lo reciben de la implementación de
        *la clase actual (this)*/
        elBoton.addActionListener(this);
        add(panel);
        panel.add(elBoton);
    }
    /*Todos los objetos que tienen un ActionListener de la clase
    *remitirán el flujo del programa a este método
    *que es obligatorio y su sintaxis es así como se muestra
    */
    public void actionPerformed(ActionEvent evt){
        JOptionPane.showMessageDialog(this,"Gracias");
    }
    public static void main(String[] args) {
        new DemoEventoCapturado();
    }
}

```

## ***Aplicación de ejemplo***

### **Abstracción**

Para mostrar una aplicación funcional haremos un visor de *resultsets* para el servidor de bases de datos MySQL y trataremos de que esté orientado a objetos. Visualmente, no es difícil ver que se necesitarán al menos dos ventanas: aquella para la autenticación del usuario y la que sirve para escribir las consultas y mostrar sus resultados. Desde esta perspectiva, podemos establecer cuatro ámbitos:

- Conexión a datos
- Autenticación de usuario
- Consultas SQL
- Vista de resultados

Con esto hemos encapsulado lo que podríamos llamar los meta-objetos de nuestra aplicación. Esto significa que cada uno de los ítems anteriores se debe estructurar con al menos una clase, o con algunas en colaboración. Entonces, atenderemos cada una de ellos por separado.

### **Conexión**

Para conectarse a cualquier servidor de base de datos hace falta una aplicación que sirva como enlace. A ese tipo de clases se les llama conectores. En el caso de MySQL, debe descargarse de su sitio *web*. En dicha descarga, se obtiene todo el código fuente de las clases necesarias pero sólo se requiere el archivo [mysql-connector-java.jar](http://www.mysql.com/connector-java.jar). Este archivo debe ponerse en la siguiente ruta:

[JAVA-HOME]\jre\lib\ext

Hecho lo cual, estamos listos para escribir el siguiente código:

### Conector.java

```
package visorconsultas.controlador;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
public class Conector {
    Connection conexion;
    String error;
    /*Creamos un constructor que recibe cuatro parámetros:
    *El nombre del servidor MySQL, el del usuario, la contraseña, y
    *la base de datos*/
    public Conector(String host, String usuario, String pw, String base) {
        try {
            /*Esta línea crea una asociación entre nuestra aplicación
            * y la clase Driver que está empaquetada en el jar de conexión.*/
            Class.forName("com.mysql.jdbc.Driver");
            /*La conexión se obtiene con una cadena que usa
            *los parámetros que recibe el constructor*/
            conexion=DriverManager.getConnection("jdbc:mysql://"+host+"/"+base,usuario,pw);
        }
        catch(ClassNotFoundException e){//Sucede si no se encuentra el driver
            error=e.getMessage();
        }
        catch(SQLException e){//Sucede si la conexión falla
            error=e.getMessage();
        }
    }
    //Este método devuelve la conexión
    public Connection getConexion(){
        return conexion;
    }
    public void cierraConexion(){
        try {
            conexion.close();
        }
        catch (Exception ex) {
```

```

    }
}
//Este método devuelve el error que impide la conexión
public String getMensajeError(){
    return error;
}
}
}

```

El código anterior trata de obtener una conexión a MySQL usando las clases [Class](#), [Connection](#), [DriverManager](#), [ClassNotFoundException](#) y [SQLException](#); el mecanismo es como sigue:

1. Con el método *forname()* de [Class](#), se crea una asociación entre nuestra clase *Conector* y el *driver* de conexión MySQL. Si no lo logra, el error es capturado con la [ClassNotFoundException](#)
2. La conexión se realiza con el método *getConnection()* de [DriverManager](#); si el servidor no está disponible o refuta la solicitud, la conexión no se consuma y se dispara la [SQLException](#)
3. El método *dameConexion()*, es el que se encargará de poner a disposición la conexión para las clases que la requieran; mientras que *cierraConexion()* la cancela. La ejecución de los objetos de la librería *sql* exigen ser ejecutados dentro de un bloque *try...catch*, por eso está así la única línea que necesaria para el terminar la conexión.
4. Por supuesto, se debe tener acceso a una base de datos de un servidor MySQL mediante un nombre de usuario y una contraseña. Estos cuatro datos: base, servidor, usuario y *password*, los recibe como parámetro el constructor y luego los usa el método *getConnection()* de la clase *DriverManager*.



## Consultas SQL

Para hacer consultas a la base de datos hacen falta una serie de clases que colaboran para hacer acciones sucesivas: [Connection](#), [Statement](#), [ResultSet](#), [ResultSetMetaData](#) y la [SQLException](#).

### ConsultasSQL.java

```
package visorconsultas.controlador;
import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
public class ConsultaSQL {
    private Connection conexion;
    private ResultSet resultadoConsulta;
    private ResultSetMetaData metaDatos;
    private String consulta;
    //Crea dos arreglos
    private String[][] datosDevueltos;
    private String [] nombresColumnas;
    private String error;
    public ConsultaSQL(Connection conRecibida, String consultaRecibida){
        conexion=conRecibida;
        consulta=consultaRecibida;
        try {
            //Crea una instancia para mandar sentencias al servidor MySQL
            Statement sentencia=conexion.createStatement();
            //Ejecuta la consulta y devuelve el ResultSet
            resultadoConsulta=sentencia.executeQuery(consulta);
            //Obtiene los metadatos del ResultSet
            metaDatos=resultadoConsulta.getMetaData();
            error=null;
        }
        catch (SQLException e) {
            error=e.getMessage();
        }
    }
}
```

```

public String[][] getDatosDevueltos(){
    if(error==null){
        try {
            //Devuelve el número de columnas del resultset
            int columnas=metaDatos.getColumnCount();
            //Lleva el cursor a la última fila del resultset
            resultadoConsulta.last();
            //Obtiene el número de fila actual( que aquí es la última)
            int filas=resultadoConsulta.getRow();
            //Dimensiona el arreglo datosDevueltos con los enteros obtenidos
            datosDevueltos=new String[filas][columnas];
            //Ubica el cursor antes del a primera fila
            resultadoConsulta.beforeFirst();
            for(int i=0;i<filas;i++){
                //Va a la siguiente fila
                resultadoConsulta.next();
                for(int j=0;j<columnas;j++){
                    //Obtiene el valor de cada una de las columnas en la fila actual
                    datosDevueltos[i][j]=resultadoConsulta.getString(j+1);
                }
            }
        }
        catch (Exception e){
        }
    }
    return datosDevueltos;
}

public String[] getNombresColumnas(){
    if(error==null){
        try{
            //Devuelve el número de columnas
            int columnas=metaDatos.getColumnCount();
            nombresColumnas=new String[columnas];
            for(int i=0;i<columnas;i++){
                //Obtiene el nombre de cada una de las columna
                nombresColumnas[i]=metaDatos.getColumnLabel(i+1);
            }
        }catch(SQLException ex){
        }
    }
}

```

```

        return nombresColumnas;
    }
    public String getMensajeError(){
        return error;
    }
}

```

El constructor de la clase `ConsultaSQL` recibe como parámetro un objeto de tipo `Connection` y otro de `String`; éste contendrá una sentencia `SELECT`; `Statement` es una interfaz que puede ejecutar sentencias en lenguaje SQL a través de una conexión; su método `executeQuery()` recibe la cadena `SELECT` y devuelve un objeto de tipo `ResultSet`, que contiene una imagen de los datos devueltos por el servidor MySQL en formato de tabla, es decir, filas y columnas; además los `ResultSet` tienen un apuntador o cursor que sirve para recorrer los registros de la tabla; el recorrido se hace con métodos como `next()`. No obstante, no toda la información necesaria está en los datos que ya se obtuvieron (por ejemplo, falta el nombre de las columnas); para obtener dicha información se usan los objetos del tipo `ResultSetMetaData`.

En la clase que acabamos de escribir, hay dos métodos: `getDatosDevueltos()` y `getNombresColumnas()` que darán la información solicitada con la consulta en dos arreglos, uno contiene la tabla y el otro la lista de los nombres de los atributos de dicha tabla. Con ellos se podrán mostrar los datos.

Aquí debemos hablar de los arreglos en Java. Para obtener una colección de datos hay variantes en la declaración y el manejo respecto de las variables individuales. Para empezar, se usan paréntesis cuadrados para su declaración:

```
String [] nombresColumnas
```

Si se requiere un arreglo bidimensional se indica desde la creación:

```
String[][] datosDevueltos
```

Una vez que el arreglo ha sido declarado, se debe dimensionar:

```
datosDevueltos= new String[5][4];
```

Hecho lo cual, se debe inicializar cada uno de los elementos agregando el índice correspondiente en los corchetes:

```
datosDevueltos[0][0]="Hola"
```

Como en todos los lenguajes de programación, los arreglos se leen o escriben casi siempre usando el bucle *for* como lo hicimos en el código anterior.

## Autenticación

Como ya vimos, para que el servidor de MySQL permita el acceso se necesitan cuatro datos: Nombre del servidor, nombre del usuario, contraseña y nombre de la base de datos; mismos que conviene pedir desde un cuadro de diálogo que se presente antes de que la ventana principal aparezca. Para esto crearemos dos clases: un panel para los componentes y un cuadro de diálogo para mostrarlos.

### Panel autenticador

Esta clase la haremos con objetos del tipo [JDialog](#), [JPanel](#), [GridLayout](#), [JPasswordField](#), [JLabel](#), [JTextField](#) y [JButton](#). Para eso, la clase será un panel que contendrá todos los componentes, y al que le asignaremos la manera como deseamos que queden acomodados: en forma de rejilla (para eso sirve *GridLayout*), y al que agregamos 2 botones y cuatro campos de texto, uno de los cuales es de tipo *password*:

PanelAutenticador.java

```
package visorconsultas.vista;  
import javax.swing.JPanel;
```

```

import javax.swing.JLabel;
import javax.swing.JTextField;
import javax.swing.JPasswordField;
import javax.swing.JButton;
import java.awt.GridLayout;

//Obsérvese que esta clase es un panel
public class PanelAutenticador extends JPanel {
    //Crea los componentes necesarios:
    //3 cuadros de texto
    public JTextField servidor, usuario, base;
    //Un cuadro de texto para contraseñas
    public JPasswordField password;
    //Dos botones
    public JButton aceptar, cancelar;
    public PanelAutenticador() {
        iniciaComponentes();
        agregaComponentes();
    }
    private void iniciaComponentes(){
        servidor=new JTextField();
        usuario=new JTextField();
        password=new JPasswordField();
        base=new JTextField();

        aceptar=new JButton("Aceptar");
        aceptar.setMnemonic('a');//Subraya la A para activar el atajo de tecla <ALT-a>
        cancelar=new JButton("Cancelar");
        cancelar.setMnemonic('c');//Subraya la C para activar el atajo de tecla <ALT-c>
    }
    private void agregaComponentes(){
        /*Esta línea convierte al panel en una rejilla de cinco filas y dos columnas,
        *que acomoda los componentes, conforme son agregados, de izquierda a
        *derecha y de arriba abajo
        */
        setLayout(new GridLayout(5,2));//La manera de acomodar los componentes es una rejilla de 5x2
        //Agrega los componentes al panel según la rejilla de izquierda a derecha y de arriba a abajo
        add(new JLabel("Servidor",JLabel.RIGHT));
        add(servidor);

```

```

        add(new JLabel("Usuario",JLabel.RIGHT));
        add(usuario);
        add(new JLabel("Contraseña",JLabel.RIGHT));
        add(password);
        add(new JLabel("Base de datos",JLabel.RIGHT));
        add(base);
        add(aceptar);
        add(cancelar);
    }
}

```

## Diálogo autenticador

Después creamos un cuadro de diálogo que lo único que hace es contener y mostrar al panel anterior:

```

package visorconsultas.vista;
import javax.swing.JDialog;
public class DialogoAutenticador extends JDialog { //Obsérvese que hereda de JDialog
    public PanelAutenticador panel;
    /*Este programa sólo sirve para
    *mostrar el panel autenticador
    *en un cuadro de diálogo*/
    public DialogoAutenticador() {
        panel=new PanelAutenticador();//Una instancia de nuestro panel autenticador
        add(panel);//agrega el panel autenticador
        setSize(300,150); //Dimensiona el diálogo
    }
}

```

## Vista de resultados

Para mostrar los resultados, creamos cuatro clases: un área de texto para que el usuario escriba las consultas, una tabla para mostrar los resultados, un panel de botones para la ejecución de las acciones y una ventana para mostrar a todos los anteriores.

## Área de texto para las consultas

Como los objetos de tipo [\*JTextArea\*](#) son multilínea, puede suceder que el contenido exceda la zona visible; por eso se deben poner dentro de un objeto de la clase [\*JScrollPane\*](#), que es un panel que pone a la disposición de los componentes con partes ocultas las barras de desplazamiento horizontal y vertical.

AreaConsulta.java

```
package visorconsultas.vista;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
public class AreaConsulta extends JScrollPane{//Esta clase es un panel desplazable
    public JTextArea texto;
    public AreaConsulta() {
        texto=new JTextArea(4,30);
        texto.setLineWrap(true);//Hace que las líneas corten en el límite del área
        texto.setWrapStyleWord(true);//Hace que corten sólo en palabras completas
        setViewportView(texto);//Dentro de las barras se verá le área de texto
    }
}
```

## Tabla para mostrar los resultados

Las [\*JTable\*](#) también deben estar dentro de un [\*JScrollPane\*](#) y su función principal es mostrar los datos guardados en objetos del tipo [\*DefaultTableModel\*](#); éstos almacenan los valores de las celdas y dimensionan la tabla. En general, conviene que las tablas estén separadas de su modelo, por eso haremos dos clases.

### Modelo

ModeloTabla.java

```
package visorconsultas.modelo;
import javax.swing.table.DefaultTableModel;
public class ModeloTabla extends DefaultTableModel { //Hereda de DefaultTableModel
    public ModeloTabla() {
```

```
        //Dimensiona la tabla para la presentación inicial
        setColumnCount(7);
        setRowCount(30);
    }
}
```

## Tabla

### TablaResultados.java

```
package visorconsultas.vista;
import javax.swing.JTable;
import javax.swing.JScrollPane;
import visorconsultas.modelo.ModeloTabla;
import javax.swing.table.TableModel;
public class TablaResultados extends JScrollPane {
    public ModeloTabla modelo;//Crea una instancia del modelo
    public JTable tabla;
    public TablaResultados() {
        modelo=new ModeloTabla();
        tabla=new JTable(modelo);//Se asigna el modelo a la tabla al momento de construirla
        //Las columnas se autoajustan
        tabla.setAutoResizeMode(JTable.AUTO_RESIZE_ALL_COLUMNS);
        setViewportView(tabla); //La tabla se verá dentro del panel de barras de desplazamiento
    }
}
```

## Panel de botones

Esta es una clase que se asemeja en su diseño a PanelAutenticación; es, sencillamente, un contenedor con dos [JButton](#). Una diferencia central es que [JPanel](#) tiene por omisión un manejador de diseño que organiza los componentes como un flujo que se reparte en el espacio disponible desde el centro de la parte superior, y que es de la clase [FlowLayout](#).

### PanelBotonesConsulta.java

```
package visorconsultas.vista;
```



```

import javax.swing.JPanel;
import javax.swing.JButton;
public class PanelBotonesConsulta extends JPanel{
    public JButton consultar, salir;
    public PanelBotonesConsulta() {
        iniciaComponentes();
        agregaComponentes();
    }
    private void iniciaComponentes(){
        consultar =new JButton("Consultar");
        consultar.setMnemonic('c');
        salir =new JButton("Salir");
        salir.setMnemonic('s');
    }
    private void agregaComponentes(){
        add(consultar);
        add(salir);
    }
}

```

## Ventana de consultas

La ventana para mostrar las consultas contiene al área de texto, a los botones y a la tabla que acabamos de hacer. Es pertinente decir aquí que los objetos del tipo *JFrame* usan el manejador de diseño *BorderLayout* que usa los bordes de los contenedores para su ordenamiento. Por eso, el método *add()* recibe como segundo parámetro cuál borde usarán los componentes para agruparse en la ventana: *North* para arriba, *East* para la derecha, etc. Obsérvese que uno de los beneficios de haber creado las demás clases heredadas de algún panel facilita su asimilación al contenedor principal.

Es responsabilidad de la ventana capturar sus eventos; así que implementa la interfaz *ActionListener*, sobrescribe el método *actionPerformed()* y agrega los *listener* a los componentes que reciben eventos del *Mouse*: los dos botones del panel autenticador, y los que controlan la consulta y la salida del programa.

El método *inicio()* hace visible a la ventana pero le sobrepone al diálogo autenticador, que es modal, es decir, que toma el control de la aplicación. Será la intervención del usuario la que determinará el flujo del sistema.

Por el último es de notar que las líneas que se refieren al *Controlador* están comentadas porque esta clase no existe todavía.

#### VentanaConsultas.java

```
package visorconsultas.vista;
import javax.swing.JFrame;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
//import visorconsultas.controlador.Controlador;
public class VentanaConsultas extends JFrame implements ActionListener {
    public DialogoAutenticador autenticador;
    public TablaResultados resultados;
    public AreaConsulta area;
    public PanelBotonesConsulta botones;
//    public Controlador controlDe;
    public VentanaConsultas() {
        iniciaComponentes();
        agregaComponentes();
        agregaListeners();
        inicio();
    }
    private void iniciaComponentes(){
        autenticador=new DialogoAutenticador();
        resultados=new TablaResultados();
        area=new AreaConsulta();
        botones=new PanelBotonesConsulta();
    }
    private void agregaComponentes(){
        add(resultados,"South");
        add(area,"West");
        add(botones,"East");
        pack();
        setLocation(100,100);
    }
}
```

```

    }
    private void agregaListeners(){
        this.autenticador.panel.aceptar.addActionListener(this);
        this.autenticador.panel.cancelar.addActionListener(this);
        botones.consultar.addActionListener(this);
        botones.salir.addActionListener(this);
    }
    public void inicio(){
//        controlDe=new Controlador();
        setVisible(true);
        autenticador.setLocationRelativeTo(this);
        autenticador.setModal(true);
        autenticador.setVisible(true);
    }
    public void actionPerformed(ActionEvent evt){
        //controlDe.acciones(this,evt);
    }
}

```

## Controlador

Hasta ahora tenemos los mecanismos de conexión a base de datos y la posibilidad de mostrarlos; toca a una clase controladora llevar el proceso de autenticación y de obtención de la información. Esto se realiza con el método *acciones()* que discrimina cada uno de los cuatro botones en la aplicación (aquéllos a los que le agregamos *listener* en la ventana y a los que nos referiremos con su nombre en cursiva). Los objetos *cancelar* y *salir*, cerrarán la aplicación. Por su parte, *aceptar* tomará los valores de los campos de texto de autenticación y los enviará como parámetro a un objeto de la clase *Conector*. Si la conexión se logra, el diálogo se cerrará y le entrega el control a la ventana principal, si no, muestra el error en un cuadro de diálogo. Por su parte, *consultar* manda el contenido del área de texto y la conexión a una instancia de

*ConsultaSQL* que muestra los datos en la tabla si los obtiene, si no, indica a su vez el error en otro cuadro de diálogo.

### Controlador.java

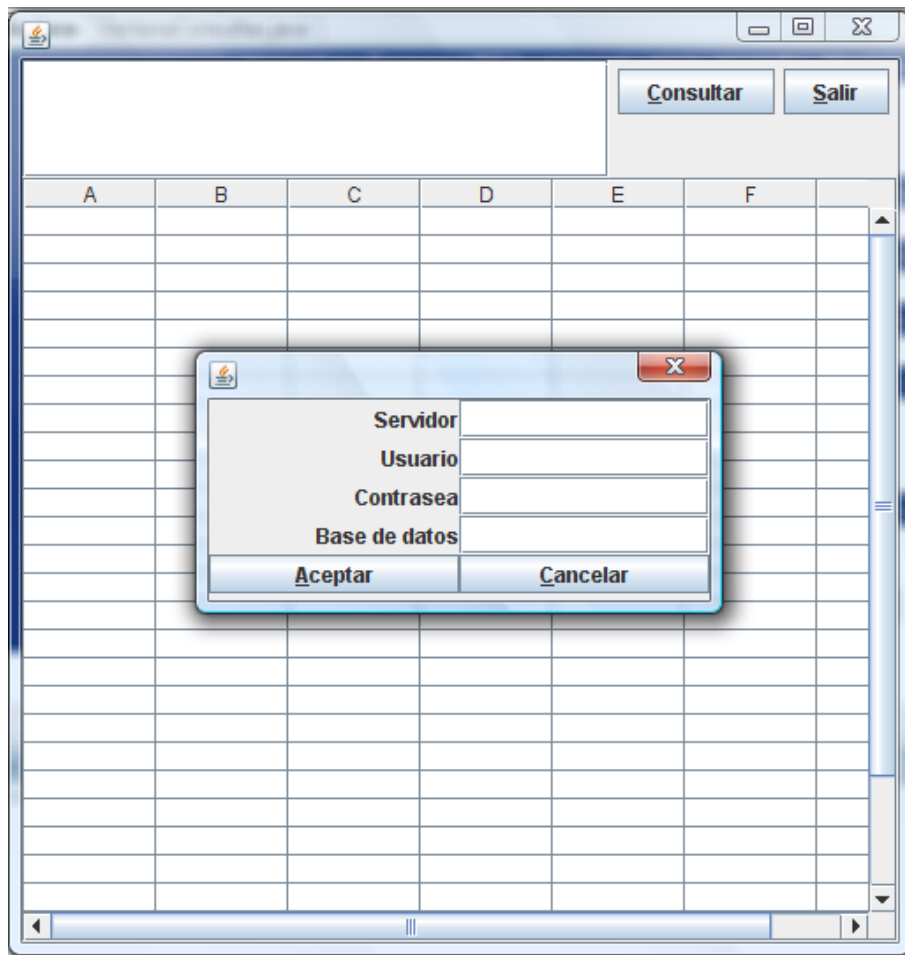
```
package visorconsultas.controlador;
import visorconsultas.vista.VentanaConsultas;
import java.awt.event.ActionEvent;
public class Controlador {
    Conector conMySQL;
    public Controlador() {
    }
    public void acciones(VentanaConsultas visor, ActionEvent evt){
        Object clicado=evt.getSource();
        if(clicado==visor.autenticador.panel.aceptar){
            String host=visor.autenticador.panel.servidor.getText();
            String usuario=visor.autenticador.panel.usuario.getText();
            String pw=new String(visor.autenticador.panel.password.getPassword());
            String base=visor.autenticador.panel.base.getText();
            conMySQL=new Conector(host,usuario,pw,base);
            if(conMySQL.getConexion()!=null)
                visor.autenticador.dispose();
            else
                muestraError("El error que manda MySQL es:\n"+conMySQL.getMensajeError());
        }
        if(clicado==visor.botones.consultar){
            ConsultaSQL consulta=
            new ConsultaSQL(conMySQL.getConexion(),visor.area.texto.getText());
            if(consulta.getMensajeError()==null)
                visor.resultados.modelo.setDataVector
                    (consulta.getDatosDevueltos(),consulta.getNombresColumnas());
            else
                muestraError("El error que manda MySQL es:\n"+consulta.getMensajeError());
        }
        if(clicado==visor.autenticador.panel.cancelar||clicado==visor.botones.salir)
            System.exit(0);
    }
    private void muestraError( String e){
```

```
        javax.swing.JOptionPane.showMessageDialog(null,e);
    }
    public static void main(String[] args){
        new visorconsultas.vista.VentanaConsultas();
    }
}
```

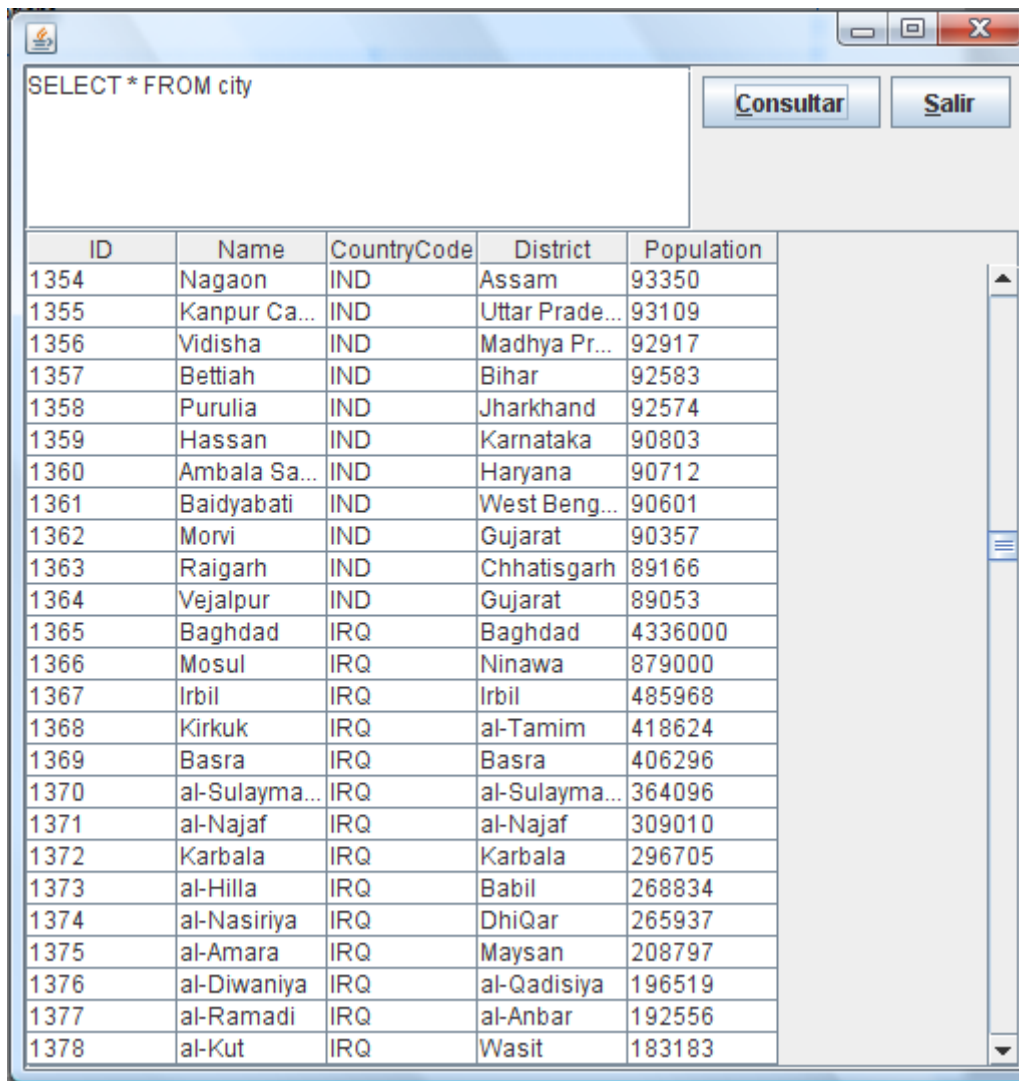
## Compilación y ejecución

Cada una de las clases que acabamos de crear debe ser compilada en el orden sucesivo en que se presenta. Al final, a *VentanaConsulta* deben quitársele las diagonales de comentarios de las líneas que invocan al *Controlador* y debe volverse a compilar; después de lo cual, ésta última clase se puede ejecutar, en tanto que es ella la que tiene el método *main()*.

La apariencia inicial es como sigue:



Y este es el resultado después de hacer una consulta:



ID	Name	CountryCode	District	Population
1354	Nagaon	IND	Assam	93350
1355	Kanpur Ca...	IND	Uttar Prade...	93109
1356	Vidisha	IND	Madhya Pr...	92917
1357	Bettiah	IND	Bihar	92583
1358	Purulia	IND	Jharkhand	92574
1359	Hassan	IND	Karnataka	90803
1360	Ambala Sa...	IND	Haryana	90712
1361	Baidyabati	IND	West Beng...	90601
1362	Morvi	IND	Gujarat	90357
1363	Raigarh	IND	Chhatisgarh	89166
1364	Vejalpur	IND	Gujarat	89053
1365	Baghdad	IRQ	Baghdad	4336000
1366	Mosul	IRQ	Ninawa	879000
1367	Irbil	IRQ	Irbil	485968
1368	Kirkuk	IRQ	al-Tamim	418624
1369	Basra	IRQ	Basra	406296
1370	al-Sulayma...	IRQ	al-Sulayma...	364096
1371	al-Najaf	IRQ	al-Najaf	309010
1372	Karbala	IRQ	Karbala	296705
1373	al-Hilla	IRQ	Babil	268834
1374	al-Nasiriya	IRQ	DhiQar	265937
1375	al-Amara	IRQ	Maysan	208797
1376	al-Diwaniya	IRQ	al-Qadisiya	196519
1377	al-Ramadi	IRQ	al-Anbar	192556
1378	al-Kut	IRQ	Wasit	183183

# Índice

Presentación.....	1
¿Qué y para qué es Java? .....	2
Objetivos y metodología del curso .....	2
Primera parte. Iniciación al lenguaje Java .....	4
Instalación.....	4
Definición breve de la Programación Orientada a Objetos (POO) .....	8
El programa Hola Mundo y los conceptos de abstracción y encapsulamiento .....	9
Elementos del lenguaje Java.....	11
Identificadores .....	12
Sentencias .....	12
Bloques de código .....	13
Comentarios.....	13
Expresiones.....	14
Operadores.....	14
Metacaracteres .....	14
Palabras reservadas.....	14
Sintaxis .....	16
Ejecución y flujo de datos .....	22
Ejecución .....	22
Flujo de datos .....	25
Tipos de datos.....	26
Generalidades .....	26



Los tipos de datos y los métodos .....	28
Caso de método que sí devuelve un tipo y no recibe parámetros.....	28
Caso de método que no devuelve un tipo y sí recibe parámetros.....	29
Caso de método que sí devuelve un tipo y sí recibe parámetros .....	30
Estructuras de control .....	31
Introducción.....	31
El bloque <i>if...else</i> .....	34
<i>if.. else</i> anidado.....	35
Operadores lógicos .....	36
AND .....	37
OR.....	38
La estructura de control <i>while</i> .....	39
La estructura de control <i>try...catch</i> .....	40
El bloque <i>for</i> .....	42
El bloque <i>switch</i> .....	43
Segunda parte. Aplicación de ejemplo .....	46
Consideraciones preliminares.....	46
Herencia y polimorfismo .....	46
Captura de eventos.....	50
Aplicación de ejemplo .....	53
Abstracción.....	53
Conexión.....	53
Consultas SQL.....	56
Autenticación.....	59
Panel autenticador.....	59

Diálogo autenticador.....	61
Vista de resultados.....	61
Área de texto para las consultas .....	62
Tabla para mostrar los resultados .....	62
Panel de botones .....	63
Ventana de consultas .....	64
Controlador.....	66
Compilación y ejecución .....	68
Y este es el resultado después de hacer una consulta:.....	70
Índice .....	71