Universidad Nacional del Litoral
Facultad de Ingeniería y Ciencias Hídricas
Departamento de Informática

Tecnicatura en Informática Aplicada al Diseño Multimedia y Sitios Web

Elementos de Programación

Unidad 6: Introducción a la Programación Orientada a Objetos

Docente: Ing. Risso, Oscar Luis
Año: 2024

Índice

Unidad 6: Introducción a la Programación Orientada a Objetos

En este Tema

En esta unidad nos introduciremos en el modelo de la Programación Orientada a Objetos. Para ello aprenderemos a definir nuestras propias clases e instanciar nuestros objetos para resolver problemas e implementar su solución en aplicaciones Java.

Todo en Java forma parte de una clase, es una clase o describe como funciona una clase. El conocimiento de de las clases es fundamental para comprender el funcionamiento de un programa en Java. El lenguaje Java proporciona una serie de paquetes de clases que incluyen ventanas, utilidades, un sistema de entrada / salida, herramientas y comunicaciones. En esta unidad temática nos proponemos aprender a crear nuestros propios paquetes de clases y luego utilizarlos instanciando objetos en programas sencillos. Para ello debemos aprender a aplicar el modelo de orientación a objetos a problemas reales. Es decir, antes de crear clases (tipos de objetos) deberemos aprender a descubrir cuáles serían las clases que se necesitan para resolver un determinado problema y qué características deben tener estas clases. Una vez que sepamos encontrar las clases que modelan un problema dado, comenzaremos a estudiar cómo se implementan estas clases en Java.

Todas las acciones de los programas Java se colocan dentro de un bloque de una clase. Un Objeto es una instancia de una clase. Todos los métodos (funciones) se definen dentro del bloque de una clase, Java no soporta funciones o variables globales fuera de la clase, si bien podemos considerar las variables declaradas al principio de la clase como globales a esa clase, así pues el esqueleto de cualquier aplicación Java se basa en la definición de una clase.

Objetivos

Al finalizar este tema Ud. debe lograr:

- Conocer los fundamentos de la Programación Orientada a Objetos (POO).
- Conocer y utilizar la metodología de la programación orientada a objetos para hallar soluciones computacionales a problemas sencillos.
- Domine y utilice la sintaxis de Java para implementar programas Orientados a Objetos

1. Diseño de Clases

1.1. Clases

Una clase es un tipo genérico que representa a un conjunto de objetos de similares características. Luego de haber definido una clase, es posible obtener tantos objetos (instancias) de dicha clase como sea necesario.

1.2. Abstracciones

Cuando se piensa en una clase hay que recordar que se trata de una abstracción de un objeto de la vida real y no del objeto en sí mismo. Cuando pensamos en la "clase vaca" no es necesario que pensemos en ninguna de las vacas reales que hemos visto alguna vez. Pensamos en la clase vaca como una abstracción que nos permite hablar sobre vacas y no necesariamente especificar una vaca en particular. Cuando hacemos referencia a esta abstracción, todos estamos de acuerdo en que la vaca tiene 4 patas, tiene un hocico, tiene colores de pelo, camina, da leche, muge, come, duerme, etc.

Generalizando podemos observar que nos relacionamos con la realidad mediante abstracciones. Si cuando contamos una historia y pronunciamos la palabra "mesa", cada uno puede imaginarse una mesa muy diferente pero en general no necesitaremos dar demasiados detalles de ninguna mesa concreta para poder seguir

contando la historia. En este caso, esa mesa de la que hablamos representa una abstracción que nos permite comunicarnos.

Estas ideas acerca de las abstracciones son aprovechadas por el modelo de orientación a objetos en programación para simplificar el análisis de los problemas. Esta metodología aprovecha la forma natural que tenemos para interactuar con la realidad. Declaramos una clase en base al problema a resolver y por tanto destacamos sólo la información y las operaciones de interés para arribar a la solución del problema.

Importante

Una abstracción es un ejercicio intelectual que nos permite representar un objeto, destacando los aspectos importantes del mismo, para estudiar un caso o analizar una situación en la cual el objeto forma parte.

1.3. Cómo identificar una Clase?

Al usar el diseño orientado a objetos para resolver problemas sencillos, debemos como primer tarea plantear clases de objetos que conformarán nuestro modelo de solución. Para ello proponemos las etapas siguientes:

Defina claramente el problema. Entienda el enunciado y re-escríbalo claramente. Efectúe un análisis de la información disponible (datos) y de los requerimientos (resultados). Investigue y plantee las relaciones entre datos y resultados sumergiéndose en el dominio del problema. Trate de identificar clases en base a lo siguiente:

Busque en el enunciado entidades o sustantivos y forme una lista preliminar.

Descarte las entidades obviamente inservibles, aquellas que se incluyen claramente en otras más generales, las redundantes y aquellas para las cuales es casi imposible proponer funcionalidades. No modele al usuario.

Seleccione aquellas entidades que incluyen claramente la información disponible y los resultados esperados. Haga un refinamiento final y observe si las clases propuestas resumen una abstracción del dominio del problema a resolver.

En este proceso de diseño juegan un rol fundamental el sentido común y la experiencia del desarrollador.

1.4. Asignando Responsabilidades

Una vez seleccionadas las posibles clases de nuestro modelo debemos indicar las responsabilidades para cada una de ellas: es decir qué debe conocer (datos o atributos) y qué debe saber hacer (funcionalidades).

Para encontrar los atributos identifique los datos y resultados relacionados con la clase propuesta. En el caso de las funcionalidades podemos comenzar por:

- Inicializar los datos.
- Algoritmos de cálculo para determinar los resultados.
- Devolver o informar estos resultados.
- Ajuste el conjunto de atributos y funcionalidades para evitar redundancia de información.

1.5. Encapsulamiento

Cuando declaramos una clase cuyos atributos no pueden ser accedidos desde el exterior decimos que la clase encapsula estos atributos. El concepto de encapsulamiento es fundamental en programación orientada a objetos y tiene por finalidad facilitar la reusabilidad y depuración del código.

Cuando utilizamos una clase, sabemos que esta tiene determinadas responsabilidades y nos interesa que las cumpla sin importar cómo lo hace. Para esto se pueden declarar partes privadas que no pueden ser accedidas excepto por métodos propias de la clase. En Java el encapsulamiento es opcional. Se debe plantear explícitamente cual será la parte privada que se desea proteger de una clase. Sin embargo, a pesar de ser

opcional en Java, la encapsulación es una buena práctica para el diseño e implementación de programas que utilizan el modelo de objetos.

2. Implementación en Java

Veamos la sintaxis JAVA para definir clases, atributos y métodos. Luego aplicaremos los conceptos teóricos anteriores para Diseñar e implementar nuestras Clases.

2.1. La cláusula pakage

Un package es una agrupación de clases afines. Equivale al concepto de librería existente en otros lenguajes o sistemas. Una clase puede definirse como perteneciente a un package y puede usar otras clases definidas en ese o en otros packages.

Los packages delimitan el espacio de nombres (space name). El nombre de una clase debe ser único dentro del package donde se define. Dos clases con el mismo nombre en dos packages distintos pueden coexistir e incluso pueden ser usadas en el mismo programa.

Una clase se declara perteneciente a un package con la cláusula package, cuya sintaxis es:

```
1 package nombre_package;
```

La cláusula package debe ser la primera sentencia del archivo fuente. Cualquier clase declarada en ese archivo pertenece al package indicado.

Por ejemplo, un archivo que contenga las sentencias:

```
1 package miPackage;
2  . . .
3  class miClase {
4  . . .
```

Declara que la clase miClase pertenece al package miPackage.

La claúsula package es opcional. Si no se utiliza, las clases declaradas en el archivo fuente no pertenecen a ningún package concreto, sino que pertenecen a un package por defecto sin nombre.

La agrupación de clases en packages es conveniente desde el punto de vista organizativo, para mantener bajo una ubicación común clases relacionadas que cooperan desde algún punto de vista.

2.2. Cláusula import

Cuando se referencia cualquier clase dentro de otra se asume, si no se indica otra cosa, que ésta otra está declarada en el mismo package. Por ejemplo:

```
1 package Geometría;
2  . . .
3  class Circulo {
4  Punto centro;
5  . . .
6  }
```

En esta declaración definimos la clase Circulo perteneciente al package Geometría. Esta clase usa la clase Punto. El compilador y la JVM asumen que Punto pertenece también al package Geometría, y tal como está hecha la definición, para que la clase Punto sea accesible (conocida) por el compilador, es necesario que esté definida en el mismo package.

Si esto no es así, es necesario hacer accesible el espacio de nombres donde está definida la clase Punto a nuestra nueva clase. Esto se hace con la cláusula import. Supongamos que la clase Punto estuviera definida de esta forma:

```
package GeometriaBase;
class Punto {
int x , y;
}
```

Entonces, para usar la clase Punto en nuestra clase Círculo deberíamos poner:

```
package GeometriaAmpliada;
import GeometriaBase.*;
class Circulo {
Punto centro;
. . .
}
```

Con la cláusula import GeometriaBase.*; se hacen accesibles todos los nombres (todas las clases) declaradas en el package GeometriaBase. Si sólo se quisiera tener accesible la clase Punto se podría declarar: import GeometriaBase.Punto;

También es posible hacer accesibles los nombres de un package sin usar la cláusula import calificando completamente los nombres de aquellas clases pertenecientes a otros packages. Por ejemplo:

```
package GeometriaAmpliada;
 class Circulo {
 GeometriaBase.Punto centro;
 . . .
 }
```

Sin embargo si no se usa import es necesario especificar el nombre del package cada vez que se usa el nombre Punto.

La claúsula import simplemente indica al compilador donde debe buscar clases adicionales, cuando no pueda encontrarlas en el package actual y delimita los espacios de nombres y modificadores de acceso. Sin embargo, no tiene la implicación de 'importar' o copiar código fuente u objeto alguno. En una clase puede haber tantas sentencias import como sean necesarias. Las cláusulas import se colocan después de la cláusula package (si es que existe) y antes de las definiciones de las clases.

2.3. Clases

La definición de la clase consta de dos partes, la declaración y el cuerpo, Sintaxis:

DeclaracionDeClase{
CuerpoDeClase
}
	class nombreClase{
	cuerpo de la clase
	}

La declaración de Clase indica al compilador el nombre de la clase, la clase de la que deriva, su superclase, los privilegios de acceso a la clase, pública, abstracta o final y si la clase implementa o no, una o varias interfaces. El nombre de la clase debe ser un identificador válido, por convención el identificador de la clase en Java empieza con una letra mayúscula.

En Java, cada clase deriva directa o indirectamente de la superclase Object. La clase padre inmediatamente superior a la que se está declarando se conoce como *superclase (superclass)*, si no se especifica de quien deriva una clase se sobreentiende que deriva de la clase *Object*, (definida en el paquete Java.lang). En la declaración de clase se utiliza la palabra *extends* para especificar la superclase de la cual deriva. Sintaxis:

```
Class UnaClase extends SuperClase {
```

```
  //Cuerpo de la clase.
}
```

Cuando una clase (hija) deriva de otra (padre) hereda sus métodos y atributos, estudiaremos con detalle la *herencia* en la próxima unidad.

2.4. Tipos de Clases

Hasta aquí, en los ejemplos solo hemos utilizado la palabra *public* para calificar el nombre de las clases que se han visto, pero existen cuatro tipos de clases en total:

- **public**: En toda clase public, lo declarado en ella será accesible desde otras clases, ya sea por herencia, desde clases declaradas fuera del paquete que contiene a esas clases públicas. Para poder acceder desde otros paquetes, primero tienen que ser importadas. Sintaxis:

```
Public class UnaClase extends SuperClase Implements unaInterfaz, otraIntefaz {
  //cuerpo de la clase
}
```

  Aquí la palabra reservada public se utiliza en un contexto diferente que cuando la empleamos dentro de la clase junto con private y protected.

- **abstract**: Una clase abstract tiene al menos un método abstracto. Una clase abstracta no se instancia. Sino que se utiliza como clase base en la herencia (se tratará en la próxima unidad).

- **final**: Una clase final se declara como clase que finaliza una cadena de herencias. Es lo opuesto a una clase abstract y nadie puede heredar de una clase final.

- **synchronizable**: Este modificador especifica que todos los métodos definidos en la clase son sincronizados, es decir, que no se puede acceder a ellos al mismo tiempo desde distintas tareas, el sistema se encarga de colocar los flags necesarios para evitarlo. Este mecanismo hace que desde distintas tareas se puedan modificar variables sin que se sobrescriban. Si no se utilizan modificadores, Java asumirá que la case es:

  - No final.
  - No abstracta.
  - Subclase de la clase object.
  - No implementa interfaz.

2.5. Métodos y Atributos. Variables miembro

Una clase en Java puede contener atributos (variables miembro) y métodos, los atributos pueden ser de tipos primitivos o clases y los métodos son funciones.

Ejemplo:

```
public class UnaClase{
  int a; //atributo
  int resultado; //atributo

  public UnaClase{ //Metodo constructor
    a = 5;
  }

  public void suma(int p){ //Método suma
```

```
10      int aux;
11      aux = a + p;
12      resultado = aux;
13    }
14  }
```

La clase “UnaClase” contiene 2 atributos de tipo entero: “a” y “resultado” y 2 métodos: UnaClase, con el mismo nombre que la clase, es el constructor, este método se ejecutara automáticamente cuando se instancia la clase. Y suma() que guarda los cálculos en la variable miembro resultado.- aux es una variable local o sea no es variable miembro de la clase.

La Sintaxis completa de la declaración de una variable miembro es: (los ítems opcionales están entre corchetes [])

```
[especificador de acceso] [static] [final] [transient] [volatile] tipo
    nombreVariable [= valor inicial];
```

2.6. Ámbito de una variable

Los bloques de sentencias compuestas en Java se delimitan por llaves { }. Las variables en Java solo son válidas desde el punto donde están declaradas hasta el final de la sentencia compuesta que las engloba. Se pueden anidar estas sentencias compuestas y cada una puede contener su propio conjunto de declaraciones de variables locales, pero no se puede declarar una variable con el mismo nombre que una de ámbito exterior.

2.6.1. Variables de instancia

La declaración de un dato miembro de una clase sin utilizar *static*, crea una variable de instancia. Cada vez que se crea una instancia de la clase se reserva memoria diferente (se crea una copia) para todas los atributos o variables de instancia de la clase con cada objeto instanciado.-

Java accede a las variables de instancia asociadas a un objeto utilizando el nombre del objeto, el operador punto (.) y el nombre de la variable. Sintaxis:

```
unObjeto.unaVariable
```

2.6.2. Variables estáticas

La declaración de un dato miembro de una clase usando static, crea una variable de clase o variable estática de una clase. Para todos los objetos instanciados de esa clase se asigna el mismo lugar de memoria para las variables estáticas, es decir para todos los objetos creados de esa clase existe una sola copia de la variable de clase, que comparten. Sintaxis:

```
class Tarjeta{
  static double disponible;
  String propietario;
  ...
}
```

En este ejemplo puedo crear varias instancias de la clase tarjeta, por ejemplo una para el titular, otra para la esposa, otra para el hijo, pero todas compartirán el atributo disponible, de tal forma que cuando llegue a 0 ese será el valor de “disponible” en todas las instancias.-

No es necesario instanciar ningún objeto de la clase para acceder a la variable de clase. (Ejemplo 6.02)

2.6.3. Constantes

En el lenguaje Java se utiliza la palabra clave final para indicar que una variable de comportarse como una constante. Esto quiere decir que no admitirá modificación después de ser declarada e inicializada. Por convención los identificadores se escriben con letras mayúsculas.

```
CaracterPublico/Privado static final TipoDeLaConstante = valorDeLaConstante;
```

Ejemplo:

```
class Calculo{
  final Double PI = 3.14159265358979323846;
  ...
}
```

Se usa la palabra clave final con una variable o una clase, se pueden crear constantes de clase, haciendo un uso altamente eficiente de memoria porque no se necesitan múltiples copias en memoria de las constantes.
La palabra final se puede aplicar a un método significando así que este método no se puede sobrescribir.

2.7. Métodos

Los métodos son funciones que pueden ser llamados dentro de la propia clase o por otras clases. La implementación de un método consta de dos partes: declaración y cuerpo. Sintaxis minima:

```
tipoRetorno nombreMetodo([ lista de argumentos ]){
  cuerpo del método;
  ...
}
```

Los métodos pueden tener numerosos atributos a la hora de declararlos, incluyendo el control de acceso, si es estático o no, etc.
La lista de argumentos o parámetros es opcional, puede limitarse a paréntesis ( ). El uso de parámetros se utiliza para pasar información al interior del cuerpo del método. Sintaxis completa, los ítems opcionales están entre corchetes []

```
[especificador de acceso] [ static ] [ abstract ] [ final ] [ native ] [
    synchronized] tipoRetorno nombreMetodo ([lista de argumentos ]) [ throws
    listaExcepciones]
```

- [$especificadordeacceso$] determina si otros métodos pueden acceder a este y como pueden hacerlo (public, private, protected)
- [*static*] determina que este método y que puede ser accedido por otras clases sin necesidad de crear una instancia de esta clase.
- [*abstract*] indica que el método no está definido en la clase, sino que se redefinirá en una subclase.
- [$final$] evita que este método sea sobrescrito.
- [*native*] son métodos escritos en otro lenguaje, Java soporta C y C++.
- [*synchronized*] se utiliza en el soporte multitarea.
- [$throwslistaExcepciones$] excepciones que puede generar y manipular el método.

2.7.1. Argumentos variables

Java admite un número variable de argumentos en su invocación, el método no necesita conocer la longitud de la lista de argumentos.

2.7.2. Valor de retorno de un método

En Java es fundamental que a la hora de la declaración se indique el tipo de dato que ha de devolver el método. Si no devuelve ningún valor se definirá de tipo void. Todos los tipos primitivos se devuelven por valor y todos los objetos se devuelven por referencia. Por valor el método trabaja con una copia en una variable auxiliar del tipo indicado, en cambio por referencia trabaja con la dirección de memoria dinámica donde se encuentra almacenado el objeto.

Para devolver un valor se utiliza la palabra clave return, esta va seguida de una expresión que será evaluada por el compilador para saber el valor de retorno.

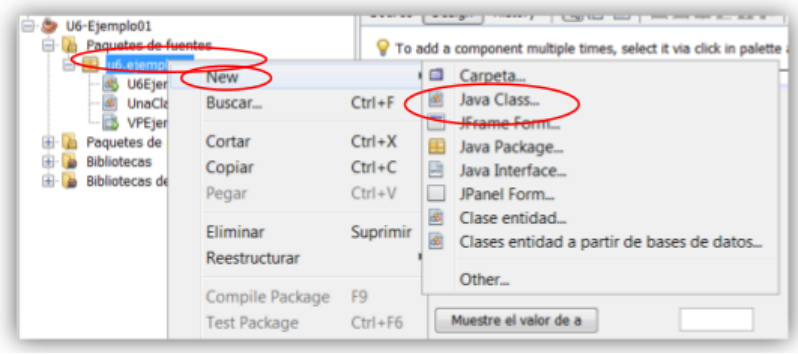
Nota

Veremos una serie de ejemplos en JAVA, sin el diseño de clases.

Ejemplo 6.01

Crea una clase con un atributo privado y le asigna un valor.

1. Creamos un Proyecto con el nombre de U6-Ejemplo01 y un JForm con el nombre de VPEjemp01.
2. Recuerda de instanciar en el main() de U6-Ejemplo01 un objeto de tipo VPEjemp01 y colocar su método .setVisible(), en true (verdadero).-
3. Luego crea dentro de: Paquete de Fuentes → u6.ejemplo01 → New → Java Class y le colocamos el nombre UnaClase

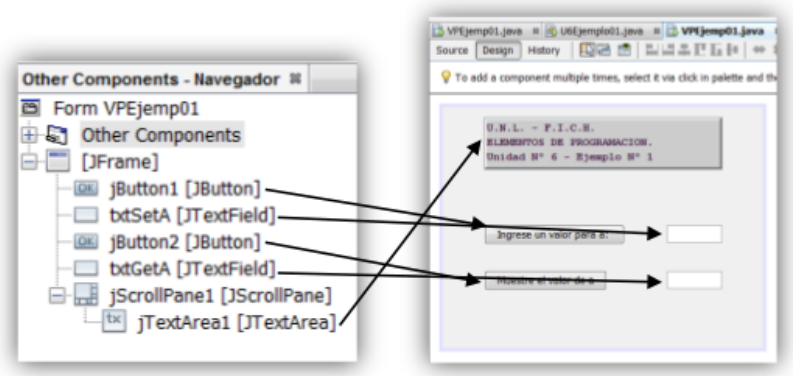


4. Abrimos la ventana de código y completamos la clase:

```
1  public class UnaClase {
2   private int a; //Atributo privado de la clase
3   //Método de acceso público, al cual le paso el parámetro x que asigno a la
        variable miembro "a"
4   public void set_a(int x){
5     a = x;
6   }
7   // este método es publico y retorna el valor de a que es un entero.-
8   public int get_a(){
```

```
    return a;
  }
}
```

5. En el formulario:



6. En el código del formulario:

```
public class VPEjemp01 extends javax.swing.JFrame {
 /**
 * Creates new form VPEjemp01
 */
 //*************************************************************************
 //1- Declaracion de componentes globales para el formulario
 // Instancio el Objeto A de tipo UnaClase
 UnaClase A;
```

7. Inicializo componentes globales para el formulario

```
  public VPEjemp01() {
    initComponents();
    //*************************************************************************

    //2-Intancio mis componentes globales
    // Instancio el Objeto A de tipo UnaClase
    A= new UnaClase();
  }
```

8. jButton1ActionPerformed()

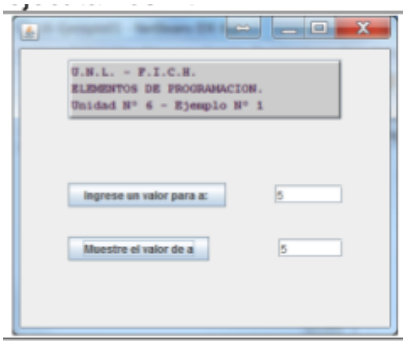
```
  private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    //Llamo al método publico set_a y le paso el número entero 10
    A.set_a(Integer.parseInt(txtSetA.getText()));
  }
```

9. jButton2ActionPerformed()

```
  private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
```

```
    // TODO add your handling code here:
    //Llamo al método publico get_a y devuelve su valor entero
    txtGetA.setText(Integer.toString(A.get_a()));
  }
```

10. Compilamos F11 y ejecutamos F6:



2.7.3. Métodos de Instancia

Cuando en java no se incluye la palabra static cada instancia de la clase, cada objeto genera un método de instancia en memoria, aunque no se creen múltiples copias del método en memoria, el resultado es como si fuera así.

2.7.4. Métodos estáticos

Cuando una función es incluida en una definición de clase Java y se utiliza la palabra static se obtiene un método estático o método de clase.

Estos métodos pueden ser invocados sin necesidad de instanciar un objeto. En java se puede invocar un método de clase utilizando el nombre de la clase un punto y el nombre del método: Sintaxis

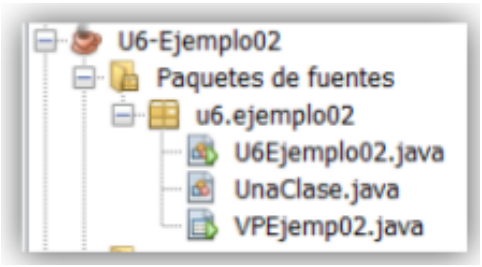
```
UnaClase.unMetodoDeClase();
```

Los métodos estáticos solo pueden manejar variables estáticas, todas las clases que se derivan de una declarada estática, comparten la misma zona de memoria. Los métodos estáticos se utilizan solo para acceder a variables estáticas.-

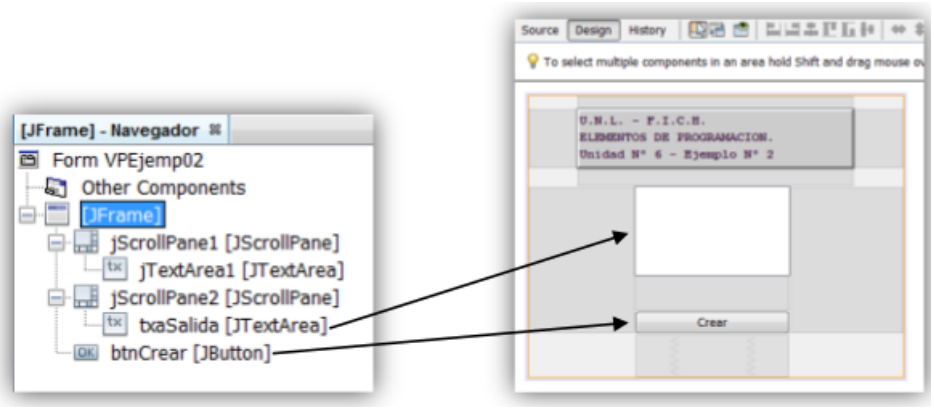
Ejemplo 6.02

Crea una clase con atributo y método estático

1. Repetimos los pasos 1, 2 y 3 del Ejemplo 6.02. con el nombre de U6-Ejemplo02, un JForm con el nombre de VPEjemp02 y una Java Class llamada UnaClase.



A continuación editamos el JForm y agregamos los siguientes componentes:



2. Abrimos la ventana de código y completamos la clase:

```
public class UnaClase {
 private static int contador = 0; //variable privada y estática
 UnaClase(){ //costructor de la clase
   contador++; //Al crear una instancia el contador cuenta
 }
 public static int getContador(){// retorno el valor de contador que es static
   return contador; // También el método es estático
 }
}
```

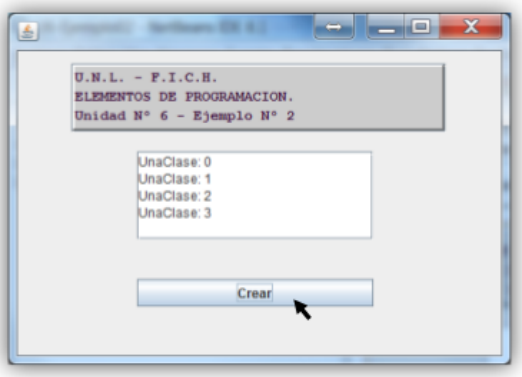
3. En el código del formulario:

```
public class VPEjemp02 extends javax.swing.JFrame {
 UnaClase A[];
 String salida;
 int c;

 public VPEjemp02() {
   initComponents();
   //2-Implemento instancia de UnaClase en el objeto A
   c = 0;//Contador para el indice del vector
   A= new UnaClase[100];//Creo arreglo de objetos vacio
   //Muestro la cantidad de Objetos
   salida= "UnaClase: "+Integer.toString(UnaClase.getContador())+"\n";
   txaSalida.setText(salida);
 }
```

4. btnCrearActionPerformed():

```
private void btnCrearActionPerformed(java.awt.event.ActionEvent evt) {
 // TODO add your handling code here:
 A[c]= new UnaClase();//Crea instancia de clase
 salida+= "UnaClase: "+Integer.toString(UnaClase.getContador())+"\n"; //
     Muestra miembro static
 c++;
 txaSalida.setText(salida);
 }
```

5. Compilamos F11 y ejecutamos F6:



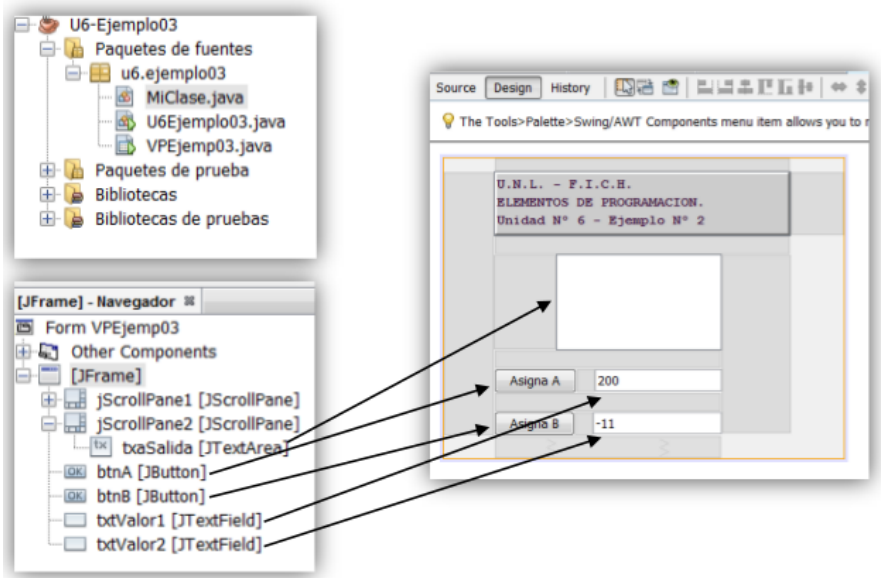
2.7.5. Paso de parámetros

En Java todos los parámetros deben estar declarados y definidos dentro de la clase, los argumentos son como variables locales declaradas en el cuerpo del método que están inicializadas con el valor que se le pasa en la invocación del método. Todos los argumentos de tipo primitivo en Java deben pasarse por valor, mientras que los objetos deben pasarse por referencia. Cuando se pasa un objeto por referencia lo que se pasa en la dirección de memoria en donde reside el objeto. Al modificar el parámetro pasado por referencia se modifica el valor en el bloque de llamada ya que ocupan la misma posición de memoria. Si se modifica una variable que haya sido pasada por valor no se modifica la variable original correspondiente en el bloque de donde fue llamado el método.-

Ejemplo 6.03

En el siguiente ejemplo mostramos el paso de parámetros de tipo primitivo, de tipo objeto, con pasaje por valor y referencia respectivamente.

1. Repetimos los pasos 1, 2 y 3 del Ejemplo 6.02. con el nombre de U6- Ejemplo03, un JForm con el nombre de VPEjemp03 y una Java Class llamada MiClase. A continuación editamos el JForm y agregamos los siguientes componentes



2. Abrimos la ventana de código de la clase y completamos

```
package u6.ejemplo03;
/**
*
* @author Prof. Gerardo Sas
*/
public class MiClase {
 private int valor = 0; //Atributo privado.
 private static int instancia = 0; //Atributo privado y estático.

 MiClase(){ //constructor sin parámetros inicializa la clase incrementando 1.
   instancia++;
 }

 void setValor(int x){
   valor = x; //Asigna el valor de x a valor
 }

 int getValor(){
   return valor; //Retorna valor
 }

 static int getInstancia(){
   return instancia; //Retorna instancia
 }

 //Metodo para comparar el atributo valor y ver si son iguales
 public boolean equals(MiClase o){ //el parámetro es un objeto de la misma
     clase
   return this.valor == o.valor;
 }

}
```

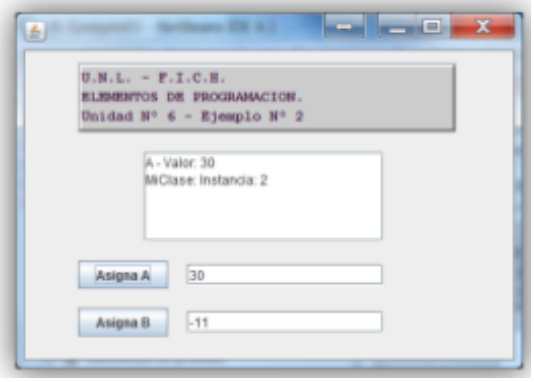
3. En el formulario codificamos:

```
package u6.ejemplo03;
/**
*
* @author Prof. Gerardo Sas
*/
public class VPEjemp03 extends javax.swing.JFrame {
 /**
 * Creates new form VPEjemp03
 */
 //1- Declaro variable de alcance general en el formulario
 MiClase A;
 MiClase B;

 public VPEjemp03() {
   initComponents();
   //2- Instancio las variables al inicializar formulario
   A= new MiClase();//Creo el objeto A
   B= new MiClase();//Creo el objeto A
 }
```

4. En los botones codificamos:

```
private void btnAActionPerformed(java.awt.event.ActionEvent evt) {
  // TODO add your handling code here:
  A.setValor(Integer.parseInt(txtValor1.getText()));//pasaje por valor
  String salida= "A - Valor: "+ A.getValor()+"\n"+"MiClase: Instancia: "+
      MiClase.getInstancia();
  if(B.equals(A))
    salida += "\nValores Iguales";
  else
    salida += "\nValores Diferentes";
  txaSalida.setText(salida);
}

private void btnBActionPerformed(java.awt.event.ActionEvent evt) {
  // TODO add your handling code here:
  B.setValor(Integer.parseInt(txtValor2.getText()));//pasaje por valor
  String salida= "B - Valor: "+ B.getValor()+"\n"+"MiClase: Instancia: "+
      MiClase.getInstancia();
  if(B.equals(A))
    salida += "\nValores Iguales";
  else
    salida += "\nValores Diferentes";
  txaSalida.setText(salida);
}
```

5. Compilamos F11 y ejecutamos F6:



2.7.6. Constructor y Sobrecarga

Ya hemos estado usando este método que tiene el mismo nombre que la clase a la que pertenece, no tiene tipo específico de retorno, ni siquiera void y dijimos que se ejecuta cuando se instancia un objeto de esa clase. Java soporta sobrecarga de métodos, es decir que dos o mas métodos pueden tener el mismo nombre, pero distinta lista de argumentos en su invocación. Cuando se invoque al método, el compilador, en base a la cantidad y tipo de argumentos, decidirá que versión va a utilizar. Su función primordial es inicializar el nuevo objeto que se instancia de la clase.

Ejemplo 6.04

Modificamos el ejemplo 6.03 sobrecargando el constructor con uno que tenga un parámetro para inicializar el atributo valor al instanciar el objeto B.

```
public class MiClase{
 // Atributos privados
 private int valor = 0;
 private static int instancia=0;

 // Constructor sin parámetros inicializa la clase incrementando 1
 MiClase(){ //Constructor sin parámetros
   instancia++;
 }

 // Constructor con parámetros inicializa la clase incrementando 1
 MiClase(int c){ //Constructor con parámetro (c)
   valor = c;
   instancia++;
 }

 void setValor(int x){
   valor = x;//Asigna el valor de x a valor
 }
 int getValor(){
   return valor;//Retorna valor
 }
 static int getInstancia(){
   return instancia;//Retorna instancia
 }
 //Metodo para comparar el atributo valor y ver si son iguales
 public boolean equals(MiClase o){//el parámetro es un objeto de la misma clase
   return (this.valor == o.valor)
 }
}
```

Cuando creo el objeto

```
...
 MiClase B;
...
 B = new MiClase(1000);//Instancio B, constructor con parámetro
```

Nota

Ahora vamos a combinar el diseño de clases con la sintaxis Java para resolver problemas, preste atención a los pasos para lograrlo

Ejemplo 6.05

Problema

Se necesita modelar una clase cilindro con el objetivo de obtener su volumen. Se conoce el radio y la altura del mismo. Se requiere una interfaz que permita ingresar al usuario los datos y obtener el volumen correspondiente.

Análisis

Usted habrá reconocido rápidamente los datos y resultado del problema: radio, altura y volumen. También en esta etapa encontramos las relaciones entre datos y resultados:

$Volumen = \pi * (Radio)^2 * Altura$

Hallar Clases y responsabilidades

La lista de sustantivos presentes en el enunciado es:

Cilindro, objetivo, volumen, radio, altura, cilindro, interfaz, usuario

Analicemos cuales de estos sustantivos pueden ser candidatos a constituir una clase para modelar este problema. Claramente el término "objetivo" forma parte de una expresión corriente y no merece la pena ser considerado como una posible clase. En general es útil también simplificar el número en los candidatos por lo que dejaremos simplemente "radio" y "altura".

Simplificando también las repeticiones podemos obtener una primer lista de candidatos como la siquiente:

Cilindro, volumen, radio, altura, interfaz, usuario

Debemos considerar que la interfaz, ni mucho menos el usuario, son objetos a modelar. El usuario nunca es modelado y las clases para la interfaz ya están modeladas en Java. Para crear las interfaces de usuario utilizamos las clases de la biblioteca de componentes Java (swing).

Ahora nos quedamos con:

Cilindro, volumen, radio, altura

La entidad que realmente hay que modelar es el cilindro. Las otras tres entidades (volumen, radio, altura) bien pueden ser atributos del cilindro y ser modelados simplemente como un número real.

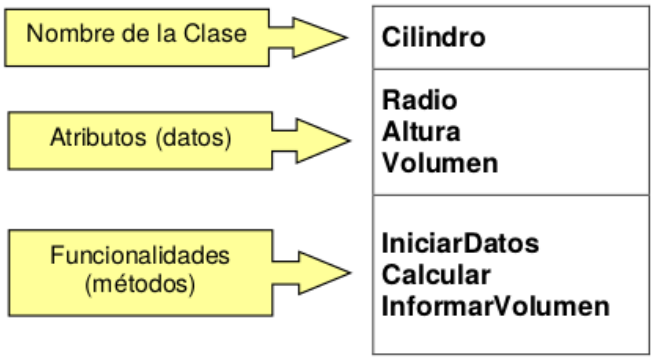
Por otro lado se deberá calcular el volumen a partir del radio y la altura, por lo tanto, la clase cilindro deberá poseer una funcionalidad (método) para calcular el volumen.

Finalmente debemos considerar funcionalidades básicas para asignar los atributos e informar los el volumen del cilindro.

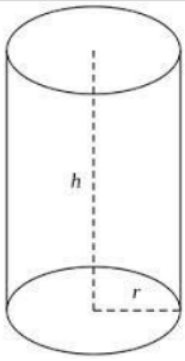
Nota

Esto es lo que debe obtener antes de comenzar a escribir código en JAVA.

Resumimos este diseño sencillo mediante un esquema como el siguiente:



El radio y la altura, son los datos del problema y será necesario asignarlos. Con respecto al volumen está claro que no se requerirá asignarle un valor desde fuera de la clase pero seguramente será necesario calcularlo e informarlo ya que es el resultado del problema.



$$V = A_B . h$$

$$V = \pi . r^2 . h$$

A_B : Área de la Base

h : altura

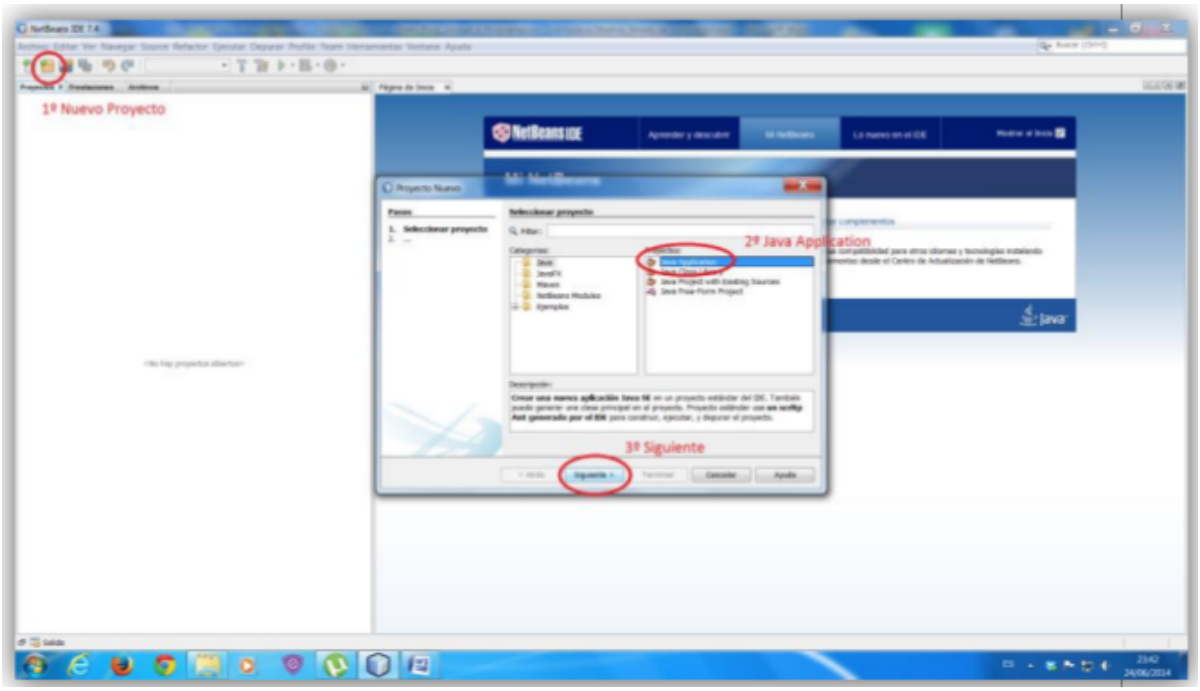
r : radio

Implementación en Java

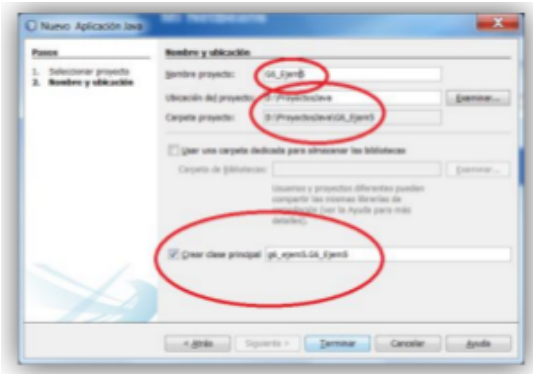
1. Planteamos como codificamos en Java la estructura del diseño anterior.

```
public class Cilindro{
 private double radio, altura, volumen;

 public void setRA(double r, double a){
   radio= r;
   altura= a;
 }

 public double getRadio(){
   return radio;
 }

 public double getAltura(){
   return altura;
 }

 public void calcularVolumen(){
   volumen= Math.PI * Math.pow(radio,2) * altura;
 }

 public double getVolumen(){
   return volumen;
 }

}
```

Resolución de este ejercicio en Java con interfaz gráfica
Primero creamos el proyecto con los métodos ya vistos, ver captura de pantalla siguiente:

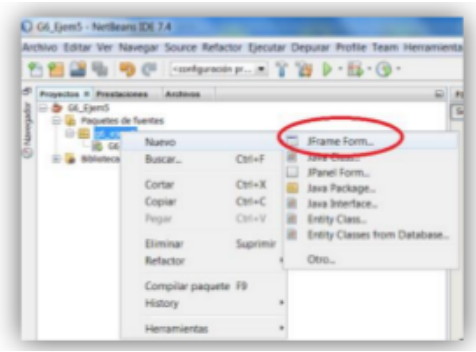
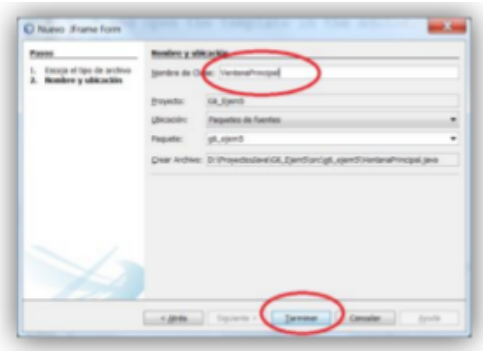


Luego colocamos el nombre del proyecto y verificamos la ubicación en la carpeta correcta. También vamos a crear la clase principal con el mismo nombre.

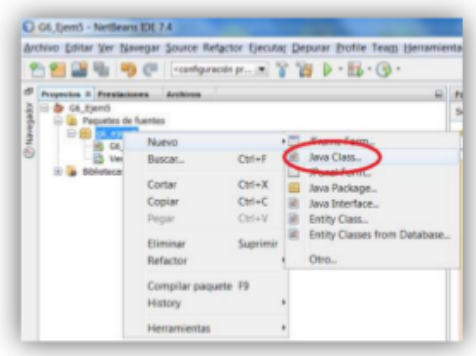
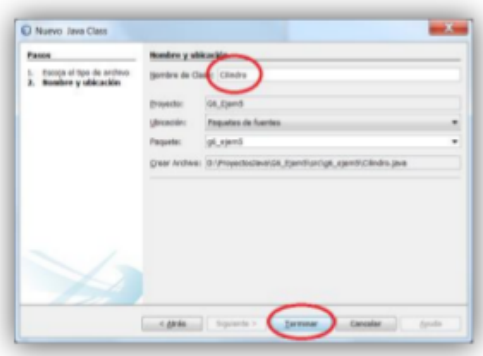


Pulsamos Terminar y se crea el proyecto.

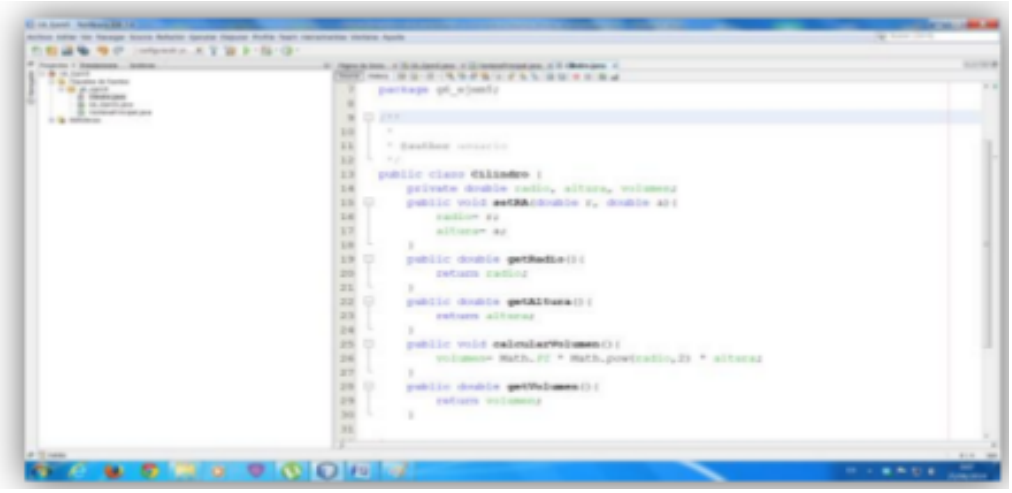
Luego sobre el paquete de fuentes g6_ejem5 click derecho y creo un JFrame al cual llamaremos Ventana-Principal.

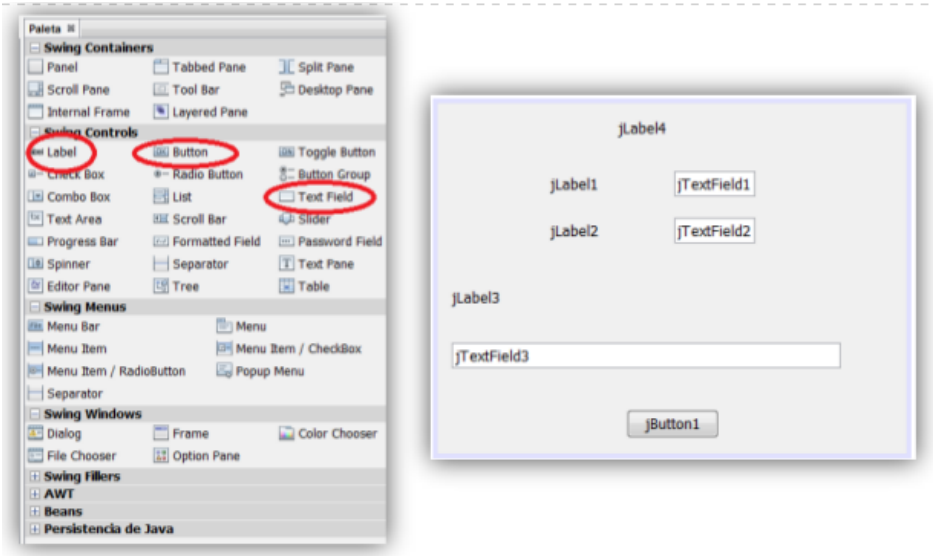
A continuación repetimos el paso anterior pero creamos una Java Class (clase java) llamada Cilindro.

Ahora copiaremos el código de la clase cilindro que escribimos en el punto 1)



A continuación en VentanaPrincipal, diseñaremos una pantalla como esta:

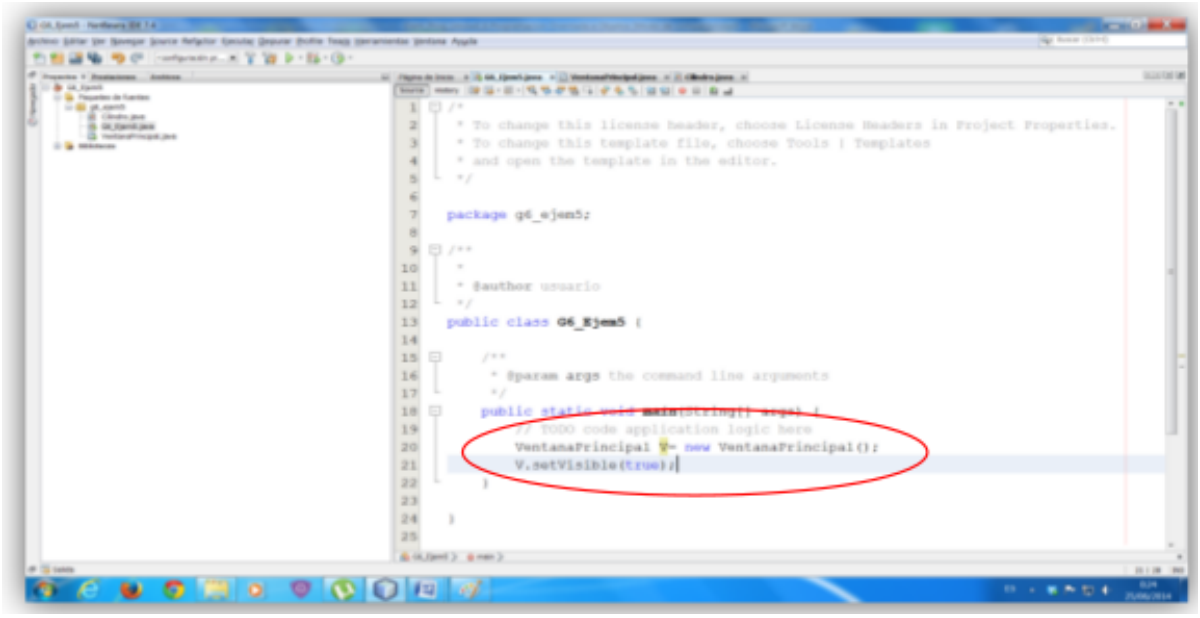


Cambiamos los textos de los objetos

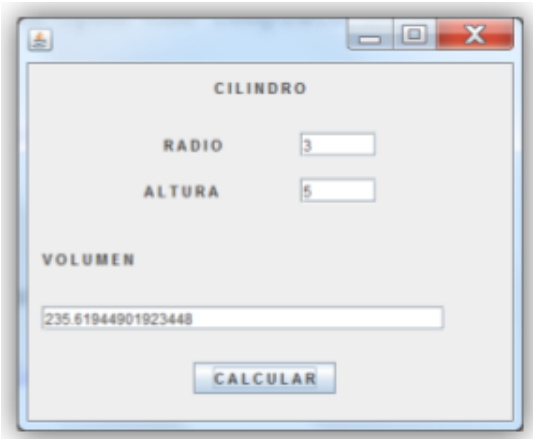
Haciendo doble click en el botón "Calcular"y escribimos el siguiente código.

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    Cilindro C = new Cilindro();
    double alt, rad;
    rad= Double.parseDouble(jTextField1.getText());
    alt= Double.parseDouble(jTextField2.getText());
    C.setRA(alt, rad);
    C.calcularVolumen();
    jTextField3.setText(Double.toString(C.getVolumen()));
}
```

Para finalizar en el main() de la clase principal G6_Ejem5 escribimos el siguiente código:



Compilamos F11 y ejecutamos con F6.-



3. Resumen

- En la programación orientada a objetos el primer paso para resolver un problema es encontrar las clases que describirán el modelo.
- En la búsqueda de las clases candidatas debe realizarse un refinamiento sucesivo. Todo comienza con una lista de los sustantivos del enunciado del problema.
- Después de obtener una lista con las clases del modelo hay que asignar “que conoce” y “que sabe hacer” cada una de ellas.
- Las clases se declaran con la palabra reservada class y se recomienda utilizar una letra mayúscula en el comienzo del nombre Ej: class Cilindro.
- Es muy importante realizar una adecuada encapsulación en cada clase. Esto ayuda a evitar comportamientos impredecibles ya que siempre se tiene control sobre el valor de las variables con que la clase trabaja.

- Java provee de palabras reservadas específicas para realizar la encapsulación: public, private y protected. Sin embargo, cuando no se indica, todo es público.
- Todos los objetos en Java son dinámicos y por lo tanto es necesario declararlos y crearlos con la instrucción new.
- En Java, cada clase deriva directa o indirectamente de la superclase Object
- Una clase en Java puede contener atributos (variables miembro) y métodos, los atributos pueden ser de tipos primitivos o clases y los métodos son funciones.-
- Método constructor: tiene el mismo nombre que la clase a la que pertenece, no tiene tipo específico de retorno, se ejecuta cuando se instancia un objeto de esa clase.
- Java soporta sobrecarga de métodos, es decir que dos o mas métodos pueden tener el mismo nombre.