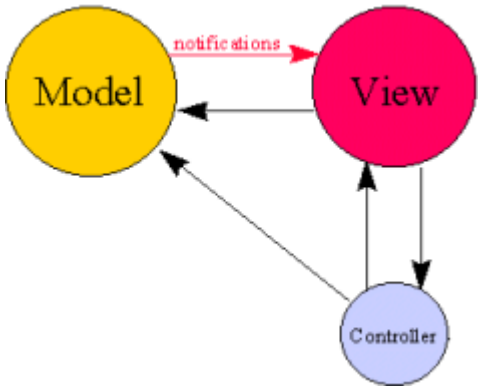
Modelo – Vista – Controlador (MVC)

MVC (Modelo-Vista-Controlador), en palabras simples, es la forma(Patron de Diseño) que utilizamos los programadores para implementar nuestras aplicaciones, además permite separar nuestra aplicación en un modelo, una vista y con controlador. Este patrón fue introducido por primera vez en el lenguaje "Smalltalk".



3 tipos de entidades, cada una especializada en su tarea.

Modelo:
Es el encargado de administrar la lógica de la aplicación. Tiene como finalidad servir de abstracción de algún proceso en el mundo real; además tiene acceso a nuestra Base de Datos, agregando q tiene las funciones que controlan la integridad del sistema.
Vista
Sencillamente es la representación visual del modelo. Es la encargada de representar los componentes visuales en la pantalla, Esta asociada a un Modelo, esto le permite que al momento de cambiar el Modelo, la vista re-dibuja la parte efectada para reflejar los cambios.
Controlador
Es el escuchador a los eventos que genere el usuario, es decir, es el que permite que interactúen el usuario con el sistema. Interpreta los eventos (la entradas) a través del teclado y/o ratón.
Por que usar MVC?
Porque fue diseñada para reducir el esfuerzo al momento de programar. Además porque permite una clara separación entre los componentes de un programa; lo cual nos permite implementarlos por separado.

Permite el trabajo en equipo.

Java implementa MVC?

La mayoría de los componentes SWING, han tomado como patrón de diseño MVC, lo cual es una gran ventaja para los programadores, porque permite implementar el PROPIO modelo de datos para cada componente swing.

Clases internas, clases anidadas o nested classes

Las clases internas se agregaron a Java para mejorar la claridad del código y para hacer más concisos los programas.

Básicamente es una clase como cualquier otra. Se declara de la misma forma que las demás pero dentro de otra clase, específicamente dentro del bloque de código especificado por las llaves, incluyendo los bloques que son partes de un método. Las clases definidas dentro de

la clase y fuera de un método tienen ciertas diferencias con las definidas dentro de un método y son llamadas clases miembro.

Las clases internas se nombran utilizando el nombre totalmente calificado de la clase que la contiene seguido de un punto y el nombre de la clase. Esto es correcto inclusive si hay un paquete con el mismo nombre que la clase interna ya que una clase interna pertenece a la clase que la contiene de una forma similar en la que una clase pertenece a un paquete. Esto no se refleja totalmente en el nombre de la clase ya que el punto que separa la clase interna de la clase que la contiene se sustituye por un singo de pesos lo que trae problemas si se desea cargar la clase utilizando el método Class.forName(). También se usa si se obtiene el nombre de la clase utilizando getClass.getName(). Este signo se toma para evitar ambigüedades entre nombres de paquetes y clases internas. También reduce los conflictos con sistemas que utilizan como carácter especial el punto.

En el código las clases internas proporcionan beneficios de organización, pero también tienen la habilidad de acceder a miembros de las clases externas, esto quiere decir que cuando una instancia de una clase interna es creada, normalmente una instancia de la clase externa, o sea la clase que la contiene, actúa como contexto. Esta instancia de clase externa es accesible desde el objeto interno. Esta accesibilidad de los miembros de la clase externa es crucial y muy útil. Es posible porque la clase interna tiene una referencia escondida a la clase externa que fue el contexto actual donde al objeto de la clase interna fue creado asegurándonos que la clase interna y la externa estén juntas.

Se puede querer crear una instancia de una clase interna desde un método estático, o en situaciones donde no hay objeto this disponible.

Podemos tener esta situación en un método main() o si se necesita crear una clase interna de un método o de un objeto de una clase que no esta relacionada. Se puede lograr esto utilizando el operador new como si fuera un método miembro de la clase externa.

```
ClaseExterna.ClaseInterna interna = new ClaseExterna.new ClaseInterna();
interna.metodo();
```

Si no se especifica una clase externa se asume el prefijo this, esta referencia debe ser válida en el momento de utilizarla. Un método estático no tiene referencia this, por lo que se debe tener cuidado en estas condiciones.

Clases miembro

Hay ciertas características que son únicas para las clases miembro. Los miembros de una clase, pueden ser marcados con modificadores para controlar su acceso. También pueden ser marcados como estáticos lo que significa que la clase no tiene ninguna referencia a una instancia de la clase externa por lo que los métodos de la clase interna no pueden utilizar la palabra clave this para acceder a variables de instancia de la clase externa pero si a miembros estáticos de la clase externa.

Realmente, el resultado es una clase de nivel más alto con un esquema de nombres modificados que de hecho se pueden utilizar como extensión de los paquetes.

Clases definidas dentro de métodos

Las clases definidas dentro de un método son locales al método por lo que son privadas al método y no se les pueden modificar el acceso como tampoco pueden ser marcadas como estáticas.

Por otro lado los objetos creados desde una clase interna dentro de un método pueden tener algún acceso a las variables del método que la encierra. La regla es simple, cualquier variable puede ser accedida por métodos dentro de la clase interna, siempre que esa variable esté marcada como final lo que puede ser una restricción muy fuerte. El problema es que un objeto creado dentro de un método vive durante la invocación del método. Las variables y argumentos del método son destruidos cuando se sale del método, estas variables no son válidas para acceder desde el método interno, o sea el método que realizó la llamada, cuando el método es finalizado. Permitiendo el acceso a solo a variables finales, es posible copiar los valores de de esas variables en el propio objetos, y de esta forma extender sus tiempos de vida.

En un método es posible crear clases internas anónimas, o sea clases sin un nombre específico. Estas pueden ser declaradas para extender otra clase o implementar una sola interfase. La sintaxis no permite colocar dos al mismo tiempo, o implementar mas de una interfase explícita a no ser que este extendiendo una clase que implementa interfases. Si se declara una clase que implemente una sola interfase explícita entonces esta es una subclase de java.lang.Object.

La definición, construcción y el primer uso de una clase anónima sucede en el mismo lugar como muestra el ejemplo.

```
public void unMetodo() {
   boton.addActionListener(
      new ActionListener() {
         public void accionPerformed(ActionEvent e) {
            System.out.println("Se apretó el botón");
         }           }    );   }
```

La clase no tiene nombre pero es referenciada simplemente para utilizar el nombre de la clase o interfase de donde la clase es derivada. La clase o interfase se extiende sin utilizar la palabra clave implements o extends.

Esto es útil para no tener que pensar nombres triviales para las clases. Hay que tener cuidado de no abusar de esta facilidad ya que no se puede crear una instancia de una clase anónima. También se debe mantener pequeño el código de la clase. Si una clase anónima supera las diez o quince líneas probablemente no debería ser anónima.

Dado que las clases internas anónimas existen dentro del alcance del método no se puede tener una clase miembro anónima.

No se puede definir un constructor para las clases internas anónimas ya que no se podría especificar cual es el constructor, igualmente la clase puede ser construida con argumentos bajo ciertas condiciones así como también iniciar valores.

La forma general para declarar una clase anónima sería la siguiente:

```
Xxxx unaXxxx = new Xxxx() { /* cuerpo de la clase */ }
```

Donde Xxxx es el nombre de la clase o interfase que se extiende.

Si la clase anónima extiende otra clase y la clase padre tiene constructores que toman argumentos, se puede invocar uno de estos constructores especificando la lista de argumentos de la clase en la clase interna anónima de la siguiente forma:

```
public void unMetodo() {
   boton.addActionListener(
      new unActionListener("Parametro") {
         /* Codigo de la extensión */
      }    );   }
```

Si se necesita iniciar algún tipo de valor en una clase anónima se puede utilizar un bloque sin nombre en el alcance de la clase. El siguiente ejemplo muestra esto:

```
boton.addActionListener(
   new unActionListener("Parametro") {
      {
         /* Iniciamos valores */
      }    }    );
```

Esto se puede realizar en cualquier clase, pero la técnica es particularmente útil con clases internas anónimas donde es la única herramienta que se tiene para proporcionar control sobre el proceso de construcción.

Interfaces gráficas

Para diseñarlas, java proporciona una biblioteca de clases denominada JFC (Java Fundation Classes – clases base de java). Se agrupan en las APIs: Swing, AWT(another windowing toolkit, otro juego de herramientas para manejar ventanas- tambien abstract- juego de herram. abstractas), Java 2D, accesibilidad y soporte para arrastrar y colocar.

Estructura de una aplicación

Una aplicación que muestra una interfaz gráfica cuando se ejecuta, no es más que un objeto de una clase derivada de JFrame.

Ejemplos
(Libro: Java 2-Curso de Programación-3ra Edición Actualizada Fco. Javier Ceballos)

```
package mvcceballos;
import java.util.*;
import java.awt.*;
public class CAplicacion extends javax.swing.JFrame        // declaración
{

   /** Crear un nuevo formulario de la clase CAplicación */
  public CAplicacion()      // constructor
  {
    setSize(300, 200);      // tamaño del formulario
    setTitle("Aplicación"); // título del formulario
```

```
 initComponents();       // iniciar los controles o componentes
 f.add("a");f.add("b");f.add("c");f.add("d");f.add("e");f.add("f");
 f.add("g");f.add("h");f.add("i");
}
/** Este método es llamado desde el constructor CAplicación */
private void initComponents()                  // redefinición
{
 jEtSaludo = new javax.swing.JLabel();
 jBtSaludo = new javax.swing.JButton();

 getContentPane().setLayout(null);

 setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);

 jEtSaludo.setFont(new java.awt.Font("Dialog", 1, 18));
 jEtSaludo.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
 jEtSaludo.setText("etiqueta");
 getContentPane().add(jBtSaludo);
 jEtSaludo.setBounds(42, 36, 204, 30);
 jBtSaludo.setMnemonic('c');
 jBtSaludo.setText("Haga clic aquí");
 jBtSaludo.setToolTipText("botón de pulsación");
 jBtSaludo.addActionListener(new java.awt.event.ActionListener()
 {
    public void actionPerformed(java.awt.event.ActionEvent evt)
  {
          jBtSaludoActionPerformed(evt);
  }
 });
 getContentPane().add(jEtSaludo);
 jBtSaludo.setBounds(42, 90, 204, 30);

}
*→ private void jBtSaludoActionPerformed (java.awt.event.ActionEvent evt)
{                                          ///redefinicion

 float rojo = (float)Math.random();
 float verde = (float)Math.random();
 float azul = (float)Math.random();
 jEtSaludo.setForeground(new java.awt.Color(rojo, verde, azul));
 int m = (int)(Math.random()*10);
 if (m>=0 && m <=8)
    jEtSaludo.setText(f.get(m));
 else
    jEtSaludo.setText("INDICE AFUERA");

}

/** Exit the Application */
```

Redefine dentro de la invocación del parámetro *

```
  private void exitForm(java.awt.event.WindowEvent evt)
  {
    System.exit(0);
  }

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

  }

  // Declaración de variables
  private javax.swing.JButton jBtSaludo;
  private javax.swing.JLabel jEtSaludo;
  public static Vector<String> f = new Vector<String>(10,1);
}



package mvcceballos;

public class Reloj extends javax.swing.JApplet
                implements Runnable
{
  public void init()
  {
    initComponents();
  }

  private void initComponents()
  {
    horaActual = new javax.swing.JLabel();

    horaActual.setFont(new java.awt.Font("Arial", 1, 48));
    horaActual.setHorizontalAlignment(
                      javax.swing.SwingConstants.CENTER);
    horaActual.setText("00:00:00");
    getContentPane().add(horaActual, java.awt.BorderLayout.CENTER);
  }

  public void start()               //inicio
  {
    if (hilo == null)
    {
      hilo = new Thread(this," ");
      hilo.start();
    }
  }
```

```
 public void run()              //ejecución
 {
  Thread hiloActual = Thread.currentThread();
  while (hilo == hiloActual)
  {
   // Obtener la hora actual
   java.util.Calendar cal = java.util.Calendar.getInstance();
   java.util.Date hora = cal.getTime();
   java.text.DateFormat df = java.text.DateFormat.getTimeInstance();
   horaActual.setText(df.format(hora));
   try
   {
    Thread.sleep(1000);
   }
   catch (InterruptedException e){ }
  }
 }

 public void stop()             //parada
 {
  hilo = null;
 }

 // Declaración de variables
 private javax.swing.JLabel horaActual;
 private Thread hilo = null;
 private java.awt.Font fuente;
}
```

En este post trataremos estos temas
Clases Internas
Clases internas dentro de un método
Clases anónimas

Clases Internas

```
package innerclasstest;
public class Main {
        private class MyInnerClass
        {
                private String label;

                public MyInnerClass(String label)
                {
                        this.label=label;
                }
                public MyInnerClass()
                {
                        this.label="Soy un constructor sin parametros :(";

                }
```

```
                public void printLabel()
                {
                        System.out.println(this.label);
                }
        }
        public Main()
        {
                MyInnerClass uno=new MyInnerClass("Soy un constructor con
parametros");
                MyInnerClass dos=new MyInnerClass();
                uno.printLabel();
                dos.printLabel();

        }
        public static void main(String[] args) {
                Main test=new Main();

        }
}
```

En el ejemplo anterior notamos tres puntos importates.
La clase interna (MyInnerClass) se declara exactamente como cualquier otra clase
Se instancia como cualquier otra clase dentro de la clase contenedora.
Puede declararse como privada, publica o protegida.

Como se ha mencionado puede llamar a los elementos de la clase contenedora aunque sean declarados como privados, para hacer la llamada tenemos que hacerlo de la siguiente manera: *NombredelaClaseContenedora.**this**.nombreMetodoOPropiedad*.
Seria algo como asi,

```
package innerclasstest;
public class Main {
        /*Clase Interna*/
        private class MyInnerClass
        {

                        public void callToCall(String parametro)
                        {
                                Main.this.call(parametro);/*OJO la
llamada*/
             }
        }
        /*Fin de la clase Interna*/
 private void call(String parametro)
        {
                        System.out.println(parametro);
}

        public Main()
        {
MyInnerClass test=new MyInnerClass();
            uno.callToCall("Estoy Llamando un metodo de la clase
contenedora");
}
        public static void main(String[] args) {
                Main test=new Main();
```

```
        }
}
```

Como vemos call(String parametro) es privado pero es facilmente llamado por la clase interna

Clases internas dentro de un método

Como hemos visto una clase puede contener una clase, pero dentro de java podemos declarar una clase dentro de una método, en este ciclo de vida de la clase depende de la ejecución del método, o en todo caso dependerá de cuanto dure el bloque donde estara declarada la clase, antes o después del bloque la clase no existe por ende no puede ser instanciada, un ejemplo seria algo asi

```
package innerclasstest;
public class Main {

    public void ContrustruirUnaClase()
    {
        class Metodo
        {
            public void print(){System.out.println("hola mundo");};
        }
        new Metodo().print();
    }

        public Main()
        {
        this.ContrustruirUnaClase();
        }
        public static void main(String[] args) {
                Main test=new Main();

        }
}
```

Aunque seria un uso poco común el anterior ejemplo, y la verdad no tendría sentido, el uso mas común de contener una clase dentro de un método seria retornar una instancia de esta clase. Para esto necesitamos que nuestra clase interna herede una clase (ya sea abstracta o sencilla) o implemente una interfaz. El codigo de la interfaz y la clase abstracta seria algo como así.

```
abstract class  AbstractClass {
    protected int res=0;

    public  abstract void addToSum(int b);

    public int getRes() {
        return res;
    }
}
```

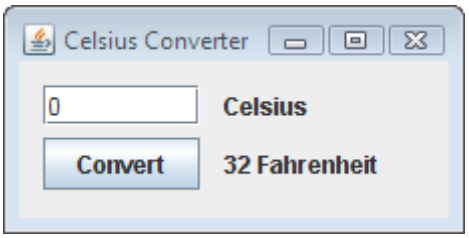
Y el de la interfaz

```
public interface IInterface {
```

```
    int Sum(int b,int c);
}
```

Ahora veamos como seria la implementación y una prueba, cabe destacar que este tipo de implementación facilita el patrón de fabrica abstracta

```
package innerclasstest;
public class Main {

    public IInterface getInstance()
    {
        class test implements IInterface
        {
            public int Sum(int b, int c) {
                return b+c;
            }
        }
        return new test();
    }

    public AbstractClass getInstanceAbstract()
    {
        class test extends AbstractClass
        {

            @Override
            public void addToSum(int b) {
                super.res+=b;
            }

        }
        return new test();
    }

        public static void main(String[] args) {
                Main test=new Main();
                Interface test2=test.getInstance();
                System.out.println(test2.Sum(3, 5));
                AbstractClass test3=test.getInstanceAbstract();
            test3.addToSum(5);
            test3.addToSum(10);
            test3.addToSum(11);
            System.out.println(test3.getRes());
        }
}
```



The CelsiusConverter Application.

Explicación de uso de formulario

http://java.sun.com/docs/books/tutorial/uiswing/learn/settingup.html

Clases anónimas

Como hemos visto las clases internas tiene una utilidad bastante evidente, sobre todo en la capacidad que tienen de ocultar código, a las programadores que usan la herramienta NetBeans notaran que cuando este genera un JFrame genera un main y dentro del main incluye unas lineas parecidas a esta

```
  java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new NewJFrame().setVisible(true);
            }
        });
```

Como vemos este realiza un new Runnable(), pero Runnable es una interfaz y aparte de esto desarrolla un cuerpo e implementa el método run(), en java se permite este tipo de instrucciones lo cual esta clase que implementa Runnable se le conoce como una clase anónima, la cual no posee un nombre, pero se sabe que implementa o extiende de una interfaz o una clase, por lo cual se le puede hacer un upcasting. Ahora del ejemplo del subtema anterior usaremos la clase abstracta AbstractClass y la interfaz IInterface, Ahora retornaremos usando una clase anónima dentro de la estructura de los métodos

```
 package innerclasstest;
public class Main {
        public AbstractClass getInstanceAbstractTwo()
    {
        return new AbstractClass(){

            @Override
            public void addToSum(int b) {
               super.res+=b;
            }           };          }
     public IInterface getInstanceTwo()
    {
        return new IInterface()
        {
            public int Sum(int b, int c) {
                return b+c;
            }           };          }
       public static void main(String[] args) {
               Main test=new Main();

        AbstractClass test4=test.getInstanceAbstractTwo();
        test4.addToSum(10);
        test4.addToSum(11);
        test4.addToSum(12);
        System.out.println(test4.getRes());

        IInterface test5=test.getInstanceTwo();
        System.out.println(test5.Sum(10, 21));
       }       }
```

Como vemos es una manera abreviada de retornar una clase implementada en el cuerpo de un método. Una desventaja de este método es que al no tener una clase con un nombre el compilador no permitirá sobrecargar el constructor vacio, para esto deberemos declarar un bloque sin nombre y definir los parámetros en el metodo contenedor como final, ya que los parámetros tienen que existir para ser llamados en el bloque vació. Por inconveniente el constructor no puede ser sobrecargado y no puede existir más de un constructor. Un ejemplo seria esto.

```
package innerclasstest;
public class Main {

    public IInterface getInstanceTwo(final int b,final int c)
    {
        return new IInterface()
        {
            private int k;
            private int l;
            {
                this.k=b;
                this.l=c;
            }
             public int Sum(int b, int c) {
                return k+l+c+b;
            }          };           }
       public Main()
       public static void main(String[] args) {
Main test=new Main();
           IInterface test6=test.getInstanceTwo(10, 20);
           System.out.println(test6.Sum(10, 21));
       }
}
```

En un futuro post trataremos instanciar clases internas en otras clases y clases internas dentro de clases abstractas e interfaces.