# **Week 4 (2/1 - 2/7)**

No unread replies.No replies.

 This week's activities will consist of the following:

* Read Chapter 13 - Graphics and Java 2D & Review PowerPoint
* Read Chapter 12 - Graphical User Interface Components: Part I  & Review PowerPoint
* Week 4 Q & A
* **Complete Project 5 & Project 6 Activity ( zip and submit all files required to compile & run the application).  Project 5 is due by Friday, February 12th; Project 4 is due Friday, February 12th**
* **Complete the "Java Delegation Event Model" DB by Sunday, February  7th.**

Note: Java provides support for performing graphics operations (i.e., drawing circles, ellipses and other shapes) and creating GUIs. These two chapters provide an introduction to the Java APIs that support GUIs and other graphics features. Java provides support for creating Graphical User Interfaces (GUIs). Chapter 12 describes the introductory concepts of GUI design. The concepts covered include basic GUI components (e.g., buttons, textboxes, lists, check-boxes) and the use of three layout managers (i.e., FlowLayout, BorderLayout & GridLayout). Layout managers are responsible for the location of the GUI components within a container.  This frees the developer from worrying about the positioning of the components.  The chapter also describes the Java Event handling model. The event handlers are responsible for implementing actions in response to a GUI event (e.g., mouse-click, key selection, etc.).

The Java Tutorial provides good reference material describing event handling for the GUI components.  The link is below.

[http://java.sun.com/docs/books/tutorial/using/events/index.html (Links to an external site.)](http://java.sun.com/docs/books/tutorial/uiswing/events/index.html)

The Java Tutorial also provides good reference material describing Java’s 2D Graphics capabilities.  The link is below.

[http://java.sun.com/docs/books/tutorial/2d/index.html (Links to an external site.)](http://java.sun.com/docs/books/tutorial/2d/index.html)

What is the difference between implementing the event handler using a

"Listener" vs. an "Adaptor"?

From book:

GUIs are**event driven**. When the user interacts with a GUI component, the interaction—known as an **event**—drives the program to perform a task. Some common user interactions that cause an application to perform a task include clicking a button, typing in a text field, selecting an item from a menu, closing a window and moving the mouse. The code that performs a task in response to an event is called an **event handler**, and the process of responding to events is known as **event handling**.

…

Before an application can respond to an event for a particular GUI component, you must:

**1.** Create a class that represents the event handler and implements an appropriate interface—known as an **event-listener interface**.

**2.** Indicate that an object of the class from Step 1 should be notified when the event occurs—known as **registering the event handler**.

…

summarize the three parts to the event-handling mechanism that you saw in [Section 12.6](http://proquest.safaribooksonline.com.portal.lib.fit.edu/9780133813036/ch12lev1sec6_html#ch12lev1sec6)—the event source, the event object and the event listener. The event source is the GUI component with which the user interacts. The event object encapsulates information about the event that occurred, such as a reference to the event source and any event-specific information that may be required by the event listener for it to handle the event. The event listener is an object that’s notified by the event source when an event occurs; in effect, it “listens” for an event, and one of its methods executes in response to the event. A method of the event listener receives an event object when the event listener is notified of the event. The event listener then uses the event object to respond to the event. This event-handling model is known as the **delegation event model**—an event’s processing is delegated to an object (the event listener) in the application.

For each event-object type, there’s typically a corresponding event-listener interface. An event listener for a GUI event is an object of a class that implements one or more of the event-listener interfaces from packages java.awt.event and javax.swing.event.

…

For many of the listener interfaces that have multiple methods, packages java.awt.event andjavax.swing.event provide event-listener adapter classes

An **adapter class** implements an interface and provides a default implementation (with an empty method body) of each method in the interface.

..

You can extend an adapter class to inherit the default implementation of every method and subsequently override only the method(s) you need for event handling.

“An **event adaptor**is an **implementation** of an **event listener** interface that can be …´internet Kennesaw.edu

Blogs.oracle.com

https://blogs.oracle.com/CoreJavaTechTips/entry/listeners\_vs\_adapters

Listeners vs Adapters

By John O'Conner on [Jul 17, 2007](https://blogs.oracle.com/CoreJavaTechTips/entry/listeners_vs_adapters)

by John Zukowski

The JavaBeans component model (and thus the Swing component set) is built upon properties and events. Properties have setter and getter methods for working with their values. Events require you to use listeners and to implement interfaces in order to receive notification of their occurrence. Although working with properties is simple, listener objects require a little extra discussion to understand how they work, typically in the graphical user interface (GUI) world. Specifically, this tip describes the AWT and Swing event-related classes that offer both a listener interface and an adapter implementation.

The following classes show examples of listener and adapter pairs:

[package java.awt.event](http://java.sun.com/javase/6/docs/api/java/awt/event/package-summary.html)

- ComponentListener/ComponentAdapter

- ContainerListener/ContainerAdapter

- FocusListener/FocusAdapter

- HierarchyBoundsListener/HierarchyBoundsAdapter

- KeyListener/KeyAdapter

- MouseListener/MouseAdapter

- MouseMotionListener/MouseMotionAdapter

- WindowListener/WindowAdapter

[package java.awt.dnd](http://java.sun.com/javase/6/docs/api/java/awt/dnd/package-summary.html)

- DragSourceListener/DragSourceAdapter

- DragTargetListener/DragTargetAdapter

[package javax.swing.event](http://java.sun.com/javase/6/docs/api/javax/swing/event/package-summary.html)

- InternalFrameListener/InternalFrameAdapter

- MouseInputListener/MouseInputAdapter

These class pairs offer two ways to do the same thing. First, consider a simple example that doesn't offer an adapter class. The ActionListener class has a single actionPerformed method. Using an anonymous inner class, you will typically use an ActionListener class in the following manner:

ActionListener listener = new ActionListener() {

public void actionPerformed(ActionEvent actionEvent) {

System.out.println("Event happened");

}

};

You can also use the actionPerformed method by implementing the ActionListener interface in a high-level class:

public class MyClass extends JFrame implements ActionListener {

...

public void actionPerformed(ActionEvent actionEvent) {

System.out.println("Event happened");

}

}

The ActionListener interface has a single method, and implementers of the interface must provide an implementation of that single method for it to do much of anything.

Other listener interfaces aren't quite so simplistic. For example, the MouseMotionListener interface has two methods:mouseDragged and mouseMoved. When you implement an interface, you must implement all the methods defined by the interface:

MouseMotionListener listener = new MouseMotionListener() {

public void mouseDragged(MouseEvent mouseEvent) {

System.out.println("I'm dragging: " + mouseEvent);

}

public void mouseMoved(MouseEvent mouseEvent) {

System.out.println("I'm moving: " + mouseEvent);

}

};

There are situations when your application doesn't need to track all events for a particular listener interface. Maybe your code only needs to respond to one or two of the methods in a listener interface. For instance, do you really want to know when a mouse moves, or only when it moves with a mouse button depressed? You cannot implement just one of the MouseMotionListener methods and leave the others out:

MouseMotionListener badListener = new MouseMotionListener() {

public void mouseDragged(MouseEvent mouseEvent) {

System.out.println("I'm dragging: " + mouseEvent);

}

};

This listener implementation will result in a compile-time error since an interface isn't fully implemented. With an interface like MouseMotionListener, that isn't too much of a problem, you just have to provide a stub for the method you aren't interested in:

MouseMotionListener listener = new MouseMotionListener() {

public void mouseDragged(MouseEvent mouseEvent) {

System.out.println("I'm dragging: " + mouseEvent);

}

public void mouseMoved(MouseEvent mouseEvent) {

// Do nothing

}

};

Not all listener interfaces are so small. Although MouseMotionListener has only two methods, the MouseListenerinterface has five:

* void mouseClicked(MouseEvent mouseEvent)
* void mouseEntered(MouseEvent mouseEvent)
* void mouseExited(MouseEvent mouseEvent)
* void mousePressed(MouseEvent mouseEvent)
* void mouseReleased(MouseEvent mouseEvent)

If you want to add a MouseListener to a component, your interface implementation must have five methods in it:

MouseListener mouseListener = new MouseListener() {

public void mouseClicked(MouseEvent mouseEvent) {

System.out.println("I'm clicked: " + mouseEvent);

}

public void mouseEntered(MouseEvent mouseEvent) {

System.out.println("I'm entered: " + mouseEvent);

}

public void mouseExited(MouseEvent mouseEvent) {

System.out.println("I'm exited: " + mouseEvent);

}

public void mousePressed(MouseEvent mouseEvent) {

System.out.println("I'm pressed: " + mouseEvent);

}

public void mouseReleased(MouseEvent mouseEvent) {

System.out.println("I'm released: " + mouseEvent);

}

};

If your application only needs to know whether the mouse is pressed or released over a component, the other three methods will be empty and ignored. Those methods are unnecessary code. The adapter classes can help reduce the amount of code you must write when your application needs only a small subset of all interface methods. Each adapter class fully implements its associated interface (or interfaces). Then, if you want a listener for a subset of associated methods, you just have to provide that subset. No empty stubs required. Here is just such an adapter for the required MouseListener previously described.

MouseListener mouseListener = new MouseAdapter() {

public void mousePressed(MouseEvent mouseEvent) {

System.out.println("I'm pressed: " + mouseEvent);

}

public void mouseReleased(MouseEvent mouseEvent) {

System.out.println("I'm released: " + mouseEvent);

}

};

This code still creates a MouseListener. However, instead of implementing all the interface methods that you don't care about, with the help of MouseAdapter, you only have to implement those MouseListener methods you are truly interested in.

Not every multi-method listener has an adapter. You can certainly create your own if you constantly find your self stubbing out most of an interface. Of the built-in classes, only the listeners listed at the top of this tip offer them. Also, the adapters are true classes, not interfaces. If you want your custom JButton subclass to also implementMouseListener, you cannot have that class subclass MouseAdapter, as only single inheritance is allowed. For example, the following code causes a compilation-time error because it attempts to subclass both JButton andMouseAdapter:

public class BadJButtonSubclass extends JButton, MouseAdapter {

...

public void mousePressed(MouseEvent mouseEvent) {

System.out.println("I'm pressed: " + mouseEvent);

}

}

If you truly wanted this JButton subclass to be a MouseListener, you must explicitly say so, and make sure all the methods of the interface are implemented:

public class GoodJButtonSubclass extends JButton implements MouseListener {

...

public void mouseClicked(MouseEvent mouseEvent) {

// Do nothing

}

public void mouseEntered(MouseEvent mouseEvent) {

// Do nothing

}

public void mouseExited(MouseEvent mouseEvent) {

// Do nothing

}

public void mousePressed(MouseEvent mouseEvent) {

System.out.println("I'm pressed: " + mouseEvent);

}

public void mouseReleased(MouseEvent mouseEvent) {

// Do nothing

}

...

addMouseListener(this);

...

}

Of course, you don't have to have your high-level class implement the interface itself. This may be a good example of when you should create the listener as an inner or anonymous class instead.

If you use an integrated development environment (IDE) to create your user interface, the IDE will often generate the interface framework for you. You will need to code the business logic inside the necessary interface methods. An IDE can simplify the implementation of a large interface.

For more information about this topic, read the [How to Write a Mouse Listener](http://java.sun.com/docs/books/tutorial/uiswing/events/mouselistener.html) lesson of [The Java Tutorial](http://java.sun.com/docs/books/tutorial/index.html).

Adapters aren't limited to mouse listening. However, the MouseAdapter is a frequent example because theMouseListener interface has so many methods. The WindowListener interface is also another large interface, and it has an associated WindowAdapter class.

==========================

Book:

When a class implements an interface, the class has an is-arelationship with that interface. All direct and indirect subclasses of that class inherit this interface. Thus, an object of a class that extends an event-adapter class is anobject of the corresponding event-listener type (e.g., an object of a subclass of*MouseAdapter* is a *MouseListener*).

http://www.ccs.neu.edu/course/com3118/EventHandling.html

**Adapter Class**

Provides an empty implementation of all methods in an event listener interface.

Useful when u want to listen and process only some of the events that are handled by one particular event listener interface.

Define your own class as subclass of this class and provide desired implementation.

Listener Interfaces implemented by Adapter classes

|  |  |
| --- | --- |
| **Adapter Class**  ComponentAdapter  ContainerAdapter  FocusAdapter  KeyAdapter  MouseAdapter  MouseMotionAdapter  WindowAdapter | **Listener Interface**  ComponentListener  ContainerListener  FocusListener  KeyListener  MouseListener  MouseMotionListener  WindowListener |

<https://docs.oracle.com/javase/tutorial/uiswing/events/generalrules.html>

Some listener interfaces contain more than one method. For example, the MouseListenerinterface contains five methods: mousePressed, mouseReleased, mouseEntered,mouseExited, and mouseClicked. Even if you care only about mouse clicks, if your class directly implements MouseListener, then you must implement all five MouseListenermethods.

The resulting collection of empty method bodies can make code harder to read and maintain. To help you avoid implementing empty method bodies, the API generally includes an *adapter* class for each listener interface with more than one method. (The [Listener API Table](https://docs.oracle.com/javase/tutorial/uiswing/events/api.html) lists all the listeners and their adapters.) For example, the MouseAdapter class implements theMouseListener interface. An adapter class implements empty versions of all its interface's methods.

To use an adapter, you create a subclass of it and override only the methods of interest, rather than directly implementing all methods of the listener interface. Here is an example of modifying the preceding code to extend MouseAdapter. By extending MouseAdapter, it inherits empty definitions of all five of the methods that MouseListener contains.

============================================

What is the difference between implementing the event handler using a

"Listener" vs. an "Adaptor"?

Choose a unique(i.e., different from classmate's entries)  listener type and

describe the event that causes the methods of the listener to be invoked.

Listener type:

Describe the event that causes the listener to be invoked:

In addition, describe a scenario where that listener type will be used in an application.:

## Tracking Focus Changes to Multiple Components

In some situations an application may need to track which component has the focus. This information might be used to dynamically update menus or perhaps a status bar. If you need to track the focus only on specific components, it may make sense to implement a [focus event listener](https://docs.oracle.com/javase/tutorial/uiswing/events/focuslistener.html).

Displaying information about a picture when the picture is clicked

Book:

You can type in a TextField only if it’s “in focus”—that is, it’s the control that the user is interacting with. When you click an interactive control, it receives the focus. Similarly, when you press the Tab key, the focus transfers from the current focusable control to the next one—this occurs in the order the controls were added to the GUI.

https://docs.oracle.com/javase/tutorial/uiswing/events/focuslistener.html

A focus listener, registered on each component, reports every focus-gained and focus-lost event. For each event, the other component involved in the focus change, theopposite component, is reported. For example, when the focus goes from a button to a text field, a focus-lost event is fired by the button (with the text field as the opposite component) and then a focus-gained event is fired by the text field (with the button as the opposite component). Focus-lost as well as focus-gained events can be temporary. For example, a temporary focus-lost event occurs when the window loses the focus. A temporary focus-gained event occurs on popup menus.

**What is the difference between implementing the event handler**

**using a "Listener" vs. an "Adaptor"?**

Some Listeners have more than one method so there is also an adaptor class:

The following classes show examples of listener and adapter pairs:

[package java.awt.event](http://java.sun.com/javase/6/docs/api/java/awt/event/package-summary.html)

- ComponentListener/ComponentAdapter

- ContainerListener/ContainerAdapter

- FocusListener/FocusAdapter

- HierarchyBoundsListener/HierarchyBoundsAdapter

- KeyListener/KeyAdapter

- MouseListener/MouseAdapter

- MouseMotionListener/MouseMotionAdapter

- WindowListener/WindowAdapter

[package java.awt.dnd](http://java.sun.com/javase/6/docs/api/java/awt/dnd/package-summary.html)

- DragSourceListener/DragSourceAdapter

- DragTargetListener/DragTargetAdapter

[package javax.swing.event](http://java.sun.com/javase/6/docs/api/javax/swing/event/package-summary.html)

- InternalFrameListener/InternalFrameAdapter

- MouseInputListener/MouseInputAdapter

From: [https://blogs.oracle.com/CoreJavaTechTips/entry/listeners\_vs\_adapters](https://docs.oracle.com/javase/7/docs/api/java/awt/event/FocusListener.html)

If your class implements one of the interfaces with multiple methods, it must override all the methods.

If you do not use all of the methods, the result is unused methods and extra code which is  harder to maintain.

The adaptor class for the listener enables you to just override the methods you need to use when you create a sublass of the adaptor class.(The adapter class implements empty versions of all its interface's methods.)

**Choose a unique(i.e., different from classmate's entries)**

**listener type and describe the event that causes the methods of the listener to be invoked**.

**Listener type**: FocusListener

From: <https://docs.oracle.com/javase/7/docs/api/java/awt/event/FocusListener.html>

public interface FocusListener

extends [EventListener](https://docs.oracle.com/javase/7/docs/api/java/util/EventListener.html)  
  
The listener interface for receiving keyboard focus events on a component. The class that is interested in processing a focus event either implements this interface (and all the methods it contains) or extends the abstract FocusAdapter class (overriding only the methods of interest). The listener object created from that class is then registered with a component using the component's addFocusListener method. When the component gains or loses the keyboard focus, the relevant method in the listener object is invoked, and the FocusEvent is passed to it.

From: [https://docs.oracle.com/javase/tutorial/uiswing/events/focuslistener.html](https://docs.oracle.com/javase/7/docs/api/java/awt/event/FocusListener.html)

A focus listener, registered on each component, reports every   
focus-gained and focus-lost event.   
For each event, the other component involved in the focus change,   
the*opposite component*, is reported.   
For example, when the focus goes from a button to a text field, a focus-lost event is fired by the button (with the text field as the opposite component) and   
then a focus-gained event is fired by the text field (with the button as the opposite component).   
Focus-lost as well as focus-gained events can be temporary.   
For example, a temporary focus-lost event occurs when the window loses the focus.   
A temporary focus-gained event occurs on popup menus.

**Describe the Event that causes the methods of the Focus listener to be invoked:**

The focus listener methods are:

1. focusGaned:

void focusGained([FocusEvent](https://docs.oracle.com/javase/7/docs/api/java/awt/event/FocusEvent.html" \o "class in java.awt.event) e)  
Invoked when a component gains the keyboard focus.

2. focusLost:

void focusLost([FocusEvent](https://docs.oracle.com/javase/7/docs/api/java/awt/event/FocusEvent.html" \o "class in java.awt.event) e)  
  
Invoked when a component loses the keyboard focus.

An example of losing & gaining focus is when you click on another window, or tab or click inside a window to a different text field, menu, etc.

From the course text-book:

You can type in a TextField only if it’s “in focus”—that is, it’s the control that the user is interacting with. When you click an interactive control, it receives the focus.

Similarly, when you press the *Tab* key, the focus transfers from the current focusable control to the next one—this occurs in the order the controls were added to the GUI.

**In addition, describe a scenario where that listener type will be used in an application.**

I found some examples of the focusListener on the [oracle website](https://docs.oracle.com/javase/7/docs/api/java/awt/event/FocusListener.html) that I downloaded and ran:

These let me test how the focus listener works.

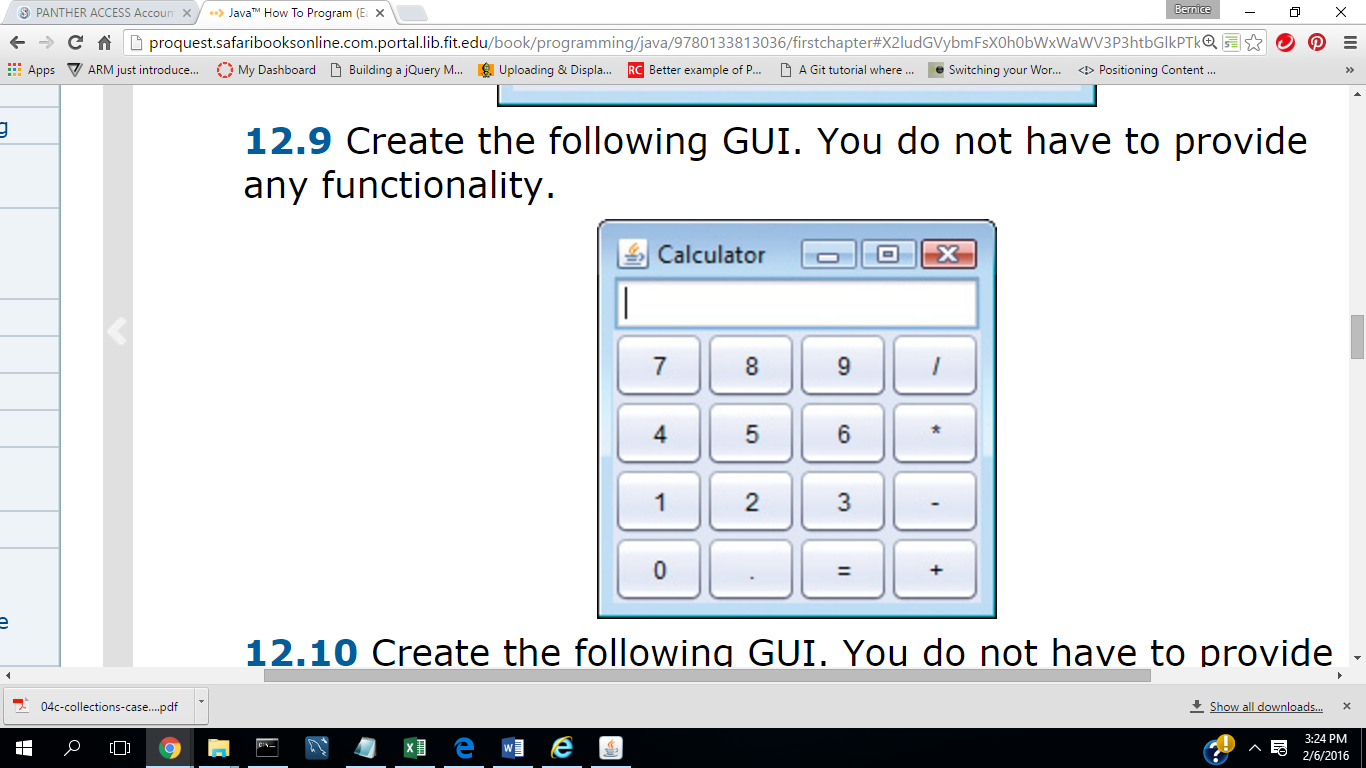
You would use a focus listener when:

In some situations an application may need to track which component has   
the focus. This information might be used to dynamically update menus or   
perhaps a status bar. If you need to track the focus only on specific   
components, it may make sense to implement a [focus event listener](https://docs.oracle.com/javase/tutorial/uiswing/events/focuslistener.html).  
  
From: [https://docs.oracle.com/javase/tutorial/uiswing/misc/focus.html](https://docs.oracle.com/javase/7/docs/api/java/awt/event/FocusListener.html)

On example of using the Focus Listener  is to display information about a picture when the picture is clicked. This information changes when different pictures are clicked.

Project 6.

**12.9** Create the following GUI. You do not have to provide any functionality.



# **Project 6 Description-2**

Provide a solution for 12.9.  The solution requires nesting panels (i.e., JPanel w/ GridLayout for buttons; JFrame w/ BorderLayout for the JPanel and JLabel).  The following describes the suggested layout managers and placement of the controls.

* Create a JPanel
* Set JPanel layout manager to GridLayout, add the buttons to this panel
* Add the JPanel to the center of the JFrame (i.e., JFrames use BorderLayout)
* Add a JLabel control to the "North" of the JFrame

Note: Do not implement event-handling for this assignment.

# **Project 5 Description**

Create an application that provIdes a solution for problem 13.11 by modifying the solution as follows.

* Provide a 5 x 5 grid instead of 8 x 8

Make all other necessary modifications necessary to run the application.  Zip all source code files and upload to the drop box.

**13.11 (Grid Using Method** ***drawLine*)** Write an application that draws an 8-by-8 grid. Use Graphicsmethod drawLine.

http://www.jamesdang.com/Pages/answerlab1.htm

1) Write a program that draws a series of eight concentric circles.  The circles should be separated by 10 pixels.  Use the drawOval method of class Graphics.

Source Program:

// Concentric.java  
// This program draws concentric circles

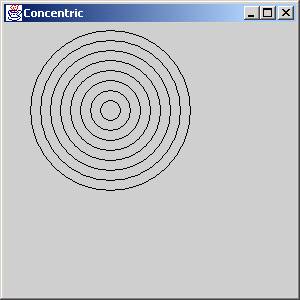
import javax.swing.\*;  
import java.awt.\*;  
import java.awt.event.\*;

public class Concentric extends JFrame   
{  
   public Concentric()  
   {  
      super( "Concentric" );  
      setSize( 300, 300 );  
      show();  
   }

   public void paint( Graphics g )  
   {  
      for ( int x = 0; x <= 160; x += 10 )   
        {  
         int y = 160 - ( x \* 2 );  
         g.drawOval( x + 30, x + 30, y, y );  
        }  
   }

   public static void main( String args[] )  
   {  
      Concentric app = new Concentric();   
      app.addWindowListener(  
         new WindowAdapter()   
         {  
            public void windowClosing( WindowEvent e )  
            {  
               System.exit( 0 );  
            }  
         }  
                                       );  
   }  
}

Output:



2) Write a program that draws lines of random lengths in random colors.

// Lines1.java  
// This program draws lines of random sizes and colors  
import javax.swing.\*;  
import java.awt.\*;  
import java.awt.event.\*;  
  
public class Lines1 extends JFrame   
{  
public Lines1()  
 {  
    super( "Drawing Lines" );  
    setSize( 200, 200 );  
    show();  
 }  
  
public void paint( Graphics g )  
 {  
for ( int y = 10; y < 200; y += 10 )   
    {  
    int x1 = ( int ) ( 1 + Math.random() \* 199 );  
  
    g.setColor( new Color( ( float ) Math.random(),   
    ( float ) Math.random(), ( float ) Math.random() ) );  
    g.drawLine( 1, y, x1, y );   
    }  
 }  
  
public static void main( String args[] )  
 {  
 Lines1 app = new Lines1();  
  
    app.addWindowListener(  
    new WindowAdapter()   
    {  
    public void windowClosing( WindowEvent e )  
        {  
        System.exit( 0 );  
        }  
    }  
                                        );   
 }  
}

3) Write a program that draws an 8-by-8 grid.  Use the drawLine method.

// Grid.java  
// This program draws an 8 x 8 grid  
import javax.swing.\*;  
import java.awt.\*;  
import java.awt.event.\*;  
  
public class Grid extends JFrame   
{  
 public Grid()  
 {  
  super( "Grid" );  
  setSize( 200, 200 );  
  show();  
 }  
  
public void paint( Graphics g )  
 {  
 int y = 30, x1 = 30;  
  
 for ( int r = 1; r <= 8; r++, y += 10 )   
 g.drawLine( 30, y, 100, y );  
  
 for ( int c = 1; c <= 8; c++, x1 += 10 )  
 g.drawLine( x1, 30, x1, 100 );  
 }  
  
public static void main( String args[] )  
 {  
 Grid app = new Grid();  
 app.addWindowListener(  
 new WindowAdapter()   
   {  
   public void windowClosing( WindowEvent e )  
      {  
      System.exit( 0 );  
      }  
   }  
                                  );   
 }  
}

4) Write a program that draws a tetrahedron ( a pyramid).  Use class GeneralPath and method draw of the class Graphics2D

// Pyramid.java  
// This program draws a tetrahedron  
import javax.swing.\*;  
import java.awt.\*;  
import java.awt.geom.\*;  
import java.awt.event.\*;  
  
public class Pyramid extends JFrame   
{   
 public Pyramid()  
 {  
    super( "Pyramid" );  
    setSize( 275, 150 );  
    show();  
 }  
  
 public void paint( Graphics g )  
 {  
   int basex[] = { 100, 200, 150, 50, 100 };  
   int basey[] = { 100, 100, 130, 130, 100 };  
   int x = 110, y = 40;  
  
   Graphics2D g2d = ( Graphics2D ) g;  
  
   GeneralPath tetra = new GeneralPath();  
  
   g2d.setColor( Color.red );  
  
   tetra.moveTo( basex[ 0 ], basey[ 0 ] );  
  
   for ( int i = 1; i < 5; i++ )   
    {  
    tetra.lineTo( x, y );  
    tetra.moveTo( basex[ i - 1 ], basey[ i - 1 ] );  
    tetra.lineTo( basex[ i ], basey[ i ] );  
    }  
  
  tetra.closePath();  
  g2d.draw( tetra );   
  }  
  
 public static void main( String args[] )  
 {  
  Pyramid app = new Pyramid();  
  
  app.addWindowListener(  
  new WindowAdapter()   
    {  
    public void windowClosing( WindowEvent e )  
        {      
        System.exit( 0 );  
        }  
    }  
                                    );  
 }

}