Event-driven programming is a programming style that uses a signal-and-response approach to programming. Swing programs use events and event handlers.

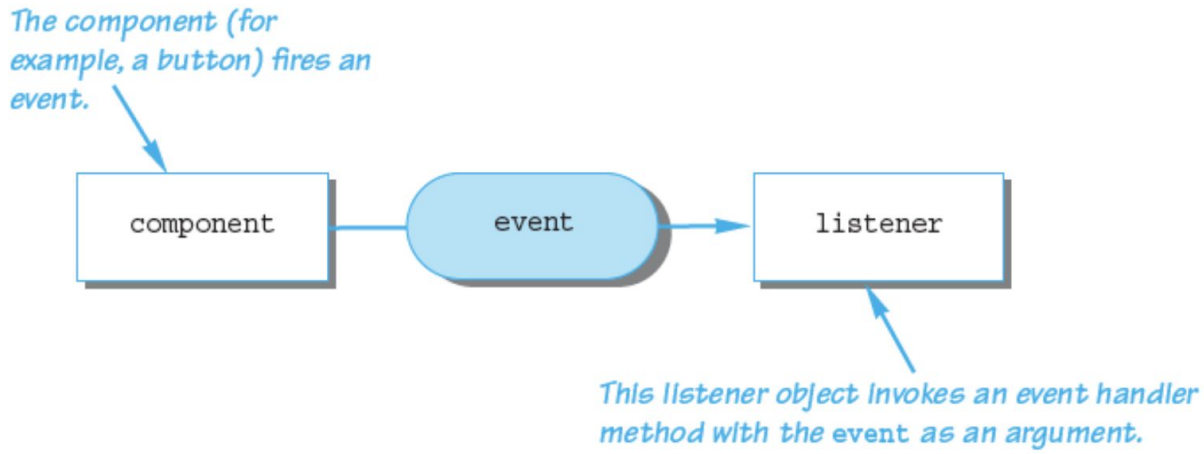
Event: Signals to objects are things called events. An event is an object that acts as a signal to another object known as a listener. The sending of the event is called firing the event. In Swing GUIs, an event often represents some action such as clicking a mouse, dragging the mouse, pressing a key on the keyboard, clicking the close-window button on a window, or any other action that is expected to elicit a response.

Listeners: The object that fires the event is often a GUI component, such as a button. The button fires the event in response to being clicked. The listener object performs some action in response to the event. For example, the listener might place a message on the screen in response to a particular button being clicked. A given component may have any number of listeners, from zero to several listeners. Each listener might respond to a different kind of event, or multiple listeners might respond to the same events.A listener object has methods that specify what will happen when events of various kinds are received by the listener. These methods that handle events are called event handlers. You the programmer will define (or redefine) these event-handler methods.

Example: An exception object is an event. The throwing of an exception is an example of firing an event (in this case, firing the exception event). The listener is the catch block that catches the event.

In event-driven programming, you create objects that can fire events, and you create listener objects to react to the events. For the most part, your program does not determine the order in which things happen. The events determine that order.

Event driven process



Event-driven programming with the Swing library makes extensive use of inheritance. The classes you define will be derived classes of some basic Swing library classes. These derived classes will inherit methods from their base class. For many of these inherited methods, library software will determine when these methods are invoked, but you will override the definition of the inherited method to determine what will happen when the method is invoked.

JFrame: An object of the class JFrame is what you think of as a window. It automatically has a border and some basic buttons for minimizing the window and similar actions. A JFrame object can have buttons and many other components added to the window and programmed for action.

Pixel: A pixel is the smallest unit of space on which your screen can write. With Swing, both the size and the position of objects on the screen are measured in pixels. The more pixels you have on a screen, the greater the screen resolution.

Some Methods in the Class JFrame

The class JFrame is in the **javax.swing package.**

- **public JFrame():** Constructor that creates an object of the class JFrame.
- **public JFrame(String title):** Constructor that creates an object of the class JFrame with the title given as the argument.

- **public void setDefaultCloseOperation(int operation):** Sets the action that will happen by default when the user clicks the close-window button. The argument should be one of the following defined constants:

    - **JFrame.DO_NOTHING_ON_CLOSE**: Do nothing. The JFrame does nothing, but if there are any registered window listeners, they are invoked. (Window listeners are explained later.)

    - **JFrame.HIDE_ON_CLOSE**: Hide the frame after invoking any registered WindowListener objects.

    - **JFrame.DISPOSE_ON_CLOSE**: Hide and dispose the frame after invoking any registered window listeners. When a window is disposed, it is eliminated but the program does not end. To end the program, use the next constant as an argument to setDefaultCloseOperation.

    - **JFrame.EXIT_ON_CLOSE**: Exit the application using the System exit method. (Do not use this for frames in applets. Applets are discussed in Chapter 20 on the website.)

If no action is specified using the method setDefaultCloseOperation, then the default action taken is JFrame.HIDE_ON_CLOSE.

- Throws an IllegalArgumentException if the argument is not one of the values listed above.
- Throws a SecurityException if the argument is JFrame.EXIT_ON_CLOSE and the Security Manager will not allow the caller to invoke System.exit.

- **public void setSize(int width, int height):** Sets the size of the calling frame so that it has the width and height specified. Pixels are the units of length used.

- **public void setTitle(String title):** Sets the title for this frame to the argument string.

- **public void add(Component componentAdded):** Adds a component to the JFrame.

- **public void setLayout(LayoutManager manager):** Sets the layout manager.(Will be done later)

- **public void setJMenuBar(JMenuBar menubar):** Sets the menu bar for the calling frame. (Will be done later)

- **public void dispose():** Eliminates the calling frame and all its subcomponents. Any memory they use is released for reuse. If there are items left (items other than the calling frame and its subcomponents), then this does not end the program.

The setVisible Method: Many classes of Swing objects have a setVisible method. The setVisible method takes one argument of type boolean. If w is an object, such as a JFrame window, that can be displayed on the screen, then the call

```
w.setVisible(true);
```

will make w visible. The call

```
w.setVisible(false);
```

will hide w.

Syntax

```
Object_For_Screen.setVisible(Boolean_Expression);
```

Example

```
public static void main(String[] args)
{
        JFrame firstWindow = new JFrame();
        .
        .
        .
        firstWindow.setVisible(true);
}
```

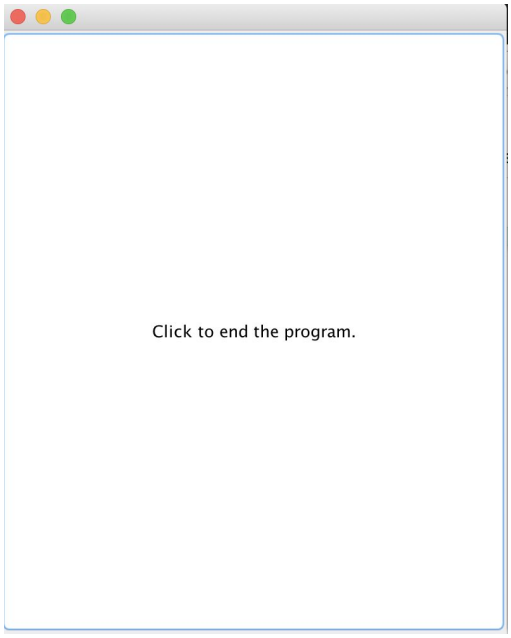
```
// Sample Code
//  This is the file FirstSwingDemo.java.
1  import javax.swing.JFrame;
2  import javax.swing.JButton;
3  public class FirstSwingDemo
4  {
5       public static final int WIDTH = 300;
6       public static final int HEIGHT = 200;

7       public static void main(String[] args)
8       {
9       JFrame firstWindow = new JFrame( ); //New JFrame
10      firstWindow.setSize(WIDTH, HEIGHT); //Setting the size of frame
11      firstWindow.setDefaultCloseOperation(
12           JFrame.DO_NOTHING_ON_CLOSE);//Nothing done when Frame closed
13      JButton endButton = new JButton("Click to end program.");//New Button
14      EndingListener buttonEar = new EndingListener(); //New Ending Listener
15      endButton.addActionListener(buttonEar); //Button registered to action
                                                //listener
16      firstWindow.add(endButton);//Button added to Frame
17      firstWindow.setVisible(true);//Visibility of frame set to true
18      }
19  }
```

//This is the file EndingListener.java.

```
1  import java.awt.event.ActionListener;
2  import java.awt.event.ActionEvent;
3  public class EndingListener implements ActionListener
4  {
5       public void actionPerformed(ActionEvent e)
6       {
7       System.exit(0);
8       }
9  }
```

//Sample Output



If you forget to program the close-window button, then the default action is HIDE_ON_CLOSE.

The JButton Class

An object of the class JButton is displayed in a GUI as a component that looks like a button. Click the button with your mouse to simulate pushing it. When creating an object of the class JButton using new, you can give a string argument to the constructor and the string will be displayed on the button.

You can add a **JButton** object to a **JFrame** by using the method **add** with the **JFrame** as the calling object and the JButton object as the argument.

A button's action is programmed by registering a listener with the button using the method **addActionListener**.

Example

```
JButton niceButton = new JButton("Click here");
niceButton.addActionListener(new SomeActionListenerClass());
someJFrame.add(niceButton);
```

Action Listeners and Action Events

Clicking a button with your mouse (or activating certain other items in a GUI) creates an object known as an event and sends the event object to another object (or objects) known as the listener(s). This is called firing the event. The listener then performs some action. When we say that the event is "sent" to the listener object, what we really mean is that some method in the listener object is invoked with the event object as the argument. This invocation happens automatically. Your Swing GUI class definition will not normally contain an invocation of this method.

Swing GUI class definition does need to do two things:

- First, for each button, it needs to specify what objects are listeners that will respond to events fired by that button; this is called registering the listener.

- Second, it must define the methods that will be invoked when the event is sent to the listener. Note that these methods will be defined by you, but in normal circumstances, you will never write an invocation of these methods. The invocations will take place automatically.

Example:

```
EndingListener buttonEar = new EndingListener( );
endButton.addActionListener(buttonEar);
```

The second line says that buttonEar is registered as a listener to endButton, which means buttonEar will receive all events fired by endButton.

Action Listener: Different kinds of components require different kinds of listener classes to handle the events they fire. A button fires events known as action events, which are handled by listeners known as action listeners.

An action listener is an object whose class implements the ActionListener interface.

For example, the class **EndingListener** in above program implements the **ActionListener interface**. The ActionListener interface has only one method heading that must be implemented, namely the following:

```
public void actionPerformed(ActionEvent e)

//Definition of actionPerformed() from above code

public void actionPerformed(ActionEvent e)
{
        System.exit(0);
}
```

If the user clicks the button **endButton**, it sends an action event to the action listener for that button. Since **buttonEar** is the **action listener** for the button **endButton**, so the action event goes to **buttonEar**. When an action listener receives an action event, the event is automatically passed as an argument to the method actionPerformed and the method actionPerformed is invoked. If the event is called **e**, then the following invocation takes place in response to **endButton firing e:**

```
buttonEar.actionPerformed(e);
```

In this case, the parameter **e** is ignored by the method **actionPerformed**. The method actionPerformed simply invokes **System.exit** and thereby ends the program. So, if the user clicks **endButton** (the one labeled "Click to end program."), the net effect is to end the program and so the window goes away.

Note that you never write any code that says:

```
buttonEar.actionPerformed(e);
```

This action does happen, but the code for this is embedded in some class definition inside the Swing and/or AWT libraries. Somewhere the code says something like

```
bla.actionPerformed(e);
```

and somehow **buttonEar** gets plugged in for the parameter **bla** and this invocation of **actionPerformed** is executed. But, all this is done for you. All you do is define the method **actionPerformed** and register **buttonEar** as a listener for **endButton**.

Note that the method actionPerformed must have a parameter of type ActionEvent, even if your definition of actionPerformed does not use this parameter. This is because the invocations of actionPerformed were already

programmed for you and so must allow the possibility of using the ActionEvent parameter e. As you will see, in other Swing GUIs the method actionPerformed does often use the event e to determine which button was clicked.

Ending a Swing Program: A GUI program is normally based on a kind of infinite loop. There may not be a Java loop statement in a GUI program, but nonetheless the GUI program need not ever end. The windowing system normally stays on the screen until the user indicates that it should go away (for example, by clicking the "Click to end program." button in above program). If the user never asks the windowing system to go away, it will never go away. When you write a Swing GUI program, you need to use **System.exit** to end the program when the user (or something else) says it is time to do so. Unlike other programs, a Swing program will not end after it has executed all the code in the program. A Swing program does not end until it executes a **System.exit.**

//Updated Code

```
//FirstWindow.java
import javax.swing.JFrame;
import javax.swing.JButton;

public class FirstWindow extends JFrame
{
   public static final int WIDTH=400;
   public static final int HEIGHT=500;

   public FirstWindow()
   {
         super(); //JFrame constructor called
         setSize(WIDTH,HEIGHT);
         setTitle("My First Window");
         setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
         JButton ButtonEnd=new JButton("Click to end the program");
        //use an anonymous object of EndingListener
         ButtonEnd.addActionListener(new EndingListener());
         add(ButtonEnd);

   }
}
```

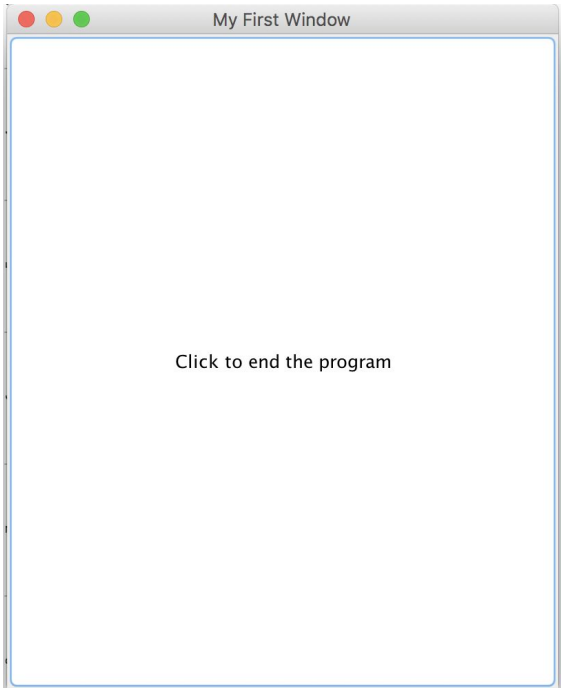
```
//DemoFirstWindow.java
public class DemoFirstWindow
{

   public static void main(String[] args)
   {
         FirstWindow w=new FirstWindow();

         w.setVisible(true);

   }
}
```

//Sample Output



FirstWindow is a derived class of the class **JFrame**. This is the normal way to define a windowing interface. The base class **JFrame gives some basic window facilities, and then the derived class adds whatever additional features you want in your window interface.**

Note that almost all the initializing for the window **FirstWindow** placed in the constructor for the class. That is as it should be. The initialization, such as setting the

initial window size, should be part of the class definition and not actions performed by objects of the class **. All the initializing methods, such as setSize and setDefaultCloseOperation, are inherited from the class JFrame.** Because they are invoked in the constructor for the window, the window itself is the calling object.

In other words, a method invocation such as

```
setSize(WIDTH, HEIGHT);
```

is equivalent to

```
this.setSize(WIDTH, HEIGHT);
```

The JLabel Class: If you want to add some text to your **JFrame**, use a label instead of a button. A label is an object of the class **JLabel**. A label is little more than a line of text.

Example (Inside a Constructor for a Derived Class of JFrame)

```
JLabel myLabel = new JLabel("Hi Mom!");
add(myLabel);
```

Color: You can set the color of a JFrame (or other GUI object). To set the background color of a JFrame, use

```
getContentPane().setBackground(Color);
```

For example, the following will set the color of the JFrame named someFrame to blue:

```
someFrame.getContentPane().setBackground(Color.BLUE);
```

Like everything else in Java, a color is an object—in this case, an object that is an instance of the class Color. **The class Color is in the java.awt package.**

The Color Constants

Color.BLACK
Color.BLUE
Color.CYAN

Color.DARK_GRAY
Color.GRAY
Color.GREEN
Color.LIGHT_GRAY
Color.MAGENTA
Color.ORANGE
Color.PINK
Color.RED
Color.WHITE
Color.YELLOW

//Sample Code 3

```
1  import javax.swing.JFrame;
2  import javax.swing.JLabel;
3  import java.awt.Color;

4  public class ColoredWindow extends JFrame
5  {
6      public static final int WIDTH = 300;
7      public static final int HEIGHT = 200;

8      public ColoredWindow(Color theColor)
9      {

10         super("No Charge for Color");
11         setSize(WIDTH, HEIGHT);
12         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

13         getContentPane().setBackground(theColor);

14         JLabel aLabel = new JLabel("Close-window button works.");
15         add(aLabel);
16     }
17     public ColoredWindow()
18     {
19         this(Color.BLUE);
20     }
21 }
```

This is an invocation of the other constructor.

This is the file **ColoredWindow.java**.

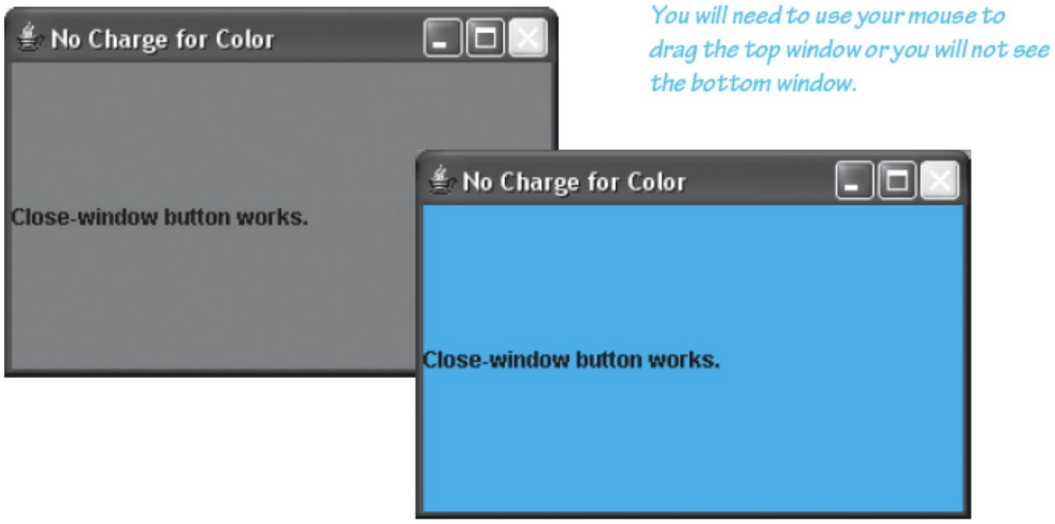
This is the file **ColoredWindow.java**.

```
1  import java.awt.Color;

2  public class DemoColoredWindow
3  {
4      public static void main(String[] args)
5      {
6          ColoredWindow w1 = new ColoredWindow();
7          w1.setVisible(true);

8          ColoredWindow w2 = new ColoredWindow(Color.GRAY);
9          w2.setVisible(true);
10     }
11 }
```

RESULTING GUI



You will need to use your mouse to drag the top window or you will not see the bottom window.

Check your understanding:

1. What Swing class do you normally use to define a window? Any window class that you define would normally be an object of this class.

2. What units of measure are used in the call to setSize() in the following code:

firstWindow.setSize(WIDTH, HEIGHT);

which is equivalent to

firstWindow.setSize(300, 200);

3. What is the method call to set the close-window button of the JFrame someWindow so that nothing happens when the user clicks the close-window button in someWindow?

4. What is the method call to set the close-window button of the JFrame someWindow so that the program ends when the user clicks the close-window button in someWindow?

5. What happens when you click the minimizing button of the JFrame in the above code?

6.. What kind of event is fired when you click a JButton?

7. What method heading must be implemented in a class that implements the ActionListener interface?

8. Change the program1 so that the window displayed has the title "My First Window".

9. Change the program2 so that the title of the JFrame is not set by the method setTitle but is instead set by the call to the base class constructor.

10. Change the program2 so that there are two ways to end the GUI program: The program can be ended by either clicking the "Click to end program." button or clicking the close-window button.

12. How would you modify the class definition in program 3 so that the window produced by the no-argument constructor is magenta instead of pink?

13. Rewrite the following two lines from program 3 so that the label does not have the name aLabel or any other name. Hint: Use an anonymous object.

```
JLabel aLabel = new JLabel("Close-window button works.");
add(aLabel);
```