

# The Java Learning Kit: Chapter 10

## Event-Driven Software

**Lesson 10.1 – Event Driven Software**

**Lesson 10.2 – Using the ActionListener Interface**

**Lab 10 – Programming Exercise: Counting Clicks**

**Lesson 10.3 – Using ActionListener for Multiple Components**

**Lesson 10.4 – Java Events for Text Boxes**

**Lesson 10.5 – Java Events for Check Boxes**

**Lesson 10.6 – Java Events for Radio Buttons**

**Lesson 10.7 – Java Events for Slider controls**

**Lesson 10.8 – Other Java Events**

*The Java Learning Kit: Chapter 10 – Event-Driven Software*

Copyright 2015 by C. Herbert, all rights reserved.

Last edited January, 2015 by C. Herbert

This document is a chapter from a draft of the Java Learning Kit, written by Charles Herbert, with editorial input from Craig Nelson, Christopher Quinones, Matthew Staley, and Daphne Herbert. It is available free of charge for students in Computer Science courses at Community College of Philadelphia during the Spring 2015 semester.

This material is protected by United States and international copyright law and may not be reproduced, distributed, transmitted, displayed, published or broadcast without the prior written permission of the copyright holder. You may not alter or remove any trademark, copyright or other notice from copies of the material.

# Introduction to Computer Science with Java

## Chapter 10 – Event-Driven Software



---

*This chapter briefly reviews the concept of an event, then describes how to activate Swing components in a GUI using event-driven programming.*

---

### Chapter Learning Outcomes

Upon completion of this chapter students should be able to:

- Describe the concept of an event in computer software, including the concepts of event triggers, event listeners, and event handlers.
- Describe how the Java *ActionListener* interface can be used to connect event listeners and event handlers to Swing GUI components.
- Create working Graphical User Interfaces in Java, using Swing components, including:
  - Buttons
  - Text boxes
  - Check boxes
  - Radio buttons
  - Slider controls.
- Implement a Java application as a working GUI.

---

## 10.1 Event Driven Software

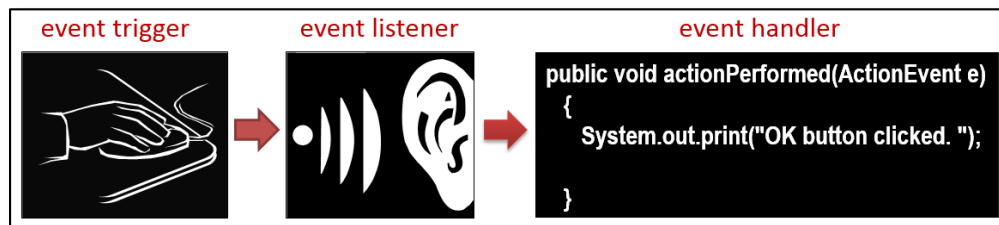
As we have seen in Chapter 7, a **graphical user interface (GUI)** is an event-driven combination of hardware and software that allows a person to interact with a computer. We have already seen how to place Swing components (labels, buttons, text boxes, and so on) in AWT containers (windows, frames, and panels) creating the visual portion of a GUI; now we will see how to activate them by implementing Java's *ActionListener* interface. First, we will review the concept of event-driven software.

**Event-driven software** listens for and responds to system events, such as someone clicking the screen pointer on a button, or pressing a certain key on a keyboard. In Java programming, an **event** is an object, which is generated by specific changes in a system that the system's software has been programmed to detect. The term *event* is also generally used to refer to the change that occurs, although in terms of programming, the change is more properly called an *event trigger*.

Event-driven software involves an event trigger, an event listener, and an event handler.

An **event trigger** is any change in a system that causes an event listener to generate an event. An **event listener** is a combination of software and hardware that senses when a specific condition, called an

event trigger, occurs. When an event listener detects the occurrence of an event trigger, it runs an **event handler**, which is a program that tells the computer what to do when the event trigger occurs.



A graphical user interface has a coordinated combination of software to display GUI components and event-driven software to activate the components. For example, a GUI could have a button, with an event trigger and event handler set to print the message *"OK button clicked"* whenever someone uses the mouse to click the button on the screen.

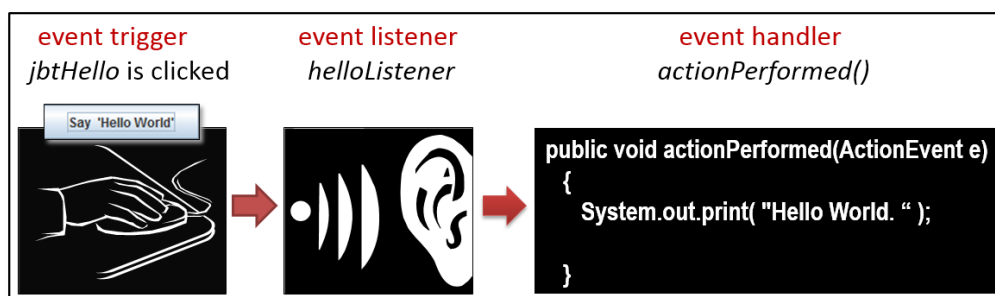
## 10.2 Using the ActionListener Interface

Java's *ActionListener* interface can be implemented to create event listeners and event handlers that can be attached to Swing components. Several that must be included in Java code for this to work:

- The component that will be used for the event trigger needs to be established in a container. For example, a *JButton* could be placed in a *JPanel* or a *JFrame*, as discussed chapter 7.
- A class must implement the *ActionListener* interface for the component's listener.
- Within the class that implements the *ActionListener* interface, the method *actionPerformed()* must be defined. This method is the event handler.
- An instance of the class that extends the *ActionListener* interface must be added to the container that holds the component. This instance must be registered to the component that will be the event trigger.

The diagrams below show an example of how this works. They describe software with a *JButton* named *jbtHello* registered to an event listener named *helloListener*. *helloListener* is an instance of the programmer-defined class *jbtListener*, which implements the *ActionListener* interface. The method *actionPerformed()* within the *helloListener* class is the event handler. When *helloListener* detects that *jbtHello* has been clicked, the code in the method *actionPerformed()* is implemented.

### Example of a Button Event



In this example, clicking the button causes the message *"Hello World"* to be displayed as console output.

## Tasks to be Completed

These tasks would be needed to build the software:

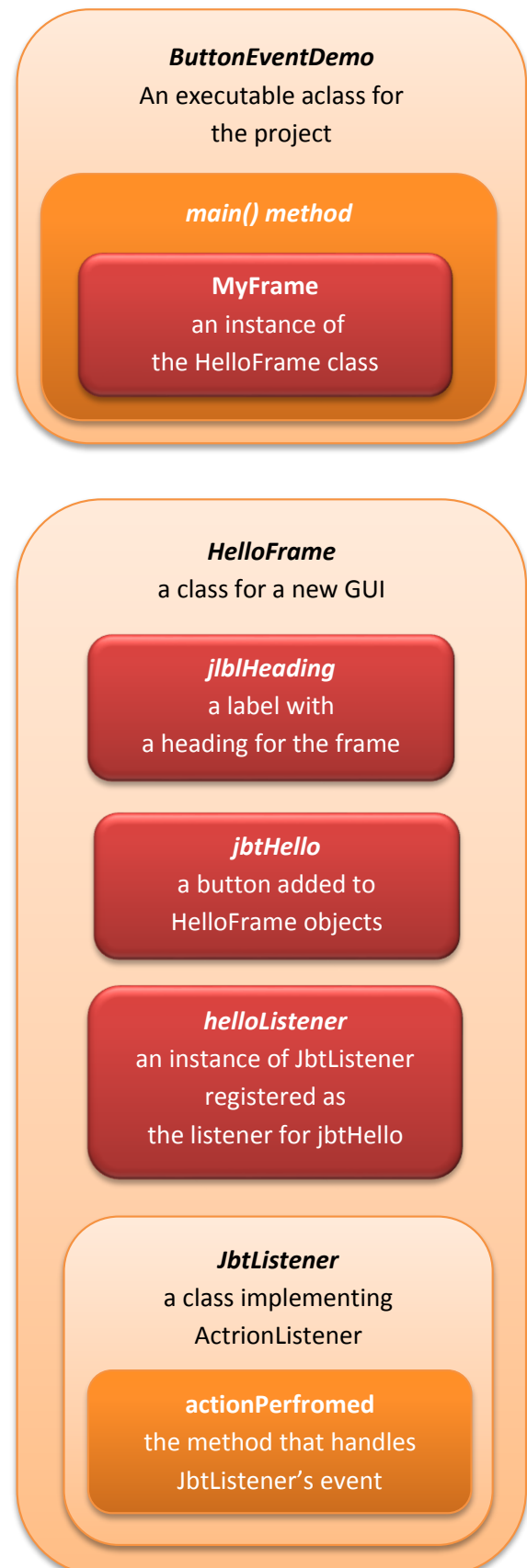
1. Define an executable class for the project named **ButtonEventDemo** with a **main()** method.
2. Define a class for a new GUI named **HelloFrame**.
3. Define the button **jbtHello** and add the button to **HelloFrame**
4. Create the class **JbtListener**, which implements **ActionListener**.
5. Add **helloListener**, an instance of **JbtListener**, to **HelloFrame** and register **helloListener** as the listener for the button **jbtHello**
6. Define the method **actionPerformed** to serve as an event handler in **helloListener**
7. Define in the **main()** method an instance of the **HelloFrame** class named **MyFrame**

As you can see, to activate the button **jbtHello** as an event-driven component, the software has three classes:

- **(ButtonEventDemo)** – a public class with the project's executable **main()** method in which **myFrame**, an instance of **HelloFrame**, will be created.
- **(HelloFrame)** – the new subclass of **JFrame** that will contain the button **jbtHello** and **helloListner**, an instance of **JbtListener** registered as the listener for **jbtHello**.
- **(JbtListener)** – a class implementing a listener for the button **jbtHello**, with the method **actionPerformed** that will be the event listener.

**JbtListener** could be defined as an inner class of the **HelloFrame** class, since it will only be used by the **HelloFrame** class. It is an inner class in the code for this example, which starts on the next page.

## Structure of the Software



```

package buttoneventdemo;

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

public class ButtonEventDemo
{
    // main method for the ButtonEventDemo class
    public static void main(String[] args)
    {
        // create an instance of HelloFrame
        HelloFrame myFrame = new HelloFrame();

        myFrame.setTitle("Button Event Demo");
        myFrame.setSize(256, 100);
        myFrame.setLocation(200, 100);
        myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        myFrame.setVisible(true);
    } // end main()
} // end class ButtonEventDemo
//*****

// create a subclass of JFrame named HelloFrame
class HelloFrame extends JFrame
{
    // a constructor for HelloFrame objects that will define the object in detail
    public HelloFrame()
    {
        // give the HelloFrame a FlowLayout
        setLayout(new FlowLayout(FlowLayout.CENTER));

        // Create a label for the demo
        JLabel jlblHeading = new JLabel("Button Event Demo");

        // Create the Hello button for the demo
        JButton jbtHello = new JButton("Say 'Hello World'");
        jbtHello.setPreferredSize(new Dimension(144, 32));

        // add components to HelloFrame objects
        add(jlblHeading);
        add(jbtHello);

        // create an instance of the JbtListener class (implements ActionListener)
        JbtListener helloListener = new JbtListener();

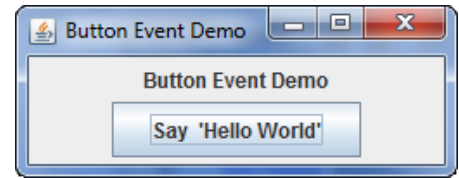
        // Register the listener for the jbtHello button
        jbtHello.addActionListener(helloListener);
    } // end HelloFrame detailed constructor
    //*****

    // create a listener for the JbtHello button as an inner class of HelloFrame
    class JbtListener implements ActionListener
    {
        // create an event handler for the Hello Button
        // ActionEvent e will be the event generated by the trigger automatically
        public void actionPerformed(ActionEvent e)
        {
            System.out.println("Hello World.");
        } // end actionPerformed(ActionEvent e)
    } // end inner class JbtListener
    //*****
} // end class HelloFrame

```

Let's see how the code from the example above actually works. The code has three classes:

- The public class *ButtonEventDemo* with the executable main method. The GUI *myFrame*, shown here, is instantiated in the main method as an instance of *HelloFrame*.
- The *HelloFrame* class which creates a class of *JFrame* graphical user interfaces.
- The *JbtListener* class, which creates listeners for components in instances of *HelloFrame*. It is an inner class within *HelloFrame*.



```
public class ButtonEventDemo
{
    public static void main(String[] args)
    {
        HelloFrame myFrame = new HelloFrame();

        myFrame.setTitle("Button Event Demo");
        myFrame.setSize(256, 100);
        myFrame.setLocation(200, 100);
        myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        myFrame.setVisible(true);
    }
}
```

*myFrame* is an instance of the *HelloFrame* class.  
It's components will be declared in the *HelloFrame* class.

The properties needed to make *myFrame* appear are declared.

```
class HelloFrame extends JFrame
{
    public HelloFrame()
    {
        setLayout(new FlowLayout(FlowLayout.CENTER));

        JLabel jlblHeading = new JLabel("Button Event Demo");

        JButton jbtHello = new JButton("Say 'Hello World'");
        jbtHello.setPreferredSize(new Dimension(144, 32));

        add(jlblHeading);
        add(jbtHello);

        JbtListener helloListener = new JbtListener();

        jbtHello.addActionListener(helloListener);

        . . .
    }
}
```

*HelloFrame* is a subclass of *JFrame*.  
It is a specialized *JFrame*.

A label and button are added to *JFrame*.

*helloListener*, an instance of the *JbtListener* class, is created in *HelloFrame*.

*helloListener* is registered to *jbtHello* by adding it as the button's *ActionListener*.  
This makes *jbtHello* an event trigger.

```
class JbtListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        System.out.println("Hello World.");
    }
} // end class JbtListener
```

*JbtListener* is declared as a class of event listeners – specifically *ActionListeners*.

*actionPerformed()* is the event handler for any instance of *JbtListener*.

## Lab 10A – Programming Exercise: Counting Clicks

In this exercise we will walk through creating software similar to the *ButtonEventDemo* in the previous section. Our software will count the number of times an OK button is clicked and display the value.

The event handler contains the code to be executed when an event occurs, so the only significant difference between this version of the program and *ButtonEventDemo* is in the event handler. We will add a counter to the event handler, then increment the counter when the button is clicked. We will change the printed message to tell the user the number of times the button has been clicked.

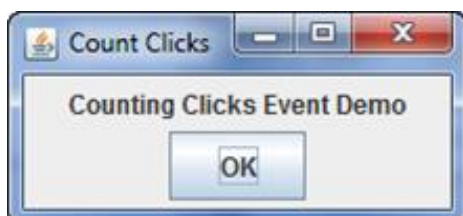
Our software will have three classes, just as in *ButtonEventDemo*:

- CountClicks – with an executable main() method.
- CountFrame – a class defining the GUI
- JbtListener – an implementation of ActionListener  
This will be an inner class of CountFrame, since it is only used by CountFrame.

Here is an outline of what we need to do:

1. Declare a public class for the project named **CountClicks** with a **main()** method.
2. Define a class for a new GUI container named **CountFrame**.
3. Define a label and the button **jbtOK** and add them to **CountFrame**.
4. Create the class **JbtListener**, which implements **ActionListener**.
5. Define the method **actionPerformed** to serve as an event handler in **JbtListener**.
6. Add **CountListener**, an instance of **JbtListener**, to **CountFrame** and register **CountListener** as the listener for the **jbtOK** button.
7. Define in the **main()** method an instance of the **CountFrame** class named **MyFrame**.

The finished GUI looks like this:



### Class CountClicks

#### main() method

**MyFrame**  
an instance of  
the CountFrame class

### Class CountFrame

**jlblHeading**  
a label for the frame

**jbtOK**  
the OK button

**CountListener**  
instance of JbtListener  
registered to jbtOK

**Class JbtListener**  
implements ActionListener

**actionPerformed**  
the method that handles  
a JbtListener event



## Creating *CountClicks* as a NetBeans Project

### STEP 1.

Download and unzip the file ***CountClicks.zip***. It contains a NetBeans project folder with some initial code and documentation for this project.

### STEP 2.

Start NetBeans. Find and open the NetBeans project ***CountClicks***. The code should look like this:

```
/*
 * CountClicks.java
 * CSCI 111 Fall 2013
 * last edited [date] by [programmer]
 * This program has a GUI with an OK button.
 * An event handler counts the number of times the OK button has been clicked,
 * and displays the result as console output.
 */

package countclicks;

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

public class CountClicks
{
    public static void main(String[] args)
    {
        // create an instance of CountFrame

    } // end main()
} // end class CountClicks
//*****

// create a subclass of JFrame named CountFrame

    // a constructor for CountFrame objects that will define the object in detail
    // with a listener and an event handler
```

A shell for the public class and its main method are in place, but details are missing. Comments below the line of asterisks indicate that a subclass of *JFrame* named *CountFrame* with a constructor defining the frame, and an inner class defining a listener, will be needed.

The required import statements are included. Notice that it is necessary to import *java.awt.event.\** to implement the *ActionListener* interface.

### STEP 3.

Modify the documentation at the top of the source code by filling in today's date and your own name in place of [DATE] and [PROGRAMMER], similar to the following:

```
* last edited November 2, 2013 by C. Herbert
```

Our task list for the project, on page 6, has seven items to be completed. The first of these is already done: Declare a public class for the project named ***CountClicks*** with a ***main()*** method.



Our next task from the task list on page 6 is to define a class for a new GUI container named **CountFrame**. It will be a specialized JFrame.

**STEP 4.**

Below the comment `// create a subclass of JFrame named CountFrame` insert the code to declare the class:

```
class CountFrame extends JFrame
{

} // end class CountFrame
```

It should look like this:

```
// create a subclass of JFrame named CountFrame
class CountFrame extends JFrame
{

} // end class CountFrame
```

Later we will create an instance of this class in the main method. Now we need a constructor for the class that has detailed instructions for creating a *CountFrame* object.

**STEP 5.**

Between the braces for the class, add the following lines of code for the constructor:

```
// a constructor for CountFrame objects that will define the object in detail
public CountFrame()
{

} // end CountFrame()
```

After step 5, your code should look like this:

```
// create a subclass of JFrame named CountFrame
class CountFrame extends JFrame
{
    // a constructor for CountFrame objects that will define the object in detail
    public CountFrame()
    {

    } // end CountFrame()
} // end class CountFrame
```

Our constructor will give `CountFrame` a `FlowLayout`, and from the picture of the interface on the bottom of page 6, we can see it needs a label and a button. This is the third item in our task list on page 6. First, we will add comments describing these things, and then we will add the instructions to do so.

**STEP 6.**

Place the following comments in the `CountFrame()` constructor (between the braces), separated by blank lines:

```
// give the CountFrame a centered FlowLayout
```

```
// Create a label for the demo
```

```
// Create the OKbutton for the demo
```

```
// create a subclass of JFrame named CountFrame
class CountFrame extends JFrame
{
    // a constructor for CountFrame objects that will define the object in detail
    public CountFrame()
    {
        // give the CountFrame a centered FlowLayout

        // Create a label for the demo

        // Create the OKbutton for the demo

    } // end CountFrame()
} // end class CountFrame
```

**STEP 7.**

Add the code to establish a `FlowLayout` following the comment `// give the CountFrame a FlowLayout`:

```
setLayout(new FlowLayout(FlowLayout.CENTER));
```

The code centers components in the frame.

**STEP 8.**

Add the code to Create a label, following the second comment `// Create a label for the demo`:

```
JLabel jlblNote = new JLabel("Counting Clicks Event Demo");
```

**STEP 9.**

Add the code to create the OK button for the demo, following the comment `// Create the OK button for the demo` and set the button's size to 64 x 32:

```
JButton jbtOK = new JButton("OK");
jbtOK.setPreferredSize(new Dimension(64, 32));
```

```

public CountFrame()
{
    // give the CountFrame a centered FlowLayout
    setLayout(new FlowLayout(FlowLayout.CENTER));

    // Create a label for the demo
    JLabel jlblNote = new JLabel("Counting Clicks Event Demo");

    // Create the OK button for the demo
    JButton jbtOK = new JButton("OK");
    jbtOK.setPreferredSize(new Dimension(64, 32));

} // end CountFrame()

```

Note that we can set the size of the button because we used a FlowLayout. GridLayout ignores setting the size for labels and buttons. Now we need to insert code to add the label and the button to CountFrame.

#### STEP 10.

Add the following comment and instructions to the method following the last three lines:

```

// add components to CountFrame objects
add(jlblNote);
add(jbtOK);

```

After step 10, the code for the method should look like this:

```

class CountFrame extends JFrame
{
    // a constructor for CountFrame objects that will define the object in detail
    public CountFrame()
    {
        // give the CountFrame a centered FlowLayout
        setLayout(new FlowLayout(FlowLayout.CENTER));


        // Create a label for the demo
        JLabel jlblNote = new JLabel("Counting Clicks Event Demo");

        // Create the OK button for the demo
        JButton jbtOK = new JButton("OK");
        jbtOK.setPreferredSize(new Dimension(64, 32));

        // add components to CountFrame objects
        add(jlblNote);
        add(jbtOK);

    } // end CountFrame()
} // end class CountFrame

```



The next item in our task list is to create the class **JbtListener**, which implements *ActionListener*. *JbtListener* will be an inner class of the *CountFrame* class. This means we need to add code for the class between the end of the *CountFrame()* constructor and the end of the *CountFrame* class. The arrow in the image of the code above shows where the inner class needs to be inserted.

**STEP 11.**

Add the following code to establish *JbtListener* as an inner class between the end of the *countFrame()* constructor and the end of the *CountFrame* class, indicated in the image above:

```
// create a listener for the JbtOK button as an inner class of CountFrame
class JbtListener implements ActionListener
{

} // end inner class JbtListener
//*****
```

When it is in place, the code should look like this:

```
} // end CountFrame()
//*****

// create a listener for the jbtOK button as an inner class of CountFrame
class JbtListener implements ActionListener
{
} // end inner class JbtListener
//*****
} // end class CountFrame
```

Notice that an error is indicated for the class name, *JbtListener*. This is because *JbtListener* implements *ActionListener*, which requires the method *actionPerformed()* to be implemented in the class. The error message is because we have not done this yet. When we add *actionPerformed()* to the class, the error will go away. This is what we will do next.

**STEP 12.**

Add the following code to put the *actionPerformed()* method in the *JbtListener()* class, between the braces for the beginning and end of the class:

```
// create an event handler for the OK Button
public void actionPerformed(ActionEvent e)
{

} // end actionPerformed(ActionEvent e)
```

After step 12, the code for the *JbtListener* class should look like this:

```
// create a listener for the JbtCount button as an inner class of CountFrame
class JbtListener implements ActionListener
{
    // create an event handler for the OK Button
    public void actionPerformed(ActionEvent e)
    {

    } // end actionPerformed(ActionEvent e)
} // end inner class CountListener
//*****
```

The listener should count the number of times the Ok button is clicked, and display a console message indicating the count, so we need to add a counter to the class and code to the method to increment the counter and display the message. The event handler method *actionPerformed()* will be run each time the OK button is clicked. It is the event handler.

**STEP 13.**

Add the following code to declare and initialize a counter immediately after the brace at the beginning of the *JbtListener* class:

```
private int count = 0;
```

**STEP 14.**

Add the following code to increment the counter and then display the console message inside the *ActionPerformed()* method:

```
count++;
System.out.println("The OK button was pressed "+ count + " times.");
```

The code for the class *JbtListener* should now look like this:


```
// create a listener for the JbtCount button as an inner class of CountFrame
class JbtListener implements ActionListener
{
    private int count = 0;

    // create an event handler for the OK Button
    public void actionPerformed(ActionEvent e)
    {
        count++;
        System.out.println("The OK button was pressed "+ count + " times.");
    } // end actionPerformed(ActionEvent e)
} // end inner class CountListener
//*****
```

The *JbtListener* class is now complete. The next item from the task list on page 6 is to add *CountListener*, an instance of *JbtListener*, to the *CountFrame()* constructor and register *CountListener* as the listener for the *jbtOK* button in *CountFrame*.

The last part of the *CountFrame()* constructor currently looks like this:

```
// add components to CountFrame objects
add(jlblNote);
add(jbtOK);
} // end CountFrame()
```



We will add the code to instantiate the listener and then to register the listener between the *add(jbtOK);* instruction and the end of the *CountFrame()* constructor, as indicated above.

**STEP 15.**

Add the following comments and instructions just before the end of the *CountFrame()* constructor:

```
// create an instance of the JbtListener class (implements ActionListener)
JbtListener CountListener = new JbtListener();

// Register the listener for the Count button
jbtOK.addActionListener(CountListener);
```

After step 15, the complete *CountFrame()* constructor should look like this:

```
public CountFrame()
{
    // give the CountFrame a centered FlowLayout
    setLayout(new FlowLayout(FlowLayout.CENTER));

    // Create a label for the demo
    JLabel jlblNote = new JLabel("Counting Clicks Event Demo");

    // Create the OK button for the demo
    JButton jbtOK = new JButton("OK");
    jbtOK.setPreferredSize(new Dimension(64, 32));

    // add components to CountFrame objects
    add(jlblNote);
    add(jbtOK);

    // create an instance of the CountListener class (implements ActionListener)
    JbtListener countListener = new JbtListener();

    // Register the listener for the Count button
    jbtOK.addActionListener(countListener);

} // end CountFrame()
```

Only the last task from the task list back on page 6 remains -- define in the *main()* method an instance of the *CountFrame* class named **MyFrame**.

Currently, the only line in the *main()* method is the comment:

```
// create an instance of CountFrame
```

We will add the code below the comment to do what the comment says, and to define the frame's title, size, and so on.

**STEP 16.**

Add the following code below the comment in the *main()* method of the *CountClicks* class:

```
CountFrame myFrame = new CountFrame();

myFrame.setTitle("Count Clicks");
myFrame.setSize(216, 100);
myFrame.setLocation(200, 100);
myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
myFrame.setVisible(true);
```

The first line creates an instance of *CountFrame* for this application named *myFrame*. The other lines set the title, size, location, and exit behavior, and make *myFrame* visible.

The software should now be complete and working. The entire code for your NetBeans project should look like the source code in the file *CountClickFinished*, included with the files for the chapter. The code is also included below.

**STEP 17.**

Test the code, fix any errors, and then run the software. Each time you click the OK button, it should print a message on the console, as shown here:



---

## 10.3 Using *ActionListeners* for Multiple Components

In general, separate implementations of *ActionListener* should be created for each component that can trigger an event, but a single implementation of *ActionListener* could be used as the event handler for several different components.

The key to understanding how events work for multiple components is to understand that the method *actionPerformed(ActionEvent e)* will always be the event handler for events in which *ActionListeners* are registered to components. Each class can only have one *actionPerformed(ActionEvent e)* method. So, if we want to have separate handlers for different components, then we need to declare separate classes that implement *ActionListeners* for each of the components.

On the other hand, if we wish to have one event listener and handler that is registered as the listener for multiple components, then we only need one class that implements *ActionListener* and has the required *actionPerformed(ActionEvent e)* method. In fact, the class that contains the components can be the listener itself, if it implements *ActionListener* and has the method *actionPerformed(ActionEvent e)*.

In the case of a single listener for multiple components, we can take advantage of the parameter *ActionEvent e* to identify which component triggered the event. *ActionEvent e* is the event that was generated. The *ActionEvent* class has a method, *getSource()*, that will identify the source of the event. Code such as the following can be used to create a single event handler that can respond to different components that are the source of the event:



```
// one event handler for multiple components
public void actionPerformed(ActionEvent e)
{
    // react to specific buttons

    if (e.getSource() == jbtLeft)
        System.out.println("Left button was pressed.");

    if (e.getSource() == jbtRight)
        System.out.println("Right button was pressed.");

} // end actionPerformed(ActionEvent e)
```

In this section we look at both ways to do this. First we will see two buttons with two different event listeners, each in a separate inner class. Then we will see two buttons with one event listener for both buttons in the same class – the class that creates the frame itself.

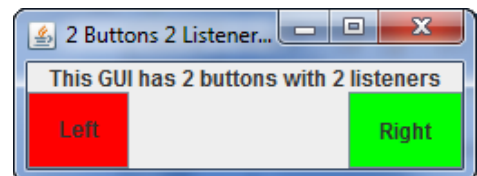
In fact, if we wish to have a single handler for separate events, then it is usually included in the same class that creates the frame, because it can be difficult to implement code such as the following if the code is in one class and the component is in another class:

```
if (e.getSource() == jbtLeft)
```

In summary, if we have more than one event listener, then those listeners should be in separate classes that each implement *ActionListener*. They are often inner classes of the class creating the GUI frame. If we wish to have a single event listener, then the same class that creates the frame can implement *ActionListener* and it must include the method *actionPerformed(ActionEvent e)*.

## Two Buttons, Two Listeners

First we will look at an example in which each component has its own listener. The file *TwoButtonsTwoListeners* included with the files for this chapter has an example of a GUI with two buttons, each with its own listener. The GUI is shown on the right.



The left button is named *jbtLeft* and the right button is named *jbtRight*. The two listeners are named *LeftListener* and *RightListener*. The code for the GUI, shown below, registers *LeftListener* for the left button and *RightListener* for the right button. Each listener is a separate *ActionListener* implementation.

When the left button is pressed, the message "Left button was pressed." appears on the console.

When the right button is pressed, the message "Right button was pressed." appears on the console.

```

class TwoButtonFrame extends JFrame
{
    // add components to the frame
    JLabel jlblNote = new JLabel("This GUI has 2 buttons with 2 listeners");
    JButton jbtLeft  = new JButton("Left");
    JButton jbtRight = new JButton("Right");

    // a detailed constructor for TwoButtonFrame objects
    public TwoButtonFrame()
    {
        // give the TwoButtonFrame a BorderLayout
        setLayout( new BorderLayout() );

        // put jlblNote in the NORTH zone, centered
        jlblNote.setHorizontalAlignment(SwingConstants.CENTER);
        add(jlblNote, BorderLayout.NORTH);

        // put jbtLeft in the WEST zone
        jbtLeft.setBackground(Color.red);
        add(jbtLeft, BorderLayout.WEST);

        // put jbtRight in the EAST zone
        jbtRight.setBackground(Color.green);
        add(jbtRight, BorderLayout.EAST);

        // create and register an instance of the LeftListener class
        LeftListener leftListener = new LeftListener();
        jbtLeft.addActionListener(leftListener);

        // create and register an instance of the RightListener class
        RightListener rightListener = new RightListener();
        jbtRight.addActionListener(rightListener);
    } // end TwoButtonFrame() constructor
    //*****

    // create a listener for the JbtLeft button as an inner class
    class LeftListener implements ActionListener
    {
        // create an event handler for the left button
        public void actionPerformed(ActionEvent e)
        {
            System.out.println("Left button was pressed.");
        } // end actionPerformed(ActionEvent e)
    } // end inner class LeftListener
    //*****

    // create a listener for the JbtRight button as an inner class
    class RightListener implements ActionListener
    {
        // create an event handler for the right button
        public void actionPerformed(ActionEvent e)
        {
            System.out.println("Right button was pressed.");
        } // end actionPerformed(ActionEvent e)
    } // end inner class LeftListener
    //*****

} // end class TwoButtonFrame

```

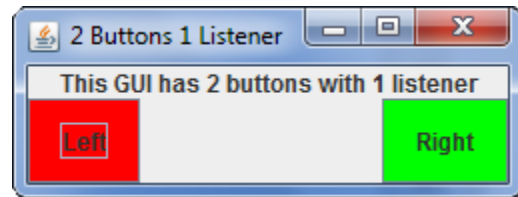
The listeners are instantiated and registered for each button.

This class is the listener with the handler for the left button.

This class is the listener with the handler for the right button.

## Two Buttons, One Listener

In this example, two components will share one listener. The file *TwoButtonsOneListener* included with the files for this chapter has an example of a GUI with two buttons that are registered to the same listener. The GUI is shown on the right.



The left button is named *jbtLeft* and the right button is named *jbtRight*. The single listener is the class itself that creates the GUI, *TwoButtonFrame*. The code for the GUI, shown below, uses the keyword *this* to register the current class, *TwoButtonFrame*, as the listener for *jbtLeft* and for *jbtRight*. In order for this approach to work, *TwoButtonFrame* must implement *ActionListener* and must include the method *actionPerformed(ActionEvent e)* to serve as the event handler.

As in the example above, when the left button is pressed, the message “*Left button was pressed.*” appears on the console. When the right button is pressed, the message “*Right button was pressed.*” appears on the console. The code to do these things is in the method *actionPerformed(ActionEvent e)*, which uses *if* commands based on the source of the event to determine which message to print.

The code for this example is on the next page.

```

class TwoButtonFrame extends JFrame implements ActionListener
{
    JLabel jlblNote = new JLabel("This GUI has 2 buttons with 1 listener");
    JButton jbtLeft  = new JButton("Left");
    JButton jbtRight = new JButton("Right");

    // a detailed constructor for TwoButtonFrame objects
    public TwoButtonFrame()
    {
        // give the TwoButtonFrame a BorderLayout
        setLayout( new BorderLayout() );

        // add jlblNote to the NORTH zone, centered
        jlblNote.setHorizontalAlignment(SwingConstants.CENTER);
        add(jlblNote, BorderLayout.NORTH);

        // add jbtLeft to the WEST zone
        jbtLeft.setBackground(Color.red);
        add(jbtLeft, BorderLayout.WEST);

        // add jbtRight to the EAST zone
        jbtRight.setBackground(Color.green);
        add(jbtRight, BorderLayout.EAST);

        // register TwoButtonFrame as the listener
        jbtLeft.addActionListener(this);
        jbtRight.addActionListener(this);

    } // end TwoButtonFrame() constructor
    //*****

    public void actionPerformed(ActionEvent e)
    {
        // react to specific buttons
        if (e.getSource() == jbtLeft)
            System.out.println("Left button was pressed.");

        if (e.getSource() == jbtRight)
            System.out.println("Right button was pressed.");

    } // end actionPerformed(ActionEvent e)
    //*****

} // end class TwoButtonFrame

```

*TwoButtonFrame*, creates a new GUI, as an extension of the *JFrame* class. It also implements *ActionListener* so that it can serve as the handler for its own events. This means it must include the method *actionPerformed(ActionEvent e)*.

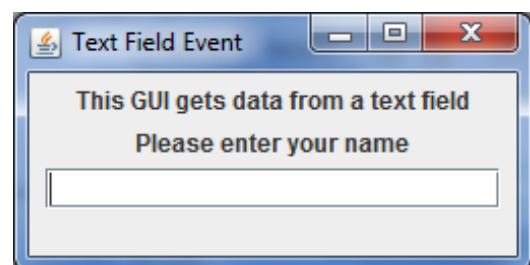
The same class, *TwoButtonFrame*, the class containing this code, is registered as the listener for both buttons.

The *actionPerformed()* method is defined in *TwoButtonFrame*, the same class that creates the GUI.

*actionPerformed()* uses *if* commands based on the source of the event to determine which message to print.

## 10.4 Java Events for Text Boxes

An example of a small GUI with a text field is shown on the right. The text field is a *JTextField*. When a user enters text in a *JTextField* – types text and then presses <enter> – an event is generated at the moment the <enter> key is pressed.



A *JTextField* has several useful properties, including **text** and **columns**. *text* of a *JTextField* is the set of characters currently in the field as a *String*. There are *getText()* and *setText()* methods for the *text* property. The *columns* property is the column width of the field as an integer. It can be specified as a parameter in a *JTextField* constructor, such as in `JTextField jtfName = new JTextField(20);`

The *getText()* method can be used in an event handler to capture as a *String* the text in the field at the time the user presses the enter key. The *setText()* method can be used to reset the field. Here is an example of code for a handler that does these things:

```
public void actionPerformed(ActionEvent e)
{
    // capture the name from the text field and reset the field
    String name = jtfName.getText();
    jtfName.setText("");

    // output a message using the name to the console
    System.out.println("Hello " + name);

    // dispose of the frame (this frame) - closes a window or frame
    this.dispose();

} // end actionPerformed(ActionEvent e)
```

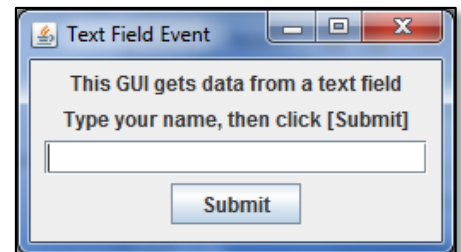
The *actionPerformed()* method is in the class *TextFieldFrame*, registered to the *jtfName* *JTextField*:

```
// register TextFieldFrame (this method) as the listener for jtfName
jtfName.addActionListener(this);
```

A user types text in the *JTextField* *jtfName* and then presses <enter>. An event is triggered when <enter> is pressed. The text in *jtfName* at the time <enter> is pressed is returned and saved in the *String* *name* by the instruction `String name = jtfName.getText()`. The instruction `jtfName.setText("");` then resets *jtfName* to be blank.

The instruction `this.dispose();` closes the currently active Window.

Alternatively, we could have a GUI with a *Submit* button that triggers the event listener, like the one shown here. The *actionPerformed()* method would still be in the class *TextFieldFrame*, but the frame in this case would be registered to the *jbtSubmit* *JButton*:



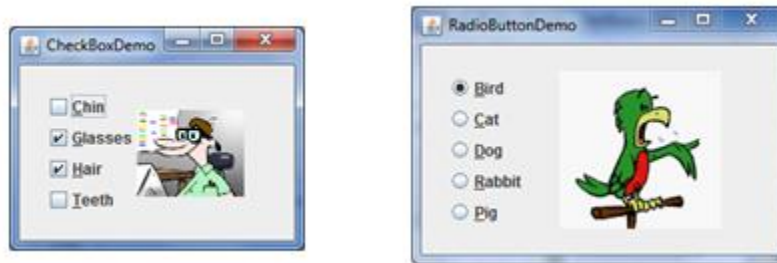
```
// register TextFieldFrame (this method) as the listener for jbtSubmit
jbtSubmit.addActionListener(this);
```

Now the event handler *actionPerformed()* will capture the text in the *JTextField* *jtfName* at the time event is triggered when the *jbtSubmit* *JButton* is clicked.

The two NetBeans projects *TextFieldListener* and *TextListenerSubmit* in the chapter files demonstrate each of these ways of reading from a text field. Of course, the *String* returned can be parsed to other data types.

## 10.5 Java Events for Check Boxes

GUI check boxes and radio buttons are both examples of GUI toggles. A GUI toggle is either checked or unchecked, and it changes state each time the user clicks the toggle. The difference between check boxes and radio buttons is that check boxes are independent of each other, while radio buttons belong to a button group, and only one button in each group may be checked at one time. They are also different shapes – check boxes are square, radio buttons are round.

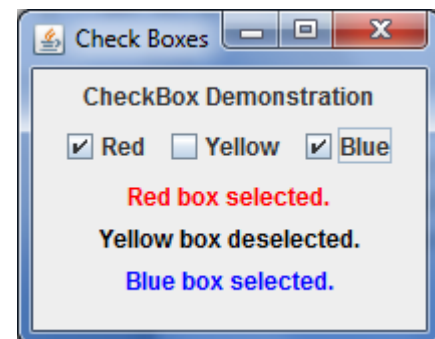


The image above shows a GUI with check boxes on the left and a GUI with radio buttons on the right. In this section we will look at how to use check boxes as Java GUI components; in the next we will see radio buttons.

With check boxes we often implement the *ItemListener* interface. The *ItemListener* interface requires the method *itemStateChanged(ItemEvent e)*, which detects changes in the state of an object. The difference between *ItemListener* and *ActionListener* can be a bit complicated, but the major difference is that *ItemListener* allows for the use of the method *setSelected()* with check boxes, which can set the boxes to be checked when they first appear.

The use of check boxes can be quite sophisticated. We look at a simple example, with three JCheckBoxes, titled *Red*, *Yellow*, and *Blue*. The titles for three checkboxes are included in their constructors when they are instantiated, as in: `JCheckBox redBox = new JCheckBox("Red");`

In our simple example, shown on the right, three labels are included in the GUI. The text and color of the labels will be affected by the check boxes.



The code for this simple example is on the next page and is included in the files for this chapter in the NetBeans project **CheckBoxCode**.

A more sophisticated checkbox example as a NetBeans project is available on the Web at:

<http://docs.oracle.com/javase/tutorial/uiswing/examples/zipfiles/components-CheckBoxDemoProject.zip>

An index of many Java code examples using Swing components for GUI programming is on the web at:

<http://docs.oracle.com/javase/tutorial/uiswing/examples/components/index.html>

Be aware – many of these examples use advanced programming techniques not covered in this course.

```

/*
 * CheckBoxCode.java
 * CSCI 111 Fall 2013
 * last edited Nov 10, 2013 by C. Herbert
 * This program shows an example of a GUI with check boxes
 * using a an itemListener implementation
 */

package checkboxcode;

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

public class CheckBoxCode {
    public static void main(String[] args)
    {
        // create an instance of CheckBoxFrame
        CheckBoxFrame myFrame = new CheckBoxFrame();

        myFrame.setTitle("Check Boxes");
        myFrame.setSize(212, 168);
        myFrame.setLocation(200, 100);
        myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        myFrame.setVisible(true);

    } // end main()
} // end class CheckBoxCode
//*****

// create a subclass of JFrames named CheckBoxFrame
class CheckBoxFrame extends JFrame implements ItemListener
{

    // add components to the frame
    JLabel heading = new JLabel("CheckBox Demonstration");
    JCheckBox redBox = new JCheckBox("Red");
    JCheckBox yellowBox = new JCheckBox("Yellow");
    JCheckBox blueBox = new JCheckBox("Blue");
    JLabel redLabel = new JLabel();
    JLabel yellowLabel = new JLabel();
    JLabel blueLabel = new JLabel();

    // a detailed constructor for CheckBoxFrame objects
    public CheckBoxFrame()
    {
        // give the CheckBoxFrame a BorderLayout
        setLayout( new FlowLayout() );

        // add the components to the frame

        add (heading);
        add(redBox);
        add(yellowBox);
        add(blueBox);
        add(redLabel);
        add(yellowLabel);
        add(blueLabel);

        // register CheckBoxFrame (this method) at the listener for the check boxes
        redBox.addItemListener(this);
        yellowBox.addItemListener(this);
        blueBox.addItemListener(this);

    } // end CheckBoxFrame() constructor

```



```
//itemStateChanged() is the event handler for itemListener implementations

public void itemStateChanged(ItemEvent e)
{
    // react to specific boxes
    if (e.getSource() == redBox )
    {
        if (e.getStateChange() == ItemEvent.SELECTED)
        {
            redLabel.setText("Red box selected.");
            redLabel.setForeground(Color.red);
        }

        else
        {
            redLabel.setText("Red box deselected.");
            redLabel.setForeground(Color.black);
        }
    } // end if

    if (e.getSource() == yellowBox )
    {
        if (e.getStateChange() == ItemEvent.SELECTED)
        {
            yellowLabel.setText("Yellow box selected.");
            yellowLabel.setForeground(Color.yellow);
        }
        else
        {
            yellowLabel.setText("Yellow box deselected.");
            yellowLabel.setForeground(Color.black);
        }
    } // end if

    if (e.getSource() == blueBox )
    {
        if (e.getStateChange() == ItemEvent.SELECTED)
        {
            blueLabel.setText("Blue box selected.");
            blueLabel.setForeground(Color.blue);
        }
        else
        {
            blueLabel.setText("Blue box deselected.");
            blueLabel.setForeground(Color.black);
        }
    } // end if

} // end itemStateChanged(ItemEvent e)
//*****

} // end class CheckBoxFrame
```

## 10.6 Java Events for Radio Buttons

Radio buttons must be placed in a button group, and only one radio button in each group may be checked at any one time. Either *ActionListener* or *ItemListener* implementations can be used with radio buttons.

Individual *JRadioButtons* must be declared, then a *ButtonGroup* must be declared, and then the buttons must be added to the *ButtonGroup*. Except for button groups and the shape of the buttons, radio buttons are toggles that work much like check boxes. However, since only one button in each group can be active at any one time, when one button is selected the other buttons are automatically deselected.

The image on the right is a GUI with radio buttons created by the code that follows. It is included in the files for this chapter as the NetBeans project *RadioButtonCode*. A more sophisticated example is available as NetBeans project online at:



<http://docs.oracle.com/javase/tutorial/uiswing/examples/zipfiles/components-RadioButtonDemoProject.zip>

```
/*
 * RadioButtonCode.java
 * CSCI 111 Fall 2013
 * last edited Nov 10, 2013 by C. Herbert
 * This program shows an example of a GUI with radio buttons
 * using an ActionListener implementation
 */

package radiobuttoncode;

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

public class RadioButtonCode {

    public static void main(String[] args)
    {
        // create an instance of RadioButtonsFrame
        RadioButtonFrame myFrame = new RadioButtonFrame();

        myFrame.setTitle("Radio Buttons");
        myFrame.setSize(224, 168);
        myFrame.setLocation(200, 100);
        myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        myFrame.setVisible(true);

    } // end main()
} // end class RadioButtonCode
//*****

// create a subclass of JFrame named RadioButtonFrame
class RadioButtonFrame extends JFrame implements ItemListener
{

    // add components to the frame
    JLabel heading = new JLabel("Radio Button Demonstration");
```

```
JRadioButton redButton = new JRadioButton("Red");
JRadioButton yellowButton = new JRadioButton("Yellow");
JRadioButton blueButton = new JRadioButton("Blue");
JLabel redLabel = new JLabel();
JLabel yellowLabel = new JLabel();
JLabel blueLabel = new JLabel();

// a detailed constructor for RadioButtonFrame objects
public RadioButtonFrame()
{
    // give the RadioButtonFrame a BorderLayout
    setLayout( new FlowLayout() );

    // add the components to the frame
    add (heading);
    add(redButton);
    add(yellowButton);
    add(blueButton);
    add(redLabel);
    add(yellowLabel);
    add(blueLabel);

    // create a button group
    ButtonGroup colorGroup = new ButtonGroup();
    colorGroup.add(redButton);
    colorGroup.add(yellowButton);
    colorGroup.add(blueButton);

    // register RadioButtonFrame as the listener for the check buttons
    redButton.addItemListener(this);
    yellowButton.addItemListener(this);
    blueButton.addItemListener(this);
} // end RadioButtonFrame() constructor

public void itemStateChanged(ItemEvent e)
{
    // react to specific button
    if (e.getSource() == redButton )
    {
        if (e.getStateChange() == ItemEvent.SELECTED)
        {
            redLabel.setText("Red button selected.");
            redLabel.setForeground(Color.red);
        }

        else
        {
            redLabel.setText("Red button deselected.");
            redLabel.setForeground(Color.black);
        }
    } // end if

    if (e.getSource() == yellowButton )
    {
        if (e.getStateChange() == ItemEvent.SELECTED)
        {
            yellowLabel.setText("Yellow button selected.");
            yellowLabel.setForeground(Color.yellow);
        }
    }
}
```

```

        else
        {
            yellowLabel.setText("Yellow button deselected.");
            yellowLabel.setForeground(Color.black);
        }

    } // end if

    if (e.getSource() == blueButton )
    {
        if (e.getStateChange() == ItemEvent.SELECTED)
        {
            blueLabel.setText("Blue button selected.");
            blueLabel.setForeground(Color.blue);
        }
        else
        {
            blueLabel.setText("Blue button deselected.");
            blueLabel.setForeground(Color.black);
        }
    }

    } // end if

} // end itemStateChanged(ItemEvent e)
//*****
} // end class RadioButtonsFrame

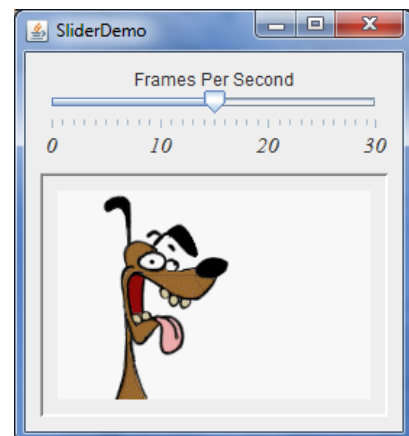
```

## 10.7 Java Events for Slider controls

A slider is a control that lets the user change a numeric value within a certain range by moving an indicator along a scale. Java has a *JSlider* class to implement sliders as Swing components. They can be placed in *JFrames* and *JPanels* like any other Swing component. The image on the right shows a GUI with a *JSlider* that changes the rate in frames per second of an animation.

Sliders have several properties:

- a minimum value
- a maximum value
- an orientation (horizontal or vertical)
- spacing for major tick marks
- spacing for minor tick marks



To set up slider events we can implement the *ChangeListener* interface, which requires the method *stateChanged*. It will constantly monitor the state of a slider and run the method as the state changes while the user moves the slider indicator.

Sliders can be used to change volume, color, or any other numeric property of an object. The method *getValue()* will return a slider's current value as an integer.

The code below shows how to set the parameters for a slider that will be used to select the red component of a color, which has a value in the range of 0 to 255:

```
// declare the slider object
JSlider redSlider = new JSlider();

// set parameters for red slider
redSlider = new JSlider(JSlider.VERTICAL, 0, 255, 0);
redSlider.setMajorTickSpacing(64);
redSlider.setMinorTickSpacing(32);
redSlider.setPaintTicks(true);
redSlider.setPaintLabels(true);
redSlider.setAlignmentX(Component.LEFT_ALIGNMENT);

// add the slider to the WEST zone of a frame with a border layout
add(redSlider, BorderLayout.WEST);
```

The instruction `redSlider = new JSlider(JSlider.VERTICAL, 0, 255, 0);` calls the `JSlider` constructor with four parameters:

- *int orientation* – the constants `JSlider.VERTICAL` and `JSlider.HORIZONTAL` are used.
- *int min* – the minimum values for the slider's range.
- *int max* – the maximum values for the slider's range.
- *int value* – the slider's initial value.

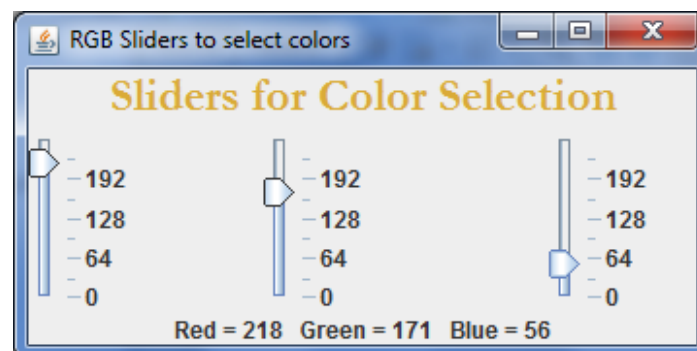
A slider has major tick marks that show with values and minor tick marks in between that do not show values. The intervals for these tick marks are set with the instructions:

```
redSlider.setMajorTickSpacing(64); and redSlider.setMinorTickSpacing(32);
```

The tick marks are shown with the instructions:

```
redSlider.setPaintTicks(true); and redSlider.setPaintLabels(true);
```

The code below is from the NetBeans project *RGBSliderCode*, included in the files for this chapter. It has a GUI with three sliders to change the Red, Green, and Blue components of the heading for the GUI. The GUI is shown here:



A NetBeans project with a more sophisticated *JSlider* example can be found online at:

<http://docs.oracle.com/javase/tutorial/uiswing/examples/zipfiles/components-SliderDemoProject.zip>

```

/*
 * RGBSliderCode.java
 * CSCI 111 Fall 2013
 * last edited Nov 10, 2013 by C. Herbert
 * This program shows an example of a GUI with sliders for RGB color selection
 * using a an ActionListener implementation
 */

package rgbslidercode;

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

public class RGBSliderCode
{
    public static void main(String[] args)
    {
        // create an instance of SliderFrame
        SliderFrame myFrame = new SliderFrame();

        myFrame.setTitle("RGB Sliders to select colors");
        myFrame.setSize(360, 180);
        myFrame.setLocation(200, 100);
        myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        myFrame.setVisible(true);

    } // end main()
} // end class RGBSliderCode
//*****

// create a subclass of JFrame named SliderFrame
class SliderFrame extends JFrame implements ChangeListener
{
    // declare components for the frame
    JLabel heading = new JLabel("Sliders for Color Selection");
    JSlider redSlider = new JSlider();
    JSlider greenSlider = new JSlider();
    JSlider blueSlider = new JSlider();
    JLabel colorZone = new JLabel("Colors");

    // a detailed constructor for SliderFrame objects
    public SliderFrame()
    {
        // give the SliderFrame a FlowLayout
        setLayout(new BorderLayout());

        // set parameters for red slider
        redSlider = new JSlider(JSlider.VERTICAL, 0, 255, 0);
        redSlider.setMajorTickSpacing(64);
        redSlider.setMinorTickSpacing(32);
        redSlider.setPaintTicks(true);
        redSlider.setPaintLabels(true);
        redSlider.setAlignmentX(Component.LEFT_ALIGNMENT);
    }
}

```

```

        // set parameters for green slider
        greenSlider = new JSlider(JSlider.VERTICAL, 0, 255, 0);
        greenSlider.setMajorTickSpacing(64);
        greenSlider.setMinorTickSpacing(32);
        greenSlider.setPaintTicks(true);
        greenSlider.setPaintLabels(true);
        greenSlider.setAlignmentX(Component.LEFT_ALIGNMENT);

        // set parameters for blue slider
        blueSlider = new JSlider(JSlider.VERTICAL, 0, 255, 0);
        blueSlider.setMajorTickSpacing(64);
        blueSlider.setMinorTickSpacing(32);
        blueSlider.setPaintTicks(true);
        blueSlider.setPaintLabels(true);
        blueSlider.setAlignmentX(Component.LEFT_ALIGNMENT);
        blueSlider.setPreferredSize(new Dimension(100, 100));

        // add the components to the frame
        heading.setHorizontalAlignment(JLabel.CENTER);
        heading.setFont(new Font("Garamond", Font.BOLD, 24));
        add(heading, BorderLayout.NORTH);

        add(redSlider, BorderLayout.WEST);
        add(greenSlider, BorderLayout.CENTER);
        add(blueSlider, BorderLayout.EAST);

        colorZone.setHorizontalAlignment(JLabel.CENTER);
        add(colorZone, BorderLayout.SOUTH);

        // register SliderFrame (this method) as the listener for the sliders
        redSlider.addChangeListener(this);
        greenSlider.addChangeListener(this);
        blueSlider.addChangeListener(this);

    } // end SliderFrame() constructor
    //*****

    // the event listener the the event handler for the sliders
    public void stateChanged(ChangeEvent e)
    {
        int r = redSlider.getValue();    // green Color value
        int g = greenSlider.getValue();  // green Color value
        int b = blueSlider.getValue();   // blue Color value

        heading.setForeground(new Color (r, g, b));
        colorZone.setText("Red = "+r+"    "+"Green = " +g+"    "+"Blue = "+b );
    } // end stateChanged(ChangeEvent e)
    //*****

} // end class SliderFrame

```



---

## 10.8 Other Java Events

There are many other Java components in the Swing packages that can be used to create graphical user interface software. More information about using Swing components, including examples of the use of Java Swing components and downloadable NetBeans projects, can be found online at:

<http://docs.oracle.com/javase/tutorial/uiswing/components/index.html>

Learning to use other Swing components involves learning what the components are and how they work. Using the components requires the programmer to design the GUI, establish the components in proper containers, and implement and register event listeners with event handlers for each active component. The programming simply involves the patience to master and deal with the details of the code.

Java also has classes of events for other devices, such as the mouse and keyboard. We can also create event-driven software for almost any purpose, not just user interfaces. Events can be triggered by any change in the state of a system, such as a change in the value of an object's property or a signal coming into a computer from an outside source.

Most of the programming for event-driven software is beyond the scope of this course. Some additional interface and graphics programming are covered in CSCI 112. Most baccalaureate Computer Science and Software Engineering programs include advanced courses in GUI programming, Human Computer Interaction, and related computer systems engineering. Many schools allow students to specialize in this area; some offer specific degrees in the field, particularly graduate degrees.

Examples of the use of the use of Java event listeners and handlers, including downloadable NetBeans projects, can be found online at:

<http://docs.oracle.com/javase/tutorial/uiswing/events/index.html>

JavaFX can be used to create Graphical user Interfaces and other interactive software, including 3D animations and environments. It requires more advanced knowledge than is covered in this course. Information about JavaFX is available online at:

<http://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm>

Extensive information on building GUIs using NetBeans is available online at:

<https://netbeans.org/kb/trails/matisse.html>

---

## Chapter Questions

1. What is a change that creates an event called?
2. What senses the change that creates an event? Is it hardware or software?
3. What is included in a working GUI?
4. What method is needed in implementations of Java's *ActionListener* interface?
5. What needs to be added to the container that holds a component in order for a component to be registered to a listener?
6. What classes do we need to create if we want separate handlers for different components in a GUI?
7. How can one listener listen for several events caused by different components?
8. What property of a `TextField` holds the text a user types into the field? What is the data type for this field? How can we use a `TextField` to capture other data types.?
9. When is an event triggered by a `TextField`? What can we do as an alternative way to trigger an event to get the current text from a `TextField`?
10. What is a GUI toggle? What are two components that act as toggles?
11. How do checkboxes and radio buttons differ in appearance? How do they differ in behavior?
12. What listener interface is often used with checkboxes instead of *ActionListener*? Why is this interface is often used with checkboxes?
13. What is a button group? Why is it used?
14. What happens to the other buttons in a group if a button is selected?
15. What is a slider control?
16. What type of variable is affected by a slider control?
17. What properties does a slider control have?
18. Which listener interface is often used with slider controls?
19. What can trigger an event other than a GUI component?
20. Where can we learn more about creating event-driven software for Java?

---

## Chapter Exercises

Each of these exercises involves creating a GUI with event-driven software to act in response to user input. `TextAreas` can be used in some cases to get the user input as a `String`, then you can parse it to integers or doubles as needed. Calculations and logic should be in the event handler. You can change messages on a screen by setting the text of a `JLabel` using the `setText()` method in the handler, such as the statement:

```
jftResult.setText(" The sine of " + x + "degrees = " + y);
```

You will need to design your GUI, including selecting a Layout manager for the GUI, such as a *BorderLayout* or *GridLayout*.

Remember, a GUI with events is a little different from sequential logic. All of the labels and textboxes, etc. are displayed on the screen, then changed by event handlers when the user triggers an event.

You can take advantage of the extensive examples in the files for this chapter to help you build your GUI.

1. Multiplication GUI

Create a GUI that functions like flash-cards to teach or test multiplication. Your software should:

- randomly select two one-digit integers ;
- display a multiplication problem based on the two integers, such as "3 x 7 =";
- get the user's answer in a text box and parse it to an integer;
- display a message telling the user that the user was correct, or display a message telling the user the correct answer.

You can use JLabels to display the problem and the messages.

## 2. Celsius to Fahrenheit GUI

Create a GUI that converts Celsius temperatures to Fahrenheit. Your GUI should:

- have two textboxes, one for city and one for Celsius temperature;
- get city and temperature from the two textboxes;
- parse the temperature to a double variable;
- in the event handler
  - calculate the temperature in degrees Fahrenheit. The formula is  $F = (\frac{9.0}{5.0} C) + 32.0$ .
  - set the text for a label displaying the city and the Fahrenheit temperature.

## 3. CCP Airways GUI

CCP Airways has a frequent flyer program that rewards people with frequent flyer miles in different categories, as follows:

- up to 10,000 miles      CCP flyer
- 10,000 to 24,999 miles   CCP Silver Flyer
- 25,000 to 49,999 miles   CCP Gold Flyer
- 50,000 or more miles    CCP Platinum Flyer

Create a Java application using a GUI with a JTextField that allows the user to enter the number of CCP Airways frequent flyer miles, then determines and displays the user's frequent flyer status.

## 4. Payroll GUI

A company determines an employee's pay based on hours times rate, with *"time and a half for overtime"*. Overtime is more than 40 hours in one week. If an employee works more than 40 hours, the gross pay is equal to the regular pay for 40 hours, plus 1.5 times the rate for overtime hours.

Create a Java GUI payroll application in NetBeans that has textboxes for an employee's hours, and rate and a handler that parses these to double variables and calculate gross pay and displays the result in a label in the GUI.

## 5. Guessing Game GUI

The guessing game that asks a user to pick a number from 1 to 100 is described in chapter 3, on pages 14, 15, and 16 with pseudocode, sample output and a flowchart. Rewrite this program to work in a GUI, with a single text box for the user to enter a guess and a single label that displays the messages for correct, lower, or higher. Each time the user enters a guess, the message should change.

Your task is to create the game in Java. You should submit the game along with your programming lab report as described in section 3.5 on page 32 of chapter 3.

#### 6. International Gymnastics Scoring GUI

We wish to write a program to quickly calculate gymnastics scores for an international competition using a GUI. The program should have text boxes to ask the user for the scores from each of the six judges, numbered 1 through 6 for anonymity. Your program should parse the scores into double variables, add each score to a sum, then calculate and display the average score.

Scores are in the range 0 to 10 with one decimal place, such as 8.5. The average should be displayed with two decimal places.

#### 7. Trigonometry GUI

Write a program to ask the user for an angle in degrees , then display the sine, cosine and tangent of the angle using a GUI.

Your program should use the functions in the Math library – Math.sin() and Math.cos() and Math.tan(). The trigonometric functions work in radians, so the angle will need to be parsed to a double then converted from degrees to radians, using the Math.toRadians() method.

For example . `xRad = Math.toRadians(x)` ; then `sin = Math.sin(xRad)`; etc.

#### 8. “Let’s make a deal!” GUI

Write a program with radio buttons and three pictures to simulate someone picking one of three doors similar to what happens in the game show *“Let’s make a deal!”*. Your program should ask the user to pick a door number 1, door number 2, or door number 3, then display what is behind the door the person picked.

You should have three places in your GUI for pictures, and your event handler should display a door in each place if a button is not selected. If a button is selected, the picture of the item behind the door should be displayed.

You need to find four pictures for this program: a door, plus three prizes behind the doors.

#### 9. The Java Math class has a hypotenuse function that can be used to determine the distance between two points. Write a GUI-based program that allows the user to submit the x and y coordinates for two points, then tells the user the distance between the points.

Your program should use a submit button to trigger the event. The formula for the distance is  $C = \sqrt{\Delta X^2 + \Delta Y^2}$  where  $\Delta X$  and  $\Delta Y$  are the difference between the x coordinates and the difference between the y coordinates. `Math.hypot(double A, double B)` returns as a double the square root of ( $A^2 + B^2$ ).

## 10. Quadratic Equation Solver GUI

A quadratic equation in general form is:  $y = ax^2 + bx + c$ . Write a program with a GUI that has text boxes for  $a$ ,  $b$ , and  $c$  and that allows the user to enter values in each text box then calculates and displays the real roots of the equation using the quadratic formula. Lab 3C on page 47 in Chapter 3 has a detailed discussion of the solution to this problem using console input. The finished code for that lab is included with the files for this chapter (Chapter 10) in the file *quadraticSolution.zip*.

Your task is to re-write the program as a GUI program with textboxes for the input of  $a$ ,  $b$ , and  $c$  and a *submit* button to trigger the event to calculate and display the solution.