EVENT HANDLING

Event handling is the process of responding to asynchronous events as they occur during the program run. An ***event*** is an action that occurs externally to your program and to which your program must respond. Events are ***asynchronous***, meaning that they can happen anytime during the program run; they are not tied to the run-time of any specific part of your program.

The basic event handling model is for every event to be tied to an ***event handler*** – some program text that is coded to correctly respond to the event. This event handler is called – by the control program, i.e. the virtual machine, operating system or whatever – whenever the event occurs.

The Java model implements this by providing various listener interfaces in the **`java.awt.event`** package. Each listener interface is meant to listen for a particular type of event and it contains a special method that is called when an event of that type occurs.

You, the programmer, set up the event handling in your program as follows:

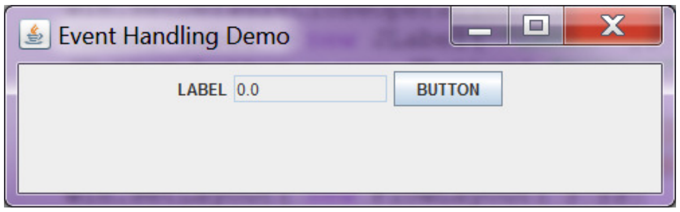
1. Choose the listener interface that listens for the type of events you need to handle.
2. Create a subclass that implements it.
3. Within your subclass, implement each method that handles an event in a manner appropriate to your need.
4. Use your newly created subclass to build an ***event listener*** object, which listens for the event to occur.
5. Register the event listener with the GUI component that can trigger the event.

With this done, the running program behaves as follows:

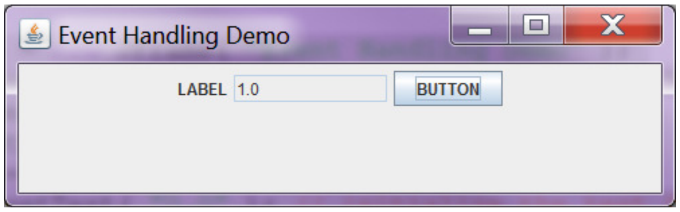
When an external event happens to a GUI component...

A. The GUI component notifies whatever event listener is registered to it and sends it the ***event object***.
B. The event listener executes its special method.
C. Your code (that you implemented in step 3 above) executes, handling the event.

Example
The application **`JFrameDemo2`** (see page 3 of the topic *GUI Applications*) displays this GUI:



We want the application to increment the number in the textbox is when the user mouse clicks the button:



The complete application is on the next page. Code that significantly differs from **`JFrameDemo2`** is in black; code that is substantially the same is in gray.

The application has undergone a major reorganization. First, to prevent the percolation of **`static`**, an application constructor is introduced (lines 27-46) and the main method is rewritten accordingly (lines 22-25). Second, the declaration of **`textfield`** is moved to line 7 so that its scope includes the private **`MyListener`** class.

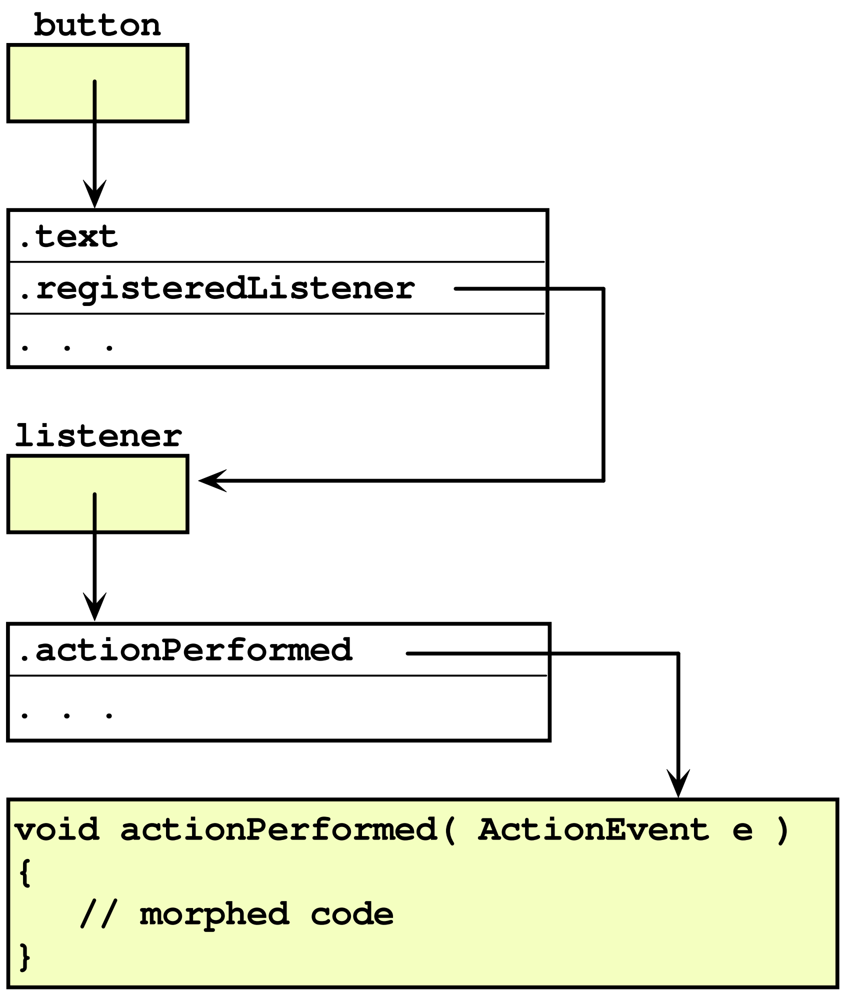
Comments within the code mark the programming steps explained above. Lines 10-20 define the event listener class, containing the handler method that grabs the text field and increments it. The event listener object is built at line 40 and registered with the button at line 42.

`MyListener` is a ***nested class***, being wholly contained within another class. It is a member of the application class and has access to everything in the application's scope.

The application constructor builds the GUI.

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

public class EventHandlingDemo1
{
   private JTextField textField; // must be accessed by MyListener

   // steps 1, 2: create subclass implementing appropriate interface
   private class MyListener implements ActionListener
   {
     // step 3: make the handler method do what I need.
      public void actionPerformed( ActionEvent e )
      {
         // get text from text field and parse into a double
         double x = Double.parseDouble( textField.getText( ) );
         // increment it and put it back
         textField.setText( Double.toString( ++x ) );
      }
   }

   public static void main ( String [] args )
   {                              // avoid percolation of 'static'
      new EventHandlingDemo1( ); // by using an app constructor
   }

   public EventHandlingDemo1( )  // the application constructor
   {
      JFrame win = new JFrame( "Event Handling Demo" );
      win.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      JLabel label = new JLabel( "LABEL" );
      JButton button = new JButton( "BUTTON" );
      textField = new JTextField( 10 );
      textField.setText( "0.0" ); // initialize the text field
      win.setLayout( new FlowLayout( ) );
      win.add( label );
      win.add( textField );
      win.add( button );
      // step 4: create an object to listen for the event
      MyListener listener = new MyListener( );
      // step 5: register the event listener with the button
      button.addActionListener( listener );
      // normally these are last
      win.setSize( 500, 150 );
      win.setVisible( true );
   }
}
```

This diagram shows the relationship between the objects in the example application created by steps marked 3 and 4.



What the JVM Does

As the program runs, the action listener (**`listener`** in the above example) sits in the background waiting for an event to occur.

When the user clicks the button named **`button`**...

A. The JVM it generates a **`java.awt.event.ActionEvent`** object.
B. The button passes the **`ActionEvent`** object to parameter **`e`** of the **`actionPerformed`** method.
C. The **`actionPerformed`** method executes.
D. The main program thread of execution waits as all this happens.
E. When **`actionPerformed`** is finished, the main thread resumes.

Both buttons and text fields generate `ActionEvent` objects and so can be handled by the same `actionPerformed` method. If it is necessary to determine which GUI component is the event source, you can use cascading if-else statements with one of the following `ActionEvent` methods:

```
Object getSource( )
// Return the GUI component that is the source of the event.
```

```
String getActionCommand( )
// Return the caption string of the GUI component that is the
// source of the event.
```

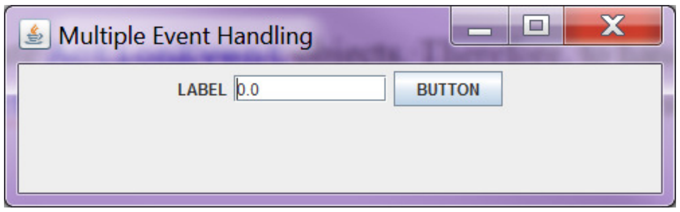
Examples

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

```
if ( e.getActionCommand( ) == "BUTTON" ) . . .
```

Example

We'll modify `EventHandlingDemo1` so that it responds to both user input to the text field and mouse clicks to the button. Specifically, it displays this GUI:



If the user enters a value in the text field then the user's entry replaces whatever value had been there. The button works the same as with the previous example.

The application is on the next page. Code that significantly differs from `EventHandlingDemo1` is in black; code that is substantially the same is in gray.

The declaration of `button` is moved to line 8 so that its scope includes the `MyListener` class.

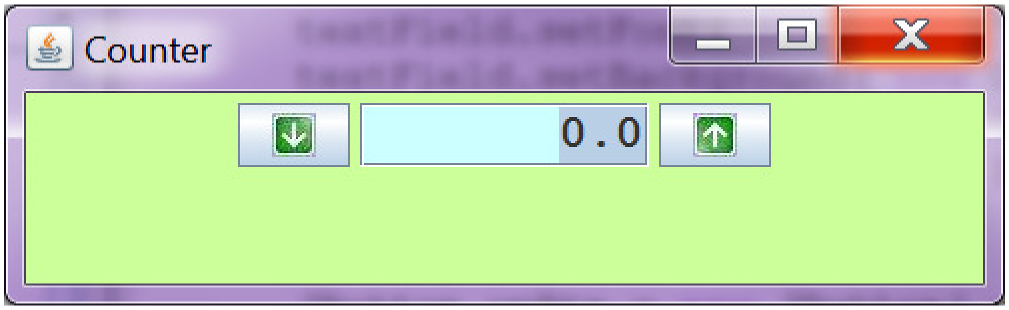
The modified listener class is on lines 10-23. It calls method `getSource` on lines 16 and 18 to determine which GUI component is the source of the event.

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

public class EventHandlingDemo2
{
   private JTextField textField; // must be accessed by MyListener
   private JButton button;       // same

   private class MyListener implements ActionListener
   {
      public void actionPerformed( ActionEvent e )
      {
         // get text from text field and parse into a double
         double x = Double.parseDouble( textField.getText( ) );
         if ( e.getSource( ) == textField ) // source is text field
            { } // don't do anything, we just want to put value back
         else if ( e.getSource( ) == button ) // source is button
            x++;  // increment the text field
         // put it back
         textField.setText( Double.toString( x ) );
      }
   }

   public static void main ( String [] args )
   {                              // avoid percolation of 'static'
      new EventHandlingDemo2( ); // by using an app constructor
   }

   public EventHandlingDemo2( )  // the application constructor
   {
      JFrame win = new JFrame( "Multiple Event Handling" );
      win.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      JLabel label = new JLabel( "LABEL" );
      button = new JButton( "BUTTON" );
      textField = new JTextField( 10 );
      textField.setText( "0.0" ); // initialize the text field
      win.setLayout( new FlowLayout( ) );
      win.add( label );
      win.add( textField );
      win.add( button );
      // create an object to listen for the event
      MyListener listener = new MyListener( );
      // register the event listener with the button and text field
      button.addActionListener( listener );
      textField.addActionListener( listener );
      // normally these are last
      win.setSize( 500, 150 );
      win.setVisible( true );
   }
}
```

Exercises

For each of the following write a Java program (application or applet) that adheres to the given specifications.

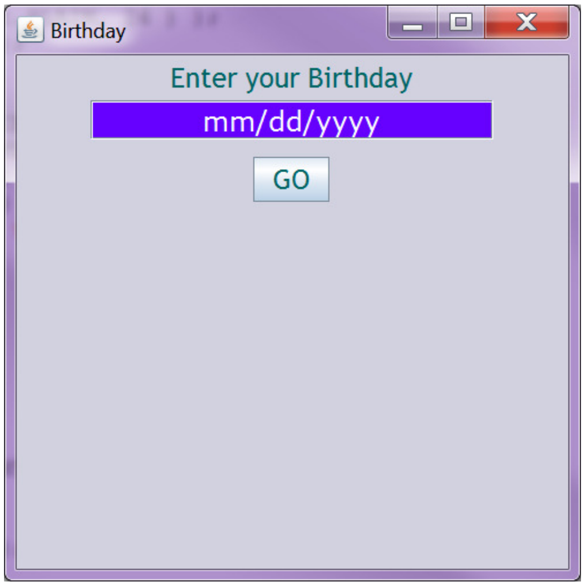
1.

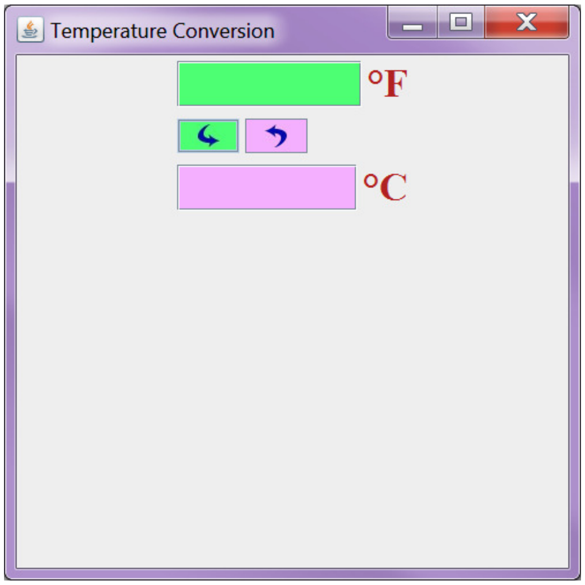


1. The text field must accept user input.
    1.1. The text field must be initialized to 0.0.
    1.2. Initially, the text field must:
        1.2.1. Contain 0.0.
        1.2.2. Have focus.
        1.2.3. Have its contents selected.
    1.3. The text field accepts any value; no input validation is performed.
2. The conditions in 1.2.2 and 1.2.3 must be true after either button is clicked or the Enter key is pressed on the text field.
3. The UP button must increment the text field value.
4. The DOWN button must decrement the text field value.
    4.1. Initially, the button must be disabled.
    4.2. The button must be disabled if the text field value is 0 or negative.

2.

1. The text field must accept user input.
    1.1. The text field contents must be center aligned.
    1.2. Initially, the text field must:
        1.2.1. Contain the string *mm/dd/yyyy*.
        1.2.2. Have focus.
        1.2.3. Have its contents selected.
    1.3. The text field accepts any value; no input validation is performed.
2. The same action must occur upon the user pressing the Enter key on the text field or clicking the GO button.



	2.1. After such action is performed, the conditions in 1.2.2 and 1.2.3 must be true. 3. The GO button must 3.1. Convert the text field value to the format *day month date, year*. For example, **`08/30/2000`** must display as **`Wednesday August 30, 2000`**. 3.2. Calculate the age of the user in years, months and days. 3.3. Display the long date and age centered beneath the GO button. 3.4. Display "Happy Birthday" if the current date is the same as the user's birth date.
3.	 1. The text fields accept user input. 1.1. The text field accepts any value; no input validation is performed. 2. The user pressing the Enter key on the Fahrenheit text field produces the same action as a mouse click on the DOWN button. 3. The user pressing the Enter key on the Celsius text field produces the same action as a mouse click on the UP button. 4. The DOWN button converts the Fahrenheit temperature to Celsius. 4.1. The Celsius value must be displayed within the Celsius text field. 4.2. The value must be formatted to two decimal places. 5. The UP button converts the Celsius temperature to Fahrenheit. 5.1. The Fahrenheit value must be displayed within the Fahrenheit text field. 5.2. The value must be formatted to two decimal places.