Chapter 11: User input

User Input

If we have a GUI, we have to respond to user inputs.

we have to make our programs responsive to the events of the user and the system itself.

Swing handles events with a set of interfaces called event listeners.

We create a listener object and associate it with the interface component we are monitoring.

Event Listeners

to respond to a user event in Java

we must implement the interface that works with these types of events..

Interface types that work with user responses are **EVENT LISTENERS**

each listener handles a specific kind of event.

Event Listener Interfaces

ActionListener – **Action events** are generated when a user performs an action on a component such as checking a box

AdjustmentListener – **Adjustment events** are generated when a component is adjusted such as a scrollbar movement

FocusListener – **Keyboard focus** events are generated when a component gains or loses focus

ItemListener – **Item events** are generated when an item such as checkbox is changed

KeyListener – **Keyboard events** occur when a user enters text using the keyboard

MouseListener – **Mouse events** occur by mouse clicks, a mouse entering a component's area and a mouse leaving a components area

MouseMotionListener – **Mouse movement events** track all movement by a mouse over component

WindowListener – **Window events** are generated when a window is maximized, minimized, moved or closed

The implements keyword

A class can implement as many listeners as needed. The **implements** keyword in the class declaration is followed by the name of the interface. If more than one interface is needed, their names are separated by commas. The following snippet shows this:

```
public class Suspense extends JFame implements ActionListener, TextListener {

        // ….

}
```

To refer to these event listener interfaces, we can import them individually or use the import statement with a wildcard to make the entire package available: **import java.awt.event.*;**

Set up

When we declare a class as an event listener, we have to set up

a specific type of event to be heard by that class
and a matching listener to the GUI component

we must listen to the component before we add it to a container or the attributes will be disregarded when the application is executed.

Methods for event listeners

addActionListener() – JButton, JCheckBox, JComboBox, JTextField, JRadioButton and JMenuItem

addFocusListener() – All Swing components

addItemListener() – JButton, JCheckbox, JComboBox, and JRadioButton

addKeyListener() – All Swing components

addMouseListener() – All Swing components

addMouseMotionListener() – All Swing components

addTextListener() – JtextField and JTextArea components

addWindowListener() – Jwindow and JFrame components.

Event listeners

The following snippet creates a JButton object and associates and action event listener with it:

JButton zap = new JButton(“Zap”;

Zap.addActionListner(this);

All of the listener adding methods take one argument: the object that is listening for events of that kind.

Using **this** indicates that the current class is the event listener. We could use a different object as long as its class implements the right listener interface.

Event handling methods

When we associate an interface with its class, the class must contain methods that **implement every method in the interface**.

In the case of an event listener, the windowing system calls each method automatically when the corresponding user event takes place. The **ActionListner** interface has only one method: **actionPerformed()**.

All classes that implement ActionListener must have a method with the following structure:

public void **actionPerformed(ActionEvent event)** {// handle event here}

If only one component in our program's GUI has a listener for action events, we will know that this actionPerformed() method is called only in response to an event generated by that one component.

Event handling methods

When more than one component has an action event listener, we must use the method's **ActionEvent** argument to figure out which component was used and to respond accordingly.

We can use this object to discover details about the component that generated the event.

Event handling methods

Every event handling method is sent an event object of some kind. We can use the object's getSource() method to determine which component sent the event, as in the following example:

public void **actionPerformed(ActionEvent event)** {

Object source = event.getSource(); }

The object returned by the getSouce() method can be compared to components by using the == operator. The following statements extend the above code to handle the user clicks on buttons name d quitButton and sortRecords:

if (**source == quitButton**) {quit();}

If (**source == sortRecords**) {sort();}

Event handling methods

if (**source == quitButton**) {quit();}

If (**source == sortRecords**) {sort();}

The quit() method is called if quitButton object generates the event and sort() is called if the sortRecords button generates the event. Many event handling methods call a different method for each kind of event or component. This makes event handling easier to manage. Also, if a class has more than one event handling method, each one can call the same methods.

Event handling methods

To determine the class of the component that generated the event, the **instanceof** operator can be used in an event handling method. The following is an example that can be used with one button and one text field with each generating an action event:

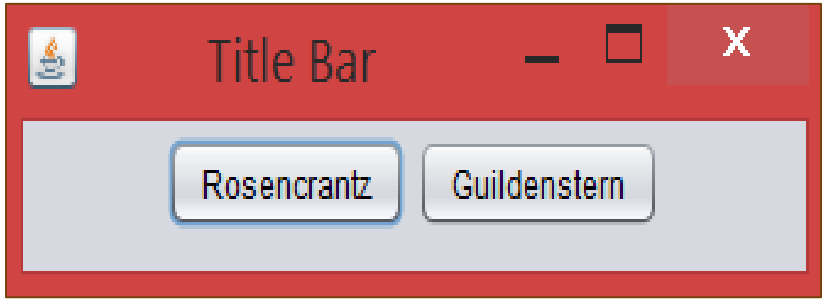
```
void actionPerformed(ActionEvent event) {
        Object source = event.getSource();
        if (source instanceof JTextField) {
                calculateScore();
        } else if (source instanceof JButton) {
                quit();
        }
}
```

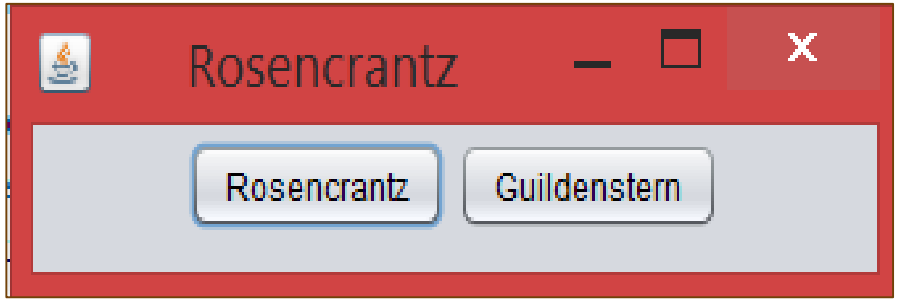
In this example, if the event generating component belongs to the JTextField class, the calculateScore() method is called otherwise if it belongs to JButton, quit() is called.

Exercise 1: TitleChanger.java, p. 248-9

This application, called TitleChanger.java, displays a frame with two JButton components that are used to change the text on the frame's title bar.

Using Luna, create the TitleChanger.java file. When a clean compile has been achieved, run the application to test all possible outcomes. The results are shown below.





Let's examine the code

Only 13 lines are needed to respond to the action events:

Line 1: Imports the java.awt.event package

Line 5: Implement the ActionListener interface

Lines 15 – 16: Add action listeners to both JButton objects

Lines 25 – 33: Respond to the action events that occur from the two JButton objects. The evt object's getSource() method determines the event's source. If it is equal to b1, the frame's title is set to Rosencrantz, if b2 the title is set to Guildenstern. From here a call to **repaint()** is needed to force the frame redraw to occur after the title has been changed

Let's examine the code

Only 13 lines are needed to respond to the action events:

Line 1: Imports the java.awt.event package

Line 5: Implement the ActionListener interface

Lines 15 – 16: Add action listeners to both JButton objects

Lines 25 – 33: Respond to the action events that occur from the two JButton objects. The evt object's getSource() method determines the event's source. If it is equal to b1, the frame's title is set to Rosencrantz, if b2 the title is set to Guildenstern. From here a call to **repaint()** is needed to force the frame redraw to occur after the title has been changed.

Action events

ACTION EVENTS are invoked when a user completes an action using components such as buttons, check boxes, menu items text fields and radio buttons

To use these events, the class must implement the ActionListener interface and the addActionListener() method is called on each component that should generate an action event unless it is to be ignored.

The actionPerformed(ActionEvent) method is the only method of the ActionListener interface and takes the following form:

```
public void actionPerformed(ActionEvent event) {

        // handle event here

}
```

Action events

As we've seen, we have the getSource() method to determine the source of the action. We can also use the **getActionCommand()** method to discover the information about the event's source.

By default, the action command is the text associated with the component such as the label on a button.

We can set a different action command for a component by calling the **setActionCommand(String)** method where String should be the text desired for the action command. The following shows us how a button and menu item is created and gives both of them to the action command "Sort Files":

JButton sort = new JButton("Sort");

JMenuItem menuSort = new JMenuItem("Sort");

sort.setActionCommand("Sort Files");

menuSort.setActionCommand("Sort Files");

Focus events

FOCUS EVENTS occur when any component gains or loses input focus on a GUI.

FOCUS is the word we use to describe the active component.

Focus applies to all components that can receive input.

We give a component its focus by calling its **requestFocus()** method with no arguments as shown with the following:

JButton ok = new JButton(“ok”);

ok.**requestFocus();**

Focus events

In order to handle the focus event, a class must implement the **FocusListener** interface which has **two** methods:
focusGained(FocusEvent)

focusLost(FocusEvent).

public void focusGained(FocusEvent event) {// … }

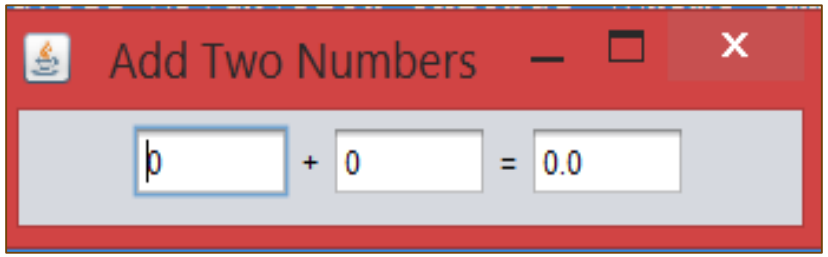
public void focusLost(FocusEvent event) {// … }

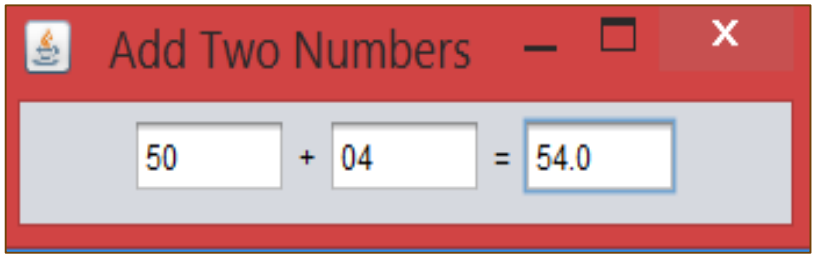
We use the getSource() method to determine which object has gained or lost focus where the object is sent as an argument to the two methods.

Exercise 1: Calculator.java, P. 250-252

In this exercise, we create a Java application that displays the sum of two numbers. We create focus events to determine when the sum needs to be recalculated.

Using Luna, create the following application calling it Calculator.java. Once complete run the code ensuring that all paths of execution are tested. The results should resemble those shown below.





Let's examine the code

The focus listeners are added to the first two fields, value1 and value2, and the class implements the FocusListener interface.

Lines 36 – 46: Call focusGain() whenever either of the fields gains input focus. Calculate the sum by adding the values in the other two fields. If either contains an invalid value (a string) a NumberFormatException is thrown and all three fields are set to zero. – **did you remember to test the alternate path (error exceptions)?**

focusLost() does the same thing by calling focusGain() with the focus event as an argument.

Items Events

Item events happen when an item is selected/deselected on a component such as a button, check box or radio button.

A class must implement the ItemListener interface to handle these types of events.

The **itemStateChanged(ItemEvent)** is the only method in ItemListener interface and takes the following form:

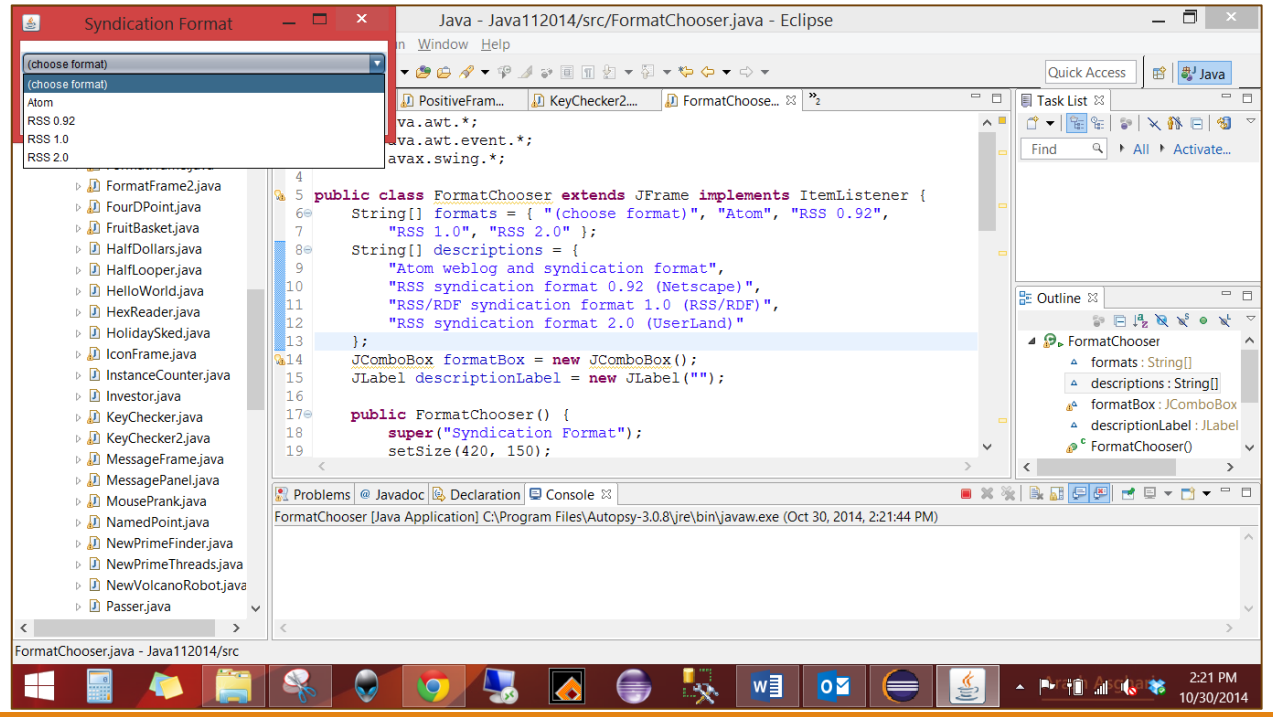
Public void **itemStateChagned(ItemEvent event)** {// handle event here}

Again, we use the **getItem()** method on the ItemEvent object to determine if a change has occurred. We can also use the **getStateChange()** method that returns an integer that equals either the class variable **ItemEvent.DESELECTED** or **ItemEvent.SELECTED**.

Exercise 1: FormatChooser.java, P. 252-4

This application illustrates the use of item events by displaying information about a selected combo box item in a label. It extends the application on Chapter 9 when we began working with Swing.

Using Luna create the following naming it FormatChooser.java. When you get a clean compile, test the application ensuring that all paths (including alternate paths) achieve the expected results as shown below.



Let's examine the code

Lines 22 – 25: Create a combo box from an array of strings and add an item listener to the component

Lines 31 – 36: Receive Item Events with itemStateChange(ItemEvent) which changes the label's text based on the index number of the selected item. Index 1 corresponds to "Atom" and so on.

Key events

KEY EVENTS happen when a key is pressed on the keyboard.

Any component can generate these events and a class must implement the **KeyListener** interface.

The interface has three methods: **keyPressed(KeyEvent), keyReleased(KeyEvent)** and **keyTyped(KeyEvent)** and take on the following formats:

public void keyPressed(KeyEvent event) {// … }

public void keyReleased(KeyEvent event) {// … }

public void keyTyped(KeyEvent event) {// … }

The **getKeyChar()** method returns the character of the key associated with the event.

If a Unicode value cannot be represented by the key, the getKeyChar() returns a character value equal to the class variable **KeyEvent.CHAR_UNDEFINED**

Key events

To generate key events, a component must be able to receive input focus – text fields, text areas and other components that accept keyboard input support this ability automatically.

For other components such as labels and panels, the **setFocusable(boolean)** method should be called with an argument of true as shown below:

JPanel pane = new JPanel();

pane.setFocusable(true);

Mouse events

MOUSE EVENTS are generated by a mouse click, a mouse movement entering a component's area or a mouse leaving the area. Any component can generate these events which are implemented via a class through the **MouseListener** interface which has five methods:

mouseClicked(MouseEvent)

mouseEntered(MouseEvent)

mouseExited(MouseEvent)

mousePressed(MouseEvent)

mouseReleased(MouseEvent)

Each takes the same form as the mouseReleased(MouseEvent) presented below:

public void **mouseReleased(MouseEvent event)**

{// handle event here}

Mouse events

We can use the following methods on the MouseEvent objects:

getClickCount() – Returns an int of the number of clicks

getrPoint() – Returns a Point object (x,y coordinates) within the component where the mouse was clicked

getX() – Returns the x position

getY() – Returns the y position

Mouse motion events

MOUSE MOTION EVENTS occur when the mouse is moved over a component.

---

this event and is implemented via the **MoustMotionListener** interface in a class. MouseMotionListener has two methods: **mouseDragged(MouseEvent)** and **mouseMoved(MouseEvent)** and take the following forms:

public void **mouseDragged(MouseEvent event)** {

    // handle event here}

public void **mouseMoved(MouseEvent event)** {

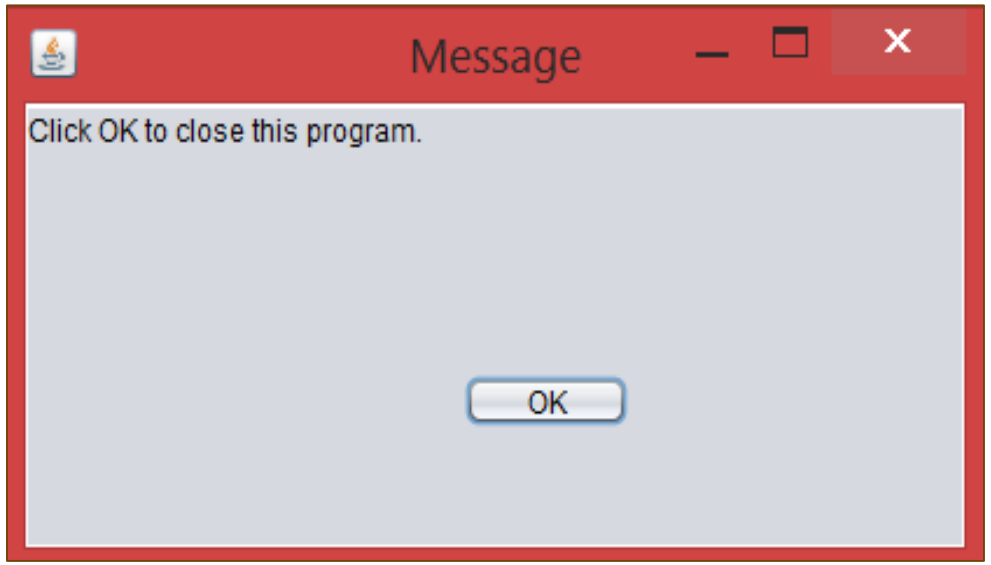
    // handle event here}

Unlike the other event listener interfaces we've looked at, MouseMotionListener does not have its own event type opting to use the MouseEvent.

Exercise 1: MousePrank.java, P. 256-8

In this exercise, we demonstrate how to detect and respond to mouse events. MousePrank.java consists of two classes, MousePrank and PrankPanel, that implements a user interface button that tries to avoid being clicked – Don't try this in real applications… the end user will not be happy!

In Luna, create the MousePrank.java application. Be sure to test it.



MousePrank.java

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

public class MousePrank extends JFrame implements ActionListener
{
   public MousePrank() {
      super("Message");
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setSize(420, 220);
      BorderLayout border = new BorderLayout();
      setLayout(border);
      JLabel message = new JLabel("Click OK to close this program.");
      add(BorderLayout.NORTH, message);
      PrankPanel prank = new PrankPanel();
      prank.ok.addActionListener(this);
      add(BorderLayout.CENTER, prank);
      setVisible(true);
   }

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

   public Insets getInsets() {
      return new Insets(40, 10, 10, 10);
   }

   private static void setLookAndFeel() {
      try {
         UIManager.setLookAndFeel(
            "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
         );
      } catch (Exception exc) {
         System.err.println("Couldn't use the system "
            + "look and feel: " + exc);
      }
   }
```

Let's examine the code

This is one of our most complex programs to date. The application is a frame that holds two components arranged with a border layout. The label "Click OK to close the program" and a panel with the OK button on it.

To make the button misbehave we implement it with the PrankPanel class (a subclass of JPanel) that includes a button that is drawn at a specific position on the panel instead of being placed by the layout manager as we talked about yesterChapter.

Firstly, we set the layout manager to null to cause it to stop using the flow layout as its default.

This is followed by placing the button on the panel using **setBounds(Rectangle)** which is the same method that determines where a frame or window will appear on a desktop. A Rectangle object is created with four arguments: its positions for x and y; and width and height.

Lets examine the code

By creating the Rectangle object as the argument to setBounds() is more efficient than creating an object with a name and using that object as the argument. We don't' need to use the object anywhere else in the class so we really don't need to name it.

The class has instance variables that hold the button's x-y position, button and buttonY. They start at (110, 110) and change whenever the muse comes within 50 pixels of the center of the button. Mouse movements are tracked by implementing the **MouseListener** interface its two methods, **mouseMoved(MouseEvent)** and **mouseDragged(MouseEvent)** however the panel ignores mouseDragged() since we don't need it. When the mouse moves, the mouse event object's getX() and getY() return its current x-y position which is stored in the instance variables mouseX and mouseY.

Lets examine the code

moveButton(int, int, int) takes three arguments – the **button's x or y position**, **the mouse's x or y position** and **the panel's width or height**. The method moves the mouse button away from the mouse in either a vertical or horizontal directions depending on whether it is called with x-coordinates and the panel's height or y-coordinates and the width.

In Line 97, after the button has been moved, the **repaint()** method is called which causes the panel's **printComponent(Graphics)** method to be called (lines 95-98). Every component has a **paintComponent()** method that can be overridden to draw the component.

Window Events

WINDOW EVENTS happen when a user opens or closes a window object such as a JFrame or a JWindow. Any component can generate these events, like the others, and a class must implement **WindowListener** interface to support it. The WindowListener supports the following methods:

windowActivated(WindowEvent)

windowClosed(WindowEvent)

windowClosing(WindowEvent)

windowDeactivated(WindowEvent)

windowDeiconified(WindowEvent)

windowIconified(WindowEvent)

windowOpened(WindowEvent)

Window events

They all take the following form using windowOpened() as an example:

public void **windowOpened(WindowEvent event)** {

    // handle event here}

The windowClosing() and windowClosed() methods are similar however one is called as the window is closing and the other is called after it is closed completely. In windowClosing() we can take action to prevent the window from being closed.

Adapter Classes

As we know, any class that implements an interface must include all its methods, even if it doesn't use it in response to some action.

This often results in the creation of many empty methods when we need to work with an event handling interfaces such as WindowListener (7 methods).

To sort of overcome this limitation, Java gives us **ADAPTERS** which are Java classes that contain empty do-nothing implementations of specific interfaces.

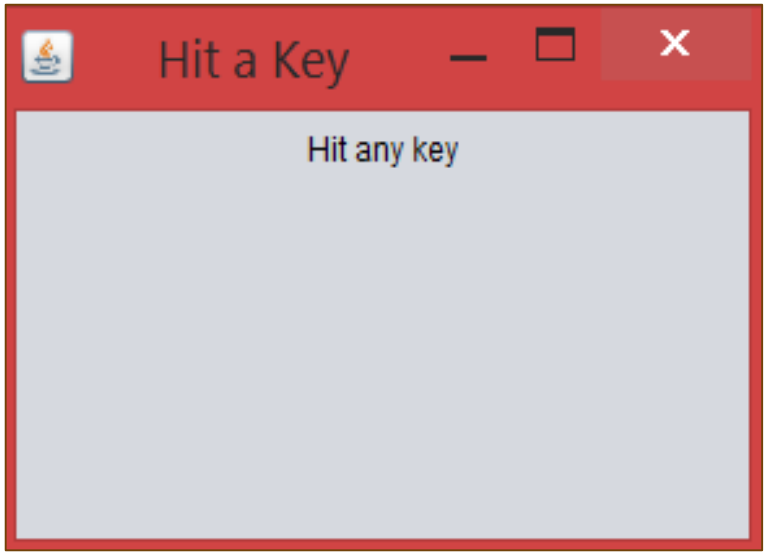
By subclassing an adapter class, we can implement only the event handling methods we need **by overriding those methods**. By doing this, the remainder inherit those do-nothing methods.

The java.awt.event package is where we find F**ocusAdapter, KeyAdapter, MouseAdapter, MouseMotionAdaptor** and **WindowsAdapter** for our use. As we can see, they correspond to the expected listeners for focus, keyboard, mouse, mouse motion and window events.

Exercise 1: KeyChecker.java, P. 260-1

The KeyChecker.java application displays the most recently pressed key by monitoring the keyboard event through a subclass of KeyAdapter.

Using Luna, key in the following code saving it as KeyChecker.java. Once a clean compile is achieved, please run the application ensuring that all paths of execution are tested.



Let's examine the code

Line 31: KeyChecker is implemented as a main class

Lines 37 – 38: KeyMonitor is implemented as a helper class where KeyMonitor is an adapter class for keyboard events that implements the KeyListener interface

Lines 44 – 47: keyTyped() overrides the same method in KeyAdapter which does nothing. When a key is pressed, the key is discovered by calling getKeyChar() of the user event object. This key becomes the text of setLabel in keyChecker.

Inner classes

The challenge in working with use input is to keep the code short and as simple as possible – KISS Theory (Keep it Super Simple).

Always keep in mind that not only programmers but testers who may or may not be programmers may have to examine the code. They have an understanding of basic programming but their job is to test our code.

Since we must implement the all of the methods when we use an event listener, we are forced into a lot of coding. We have some methods such as adapter classes (used in KeyChecker.java) to shorten the code when we need to handle events but there is another method called **INNER CLASSES**

Inner classes

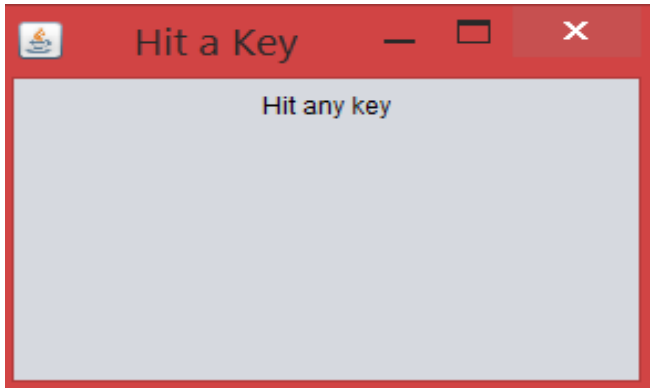
As a refresher, an Inner Class is defined within a class as if they were a method or variable. In the following statement, we create an adapter class in an inner class.

```
KeyAdapter monitor new KeyAdpater() {
        public void keyTyped(KeyEvent event) {
                keyLabel.setText (“ “ + event.getKeyChar());
        }
};
```

Exercise 1: KeyChecker2.java

In this revision of KeyChecker, the KeyAdapter object overrides one method, keyTyped(KeyEvent), to receive the keyboard input. The new version has two advantages over the original – it is shorter since it does not require the creation of a separate class which is less code to maintain and it does not need to make use of the this variable in the inner class in order to change the label on Line 15 since the inner class can access the variables and methods of its own class (*Remember the discussion on the levels of scope?*)

Code the following using Luna. Remember to test all branches including any exceptions. We should end up with the same output as the original KeyChecker application.



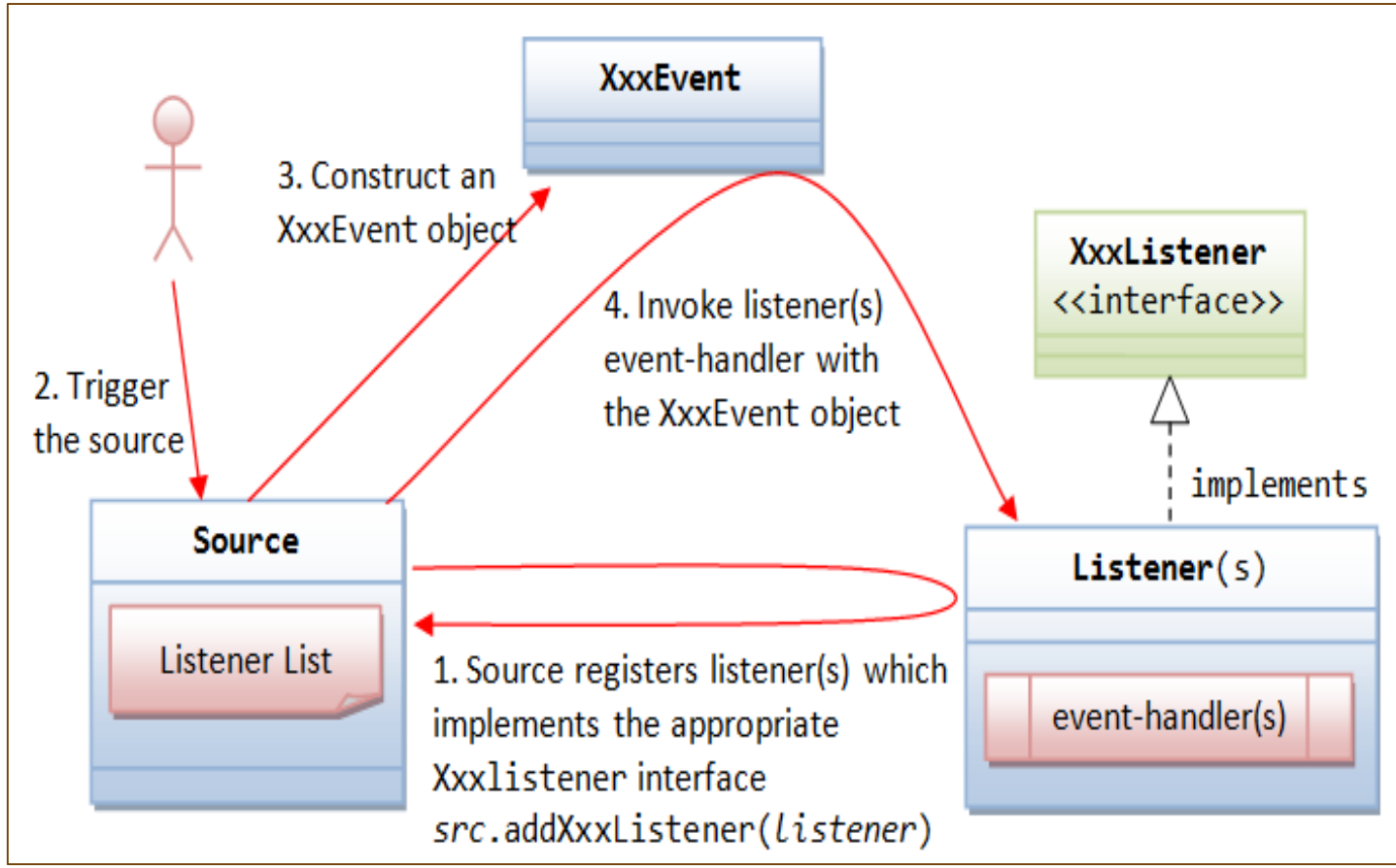
More on inner classes

In addition, an inner class can be anonymous which ultimately the objects of the class that are not assigned to a variable. The anonymous inner class is an object that implements the ActionListener interface.

Doing this, we override the object's actionPerformed() to do some action for the event we are considering. Since each event, (each button on a mouse for example), has its own listener, it is simpler to implement and code than using one listener for multiple components.

SUMMARY

In this chapter we will explore the on how to handle user events. This is one of the most key actions of any program that interacts with a user. We used the following basic steps to achieve our desired results as shown in the graphic:



SUMMARY

A listener interface is added to the class that will contain the event handling methods

---

A listener is added to each component that will generate events

The methods are added with each containing an EventObject as its only argument to the method

Methods of the EventObject class, (i.e. getSource()), are used to learn which components and what kind of event generated

These steps can be used with each of the listener interfaces.

INDEPENDENT EXERCISES – 2 HOURS, P. 264

A. Create an application that uses FocusListener to ensure that a text field's value is multiplied by -1 and is redisplayed whenever a user changes it to a negative value.

B. Create a calculator that adds or subtracts the contents of two text fields whenever the appropriate button is clicked, displaying the result as a label.

Solutions to these exercises can be found in Appendix A – Chapter 11 Independent Exercises