This will cover java GUI event handling concepts + chapters of "Advanced Java Programming" –Unit: 3 (BSc. CSIT, TU)

Event Handling

Normally, a user interacts with an application's GUI to indicate the tasks that the application should perform. For example, when you write an e-mail in an e-mail application, clicking the Send button tells the application to send the e-mail to the specified e-mail addresses. GUIs are event driven. When the user interacts with a GUI component, the interaction—known as an event—drives the program to perform a task. Some common user interactions that cause an application to perform a task include clicking a button, typing in a text field, selecting an item from a menu, closing a window and moving the mouse. The code that performs a task in response to an event is called an event handler, and the overall process of responding to events is known as **event handling.**

Any program that uses GUI (graphical user interface) such as Java application written for windows, is event driven. **Event** describes the change in state of any object. For Example: Pressing a button, entering a character in Textbox, Clicking or Dragging a mouse, etc.

The events can be broadly classified into two categories:

1. **Foreground Events -** Those events which require the direct interaction of user. They are generated as consequences of a person interacting with the graphical components in Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page etc.
2. **Background Events -** Those events that require the interaction of end user are known as background events. Operating system interrupts, hardware or software failure, timer expires, an operation completion are the example of background events.

Delegation Event Model

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism have the code which is known as event handler that is executed

when an event occurs. Java Uses the **Delegation Event Model** to handle the events. This model defines the standard mechanism to generate and handle the events.

The Delegation Event Model has the following key participants namely:

Source - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to it's handler. Java provide as with classes for source object.

Listener - It is also known as event handler. Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received, the listener process the event and then returns.

The benefit of this approach is that the user interface logic is completely separated from the logic that generates the event. The user interface element is able to delegate the processing of an event to the separate piece of code. In this model, Listener needs to be registered with the source object so that the listener can receive the event notification. This is an efficient way of handling the event because the event notifications are sent only to those listener that want to receive them.

Components of Event Handling

Event handling has three main components:

- **Events :** An event is a change in state of an object.
- **Events Source :** Event source is an object that generates an event.
- **Listeners :** A listener is an object that listens to the event. A listener gets notified when an event occurs.

Simply Event Handling includes : **Action→ Event→Listener**

- ✓ ***Action:*** What user does is known as action. Example, a click over button. Here, click is the action performed by the user.
- ✓ ***Event:*** The action done by the component when the user's action takes place is known as event. That is, event is generated (not seen, it is software) against action.

- ✓ ***Listener:*** It is an interface that handles the event. That is, the event is caught by the listener and when caught, immediately executes some method filled up with code. Other way, the method called gives life to the user action.

For more clarity,

- ❖ **Action by user:** Mouse click over a button
- ❖ **Event generated:** ActionEvnet
- ❖ **Listener that handles ActionEvent:** ActionListener
- ❖ **Method called implicitly:** actionPerformed() method.
  The code of actionPerformed() contains the actions of what is to be done (like user name and password validation etc.) when the user clicks a button.

Steps involved in event handling

- ✓ The User clicks the button and the event is generated.
- ✓ Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.
- ✓ Event object is forwarded to the method of registered listener class.
- ✓ The method is now get executed and returns.

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

1. Create a class that represents the event handler and implements an appropriate interface—known as an **event-listener interface**.
2. Indicate that an object of the class from Step 1 should be notified when the event occurs—known as **registering the event handler**.

Points to remember about listener

- In order to design a listener class we have to develop some listener interfaces. These Listener interfaces forecast some public abstract callback methods which must be implemented by the listener class.
- If you do not implement the any if the predefined interfaces then your class can not act as a listener class for a source object.

Callback Methods

These are the methods that are provided by API provider and are defined by the application programmer and invoked by the application developer. Here the callback methods represents an event method. In response to an event java jre will fire **callback method**. All such callback methods are provided in listener interfaces.

If a component wants some listener will listen to it's events the source must register itself to the listener.

Registration Methods

For registering the component with the Listener, many classes provide the registration methods. For example:

Button

public void addActionListener(ActionListener a){}

MenuItem

public void addActionListener(ActionListener a){}

TextField

public void addActionListener(ActionListener a){}

public void addTextListener(TextListener a){}

TextArea

public void addTextListener(TextListener a){}

Checkbox

public void addItemListener(ItemListener a){}

Choice

public void addItemListener(ItemListener a){}

List

public void addActionListener(ActionListener a){}

public void addItemListener(ItemListener a){}

Java Event Handling Code

We can put the event handling code into one of the following places:

- Within class
- Other class
- Inner Class / Anonymous inner class

Example of event handling within a class

```
import javax.swing.*;
import java.awt.event.*;
class EventDemo extends JFrame implements ActionListener{
JTextField tf;
EventDemo(){
  //create components
tf=new JTextField();
tf.setBounds(60,50,170,20);
JButton b=new JButton("click me");
b.setBounds(100,120,80,30);
  //register listener
b.addActionListener(this);//passing current instance
  //add components and set size, layout and visibility
add(b);add(tf);
setSize(300,300);
setLayout(null);
setVisible(true);
}
public void actionPerformed(ActionEvent e){
tf.setText("Welcome");
```

```
}

public static void main(String args[]){

new EventDemo();

}

}
```

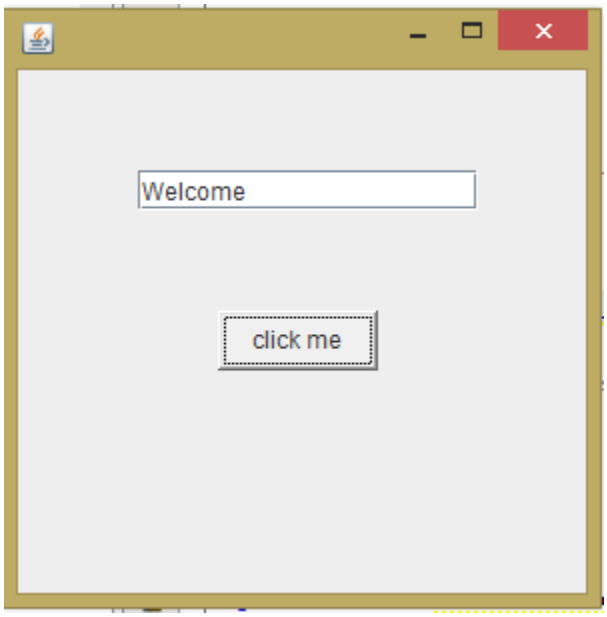

Figure 1: EventHandling Example

Example of event handling by Outer Class

```
package eventhandlingpackage;
import javax.swing.*;

/**
 *
 * @author BIPIN
 */


class AE2 extends JFrame{
```

```
JTextField tf;
AE2(){
//create components
tf=new JTextField();
tf.setBounds(60,50,170,20);
JButton b=new JButton("click me");
b.setBounds(100,120,80,30);
//register listener
Outer o=new Outer(this);
b.addActionListener(o);//passing outer class instance
//add components and set size, layout and visibility
add(b);add(tf);
setSize(300,300);
setLayout(null);
setVisible(true);
}
public static void main(String args[]){
new AE2();
}
}
```

```
package eventhandlingpackage;

import java.awt.event.*;

/**
 *
 * @author BIPIN
 */

class Outer implements ActionListener{
```

```
AE2 obj;

Outer(AE2 obj){

this.obj=obj;

}

public void actionPerformed(ActionEvent e){

obj.tf.setText("welcome");

}

}
```

Example of Event handling by Inner class(Anonymous Class)

```
package eventhandlingpackage;
/**
 *
 * @author BIPIN
 */
import javax.swing.*;
import java.awt.event.*;
class AE3 extends JFrame{
JTextField tf;
AE3(){
tf=new JTextField();
tf.setBounds(60,50,170,20);
JButton b=new JButton("click");
ActionListener ac;
   ac = new ActionListener(){
      public void actionPerformed(ActionEvent e){
         tf.setText("hello");
      }
   };
b.addActionListener(ac);
add(b);add(tf);
```

```
setSize(300,300);

setLayout(null);

setVisible(true);

}

public static void main(String args[]){

new AE3();

}

}
```

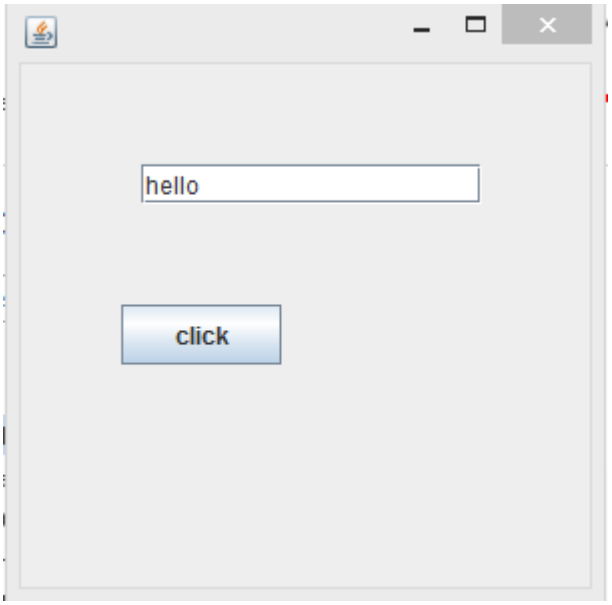

Figure 2: EventHandling Demo

How Events are handled ?

A source generates an Event and send it to one or more listeners registered with the source. Once event is received by the listener, they process the event and then return. Events are supported by a number of Java packages, like **java.util, java.awt** and **java.awt.event.**

Important Event Classes and Interface

Event Classes	**Description**	**Listener Interface**
ActionEvent	generated when button is pressed, menu-item is selected, list-item is double clicked	ActionListener
MouseEvent	generated when mouse is dragged, moved,clicked,pressed or released and also when it enters or exit a component	MouseListener
KeyEvent	generated when input is received from keyboard	KeyListener
ItemEvent	generated when check-box or list item is clicked	ItemListener
TextEvent	generated when value of textarea or textfield is changed	TextListener
MouseWheelEvent	generated when mouse wheel is moved	MouseWheelListener

WindowEvent	generated when window is activated, deactivated, deiconified, iconified, opened or closed	WindowListener
ComponentEvent	generated when component is hidden, moved, resized or set visible	ComponentEventListener
ContainerEvent	generated when component is added or removed from container	ContainerListener
AdjustmentEvent	generated when scroll bar is manipulated	AdjustmentListener
FocusEvent	generated when component gains or loses keyboard focus	FocusListener

Event Classes

The Event classes represent the event. Java provides us various Event classes but we will discuss those which are more frequently used.

EventObject class

It is the root class from which all event state objects shall be derived. All Events are constructed with a reference to the object, the source, that is logically deemed to be the object upon which the Event in question initially occurred upon.This class is defined in java.util package.

Class declaration

Following is the declaration for **java.util.EventObject** class:

```
public class EventObject

   extends Object

      implements Serializable
```

Field

Following are the fields for **java.util.EventObject** class:

protected Object source -- The object on which the Event initially occurred.

Class constructors

S.N.	Constructor & Description
1	EventObject(Object source) Constructs a prototypical Event.

Class methods

S.N.	Method & Description
1	Object getSource() The object on which the Event initially occurred.
2	String toString() Returns a String representation of this EventObject.

Methods inherited

This class inherits methods from the following classes:

java.lang.Object

AWT Event Classes:

Following is the list of commonly used event classes.

Sr. No.	Control & Description
1	AWTEvent It is the root event class for all AWT events. This class and its subclasses supercede the original java.awt.Event class.
2	ActionEvent The ActionEvent is generated when button is clicked or the item of a list is double clicked.
3	InputEvent The InputEvent class is root event class for all component-level input events.
4	KeyEvent On entering the character the Key event is generated.
5	MouseEvent This event indicates a mouse action occurred in a component.
6	TextEvent The object of this class represents the text events.

7	WindowEvent The object of this class represents the change in state of a window.
8	AdjustmentEvent The object of this class represents the adjustment event emitted by Adjustable objects.
9	ComponentEvent The object of this class represents the change in state of a window.
10	ContainerEvent The object of this class represents the change in state of a window.
11	MouseMotionEvent The object of this class represents the change in state of a window.
12	PaintEvent The object of this class represents the change in state of a window.

Event Listeners

The Event listener represent the interfaces responsible to handle events. Java provides us various Event listener classes but we will discuss those which are more frequently used. Every method of an event listener method has a single argument as an object which is subclass of **EventObject** class. For example, mouse event listener methods will accept instance of MouseEvent, where *MouseEvent* derives from *EventObject.*

EventListner interface

It is a marker interface which every listener interface has to extend. This class is defined in *java.util* package.

Following is the declaration for **java.util.EventListener** interface:

```
public interface EventListener
```

AWT Event Listener Interfaces:

Following is the list of commonly used event listeners.

Sr. No.	Control & Description
1	ActionListener This interface is used for receiving the action events.
2	ComponentListener This interface is used for receiving the component events.
3	ItemListener This interface is used for receiving the item events.
4	KeyListener This interface is used for receiving the key events.

5	MouseListener This interface is used for receiving the mouse events.
6	TextListener This interface is used for receiving the text events.
7	WindowListener This interface is used for receiving the window events.
8	AdjustmentListener This interface is used for receiving the adjusmtent events.
9	ContainerListener This interface is used for receiving the container events.
10	MouseMotionListener This interface is used for receiving the mouse motion events.
11	FocusListener This interface is used for receiving the focus events.

What is the problem with listener interfaces?

Actually problem does not occur with every listener, but occurs with a few. A few listeners contain more than one abstract method. For example, WindowListener, used for frame closing, comes with 7 abstract methods. We are interested in only one abstract method to close the frame, but being WindowListener is an interface, we are forced to override remaining 6 methods also, just at least with empty body. This is the only problem, else fine. Some listeners like ActionListener comes with only one abstract method and with them no problem at all.

Adapter Classes

Java **adapter classes** provide the default implementation of listener interfaces. If you inherit the adapter class, you will not be forced to provide the implementation of all the methods of listener interfaces. So it saves code.

Adapters are replacement to listeners. The advantage with adapter is we can override any number of methods we would like and not all. For example, if we use WindowAdapter(instead of WindowListener), we can override only one method to close the frame.

Adapters make event handling simple. Any listener has more than one abstract method has got a corresponding adapter class. For example, MouseListener with 5 abstract methods has got a corresponding adapter known as MouseAdapter. But ActionListener and ItemListener do not have corresponding adapter class as they contain only one abstract method.

Adapters are abstract classes introduced from JDK 1.1.

The adapter classes are found in *java.awt.event, java.awt.dnd* and *javax.swing.event* packages. The Adapter classes with their corresponding listener interfaces are given below.

java.awt.event Adapter classes

Adapter class	Listener interface
WindowAdapter	WindowListener
KeyAdapter	KeyListener

MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
FocusAdapter	FocusListener
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
HierarchyBoundsAdapter	HierarchyBoundsListener

java.awt.dnd Adapter classes

Adapter class	Listener interface
DragSourceAdapter	DragSourceListener
DragTargetAdapter	DragTargetListener

javax.swing.event Adapter classes

Adapter class	Listener interface
MouseInputAdapter	MouseInputListener
InternalFrameAdapter	InternalFrameListener

Listener Interfaces vs Adapter Classes

Any class which handles events must implement a listener interface for the event type it wishes to handle. A class can implement any number of listener interfaces. To implement a listener interface the class must implement every method in the interface. Sometimes you may want to implement only one or two of the methods in an interface. You can avoid having to write your own implementation of all the methods in an interface by using an adapter class. An adapter class is a class that already implements all the methods in its corresponding interface. Your class must extend the adapter class by inheritance so you can extend only one adapter class in your class (Java does not support multiple inheritance). Here are the Java adapter classes and the listener interfaces they implement. Notice that adapters exist for only listener interfaces with more than one method.

LISTENER INTERFACE	CORRESPONDING ADAPTER
WindowListener (7)	WindowAdapter
MouseListener (5)	MouseAdapter
MouseMotionListener (2)	MouseMotionAdapter
KeyListener (3)	KeyAdapter
FocusListener (2)	FocusAdapter

The values in the parentheses indicate the abstract methods available in the listener

Using Action Commands

- Java allows us to associate commands with events caused by AWT buttons and menu items.
- In case of buttons commands are captions of buttons by default.
- But using captions as command is not good idea because when our program changes captions, it is also needed to change event handlers.
- Default commands are changed by using *setActionCommand()* method.
- We can get command for buttons and menu items by using the method getActionCommand().
- We can associate multiple components that are supposed to perform same action with same command.

Example:

```
package eventhandlingpackage;
import javax.swing.*;
import java.awt.event.*;
public class ActionCommandDemo extends JFrame implements ActionListener
{
   JLabel lbl =new JLabel();
   ActionCommandDemo(){
   lbl.setBounds(70,50,180,25);
   setLayout(null);
   JButton btn1 = new JButton("Next");
   btn1.setBounds(50, 120, 80, 30);
   JButton btn2 = new JButton("OK");
   btn2.setBounds(180, 120, 80, 30);
   add(lbl);add(btn1);add(btn2);
   btn1.addActionListener(this);
   btn2.addActionListener(this);
   btn1.setActionCommand("nextStep");
   btn1.setActionCommand("nextStep");
   setSize(300,300);
   setVisible(true);
```

```
 }
public void actionPerformed(ActionEvent e)
{
   if(e.getActionCommand()=="nextStep")
      lbl.setText("Do you want to go to next step?");
    }
public static void main(String[] args){
   new ActionCommandDemo();
}
}
```

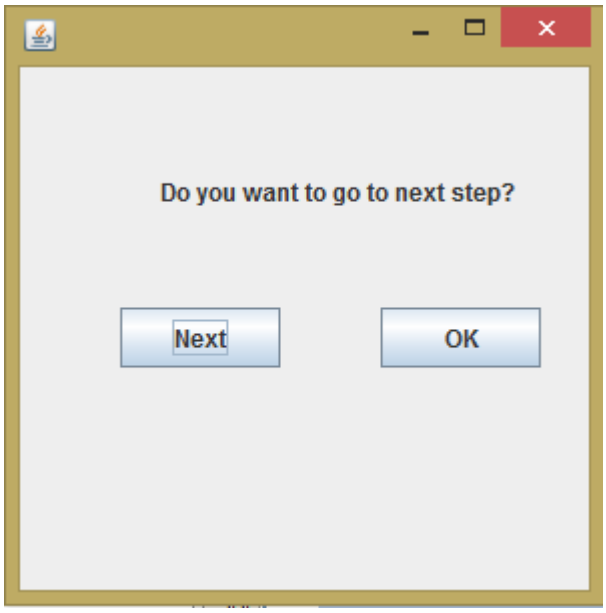

Figure 3: ActionCommandDemo

Handling Different Events

Action Events

A command button generates an ActionEvent when the user clicks it. Command buttons are created with class JButton. The text on the face of a JButton is called a button label.

The action event is generated when button is clicked or the item of a list is double clicked. The object that implements the ActionListners interface gets this ActionEvent when the event occurs and hence must handle the event by actionPerformed() method of ActionListener interface.

Java ActionListener Interface

The Java **ActionListene**r is notified whenever you click on the button or menu item. It is notified against **ActionEvent**. The ActionListener interface is found in **java.awt.event package**. It has only one method: *actionPerformed().*

actionPerformed() method

The **actionPerformed()** method is invoked automatically whenever you click on the registered component.

```
public abstract void actionPerformed(ActionEvent e);
```

Java ActionListener Example: On Button click

```
import java.awt.event.*;
import javax.swing.*;
public class ActionListenerExample {
public static void main(String[] args) {
    JFrame f=new JFrame("ActionListener Example");
    final JTextField tf=new JTextField();
    tf.setBounds(100,100, 300,40);
    JButton b=new JButton("ClickMe");
```

```
    b.setBounds(100,200,120,60);
    b.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e){
          tf.setText("You have just clicked a button!");
    }
    });
    f.add(b);f.add(tf);
    f.setSize(500,500);
    f.setLayout(null);
    f.setVisible(true);
}
}
```

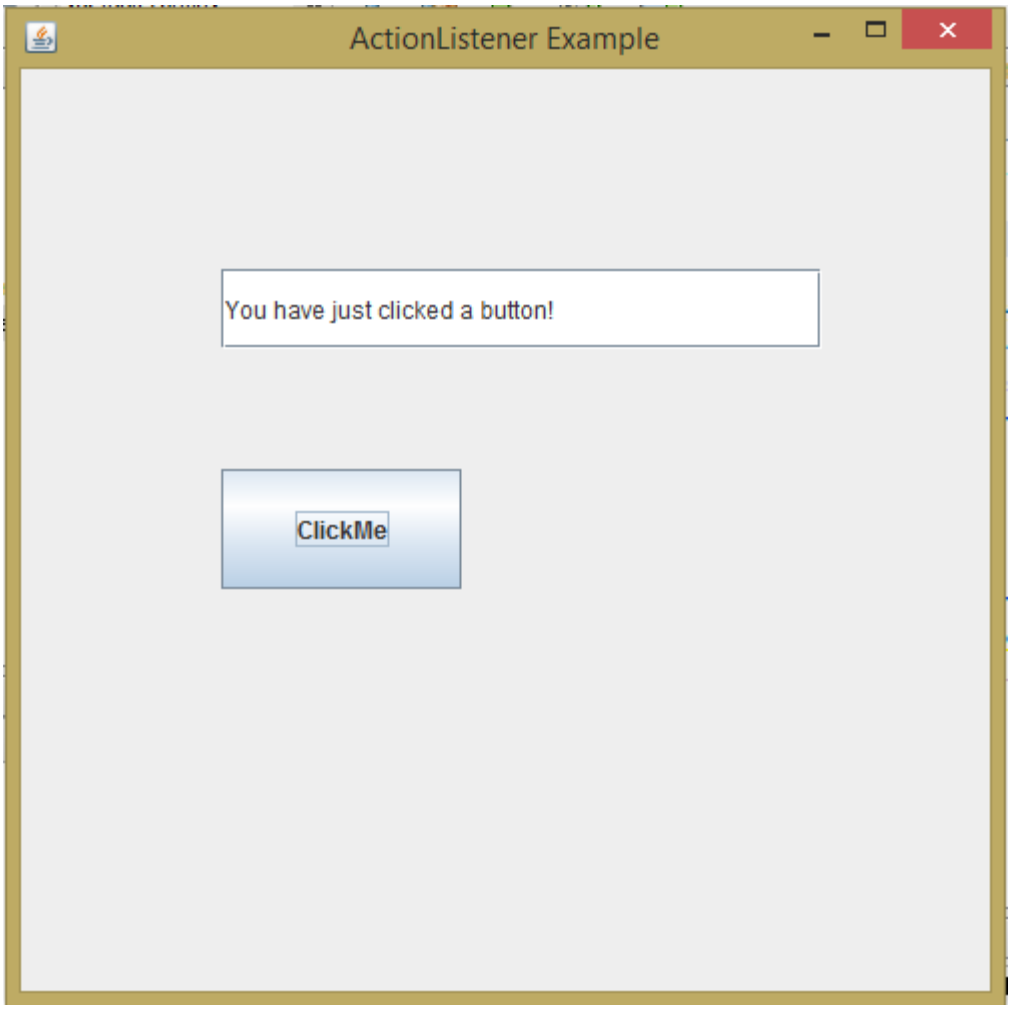

Figure 4: ActionListener on Button Click

Key Events

Keyboard generates **KeyEvent** and handled by **KeyListener** overriding 3 abstract methods.

The KeyListener interface comes with three abstract methods.

```
public abstract void keyPressed(KeyEvent e);
public abstract void keyReleased(KeyEvent e);
public abstract void keyTyped(KeyEvent e);
```

The first one **keyPressed()** is called when a key on the keyboard is pressed and second one **keyReleased()** is called when the key is released and the last one **keyTyped()** is called when a key is typed.

Example

```
package eventhandlingpackage;

import javax.swing.*;

import java.awt.event.*;

/**

 *

 * @author BIPIN

 */

public class KeyListenerExample extends JFrame implements KeyListener{

   JLabel l;

   JTextArea area;

   KeyListenerExample(){

      l=new JLabel();

      l.setBounds(20,50,100,20);

      area=new JTextArea();

      area.setBounds(20,80,400, 100);
```

```
        area.addKeyListener(this);
        setTitle("KeyListnerExample");
        add(l);add(area);
        setSize(500,300);
        setLayout(null);
        setVisible(true);
    }
    public void keyPressed(KeyEvent e) {
        l.setText("Key Pressed");
    }
    public void keyReleased(KeyEvent e) {
        l.setText("Key Released");
    }
    public void keyTyped(KeyEvent e) {
        l.setText("Key Typed");
    }

    public static void main(String[] args) {
        new KeyListenerExample();
    }
}
```

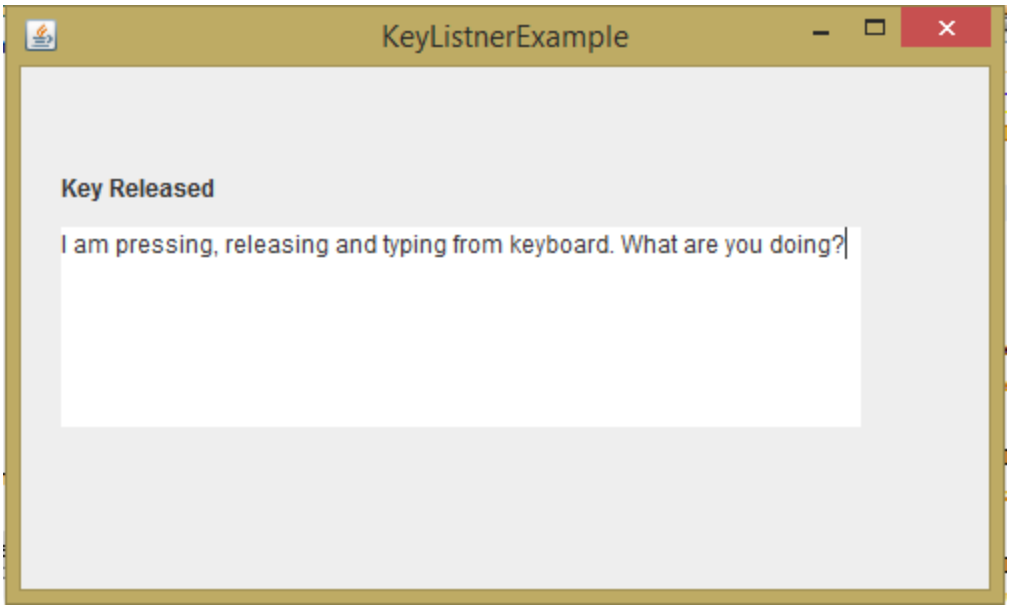

Figure 5: Example of Handling Key Events

Another Java KeyListener Example : Counting Words & Characters

```
package eventhandlingpackage;

import java.awt.Color;

import javax.swing.*;

import java.awt.event.*;

/**

 *

 * @author BIPIN

 */

public class AnotherKeyListener extends JFrame implements KeyListener{

   JLabel l;

   JTextArea area;

   AnotherKeyListener(){
```

```
    l=new JLabel();
    l.setBounds(20,50,400,20);
    l.setForeground(Color.red);
    area=new JTextArea();
    area.setBounds(20,80,400, 100);
    area.addKeyListener(this);
    setTitle("TypedWordCounter");
    add(l);add(area);
    setSize(500,300);
    setLayout(null);
    setVisible(true);
}
public void keyPressed(KeyEvent e) {
    l.setText("pressing...");
}
public void keyReleased(KeyEvent e) {
   String text=area.getText();
    String words[]=text.split("\\s");
    int wlen = words.length;
    l.setText("Words: "+wlen+" Characters:"+text.length());
}
public void keyTyped(KeyEvent e) {
```

```
        l.setText("Typing....");
    }

    public static void main(String[] args) {
        new AnotherKeyListener();
    }
}
```

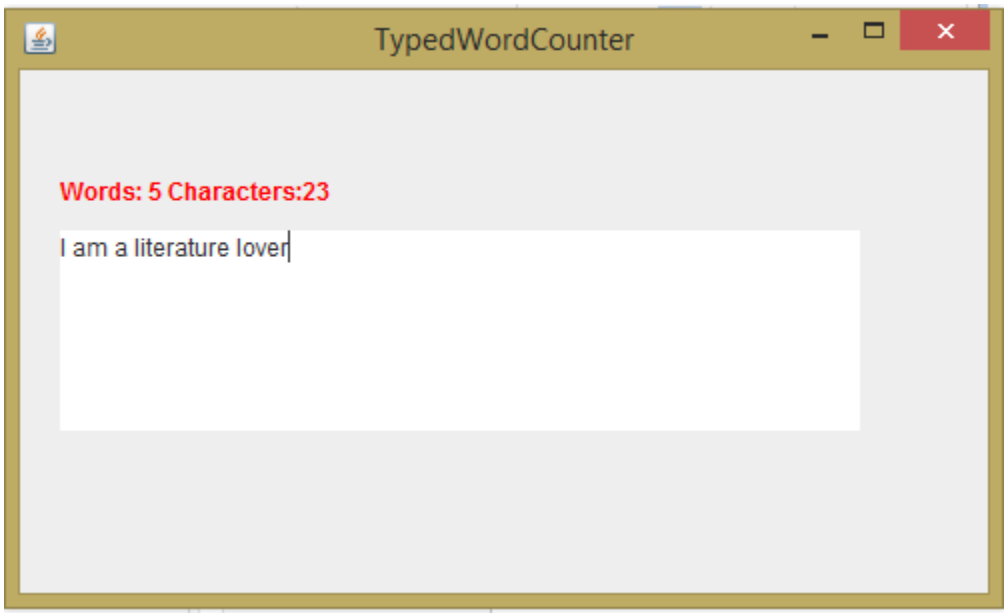

Figure 6: TypedWordCounter with KeyEvent Handling

Focus Events

Focus events occur when any component gains or loses input focus on a graphical user interface. Focus applies to all components that can receive input. **FocusEvent** class is defined in **java.awt.event** package.

To handle a focus event, a class must implement the **FocusListener** interface or extend **FocusAdapter** class. The object that implements the FocusListener interface gets the FocusEvent when the event occurs and hence must handle the event by overriding *focusGained()* and *focusLost()* methods of FocusListener interface. These objects are automatically invoked for a component, which is registered with FocusListener, when focus is gained or lost it respectively.

There are two levels of focus change events: permanent and temporary. Permanent focus change events occur when focus is directly moved from one component to another, such as through calls to requestFocus() or as the user uses the Tab key to traverse components. Temporary focus change events occur when focus is temporarily gained or lost for a component as the indirect result of another operation, such as window deactivation or a scrollbar drag. In this case, the original focus state will automatically be restored once that operation is finished, or, for the case of window deactivation, when the window is reactivated. Both permanent and temporary focus events are delivered using the FOCUS_GAINED and FOCUS_LOST event ids; the levels may be distinguished in the event using the isTemporary() method.

Example:

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


public class FocusEventDemo {
   private JFrame mainFrame;
   private JLabel headerLabel;
   private JLabel statusLabel;
```

```
private JPanel controlPanel;


public FocusEventDemo(){
   prepareGUI();
}
public static void main(String[] args){
   FocusEventDemo  fed = new FocusEventDemo();
   fed.showFocusListenerDemo();
}
private void prepareGUI(){
   mainFrame = new JFrame("Focus Event Example");
   mainFrame.setSize(400,400);
   mainFrame.setLayout(new GridLayout(3, 1));


   headerLabel = new JLabel("",JLabel.CENTER );
   statusLabel = new JLabel("",JLabel.CENTER);
   statusLabel.setSize(350,100);


   mainFrame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent windowEvent){
         System.exit(0);
      }
   });
```

```
    controlPanel = new JPanel();
    controlPanel.setLayout(new FlowLayout());

    mainFrame.add(headerLabel);
    mainFrame.add(controlPanel);
    mainFrame.add(statusLabel);
    mainFrame.setVisible(true);
  }
  private void showFocusListenerDemo(){
    headerLabel.setText("Listener in action: FocusListener");
    JButton okButton = new JButton("OK");
    JButton cancelButton = new JButton("Cancel");

    okButton.addFocusListener(new CustomFocusListener());
    cancelButton.addFocusListener(new CustomFocusListener());

    controlPanel.add(okButton);
    controlPanel.add(cancelButton);
    mainFrame.setVisible(true);
  }
  class CustomFocusListener implements FocusListener{
    public void focusGained(FocusEvent e) {
      statusLabel.setText(statusLabel.getText()
```

```
            + e.getComponent().getClass().getSimpleName() + " gained focus. ");
        }
        public void focusLost(FocusEvent e) {
          statusLabel.setText(statusLabel.getText()
            + e.getComponent().getClass().getSimpleName() + " lost focus. ");
        }
      }
}
```

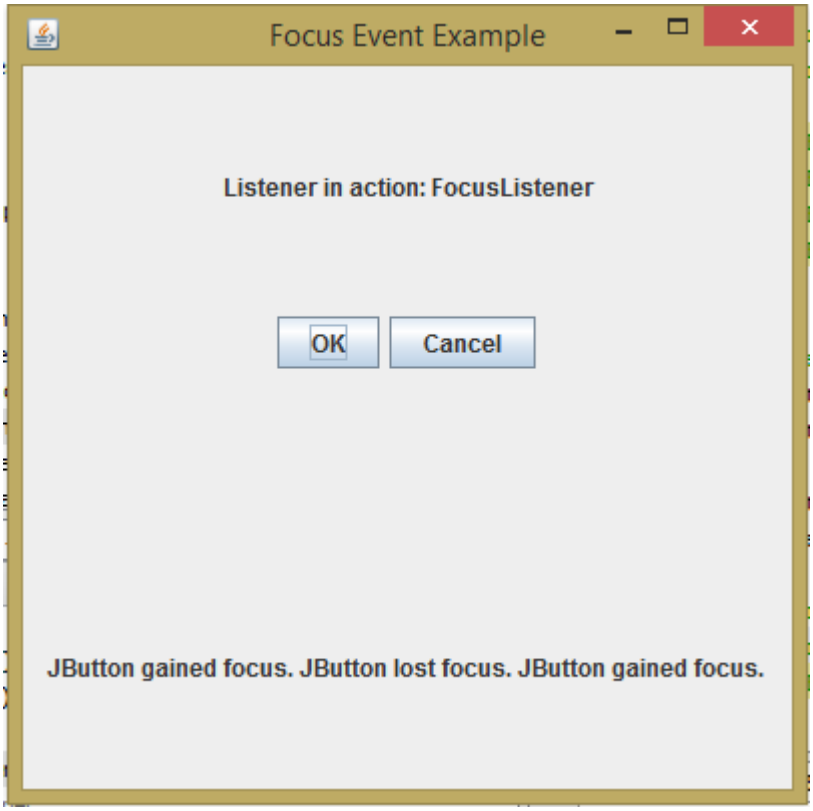

Figure 7: Focus Event Example

Window Events

The Java **WindowListener** is notified whenever you change the state of window. It is notified against **WindowEvent**. The *WindowListener* interface is found in *java.awt.event* package.

- Change of window state means-
  - Minimized , maximized, activate, deactivate, opened, closed etc.
- Window event is generated by components such as JFrame, JInternalFrame, JDialog etc.
- To handle window event, a class must implement the WindowListener interface or extend WindowAdapter calss.
- The object that implements the WindowListener gets this window when the event occurs and hence must handle the event by overriding –

  - `public abstract void windowActivated(WindowEvent e);`
  - `public abstract void windowClosed(WindowEvent e);`
  - `public abstract void windowClosing(WindowEvent e);`
  - `public abstract void windowDeactivated(WindowEvent e);`
  - `public abstract void windowDeiconified(WindowEvent e);`
  - `public abstract void windowIconified(WindowEvent e);`
  - `public abstract void windowOpened(WindowEvent e);`

  methods of WindowListener interface.

Example

```
import javax.swing.*;

import java.awt.event.*;

public class WindowEventDemo extends JFrame implements WindowListener{

   WindowEventDemo(){

      addWindowListener(this);



      setSize(400,400);

      setLayout(null);
```

```
        setVisible(true);
    }

public static void main(String[] args) {
    new WindowEventDemo();
}
public void windowActivated(WindowEvent arg0) {
    System.out.println("activated");
}
public void windowClosed(WindowEvent arg0) {
    System.out.println("closed");
}
public void windowClosing(WindowEvent arg0) {
    System.out.println("closing");
    dispose();
}
public void windowDeactivated(WindowEvent arg0) {
    System.out.println("deactivated");
}
public void windowDeiconified(WindowEvent arg1) {
    System.out.println("deiconified");
}
public void windowIconified(WindowEvent arg2) {
```

```
    System.out.println("iconified");
}
public void windowOpened(WindowEvent arg3) {
    System.out.println("opened");
}
}
```

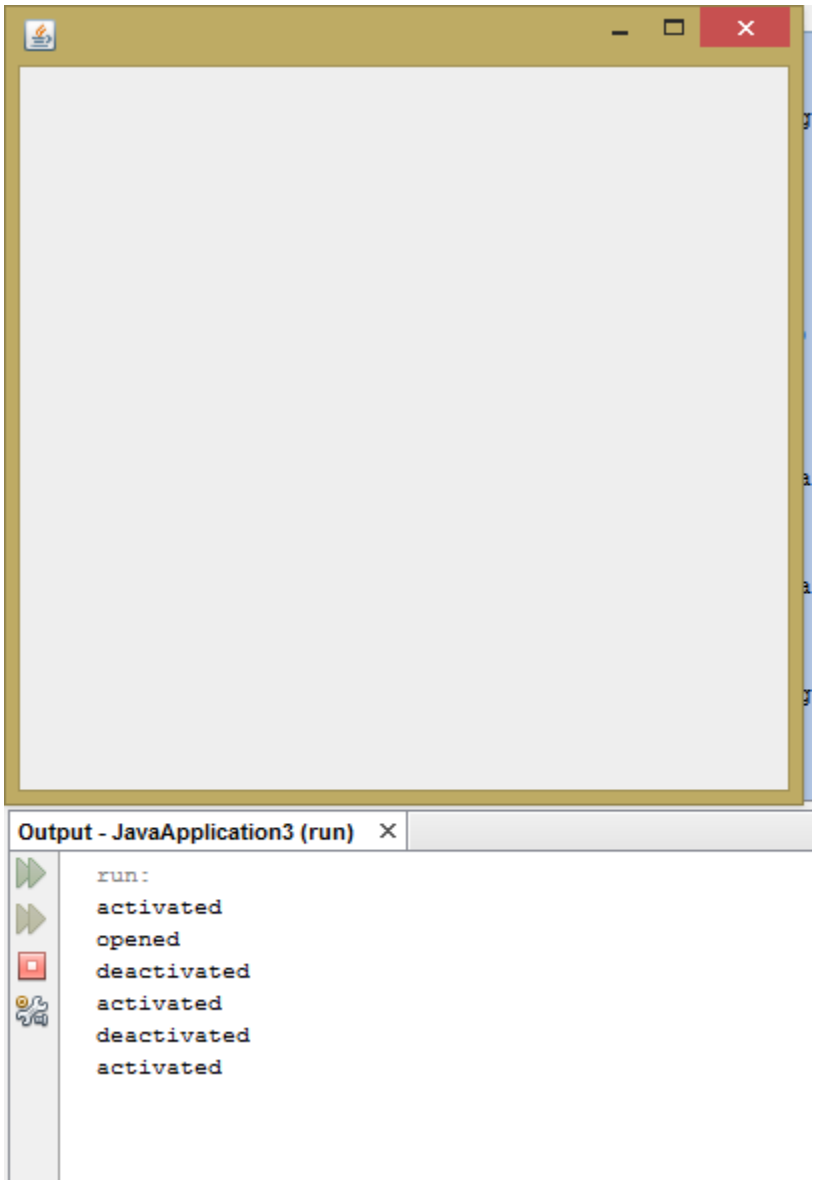

Figure 8: WindowEvent Demo

Mouse Events

An event which indicates that a mouse action occurred in a component. This event is used both for mouse events (click, enter, exit) and mouse motion events (moves and drags).

This low-level event is generated by a component object for:

Mouse Events

- ✓ a mouse button is pressed
- ✓ a mouse button is released
- ✓ a mouse button is clicked (pressed and released)
- ✓ the mouse cursor enters a component
- ✓ the mouse cursor exits a component

Mouse Motion Events

- ✓ the mouse is moved
- ✓ the mouse is dragged

Java MouseListener Interface

The Java MouseListener is notified whenever you change the state of mouse. It is notified against MouseEvent. The *MouseListener* interface is found in *java.awt.event* package. It has five methods.

Methods of MouseListener interface

The signature of 5 methods found in MouseListener interface are given below:

```
public abstract void mouseClicked(MouseEvent e);
public abstract void mouseEntered(MouseEvent e);
public abstract void mouseExited(MouseEvent e);
public abstract void mousePressed(MouseEvent e);
public abstract void mouseReleased(MouseEvent e);
```

Java MouseMotionListener Interface

The Java MouseMotionListener is notified whenever you move or drag mouse. It is notified against MouseEvent. The MouseMotionListener interface is found in java.awt.event package. It has two methods.

Methods of MouseMotionListener interface

The signature of 2 methods found in MouseMotionListener interface are given below:

```
public abstract void mouseDragged(MouseEvent e);
public abstract void mouseMoved(MouseEvent e);
```

Example of MouseListner

```
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.*;
import java.awt.event.*;
public class MouseEventDemo extends JFrame implements MouseListener{
   MouseEventDemo(){
      addMouseListener(this);
      setSize(400,400);
      setLayout(null);
      setVisible(true);
   }
   public void mouseClicked(MouseEvent e) {
      Graphics g=getGraphics();
      g.setColor(Color.RED);
      g.fillOval(e.getX(),e.getY(),20,20);
   }
   public void mouseEntered(MouseEvent e) {}
```

```
    public void mouseExited(MouseEvent e) {}
    public void mousePressed(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}

public static void main(String[] args) {
    new MouseEventDemo();
}
}
```

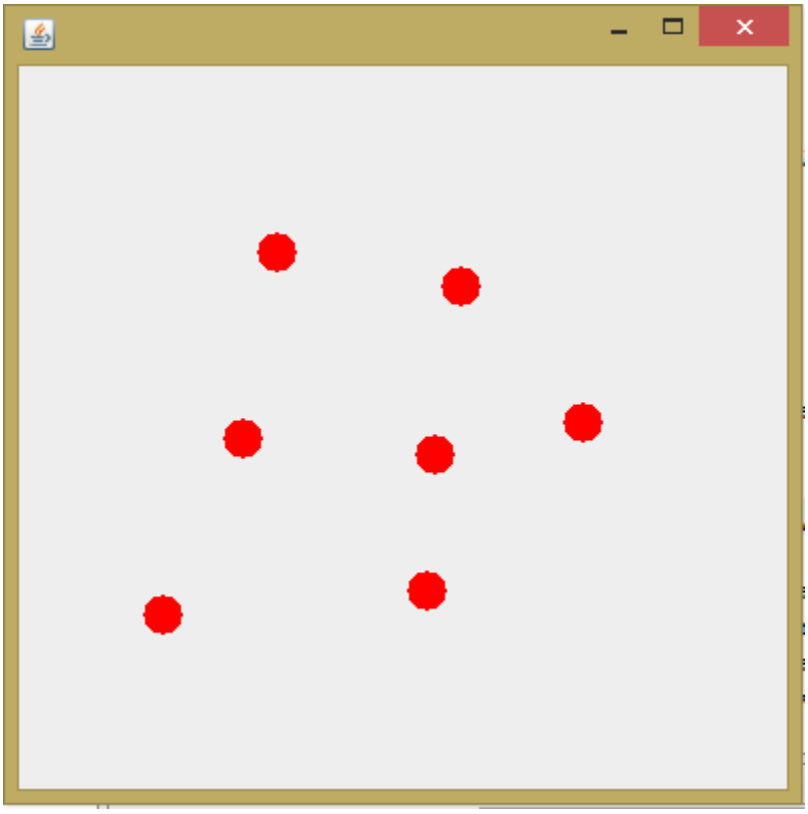

Figure 9: Mouse Event Demo

Mouse Motion Example

```
package eventhandlingpackage;

/**

 *

 * @author BIPIN

 */

import java.awt.*;

import javax.swing.*;

import java.awt.event.*;

public class MouseMotionDemo extends JFrame implements MouseMotionListener{

   MouseMotionDemo(){

      addMouseMotionListener(this);



      setSize(300,300);

      setLayout(null);

      setVisible(true);

   }

public void mouseDragged(MouseEvent e) {

   Graphics g=getGraphics();

   g.setColor(Color.ORANGE);

   g.fillOval(e.getX(),e.getY(),10,10);
```

```
}

public void mouseMoved(MouseEvent e) {}


public static void main(String[] args) {

   new MouseMotionDemo();

}

}
```

Figure 10:Mouse Motion Demo

Item Events

A semantic event which indicates that an item was selected or deselected. This high-level event is generated by an ItemSelectable object (such as a List) when an item is selected or deselected by the user. The event is passed to every ItemListener object which registered to receive such events using the component's addItemListener method.

The object that implements the ItemListener interface gets this ItemEvent when the event occurs. The listener is spared the details of processing individual mouse movements and mouse clicks, and can instead process a "meaningful" (semantic) event like "item selected" or "item deselected".

Java ItemListener Interface

The Java *ItemListener* is notified whenever you click on the checkbox. It is notified against ItemEvent. The ItemListener interface is found in *java.awt.event* package. It has only one method: *itemStateChanged().*

itemStateChanged() method

The *itemStateChanged()* method is invoked automatically whenever you click or unclick on the registered checkbox component.

```
public abstract void itemStateChanged(ItemEvent e);
```

Example

```
import javax.swing.*;
import java.awt.event.*;
public class ItemEventDemo implements ItemListener{
   JCheckBox checkBox1,checkBox2, checkBox3;
   JLabel label;
   ItemEventDemo(){
      JFrame f= new JFrame("CheckBox Item Event Example");
      label = new JLabel();
```

```
    label.setSize(400,100);
    checkBox1 = new JCheckBox("C++");
    checkBox1.setBounds(100,100, 50,50);
    checkBox2 = new JCheckBox("Java");
    checkBox2.setBounds(100,150, 100,100);
    checkBox3 = new JCheckBox("Python");
    checkBox3.setBounds(100,200, 150,150);
    f.add(checkBox1); f.add(checkBox2); f.add(checkBox3);
    f.add(label);
    checkBox1.addItemListener(this);
    checkBox2.addItemListener(this);
    checkBox3.addItemListener(this);
    f.setSize(500,500);
    f.setLayout(null);
    f.setVisible(true);
  }
public void itemStateChanged(ItemEvent e) {
    if(e.getSource()==checkBox1)
        label.setText("C++ Checkbox: "
        + (e.getStateChange()==1?"checked":"unchecked"));
    if(e.getSource()==checkBox2)
    label.setText("Java Checkbox: "
    + (e.getStateChange()==1?"checked":"unchecked"));
```

```
        if(e.getSource()==checkBox3)
        label.setText("Python Checkbox: "
        + (e.getStateChange()==1?"checked":"unchecked"));
    }
public static void main(String args[])
{
   new ItemEventDemo();
}
}
```

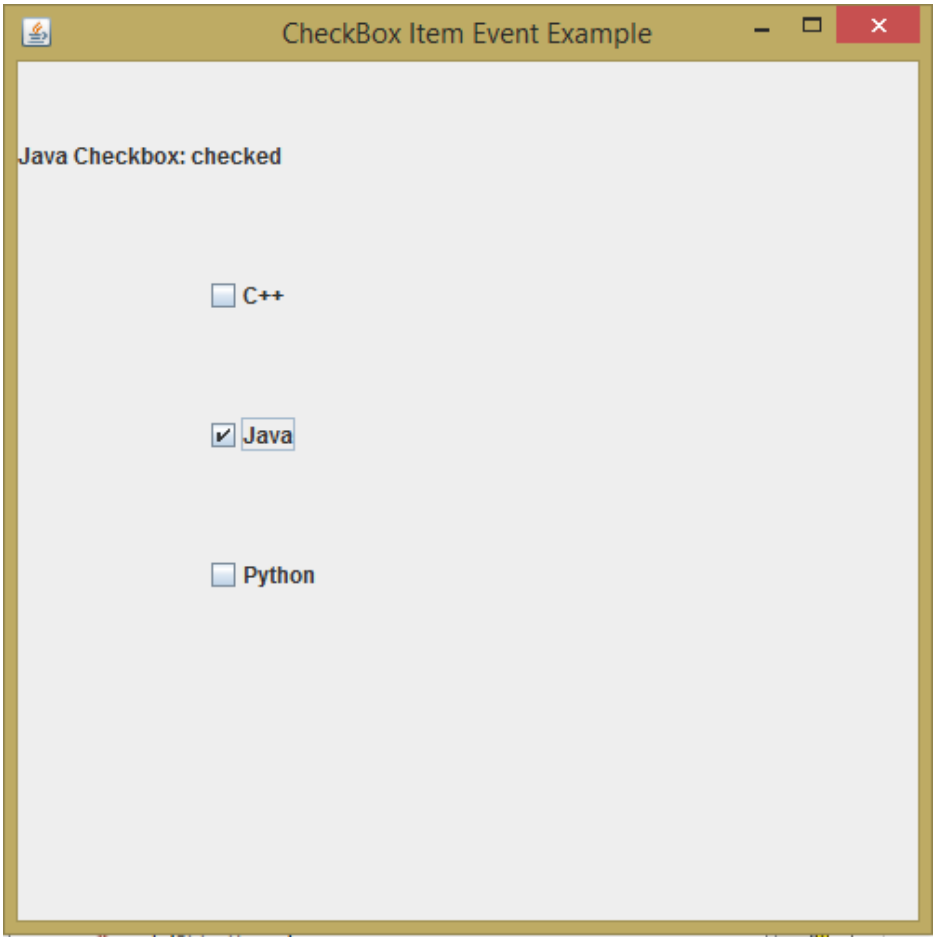

Figure 11: ItemEvent Demo

References

1. javatpoint.com
2. tutorialspoint.com
3. studytonight.com
4. way2java.com
5. “Advanced Java Programming “ by *Arjun Sing Saud*