Unit-3
Event Handling

Introduction

Event

Change in the state of an object is known as ***event*** i.e. event describes the change in state of source. Events are generated as result of user interaction with the graphical user interface components. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page are the activities that causes an event to happen.

Types of Event

The events can be broadly classified into two categories:

- ***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.
- ***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.

Event Handling

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.

Delegation Event Model

The *delegation event model*, which defines standard and consistent mechanisms to generate and process events. Its concept is quite simple: a *source* generates an event and sends it to one or more *listeners*. In this scheme, the listener simply waits until it receives an event. Once received, the listener processes the event and then returns. The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events.

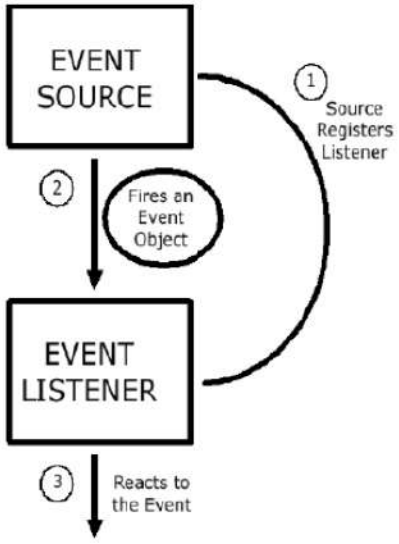
In the delegation event model, listeners need to be registered with the source object so that the listener can receive the event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them.

There are three things that are done in event handling:

1. Create a class that represents the event handler.
2. Implement an appropriate interface called “event listener interface” in the above class.
3. Register the event handler

The delegation event model has three main components:

- ***Events:*** An event is a change of 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.



Event Classes

Event classes are the classes that represent events at the core of java's event handling mechanism. At the root of the Java event class hierarchy is ***EventObject***, which is in ***java.util***. It is the superclass of all events.

The class ***AWTEvent***, defined within the *java.awt* ***package***, is a subclass of ***EventObject***. It is the superclass of all AWT-based events used by delegation event model.

The main classes in ***java.awt.event***:

Event Class	Description
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract superclass for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved.
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Event Listener Interfaces

To handle events we must implement *event listener interfaces*. It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications. The methods that receive and process events are defined in a set of interfaces found in *java.awt.event*.

The list of commonly used event listener interfaces are:

Interface	Description
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved.
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus.
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Handling Action Events

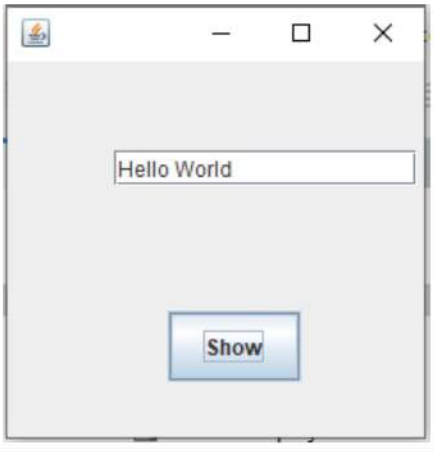
The ***ActionEvent*** is generated when button is clicked or the item of a list is double-clicked. The listener related to this class is ***ActionListener***. To handle an action event, a class must implements the ***ActionListener*** interface.

To have any component listen for an *ActionEvent*, we must register the component with an *ActionListener* as: ***component.addActionListener(new MyActionListener());*** and then write the ***public void actionPerformed(ActionEvent e);*** method for the component to react to the event.

Example

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JTextField;

class EventHandling extends JFrame implements ActionListener
{
   JTextField tf;
   EventHandling ()
   {
      tf = new JTextField ();
      tf.setBounds (60, 50, 170, 20);
      JButton button = new JButton ("Show");
      button.setBounds (90, 140, 75, 40);
      button.addActionListener (this);
      add (button);
      add (tf);
      setSize (250, 250);
      setLayout (null);
      setVisible (true);
   }
   @Override
   public void actionPerformed (ActionEvent e)
   {
      tf.setText ("Hello World");
   }
   public static void main (String args[])
   {
      new EventHandling();
   }
}
```

Output:



Handling Key Events

The ***KeyEvent*** is generated in case of key presses and key depresses on text fields such as *JTextField* and *JTextArea*. One example of *KeyEvent* is user typing in a textfield. The listener associated with this class is ***KeyListener***. To handle a key event, a class must implements the ***KeyListener*** interface.

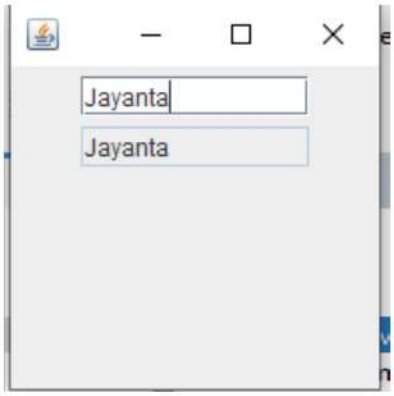
To have any component listen for a *KeyEvent*, we must register the component with *KeyListener* as: ***component.addKeyListener(new MyKeyListener());*** and then write the

public void keyPressed(KeyEvent e);
public void keyTyped(KeyEvent e);
public void keyReleased(KeyEvent e); methods for the component to react to the event.

Example

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class EventHandling extends JFrame {
   JTextField t1 = new JTextField(10);
   JTextField t2 = new JTextField(10);

   public EventHandling() {
      setLayout(new FlowLayout());
      setSize(200, 200);
      setVisible(true);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      add(t1);
      add(t2);
      t2.setEditable(false);

      t1.addKeyListener(new KeyListener() {
         @Override
         public void keyTyped(KeyEvent e) { }

         @Override
         public void keyPressed(KeyEvent e) { }

         @Override
         public void keyReleased(KeyEvent e) {
            String copy = t1.getText();
            t2.setText(copy);
         }
      });
   }
   public static void main(String[] args) {
      new EventHandling();
   }
}
```

Output:



Handling Focus Events

The ***FocusEvent*** is generated when a component gains or loses keyboard focus. The listener for this class is ***FocusListener***. To handle a focus event, a class must implements the ***FocusListener*** interface.

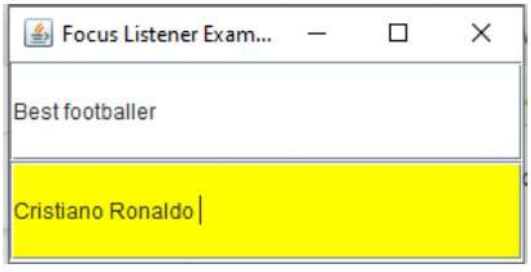
To have any component listen for an *FocusEvent*, we must register the component with an *FocusListener* as: ***component.addFocusListener(new MyFocusListener());*** and then write the

public void focusGained(FocusEvent e);
public void focusLost(FocusEvent e); methods for the component to react to the event.

Example

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

public class EventHandling extends JFrame implements FocusListener {
   private JTextField tf1;
   private JTextField tf2;

   public EventHandling() {
      setTitle("Focus Listener Example");
      setLayout(new GridLayout(2, 1));
      setSize(200, 200);
      tf1 = new JTextField();
      tf2 = new JTextField();
      add(tf1);
      add(tf2);
      tf1.addFocusListener(this);
      tf2.addFocusListener(this);
      setVisible(true);
   }

   public void focusGained(FocusEvent e) {
      if (e.getSource() == tf1) {
         tf1.setBackground(Color.YELLOW);
      } else if (e.getSource() == tf2) {
         tf2.setBackground(Color. YELLOW);
      }
   }

   public void focusLost(FocusEvent e) {
      if (e.getSource() == tf1) {
         tf1.setBackground(Color.WHITE);
      } else if (e.getSource() == tf2) {
         tf2.setBackground(Color.WHITE);
      }
   }

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

Output:



Handling Mouse Events

The ***MouseEvent*** is generated when mouse action (dragged, moved, clicked, pressed, or released) occurred in a component, also generated when the mouse enters or exits a component. The listener for this class is ***MouseListener***. To handle a mouse event, a class must implements the ***MouseListener*** interface.

To have any component listen for a *MouseEvent*, we must register the component with *MouseListener* as: ***component.addMouseListener(new MyMouseListener());*** and then write the

public void mouseClicked(MouseEvent e);
public void mousePressed(MouseEvent e);
public void mouseReleased(MouseEvent e);
public void mouseEntered(MouseEvent e);
public void mouseExited(MouseEvent e); methods for the component to react to the event.

Example

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

public class MouseListenerExample extends JFrame implements MouseListener {
   private JButton button;
   public MouseListenerExample() {
      setTitle("Handling Mouse Listener");
      setSize(300, 150);
      button = new JButton("Click me!");
      add(button);
      button.addMouseListener(this);
      setVisible(true);
   }

   public void mouseClicked(MouseEvent e) {
      button.setText("Clicked!");
   }

   public void mouseEntered(MouseEvent e) {
      button.setBackground(Color.RED);
   }

   public void mouseExited(MouseEvent e) { }

   public void mousePressed(MouseEvent e) { }

   public void mouseReleased(MouseEvent e) { }

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

Handling Window Event

The ***WindowEvent*** is generated whenever we change the state of window. Closing, activating, deactivating, opening, quiting, minimizing, maximizing a window come under this class. Window event is generated by component such as JFrame, JInternalFrame, JDialog etc. To handle a window event, a class must implements the ***WindowListener*** interface.

To have any component listen for a *WindowEvent*, we must register the component with *WindowListener* as: ***component.addWindowListener(new MyWindowListener());*** and then write the
public void windowOpened(WindowEvent e);
public void windowClosing(WindowEvent e);
public void windowClosed(WindowEvent e);
public void windowIconified(WindowEvent e);
public void windowDeiconified(WindowEvent e);
public void windowActivated(WindowEvent e);
public void windowDeactivated(WindowEvent e); methods for the component to react to the event.

Example

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class WindowListenerExample extends JFrame implements WindowListener {
   JLabel label;
   public WindowListenerExample() {
      setTitle("Handling Window Listener");
      setSize(300, 150);
      label = new JLabel("Window opened.");
      add(label);
      addWindowListener(this);
      setVisible(true);
   }
   public void windowOpened(WindowEvent e) {
      label.setText("Window opened.");
   }
   public void windowClosing(WindowEvent e) {
      label.setText("Window closing.");
   }
   public void windowClosed(WindowEvent e) { }
   public void windowIconified(WindowEvent e) {
      label.setText("Window minimized.");
   }
   public void windowDeiconified(WindowEvent e) {
      label.setText("Window restored.");
   }
   public void windowActivated(WindowEvent e) {
      label.setText("Window activated.");
   }
   public void windowDeactivated(WindowEvent e) {
      label.setText("Window deactivated.");
```

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

Handling Item Event

Item event is generated whenever user selects or deselects a selectable object such as radio button, checkbox or list. To handle an item event, a class must implement the ***ItemListener*** interface.

To have any component listen for an *ItemEvent*, we must register the component with an *ItemListener* as: ***component.addItemListener(new MyItemListener());*** and then write the

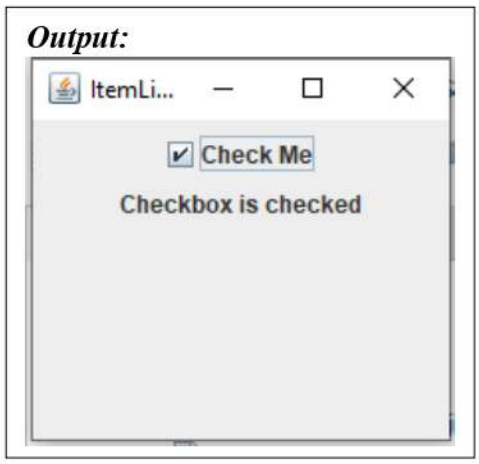
public void itemStateChanged(ItemEvent e); method for the component to react to the event.

The *stateChange* of any *ItemEvent* instance takes one of the following values: *ItemEvent.SELECTED, ItemEvent.DESELECTED*

Example

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class ItemListenerExample implements ItemListener {
    JFrame frame;
    JCheckBox checkBox;
    JLabel label;
    public ItemListenerExample() {
        frame = new JFrame("ItemListener Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        checkBox = new JCheckBox("Check Me");
        label = new JLabel();
        checkBox.addItemListener(this);
        JPanel panel = new JPanel();
        panel.add(checkBox);
        panel.add(label);
        frame.add(panel);
        frame.setSize(200, 200);
        frame.setVisible(true);
    }
    public void itemStateChanged(ItemEvent e) {
        if (e.getStateChange() == ItemEvent.SELECTED) {
            label.setText("Checkbox is checked");
        } else if (e.getStateChange() == ItemEvent.DESELECTED) {
            label.setText("Checkbox is unchecked");
        }
    }
    public static void main(String[] args) {
        new ItemListenerExample();
    }
}
```

Output:



Adapter Classes

Many of the listener interfaces have more than one method. When we implement a listener interface in any class then we must have to implement all the methods declared in that interface because all the methods in an interface are final and must be override in class which implement it. Adapter class makes it easy to deal with this situation.

An adapter class provides empty implementations of all methods defined by that interface. Adapter classes are very useful if we want to override only some of the methods defined by that interface. We can define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which we are interested.

Adapter Class	Listener Interface
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener and (as of JDK 6) MouseMotionListener and MouseWheelListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener, WindowFocusListener, and WindowStateListener

For example, the *KeyListener* interface contains three methods *KeyPressed(), KeyReleased()* and *KeyTyped()*. If we implement this interface, we have to give implementation of all these three methods. However, by using the *KeyAdapter* class, we can override only the method that is needed. This can be shown in the code example given below:

Using KeyListener

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class KeyListenerExample extends JFrame implements KeyListener {
   JLabel outputLabel;
   public KeyListenerExample() {
      setTitle("KeyListener Example");
      setSize(400, 200);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setLayout(new FlowLayout());
      outputLabel = new JLabel("Press a key");
      add(outputLabel);
      addKeyListener(this);
      setVisible(true);
   }
   @Override
   public void keyPressed(KeyEvent e) {
      outputLabel.setText("Key Pressed: " + e.getKeyChar());
   }
   @Override
   public void keyReleased(KeyEvent e) {
```

```
        outputLabel.setText("Key Released: " + e.getKeyChar());
    }
    @Override
    public void keyTyped(KeyEvent e) {}

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

Using KeyAdapter

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

public class KeyAdapterExample extends JFrame {
    JLabel outputLabel;

    public KeyAdapterExample() {
        setTitle("KeyAdapter Example");
        setSize(400, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        outputLabel = new JLabel("Press a key");
        add(outputLabel);
        addKeyListener(new CustomKeyAdapter());
        setVisible(true);
    }

    class CustomKeyAdapter extends KeyAdapter {
        @Override
        public void keyPressed(KeyEvent e) {
            outputLabel.setText("Key Pressed: " + e.getKeyChar());
        }

        @Override
        public void keyReleased(KeyEvent e) {
            outputLabel.setText("Key Released: " + e.getKeyChar());
        }
    }
    public static void main(String[] args) {
        new KeyAdapterExample();
    }
}
```

Event listener interface vs. Event adapter class

In an ***adapter class***, there is no need for implementation of all the methods presented in an interface. It is used when only some methods of defined by its interface have to be overridden.

In a ***listener***, all the methods that have been declared in its interface have to be implemented. This is because these methods are final.

Q. Write a program using swing components to add two numbers. Use text fields for inputs and output. Your program should display the result when the user presses a button.

Solution:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
class Addition extends JFrame implements ActionListener{
 JLabel l1, l2;
 JTextField t1, t2, t3;
 JButton b1;
 public Addition()
 {
  l1 = new JLabel("First Number:");
  l1.setBounds(20, 10, 100, 20);        //x, y, width, height
  t1 = new JTextField(10);
  t1.setBounds(120, 10, 100, 20);
  l2 = new JLabel("Second Number:");
  l2.setBounds(20, 40, 100, 20);
  t2 = new JTextField(10);
  t2.setBounds(120, 40, 100, 20);
  b1 = new JButton("Sum");
  b1.setBounds(20, 70, 80, 20);
  t3 = new JTextField(10);
  t3.setBounds(120, 70, 100, 20);
  add(l1);
  add(t1);
  add(l2);
  add(t2);
  add(b1);
  add(t3);
  b1.addActionListener(this);           //Registering event
  setSize(400,300);
  setLayout(null);
  setVisible(true);
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
 @Override
 public void actionPerformed(ActionEvent e)
 {
   if(e.getSource()==b1){
       int num1 = Integer.parseInt(t1.getText());
       int num2 = Integer.parseInt(t2.getText());
       int sum = num1 + num2;
       t3.setText(String.valueOf(sum));
   }
 }
 public static void main(String args[])
 {
```

```
   new Addition();
 }
}
```

Q. Write a Java program to find the sum of two numbers using swing components. Use text fields for input and output. Your program displays output if you press any key in keyboard. Use key adapter to handle events.

Solution:

```
import javax.swing.*;
import java.awt.GridLayout;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;

public class Addition
{
 JLabel l1, l2, l3;
 JTextField t1, t2, t3;
 JFrame f = new JFrame();

 Addition()
 {
  l1 = new JLabel("First Number:");
  t1 = new JTextField();
  l2 = new JLabel("Second Number:");
  t2 = new JTextField();
  l3 = new JLabel("Press any key");
  t3 = new JTextField();
  t3.addKeyListener(new keychecker());
  f.add(l1);
  f.add(t1);
  f.add(l2);
  f.add(t2);
  f.add(l3);
  f.add(t3);
  f.setSize(250,150);
  f.setLayout(new GridLayout(3,2));
  f.setLocationRelativeTo(null);
  f.setVisible(true);
  f.setDefaultCloseOperation(f.EXIT_ON_CLOSE);
}
class keychecker extends KeyAdapter{
   public void keyPressed(KeyEvent e)
   {
       int num1 = Integer.parseInt(t1.getText());
       int num2 = Integer.parseInt(t2.getText());
       int sum = num1 + num2;
       JOptionPane.showMessageDialog(f, "The sum is" +sum);
       t3.setText(null);
   }
 }
```

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

Q. Write a GUI program using swing component to find number of words and characters in a TextArea.

Solution

```
import javax.swing.*;
import java.awt.event.*;
public class TextAreaExample implements ActionListener{
JLabel l1,l2;
JTextArea area;
JButton b;
TextAreaExample() {
   JFrame f= new JFrame();
   l1=new JLabel();
   l1.setBounds(50,25,100,30);
   l2=new JLabel();
   l2.setBounds(160,25,100,30);
   area=new JTextArea();
   area.setBounds(20,75,250,200);
   b=new JButton("Count Words");
   b.setBounds(100,300,120,30);
   b.addActionListener(this);
   f.add(l1);
  f.add(l2);
   f.add(area);
 f.add(b);
 f.setSize(450,450);
 f.setLayout(null);
 f.setVisible(true);
}
public void actionPerformed(ActionEvent e){
   String text = area.getText();
   String words[]=text.split("\\s");
   l1.setText("Words: "+ words.length);
   l2.setText("Characters: "+text.length());
}
public static void main(String[] args) {
   new TextAreaExample();
}
}
```

Q. Write a program using swing components to find simple interest. Use text fields for inputs and output. Your program should display the result when the user presses a button.

Solution:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
class SimpleInterest extends JFrame implements ActionListener
{
 JLabel l1, l2, l3;
 JTextField t1, t2, t3, t4;
 JButton b1;
 public SimpleInterest()
 {
  l1 = new JLabel("Principal:");
  l1.setBounds(20, 10, 100, 20);        //x, y, width, height
  t1 = new JTextField(10);
  t1.setBounds(120, 10, 100, 20);
  l2 = new JLabel("Time:");
  l2.setBounds(20, 40, 100, 20);
  t2 = new JTextField(10);
  t2.setBounds(120, 40, 100, 20);
  l3 = new JLabel("Rate:");
  l3.setBounds(20, 70, 100, 20);
  t3 = new JTextField(10);
  t3.setBounds(120, 70, 100, 20);
  b1 = new JButton("Simple Interest");
  b1.setBounds(20, 100, 80, 20);
  t4 = new JTextField(10);
  t4.setBounds(120, 100, 100, 20);
  add(l1);
  add(t1);
  add(l2);
  add(t2);
  add(l3);
```

```
  add(t3);
  add(b1);
  add(t4);
  b1.addActionListener(this);          //Registering event
  setSize(400,300);
  setLayout(null);
  setVisible(true);
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
 @Override
 public void actionPerformed(ActionEvent e)
 {
   if(e.getSource()==b1){
      double P = Double.parseDouble(t1.getText());
      double T = Double.parseDouble(t2.getText());
      double R = Double.parseDouble(t3.getText());
      double SI = (P*T*R)/100;
      t4.setText(String.valueOf(SI));
  }
 }
 public static void main(String args[])
 {
   new SimpleInterest() ;
 }
}
```

Q. Write a GUI program using components to find sum and difference of two numbers. Use two text fields for giving input and a label for output. The program should display sum if user presses mouse and difference if user release mouse.

Solution:

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

public class SumAndDifference extends JFrame {
   JTextField num1Field, num2Field;
   JLabel outputLabel;
```

```
    public SumAndDifference() {
        setTitle("Sum and Difference");
        setSize(400, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new GridLayout(3, 2));
        JLabel num1Label = new JLabel("Number 1:");
        num1Field = new JTextField();
        JLabel num2Label = new JLabel("Number 2:");
        num2Field = new JTextField();
        outputLabel = new JLabel();
        add(num1Label);
        add(num1Field);
        add(num2Label);
        add(num2Field);
        add(outputLabel);

        num1Field.addMouseListener(new handleSumDiff());
        num2Field.addMouseListener(new handleSumDiff());

        setVisible(true);
    }
    class handleSumDiff extends MouseAdapter{
        @Override
        public void mousePressed(MouseEvent e) {
            double num1 = Double.parseDouble(num1Field.getText());
            double num2 = Double.parseDouble(num2Field.getText());
            double sum = num1 + num2;
            outputLabel.setText("Sum: " + sum);
        }

        @Override
        public void mouseReleased(MouseEvent e) {
            double num1 = Double.parseDouble(num1Field.getText());
            double num2 = Double.parseDouble(num2Field.getText());
            double difference = num1 - num2;
            outputLabel.setText("Difference: " + difference);
        }
    }

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

Q. Design a GUI form using swing with a text field, a text label for displaying the input message "Input any string", and three buttons with caption "Check Palindrome", "Reverse", "Find Vowels". Write a complete program for above scenario and for checking palindrome in first button, reverse it after clicking second button and extract the vowels from it after clicking third button.

Solution:

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

public class SwingExample extends JFrame
{
   public SwingExample()
   {
      setLayout(new GridLayout(4,1,10,20));
      JLabel inputLabel = new JLabel("Input any String: ");
      JTextField inputTextField = new JTextField(20);
      add(inputLabel);
      add(inputTextField);

      JLabel outputLabel = new JLabel("Output: ");
      JTextField outputTextField = new JTextField(20);
      add(outputLabel);
      add(outputTextField);
      outputTextField.setEditable(false);

      JButton checkPalindromeButton = new JButton("Check Palindrome");
      add(checkPalindromeButton);

      JButton reverseButton = new JButton("Reverse");
      add(reverseButton);

      JButton findVowelButton = new JButton("Find Vowel");
      add(findVowelButton);

      checkPalindromeButton.addActionListener(new ActionListener()
      {
         @Override
         public void actionPerformed(ActionEvent e)
         {
            String copyUserInput="";
            String userInput = inputTextField.getText();
            int length = userInput.length();

            for (int i = length-1; i>=0; i-- )
            {
               copyUserInput = copyUserInput + userInput.charAt(i);
            }
            if (copyUserInput.equalsIgnoreCase(userInput))
            {
```

```
            outputTextField.setText("String is palindrome.");
        }
        else
        {
            outputTextField.setText("String isn't a palindrome.");
        }
    }
});

reverseButton.addActionListener(new ActionListener()
{
    @Override
    public void actionPerformed(ActionEvent e)
    {
        String reverseUserInput="";
        String userInput = inputTextField.getText();
        int length = userInput.length();

        for (int i = length-1; i>=0; i-- )
        {
            reverseUserInput = reverseUserInput + userInput.charAt(i);
        }

            outputTextField.setText("Reverse String is: "+ reverseUserInput);
    }
});

findVowelButton.addActionListener(new ActionListener()
{
    @Override
    public void actionPerformed(ActionEvent e)
    {
        char[] vowel={'a','e','i','o','u','A','E','I','O','U'};


        String userInput = inputTextField.getText();
        int length = userInput.length();
        char[] extractedVowel= new char[length];
        String showVowel="";
        for (int i =0; i<=length-1; i++ )
        {
            for (int j = 0; j<=vowel.length-1; j++)
            {
                if(userInput.charAt(i)== vowel[j])
                {
                    extractedVowel[i] = userInput.charAt(i);
                    showVowel = showVowel + String.valueOf(extractedVowel[i]);
                }
            }
        }
```

```
            outputTextField.setText("Vowels: "+showVowel);
        }
      });
      pack();
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      setVisible(true);
   }
   public static void main(String[] args)
   {
      new SwingExample();
   }
}
```

Please let me know if I missed anything or anything is incorrect.

poudeljayanta99@gmail.com