**Event Handling**

*Delegation Event Model*, 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. A user interface element is able to “delegate” the processing of an event to a separate piece of code.

In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them.

**Events**

In the delegation model, an *event* is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface. Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse.

Events may also occur that are not directly caused by interactions with a user interface. For example, an event may be generated when a timer expires, a counter exceeds a value, software or hardware failure occurs, or an operation is completed.

**Event Sources**

A *source* is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event. A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method.

Here is the general form:

public void add*Type*Listener(*Type*Listener *el*)

Here, *Type* is the name of the event and *el* is a reference to the event listener. For example, the method that registers a keyboard event listener is called **addKeyListener( )**. The method that registers a mouse motion listener is called **addMouseMotionListener( )**.

When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as *multicasting* the event. In all cases, notifications are sent only to listeners that register to receive them.

Some sources may allow only one listener to register. The general form of such a method is this:

public void add*Type*Listener(*Type*Listener *el*)

throws java.util.TooManyListenersException

A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this:

public void remove*Type*Listener(*Type*Listener *el*)

Some examples of Event Sources are:

|  |  |
| --- | --- |
| Button | Generates action events when the button is pressed |
| Checkbox | Generates item events when the check box is selected or deselected |
| Choice | Generates item events when the choice is changed. |
| List | Generates action events when an item is double-clicked; generates item events when an item is selected or deselected. |
| MenuItem | Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected |
| Scrollbar | Generates adjustment events when the scroll bar is manipulated |
| Text Components | Generates text events when the user enters a character. |
| Window | Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit. |

**Event Listeners**

A *listener* is an object that is notified when an event occurs. 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**. For example, the **MouseMotionListener** interface defines two methods to receive notifications when the mouse is dragged or moved.

Some examples of Event Listener Interfaces are:

|  |  |
| --- | --- |
| ActionListener | Defines one method to receive action events. |
| 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. |
| TextListener | Defines one method to recognize when a text value changes |
| WindowListener | Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit. |

**The ActionListener Interface**

This interface defines the actionPerformed( ) method that is invoked when an action event occurs. Its general form is shown here:

void actionPerformed(ActionEvent ae)

**The ItemListener Interface**

This interface defines the itemStateChanged( ) method that is invoked when the state of an item changes. Its general form is shown here:

void itemStateChanged(ItemEvent ie)

**The KeyListener Interface**

This interface defines three methods. The keyPressed( ) and keyReleased( ) methods are invoked when a key is pressed and released, respectively. The keyTyped( ) method is invoked when a character has been entered.

For example, if a user presses and releases the A key, three events are generated in sequence: key pressed, typed, and released. If a user presses and releases the HOME key, two key events are generated in sequence: key pressed and released.The general forms of these methods are shown here:

void keyPressed(KeyEvent ke)

void keyReleased(KeyEvent ke)

void keyTyped(KeyEvent ke)

**The MouseListener Interface**

This interface defines five methods. If the mouse is pressed and released at the same point, mouseClicked( ) is invoked. When the mouse enters a component, the mouseEntered( ) method is called. When it leaves, mouseExited( ) is called. The mousePressed( ) and mouseReleased( ) methods are invoked when the mouse is pressed and released, respectively.The general forms of these methods are shown here:

void mouseClicked(MouseEvent me)

void mouseEntered(MouseEvent me)

void mouseExited(MouseEvent me)

void mousePressed(MouseEvent me)

void mouseReleased(MouseEvent me)

**The MouseMotionListener Interface**

This interface defines two methods. The mouseDragged( ) method is called multiple times as the mouse is dragged. The mouseMoved( ) method is called multiple times as the mouse is moved. Their general forms are shown here:

void mouseDragged(MouseEvent me)

void mouseMoved(MouseEvent me)

**The TextListener Interface**

This interface defines the textChanged( ) method that is invoked when a change occurs in a text area or text field. Its general form is shown here:

void textChanged(TextEvent te)

**The WindowListener Interface**

This interface defines seven methods. The windowActivated( ) and windowDeactivated( ) methods are invoked when a window is activated or deactivated, respectively. If a window is iconified, the windowIconified( ) method is called. When a window is deiconified, the windowDeiconified( ) method is called. When a window is opened or closed, the windowOpened( ) or windowClosed( ) methods are called, respectively. The windowClosing( ) method is called when a window is being closed. The general forms of these methods are:

void windowActivated(WindowEvent we)

void windowClosed(WindowEvent we)

void windowClosing(WindowEvent we)

void windowDeactivated(WindowEvent we)

void windowDeiconified(WindowEvent we)

void windowIconified(WindowEvent we)

void windowOpened(WindowEvent we)

**Event Classes**

The classes that represent events are at the core of Java’s event handling mechanism. Thus, we begin our study of event handling with a tour of the event classes. As you will see, they provide a consistent, easy-to-use means of encapsulating events.

At the root of the Java event class hierarchy is **EventObject**, which is in **java.util**. It is the superclass for all events. Its one constructor is shown here:

EventObject(Object *src*)

Here, *src* is the object that generates this event.

**EventObject** contains two methods: **getSource( )** and **toString( )**. The **getSource( )** method returns the source of the event. Its general form is shown here:

Object getSource( )

As expected, **toString( )** returns the string equivalent of the event.

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

Some examples of Event classes are:

|  |  |
| --- | --- |
| ActionEvent | Generated when a button is pressed, a list item is double-clicked, or a menu item is selected |
| 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 |
| 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 |
| 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. |

**Frame Programs with GUI**

**Simple Frame Program**

import java.awt.Frame;

import java.awt.event.\*;

import javax.swing.\*;

public class MyFrame1 extends Frame

{

MyFrame1(String title)

{

//call the superclass constructor

super(title); // This will set the title of frame

//set window title using setTitle method

//this.setTitle(title);

setSize(300,300);

//Newly created window will not be displayed until we call setVisible(true)

this.setVisible(true);

}

public static void main(String args[])

{

MyFrame1 window = new MyFrame1("My First Frame");

}

}

**Frame with Window Closing**

import java.awt.Frame;

import java.awt.event.\*;

public class MyFrameWindowClose extends Frame implements WindowListener

{

MyFrameWindowClose(String title)

{

//super(title);

//Register the Listeners

addWindowListener(this);

//set window title using setTitle method

this.setTitle(title);

setSize(300,300);

//Newly created window will not be displayed until we call setVisible(true)

this.setVisible(true);

}

public void windowClosing(WindowEvent e)

{

//hide the window when window's close button is clicked

this.setVisible(false);

dispose();

//System.exit(0);

}

public void windowActivated(WindowEvent we){}

public void windowClosed(WindowEvent we){}

public void windowDeactivated(WindowEvent we){}

public void windowDeiconified(WindowEvent we){}

public void windowIconified(WindowEvent we){}

public void windowOpened(WindowEvent we){}

public static void main(String args[])

{

MyFrameWindowClose window = new MyFrameWindowClose("My First Frame");

}

}

**Frame with Adapter Class Window Closing**

import java.awt.Frame;

import java.awt.event.\*;

public class MyFrameWindowCloseAdapter extends Frame

{

MyFrameWindowCloseAdapter(String title)

{

//super(title);

//Register the Listeners

addWindowListener(new MyWindowAdapter(this));

//set window title using setTitle method

this.setTitle(title);

setSize(300,300);

//Newly created window will not be displayed until we call setVisible(true)

this.setVisible(true);

}

public static void main(String args[])

{

MyFrameWindowCloseAdapter window = new MyFrameWindowCloseAdapter("My First Frame");

}

}

class MyWindowAdapter extends WindowAdapter

{

MyFrameWindowCloseAdapter myWindow = null;

MyWindowAdapter(MyFrameWindowCloseAdapter myWindow){

this.myWindow = myWindow;

}

//implement windowClosing method

public void windowClosing(WindowEvent e) {

//hide the window when window's close button is clicked

myWindow.setVisible(false);

myWindow.dispose();

}

}

**Addition of two numbers using a Frame**

import java.awt.\*;

import java.awt.event.\*;

public class AdditionFrame extends Frame implements ActionListener

{

// reference variables

TextField tf1, tf2, tf3; // 3 text fields

Button pb, mb1, mulb2, rmb3, db, eb; // 6 buttons

public AdditionFrame ( )

{

// setting the layout

setLayout(new GridLayout(6, 2, 25, 25));

setBackground(Color.cyan);

// creating objects

tf1 = new TextField(10);

tf2 = new TextField(10);

tf3 = new TextField(10);

pb = new Button("+");

mb1 = new Button("-");

mulb2 = new Button("\*");

rmb3 = new Button("%");

db = new Button("/");

eb = new Button("EXIT");

// event handling

pb.addActionListener(this);

mb1.addActionListener(this);

mulb2.addActionListener(this);

rmb3.addActionListener(this);

db.addActionListener(this);

eb.addActionListener(this);

// beautification

eb.setForeground(Color.red);

tf3.setEditable(false);

tf3.setFont(new Font("Serif", Font.BOLD,20));

// adding components, order is important

add(new Label("Enter 1st Number"));

add(tf1);

add(new Label("Enter 2nd Number"));

add(tf2);

add(pb); // adding buttons one by one

add(mb1);

add(mulb2);

add(rmb3);

add(db);

add(eb);

add(new Label("Result")); // adding last row

add(tf3);

// create the frame

setTitle("Arithmetic");

setSize(400,450);

setVisible(true);

}

// override the abstract method of AL

public void actionPerformed(ActionEvent e)

{

Button btn = (Button) e.getSource();

if(btn == eb)

{

System.exit(0); // closes the current application

}

String s1 = tf1.getText();

double d1 = Double.parseDouble(s1);

double d2 = Double.parseDouble(tf2.getText());

// Object obj = e.getSource();

// Button btn = (Button) obj;

// getSource( ) returns the object of the button (in Object form).

String s2 = " "; // a temporary local variable

if(btn == pb)

s2 = "sum = " + (d1+d2);

// assign a value to s2

else if(btn == mb1)

s2 = "Difference = " + (d1-d2);

else if(btn == mulb2)

s2 = "Product = " + (d1\*d2);

else if(btn == rmb3)

s2 = "Remainder = "+ (d1%d2);

else if(btn == db)

s2 = "Quotient = " + (d1/d2);

tf3.setText(s2); // place the result s2 in tf3

}

public static void main(String[] args)

{

new AdditionFrame ();

}

}