**UNIT V STEP MATERIAL**

**OBJECT ORIENTED PROGRAMMING THROUGH JAVA**

**SHORT ANSWERS:**

1. **Write the differences between AWT and SWINGS**

|  |  |
| --- | --- |
| **AWT** | **Swing** |
| AWT stands for Abstract windows toolkit. | Swing is also called as JFC’s (Java Foundation classes). |
| AWT components are called Heavyweight component. | Swings are called light weight component because swing components sits on the top of AWT components and do the work. |
| AWT components require java.awt package. | Swing components require javax.swing package. |
| AWT components are platform dependent. | Swing components are made in purely java and they are platform independent. |
| This feature is not supported in AWT. | We can have different look and feel in Swing. |
| These feature is not available in AWT | Swing has many advanced features like JTabel, Jtabbed pane which is not available in AWT.  |
|  |  |
|  |  |
|  |  |

1. **What is an Event? Explain event delegation model?**

**Event:**

* 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, a software or hardware failure occurs, or an operation is completed.

**Delegation Event Model:**

* The modern approach to handling events is based on 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.
1. **Explain Applet Lifecycle**

**Life cycle of an applet:**

• Applets life cycle includes the following methods

1. **init( )**

2. **start( )**

3. **paint( )**

4. **stop( )**

5. **destroy( )**

When an applet begins, the AWT calls the following methods, in this sequence:

• **init( ) :** The **init( )** method is the first method to be called. This is where variables should be initialized. This method is called only once during the run time of your applet.

• **start( ):** The **start( )** method is called after **init( )**. It is also called to restart an applet after it has been stopped. Whereas **init( )** is called once—the first time an applet is loaded—**start( )** is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at **start( )**.

• **paint( ):** The **paint( )** method is called each time applet's output must be redrawn. **paint( )** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )** is called. The **paint( )** method has one parameter of type **Graphics**. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

• **stop( ):** The **stop( )** method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When **stop( )** is called, the applet is probably running. Applet uses **stop( )** to suspend threads that don't need to run when the applet is not visible. To restart **start( )** is called if the user returns to the page

• **destroy( ):** The **destroy( )** method is called when the environment determines that your applet needs to be removed completely from memory. The **stop( )** method is always called before **destroy( )**.

1. **Write the differences between Applets and Applications**

**Differences between applets and applications:**

* An applet runs under the control of a browser, whereas an application runs stand-alone, with the support of a virtual machine. As such, an applet is subjected to more stringent security restrictions in terms of file and network access, whereas an application can have free reign over these resources.
* Applets are great for creating dynamic and interactive web applications, but the true power of Java lies in writing full blown applications. With the limitation of disk and network access, it would be difficult to write commercial applications (though through the user of server based file systems, not impossible). However, a Java application has full network and local file system access.
* No main() method: Applets do not need a main() method to exist anywhere because they are running inside another program (the browser) and are thus not a stand-alone program.
* No parameterized constructor: The browser calls the default, un parameterized constructor of an Applet, so parameterized constructors are not generally useful. Applets can be called from the web page with parameters however.
1. **Discuss some applet security issues**

**Applet security issues:**

• No access to the client's local file system

• No network access to a remote system other than the applet's host machine

• No access to the client's printer

• They cannot access client resources such as the local filesystem, executable files, system clipboard, and printers.

• They cannot connect to or retrieve resources from any third party server (any

server other than the server it originated from).

• They cannot load native libraries.

• They cannot change the SecurityManager.

• They cannot create a ClassLoader.

• They cannot read certain system properties.

• They cannot run out of security

1. **List the Swing Component Classes. Write Simple program using swings**

**The Swing component classes that are used are shown here:**

1. AbstractButton Abstract superclass for Swing buttons.
2. ButtonGroup Encapsulates a mutually exclusive set of buttons.
3. ImageIcon Encapsulates an icon.
4. JApplet The Swing version of Applet.
5. JButton The Swing push button class.
6. JCheckBox The Swing check box class.
7. JComboBox Encapsulates a combo box (an combination of a drop-down list

and text field).

1. JLabel The Swing version of a label.
2. JRadioButton The Swing version of a radio button.
3. JScrollPane Encapsulates a scrollable window.
4. JTabbedPane Encapsulates a tabbed window.
5. JTable Encapsulates a table-based control.
6. JTextField The Swing version of a text field.
7. JTree

**Java Swing Examples**

There are two ways to create a frame:

o By creating the object of Frame class (association)

o By extending Frame class (inheritance)

We can write the code of swing inside the main(), constructor or any other method.

**Simple Java Swing Example**

import javax.swing.\*;

public class FirstSwingExample {

public static void main(String[] args) {

JFrame f=new JFrame();//creating instance of JFrame

JButton b=new JButton("click");//creating instance of JButton

b.setBounds(130,100,100, 40);//x axis, y axis, width, height

f.add(b);//adding button in JFrame

f.setSize(400,500);//400 width and 500 height

f.setLayout(null);//using no layout managers

f.setVisible(true);//making the frame visible

}

}

1. **Explain Components and containers**

Components and Containers:

* A component is an independent visual control, such as a push button.
* A container holds a group of components.
* Swing components are derived from JComponent class. Note that all component classes begin with the letter J. For example a label is JLabel, a button is JButton etc.
* Swing defines two types of containers.
* The first are top level containers: JFrame, JApplet, JWindow, and JDialog. These containers do not inherit JComponent. They do, however inherit the AWT classes Container and Component. Unlike Swing’s other components which are heavy weight, the top level containers are heavy weight.
* The second type of containers are light weight inherit from JComponent. Example- Jpanel.
1. **Explain MVC architecture**

**MVC ARCHITECTURE:**

Model View Controller is a model used in swing components. Model represents the data of the component. View represents its appearance and controller represents its appearance and controller is a mediator between the model and the view. MVC represents the separation of model of an object from its view and how it is controlled.

**LONG ANSWERS:**

1. **Explain adapter classes with an example**

**Adapter Classes:**

Java provides a special feature, called an *adapter class*, that can simplify the creation of event handlers in certain situations. An adapter class provides an empty implementation of all methods in an event listener interface. Adapter classes are useful when the user wants to receive and process only some of the events that are handled by a particular event listener interface. The programmer 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 the programmer is interested. For example, the **MouseMotionAdapter** class has two methods, **mouseDragged( )** and **mouseMoved( )**. The signatures of these empty methods are exactly as defined in the **MouseMotionListener** interface. If one is interested in only mouse drag events, then extend **MouseMotionAdapter** and implement **mouseDragged( )**. The empty implementation of **mouseMoved( )** would handle the mouse motion events .

The following example demonstrates an adapter. It displays a message in the status bar of an applet viewer or browser when the mouse is clicked or dragged. However, all other mouse events are silently ignored. The program has three classes. **AdapterDemo** extends **Applet**. Its **init( )** method creates an instance of **MyMouseAdapter** and registers that object to receive notifications of mouse events. It also creates an instance of **MouseMotionAdapter** and registers that object to receive notifications of mouse motion events. Both of the constructors take a reference to the applet as an argument. **MyMouseAdapter** implements the **mouseClicked( )** method. The other mouse events are silently ignored by code inherited from the **MouseAdapter** class. **MyMouseMotionAdapter** implements the **mouseDragged( )** method. The other mouse motion event is silently ignored by code inherited from the **MouseMotionAdapter** class.

Note that both of our event listener classes save a reference to the applet. This

information is provided as an argument to their constructors & is used later to

invoke **showStatus( )** method.

// Demonstrate an adapter.

import java.awt.\*;

import java.awt.event.\*;

import java.applet.\*;

/\*

<applet code="AdapterDemo" width=300 height=100>

</applet>

\*/

public class AdapterDemo extends Applet {

public void init() {

addMouseListener(new MyMouseAdapter(this));

addMouseMotionListener(new MyMouseMotionAdapter(this));

} }

class MyMouseAdapter extends MouseAdapter {

AdapterDemo adapterDemo;

public MyMouseAdapter(AdapterDemo adapterDemo) {

this.adapterDemo = adapterDemo;

}

// Handle mouse clicked.

public void mouseClicked(MouseEvent me) {

adapterDemo.showStatus("Mouse clicked");

} }

class MyMouseMotionAdapter extends MouseMotionAdapter {

AdapterDemo adapterDemo;

public MyMouseMotionAdapter(AdapterDemo adapterDemo) {

this.adapterDemo = adapterDemo;

}

// Handle mouse dragged.

public void mouseDragged(MouseEvent me) {

adapterDemo.showStatus("Mouse dragged");

} }

As it can see by looking at the program, not having to implement all of the

methods defined by the **MouseMotionListener** and **MouseListener** interfaces

saves a considerable amount of effort and prevents your code from becoming

cluttered with empty methods.

1. **How to handle mouse events? Explain with example**

To handle mouse events, implement the **MouseListener** and the **MouseMotionListener** interfaces. The following applet demonstrates the process. It displays the current coordinates of the mouse in the applet’s status window.

Each time a button is pressed, the word “Down” is displayed at the location of the mouse pointer. Each time the button is released, the word “Up” is shown. If a button is clicked, the message “Mouse clicked” is displayed in the upper-left corner of the applet display area. As the mouse enters or exits the applet window, a message is displayed in the upper-left corner of the applet display area. When dragging the mouse, a \* is shown, which tracks with the mouse pointer as it is dragged.

Notice that the two variables, **mouseX** and **mouseY**, store the location of the mouse when a mouse pressed, released, or dragged event occurs. These coordinates are then used by **paint( )** to display output at the point of these occurrences.

// Demonstrate the mouse event handlers.

import java.awt.\*;

import java.awt.event.\*;

import java.applet.\*;

/\*

<applet code="MouseEvents" width=300 height=100>

</applet>

\*/

public class MouseEvents extends Applet implements MouseListener,

MouseMotionListener {

String msg = "";

int mouseX = 0, mouseY = 0; // coordinates of mouse

public void init() {

addMouseListener(this);

addMouseMotionListener(this);

}

// Handle mouse clicked.

public void mouseClicked(MouseEvent me) {

// save coordinates

mouseX = 0;

mouseY = 10;

msg = "Mouse clicked.";

repaint();

}

// Handle mouse entered.

public void mouseEntered(MouseEvent me) {

// save coordinates

mouseX = 0;

mouseY = 10;

msg = "Mouse entered.";

repaint();

}

// Handle mouse exited.

public void mouseExited(MouseEvent me) {

// save coordinates

mouseX = 0;

mouseY = 10;

msg = "Mouse exited.";

repaint();

}

// Handle button pressed.

public void mousePressed(MouseEvent me) {

// save coordinates

mouseX = me.getX();

mouseY = me.getY();

msg = "Down";

repaint();

}

// Handle button released.

public void mouseReleased(MouseEvent me) {

// save coordinates

mouseX = me.getX();

mouseY = me.getY();

msg = "Up";

repaint();

}

// Handle mouse dragged.

public void mouseDragged(MouseEvent me) {

// save coordinates

mouseX = me.getX();

mouseY = me.getY();

msg = "\*";

showStatus("Dragging mouse at " + mouseX + ", " + mouseY);

repaint();

}

// Handle mouse moved.

public void mouseMoved(MouseEvent me) {

// show status

showStatus("Moving mouse at " + me.getX() + ", " + me.getY());

}

// Display msg in applet window at current X,Y location.

public void paint(Graphics g) {

g.drawString(msg, mouseX, mouseY);

}

}

HSample output from this program is shown here:

Let’s look closely at this example.

The **MouseEvents** class extends **Applet** and implements both the **MouseListener** and **MouseMotionListener** interfaces.

These two interfaces contain methods that receive and process the various types of mouse events. Notice that the applet is both the source and the listener for these events. This works because **Component**, which supplies the **addMouseListener( )** and **addMouseMotionListener( )** methods, is a superclass of **Applet**. Being both the source and the listener for events is a common situation for applets. Inside **init( )**, the applet registers itself as a listener for mouse events. This is done by using

**addMouseListener( )** and **addMouseMotionListener( )**, which, as mentioned, are

members of **Component**.

They are shown here:

void addMouseListener(MouseListener *ml*)

void addMouseMotionListener(MouseMotionListener *mml*)

Here, *ml* is a reference to the object receiving mouse events, and *mml* is a reference

to the object receiving mouse motion events. In this program, the same object is

used for both.

The applet then implements all of the methods defined by the **MouseListener** and

**MouseMotionListener** interfaces. These are the event handlers for the various

mouse events. Each method handles its event and then returns.

1. **write a java program that works as a simple calculator. Use a grid layout to arrange buttons for the digits and for the + \_ \* % operatorions. a text field to display the result. Handle any possible exceptions like divide by zero.**

**import** java.awt.event.\*;

**import** java.awt.\*;

**import** javax.swing.\*;

**public** **class** Cal **extends** JFrame **implements** ActionListener{

String msg="";

**int** v1,v2,result;

JTextField t1,t2;

JButton b;

**boolean** start=**true**;

**char** OP;

Cal(){

 t1=**new** JTextField(25);

 t2=**new** JTextField(25);

 t1.setEditable(**false**);

 add(t1,BorderLayout.*NORTH*);

 JPanel p=**new** JPanel();

 p.setLayout(**new** GridLayout(4,4));

 String buttons="789\*456-123+0/c=";

 **for** (**int** i=0;i<buttons.length();i++)

 {

 b=**new** JButton(buttons.substring(i,i+1));

 b.addActionListener(**this**);

 p.add(b);

 }

 add(p,BorderLayout.*CENTER*);

 setSize(400,400);

 setVisible(**true**);

 setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*);

}

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

 String str=ae.getActionCommand();

 System.*out*.println("str="+str);

 **char** ch=str.charAt(0);

 **if**(Character.*isDigit*(ch))

 {

 **if**(start==**true**)

 t1.setText(str);

 **else**

 t1.setText(t1.getText()+str);

 start=**false**;

 }

 **else** **if**(str.equals("+"))

 {

 v1=Integer.*parseInt*(t1.getText());

 OP='+';

 start=**true**;

 }

 **else** **if**(str.equals("-"))

 {

 v1=Integer.*parseInt*(t1.getText());

 OP='-';

 start=**true**;

 }

 **else** **if**(str.equals("\*"))

 {

 v1=Integer.*parseInt*(t1.getText());

 OP='\*';

 start=**true**;

 }

 **else** **if**(str.equals("/"))

 {

 v1=Integer.*parseInt*(t1.getText());

 OP='/';

 start=**true**;

 }

 **else** **if** (str.equals("c"))

 {

 t1.setText("");

 }

 **else** **if** (str.equals("="))

 {

 v2=Integer.*parseInt*(t1.getText());

 **if**(OP=='+')

 result=v1+v2;

 **else** **if** (OP=='-')

 result=v1-v2;

 **else** **if** (OP=='\*')

 result=v1\*v2;

 **else** **if** (OP=='/')

 {

 **try**{

 result=v1/v2;

 }

 **catch**(ArithmeticException e)

 {

 JOptionPane.*showMessageDialog*(**null**, "Should not divide by zero");

 }

 }

 t1.setText(""+result);

 start=**true**;

 }}

**public** **static** **void** main(String [] args)

{

Cal c=**new** Cal();

}

}