CMR
INSTITUTE OF
TECHNOLOGY

USN | | | | | | | | | |

Improvement test – III

Sub:	OBJECT ORIENTED CONCEPTS							Code:	15CS45
Date:	30/ 05 / 2017	Duration:	90 mins	Max Marks:	50	Sem:	IV	Branch:	CSE (C,D)
Answer Any FIVE FULL Questions									

		Marks	OBE CO	OBE RBT
1 (a)	Define an Applet? Explain the Applet Skeleton along with an example?	[10]	CO6	L2
2 (a)	Define Delegation event model? Explain the following • Event classes • Sources of events • Event listener interfaces	[10]	CO4	L2
3 (a)	Write an applet program to display the message “WELCOME TO VTU”. set the background color to green and foreground to red.	[10]	CO6	L3
4 (a)	Differentiate between AWT and swings? Explain the two features of swings?	[10]	CO6	L2
5(a)	Define Swing? Explain the following with the example. 1) JTextField 2) JButton 3) JComboBox	[10]	CO6	L2
6(a)	Explain different forms of repaint method.	[05]	CO6	L2
(b)	Explain how Parameters are passed to an Applet.	[05]	CO6	L2
7 (a)	Write Swing application having two buttons “alpha” and “beta”. when either of the buttons pressed, it should display “alpha pressed” and “beta pressed” respectively	[10]	CO6	L3

1 (a) Define an Applet? Explain the Applet Skeleton along with an example?

Applet is a special type of program that is embedded in the webpage to generate the dynamic content. the browser and works at client side.

AWT-based applets (such as those discussed in this chapter) will also override the **paint( )** method, which is defined by the AWT **Component** class. This method is called when the applet's output must be redisplayed. (Swing-based applets use a different mechanism to accomplish this task.) These five methods can be assembled into the skeleton shown here:

```
// An Applet skeleton.
import java.awt.*;
import java.applet.*;
/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/
public class AppletSkel extends Applet
{
// Called first.
public void init()
{
// initialization
}

/* Called second, after init(). Also called whenever the applet is restarted. */
public void start()
{
// start or resume execution
}
// Called when the applet is stopped.
public void stop()
{
// suspends execution
}
/* Called when applet is terminated. This is the last method executed. */
public void destroy()
{
// perform shutdown activities
}
// Called when an applet's window must be restored.
public void paint(Graphics g)
{
// redisplay contents of window
}
}
```

Applet Initialization and Termination

It is important to understand the order in which the various methods shown in the skeleton are called. When an applet begins, the following methods are called, in this sequence:

1. **init( )**
2. **start( )**
3. **paint( )**

When an applet is terminated, the following sequence of method calls takes place:

1. **stop( )**
2. **destroy( )**

Let's look more closely at these methods.

init( )
The **init( )** method is the first method to be called. This is where you should initialize variables. 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 your applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored. **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.

2) Define Delegation event model? Explain the following

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 an event is received, the listener processes the event and then returns.

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. Asource 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.

Event Classes

The classes that represent events are at the core of Java's event handling mechanism. Thus, a discussion of event handling must begin with the event classes. It is important to understand, however, that Java defines several types of events. The most widely used events are those defined by the AWT and those defined by Swing. This chapter focuses on the AWT 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. Its **getID( )** method can be used to determine the type of the event. The signature of this method is shown here:

To summarize:

• **EventObject** is a superclass of all events.

• **AWTEvent** is a superclass of all AWT events that are handled by the delegation event model.

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 **MotionListener** interface defines two methods to receive notifications when the mouse is dragged or moved. Any object may receive and process one or both of these events if it provides an implementation of interface.

3)Write an Applet program to display the message "WELCOME TO VTU", set the background color to green and foreground to red.

```
/* A simple applet that sets the foreground and background colors and outputs a string. */
import java.awt.*;
import java.applet.*;
/*
<applet code="Sample" width=300 height=50>
</applet>
*/
```

```
public class Sample extends Applet
{
	String msg;
	// set the foreground and background colors.

	public void init()
	{
		setBackground(Color.cyan);
		setForeground(Color.red);
		msg = "welcome to vtu";
	}
	// Initialize the string to be displayed.

	// Display msg in applet window.

	public void paint(Graphics g)
	{
		g.drawString(msg, 10, 30);
	}
}
```

4) Differentiate between AWT and Swings? Explain the two features of 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. Also.

	Swing components are called "lightweight" because they do not require a native OS object to implement their functionality. JDialog and JFrame are heavyweight, because they do have a peer. So components like JButton, JTextArea, etc., are lightweight because they do not have an OS peer.
With AWT, you have 21 "peers" (one for each control and one for the dialog itself). A "peer" is a widget provided by the operating system, such as a button object or an entry field object.	With Swing, you would have only one peer, the operating system's window object. All of the buttons, entry fields, etc. are drawn by the Swing package on the drawing surface provided by the window object. This is the reason that Swing has more code. It has to draw the button or other control and implement its behavior instead of relying on the host operating system to perform those functions.
AWT is a thin layer of code on top of the OS.	Swing is much larger. Swing also has very much richer functionality.
Using AWT, you have to implement a lot of things yourself.	Swing has them built in.

Two Key Swing Features

Swing was created to address the limitations present in the AWT. It does this through two key features: lightweight components and a pluggable look and feel. Together they provide an elegant, yet easy-to-use solution to the problems of the AWT. More than anything else, it is these two features that define the essence of Swing.

Swing Components Are Lightweight

With very few exceptions, Swing components are *lightweight*. This means that they are written entirely in Java and do not map directly to platform-specific peers. Thus, lightweight components are more efficient and more flexible. Furthermore, because lightweight components do not translate into native peers, the look and feel of each component is determined by Swing, not by the underlying operating system. As a result, each component will work in a consistent manner across all platforms.

Swing Supports a Pluggable Look and Feel

Swing supports a *pluggable look and feel* (PLAF). Because each Swing component is rendered by Java code rather than by native peers, the look and feel of a component is under the control of Swing. This fact means that it is possible to separate the look and feel of a component from the logic of the component, and this is what Swing does. Separating out the look and feel provides a significant advantage: it becomes possible to change the way that a component is rendered without affecting any of its other aspects. In other words, it is possible to "plug in" a new look and feel for any given component without creating any side effects in the code that uses that component. Moreover, it becomes possible to define entire sets of look-and-feels that represent different GUI styles. To use a specific style, its look and feel is simply "plugged in." Once this is done, all components are automatically rendered using that style.

Pluggable look-and-feels offer several important advantages. It is possible to define a look and feel that is consistent across all platforms. Conversely, it is possible to create a look and feel that acts like a specific platform. For example, if you know that an application will be running only in a Windows environment, it is possible to specify the Windows look and feel. It is also possible to design a custom look and feel. Finally, the look and feel can be changed dynamically at run time.

Java 8 provides look-and-feels, such as metal and Nimbus, that are available to all Swing users. The metal look and feel is also called the *Java look and feel*. It is platform-independent and available in all Java execution environments. It is also the default look and feel. Windows environments also have access to the Windows look and feel. This book uses the default Java look and feel (metal) because it is platform independent.

5) Define Swing? Explain the following with the example

Java Swing is a part of Java Foundation Classes (JFC) that is used to create window-based applications. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java.

Java JTextField

The object of a JTextField class is a text component that allows the editing of a single line text. It inherits JTextComponent class.

JTextField class declaration

Let's see the declaration for javax.swing.JTextField class.

1. **public class** JTextField **extends** JTextComponent **implements** SwingConstants

Commonly used Constructors:

Constructor	**Description**

JTextField()	Creates a new TextField
JTextField(String text)	Creates a new TextField initialized with the specified text.
JTextField(String text, int columns)	Creates a new TextField initialized with the specified text and colu
JTextField(int columns)	Creates a new empty TextField with the specified number of colun

Commonly used Methods:

Methods	Description
void addActionListener(ActionListener l)	It is used to add the specified action listener to receive action textfield.
Action getAction()	It returns the currently set Action for this ActionEvent source, or is set.
void setFont(Font f)	It is used to set the current font.
void removeActionListener(ActionListener l)	It is used to remove the specified action listener so that it no long events from this textfield.

Java JButton

The JButton class is used to create a labeled button that has platform independent implementation. The application result in some action when the button is pushed. It inherits AbstractButton class.

JButton class declaration

Let's see the declaration for javax.swing.JButton class.

1. **public class** JButton **extends** AbstractButton **implements** Accessible

Commonly used Constructors:

Constructor	Description
JButton()	It creates a button with no text and icon.
JButton(String s)	It creates a button with the specified text.
JButton(Icon i)	It creates a button with the specified icon object.

Commonly used Methods of AbstractButton class:

Methods	**Description**
void setText(String s)	It is used to set specified text on button
String getText()	It is used to return the text of the button.
void setEnabled(boolean b)	It is used to enable or disable the button.
void setIcon(Icon b)	It is used to set the specified Icon on the button.
Icon getIcon()	It is used to get the Icon of the button.
void setMnemonic(int a)	It is used to set the mnemonic on the button.
void addActionListener(ActionListener a)	It is used to add the action listener to this object.

Java JComboBox

The object of Choice class is used to show popup menu of choices. Choice selected by user is shown on the top of a menu. It inherits JComponent class.

JComboBox class declaration

Let's see the declaration for javax.swing.JComboBox class.

1. **public class** JComboBox **extends** JComponent **implements** ItemSelectable, ListDataListener, ActionListen er, Accessible

Commonly used Constructors:

Constructor	**Description**
JComboBox()	Creates a JComboBox with a default data model.
JComboBox(Object[] items)	Creates a JComboBox that contains the elements in the specified array.
JComboBox(Vector<?> items)	Creates a JComboBox that contains the elements in the specified Vector.

Commonly used Methods:

Methods	**Description**
void addItem(Object anObject)	It is used to add an item to the item list.
void removeItem(Object anObject)	It is used to delete an item to the item list.
void removeAllItems()	It is used to remove all the items from the list.
void setEditable(boolean b)	It is used to determine whether the JComboBox is editable.
void addActionListener(ActionListener a)	It is used to add the ActionListener.
void addItemListener(ItemListener i)	It is used to add the ItemListener.

6 a)Explain the different forms of repaint method.

The **repaint( )** method has four forms. Let's look at each one, in turn. The simplest version of **repaint( )** is shown here:

void repaint( )
This version causes the entire window to be repainted. The following version specifies a region
that will be repainted:

void repaint(int *left*, int *top*, int *width*, int *height*)
Here, the coordinates of the upper-left corner of the region are specified by *left* and *top*, and the width and height of the region are passed in *width* and *height*. These dimensions are specified in pixels. You save time by specifying a region to repaint. Window updates are costly in terms of time. If you need to update only a small portion of the window, it is more efficient to repaint only that region.

Calling **repaint( )** is essentially a request that your applet be repainted sometime soon. However, if your system is slow or busy, **update( )** might not be called immediately. Multiple requests for repainting that occur within a short time can be collapsed by the AWT in a manner such that **update( )** is only called sporadically. This can be a problem in many situations including animation, in which a consistent update time is necessary. One solution to this problem is to use the following forms of **repaint( )**:

void repaint(long *maxDelay*)

void repaint(long *maxDelay*, int *x*, int *y*, int *width*, int *height*)

Here, *maxDelay* specifies the maximum number of milliseconds that can elapse before **update( )** is called. Beware, though. If the time elapses before **update( )** can be called, it isn't called. There's no return value or exception thrown.

6b) Explain how Parameters are passed to an Applet?

The APPLET tag in HTML allows you to pass parameters to your applet. To retrieve a parameter, use the **getParameter( )** method. It returns the value of the specified parameter in the form of a **String** object. Thus, for numeric and **boolean** values, you will need to convert their string representations into their internal formats. Here is an example that demonstrates passing parameters:

```
// Use Parameters
import java.awt.*;
import java.applet.*;
/*
<applet code="ParamDemo" width=300 height=80>
<param name=fontName value=Courier>
<param name=fontSize value=14>
<param name=leading value=2>
<param name=accountEnabled value=true>
</applet>
*/
public class ParamDemo extends Applet
{
        String fontName;
        int fontSize;
        float leading;
        boolean active;

        // Initialize the string to be displayed.
        public void start()
        {
                String param;
                fontName = getParameter("fontName");
                if(fontName == null)
                        fontName = "Not Found";
                param = getParameter("fontSize");
                try {
                        if(param != null) // if not found
                                fontSize = Integer.parseInt(param);
                        else
                                fontSize = 0;
                        } catch(NumberFormatException e) {
                fontSize = -1;
        }
        param = getParameter("leading");
        try
        {
                if(param != null) // if not found
                        leading = Float.valueOf(param).floatValue();
                else
                        leading = 0;
        } catch(NumberFormatException e) {
                leading = -1;
        }
        param = getParameter("accountEnabled");
        if(param != null)
        active = Boolean.valueOf(param).booleanValue();
        }
        // Display parameters.

        public void paint(Graphics g) {
        g.drawString("Font name: " + fontName, 0, 10);
        g.drawString("Font size: " + fontSize, 0, 26);
        g.drawString("Leading: " + leading, 0, 42);
        g.drawString("Account Active: " + active, 0, 58);
}
}
```

7a)write swing application having two buttons “alpha” and “beta” when either of the buttons pressed it should display “alpha pressed” and “beta pressed” respectively.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class EventDemo
{
        JLabel jlab;
        EventDemo()
        {
                // Create a new JFrame container.
                JFrame jfrm = new JFrame("An Event Example");

                // Specify FlowLayout for the layout manager.
                jfrm.setLayout(new FlowLayout());

                // Give the frame an initial size.
                jfrm.setSize(220, 90);

                // Terminate the program when the user closes the application.
                jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

                // Make two buttons.
                JButton jbtnAlpha = new JButton("Alpha");
                JButton jbtnBeta = new JButton("Beta");

                // Add action listener for Alpha.
                jbtnAlpha.addActionListener(new ActionListener()
                {
                        public void actionPerformed(ActionEvent ae)
                        {
                                jlab.setText("Alpha was pressed.");
                        }
                });
```

```
                    // Add action listener for Beta.
                    jbtnBeta.addActionListener(new ActionListener() {
                              public void actionPerformed(ActionEvent ae) {
                                                  jlab.setText("Beta was pressed.");
                    }
                    });
                    // Add the buttons to the content pane.
                    jfrm.add(jbtnAlpha);
                    jfrm.add(jbtnBeta);

                    // Create a text-based label.
                    jlab = new JLabel("Press a button.");

                    // Add the label to the content pane.
                    jfrm.add(jlab);
                    jfrm.setVisible(true);
                    }
                    public static void main(String args[]) {
                    // Create the frame on the event dispatching thread.
                    SwingUtilities.invokeLater(new Runnable() {
                    public void run() {
                    new EventDemo();
                    }
                    });
}
}
```