CMR
INSTITUTE OF
TECHNOLOGY

USN										

SCHEME AND SOLUTION

Internal Assesment Test – III

Sub:	Object Oriented Concepts							Code:	15CS45
Date:	30 / 05 / 2017	Duration:	90 mins	Max Marks:	50	Sem:	4(A,B)	Branch:	CSE

Answer FIVE FULL questions selecting AT LEAST TWO questions from each module

		Marks	OBE		Marks Distribution
			CO	RBT	
	MODULE I				
1.	**Give an example for using keyboard event. Write a Java program to demonstrate the key event handler**.	[10]	CO2	L3	Explanation: 5m Program: 5m

When a key is pressed, a **KEY_PRESSED** event is generated. This results in a call to the **keyPressed()** event handler. When the key is released, a **KEY_RELEASED** event is generated and the keyReleased( ) handler is executed. If a character is generated by the keystroke, then a **KEY_TYPED** event is sent and the **keyTyped( )** handler is invoked.

The following program demonstrates keyboard input. It echoes keystrokes to the applet window and shows the pressed/released status of each key in the status window.

```
// Demonstrate the key event handlers.

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

/*
<applet code="SimpleKey" width=300 height=100>
</applet>
*/

public class SimpleKey extends Applet
implements KeyListener {
String msg = "";
int X = 10, Y = 20; // output coordinates
public void init() {
addKeyListener(this);
}

public void keyPressed(KeyEvent ke) {
showStatus("Key Down");
}

public void keyReleased(KeyEvent ke) {
showStatus("Key Up");
}

public void keyTyped(KeyEvent ke) {
msg += ke.getKeyChar();
repaint();
}

// Display keystrokes.
public void paint(Graphics g) {
g.drawString(msg, X, Y);
}
}
```

Sample output is shown here:

2.	Briefly explain the role of: i) Event classes ii) Event listener interfaces	[10]	CO2	L2	Explanation: 5m each

Event classes

The Event classes represent the event. Java provides us various 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.				

PaintEvent

The object of this class represents the change in state of a window.

Event Listener Interfaces

The delegation event model has two parts:
sources and listeners.

Listeners are created by implementing one or more of the interfaces defined by the **java.awt.event** package.

When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument. Table below lists commonly used listener interfaces and provides a brief description of the methods that they define.

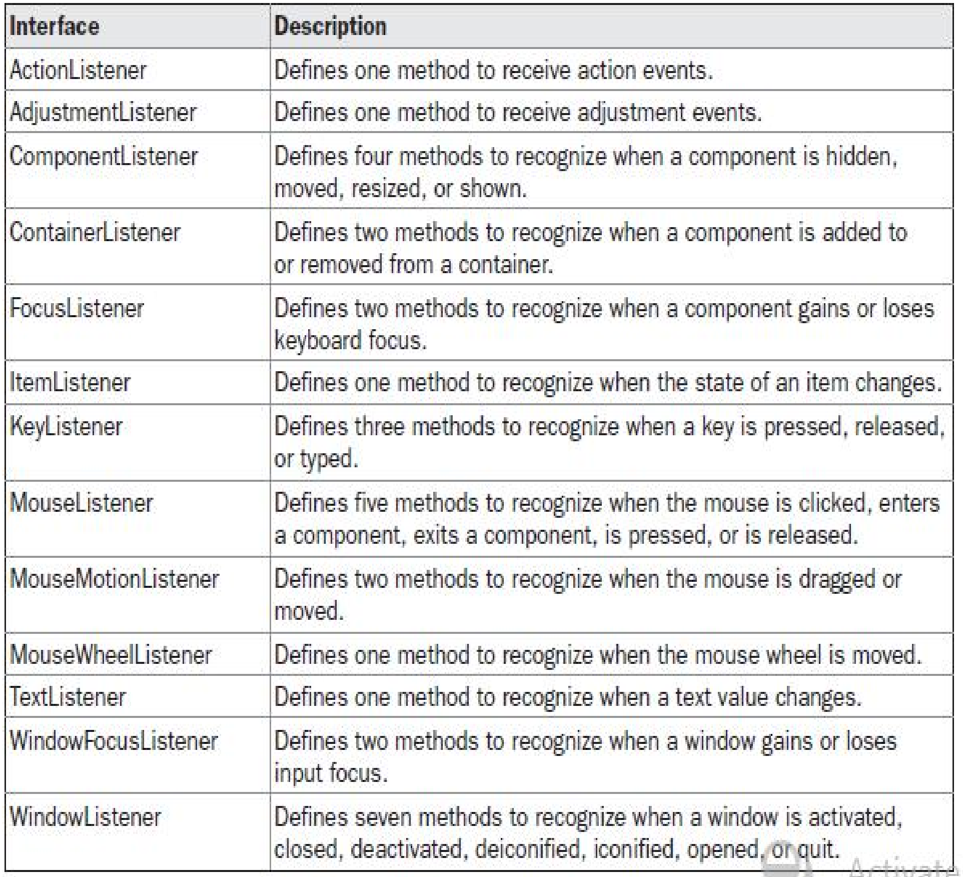
Eg:

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*)

.



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.

TABLE 22-3 Commonly Used Event Listener Interfaces

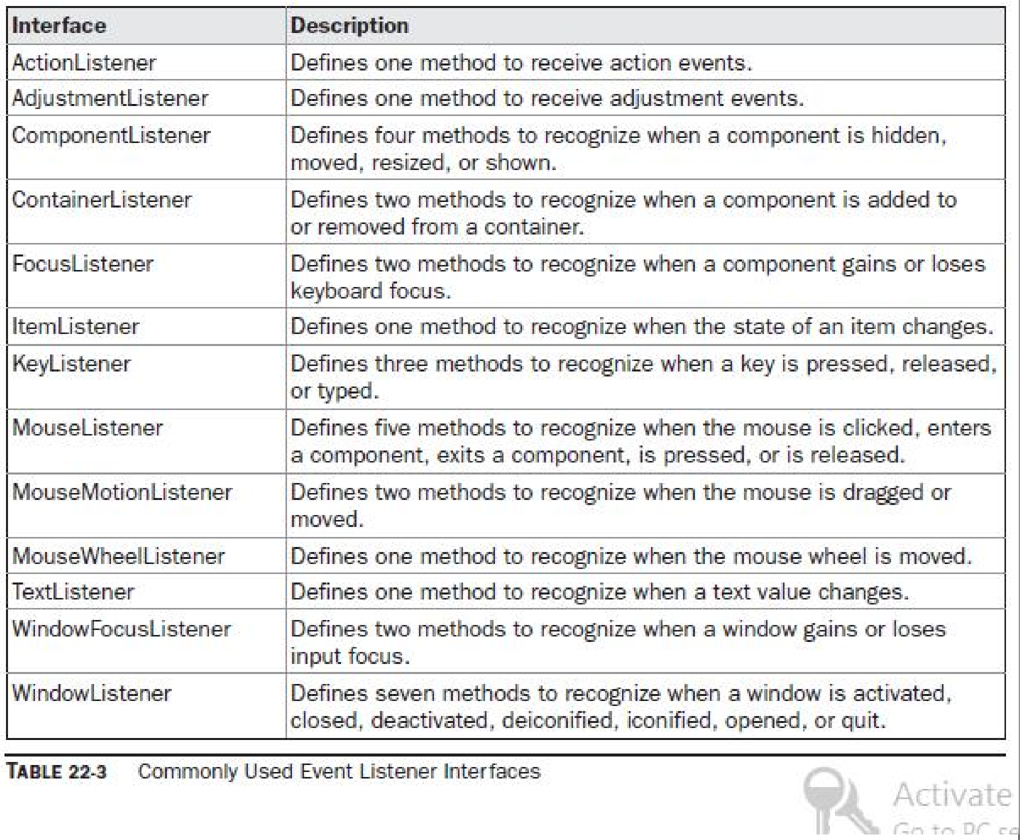
3.	**Define the delegation event model, Briefly explain the role of:** i) **Sources of event** ii) **Adapter clauses**	[10]	CO2	L2	Explanation: 5m each

Sources of event

In Java, events are handled in terms of **event sources** and **event listeners**. An event source is an object that produces an event, and an event listener is an object that wants to be informed when an event occurs.

For example, a button is an event source, and an animation object might be an event listener.

Table below lists some of the user interface components that can generate the events . In addition to these graphical user interface elements, any class derived



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.

TABLE 22-3 Commonly Used Event Listener Interfaces

ii. 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 you want to receive and process only some of the events that are handled by a particular event listener interface.One 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 you are interested. For example, the **MouseMotionAdapter** class has two methods, **mouseDragged( )** and **mouseMoved( )**, which are the methods defined by the **MouseMotionListener** interface. If you were interested in only mouse drag events, then you could simply extend **MouseMotionAdapter** and override **mouseDragged( )**. The empty implementation of **mouseMoved( )** would handle the mouse motion events.				
4.	**a. Define the 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 an event is 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. This is a more efficient way to handle events than the design used by the old Java 1.0 approach. Previously, an event was propagated up the containment hierarchy until it was handled by a component. This required components to receive events that they did not	[5]	CO2	L1	Explanation: 5m

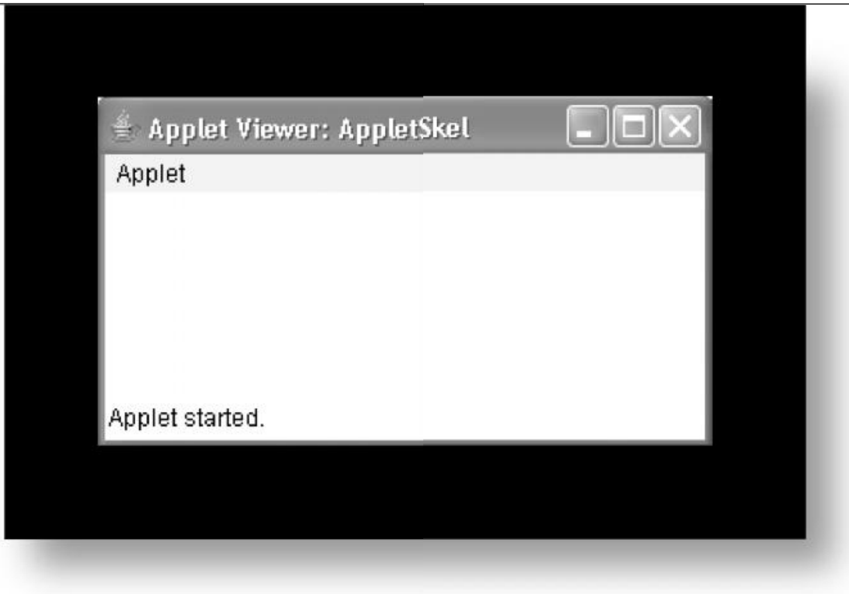
	process, and it wasted valuable time. The delegation event model eliminates this overhead.				
	b. Define swing. Explain two features of it Swing is a set of classes that provides more powerful and flexible GUI components than does the AWT. Swing provides the look and feel of the modern Java GUI. The two key features of swing are: • Lightweight components and • A pluggable look and feel • **Swing Components Are Lightweight** Swing components are *lightweight*. This means that they are written entirely in Java and do not map directly to platform-specific peers. Because lightweight components are rendered using graphics primitives, they can be transparent, which enables nonrectangular shapes. 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. This means that 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-	[5]	CO2	L2	Explanation: 5m

	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 SE 6 provides look-and-feels, such as metal and Motif, 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 and Windows Classic look and feel.				
	MODULE II				

5.	**Explain the applet skeleton and write an example program for applet**
	
	// 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
	}
		[10]	CO2	L2	Explanation: 5m
					Program: 5m

```
/* 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
}
}
```

Although this skeleton does not do anything, it can be compiled and run. When run, it generates the following window when viewed with an applet viewer:



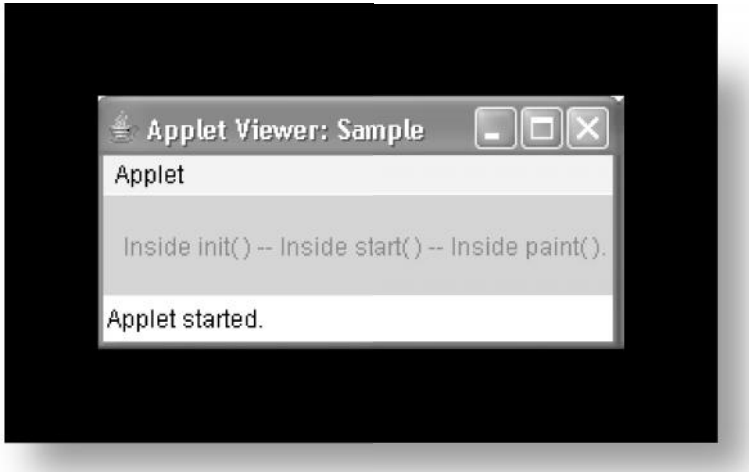
Applet started.

OR

```
/* 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 = "Inside init( ) --";
}

// Initialize the string to be displayed.
public void start() {
msg += " Inside start( ) --";
}

// Display msg in applet window.
public void paint(Graphics g) {
msg += " Inside paint( ).";
g.drawString(msg, 10, 30);
}
}
```

This applet generates the window shown here:



```
Inside init() -- Inside start() -- Inside paint().
```

Applet started.

init( )

The init( ) method is the first method to be called. This is where one 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 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. You should use stop( ) to suspend threads that don't need to run when the applet is not visible. You can restart them when 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. At this point, you should free up any resources the applet may be using. The stop( ) method is always called before destroy( ).

Overriding update( )

In some situations, your applet may need to override another method defined by the AWT, called update( ). This method is called when your applet has requested that a portion of its window be redrawn. The default

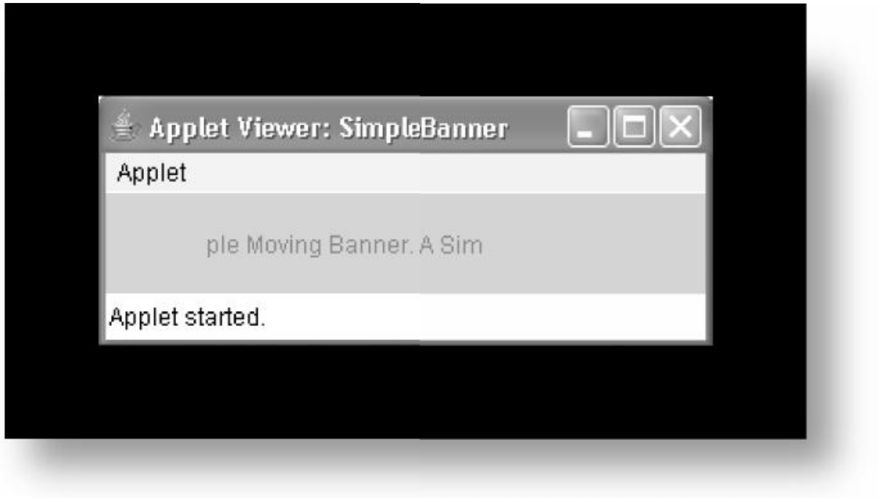
	version of update( ) simply calls paint( ). However, you can override the update( ) method so that it performs more subtle repainting. In general, overriding update( ) is a specialized technique that is not applicable to all applets, and the examples in this book do not override update( ).				
6.	Write an applet program to scroll a text across applet window	[10]	CO3	L3	Program: 10m

A Simple Banner Applet

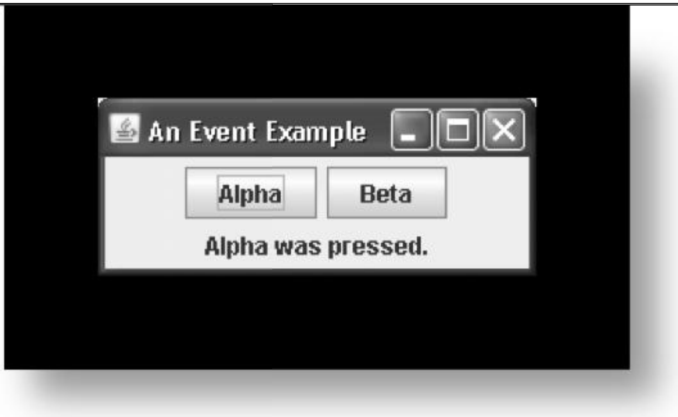
This applet scrolls a message, from right to left, across the applet’s window. Since the scrolling of the message is a repetitive task, it is performed by a separate thread, created by the applet when it is initialized. The banner applet is shown here:

```
/* A simple banner applet. This applet creates a thread that scrolls
the message contained in msg right to left across the applet's window.
*/

import java.awt.*;
import java.applet.*;

/*
<applet code="SimpleBanner" width=300 height=50>
</applet>
*/

public class SimpleBanner extends Applet implements Runnable {
String msg = " A Simple Moving Banner.";
Thread t = null;
int state;
boolean stopFlag;

// Set colors and initialize thread.
public void init() {
setBackground(Color.cyan);
setForeground(Color.red);
```

```
}

// Start thread
public void start() {
t = new Thread(this);
stopFlag = false;
t.start();
}

// Entry point for the thread that runs the banner.
public void run() {
char ch;

// Display banner
for( ; ; ) {
try {
repaint();
Thread.sleep(250);
ch = msg.charAt(0);
msg = msg.substring(1, msg.length());
msg += ch;
if(stopFlag)
break;
} catch(InterruptedException e) {}
}
}

// Pause the banner.
public void stop() {
stopFlag = true;
t = null;
}

// Display the banner.
public void paint(Graphics g) {
g.drawString(msg, 50, 30);
```

	} } Following is sample output:  ple Moving Banner. A Sim Applet started.				
7.	**Create swing application having two buttons named alpha and beta. When either of the buttons pressed, it should display “alpha pressed” and “beta pressed” respectively.** **Create swing application having two buttons named al and beta. When either of the buttons pressed, it sh display “alpha pressed” and “beta pressed” respectively.** // Handle an event in a Swing program. import java.awt.*; import java.awt.event.*;	[10]	CO3	L3	Program: 10m

```
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);

// Display the frame.
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();
}
});
}
}
```

	 Alpha was pressed.				