CMR Institute of Technology, Bangalore			
Department(s): Computer Science & Engineering			
Semester: 04	Section(s): A,B & C	Lectures/week: 04	
Subject: Object Oriented Concepts		Code: 18CS45	
Course Instructor(s): Dr, Jhansi Rani P. and Mrs. Poonam Tijare			
Course duration: 10 March 2020– 8 August 2020			
IAT 3 Question paper and scheme			

Sno	Question	Marks	OBE	CO
1	What you mean by Thread? Explain how thread can be created in JAVA? Thread -2 marks Ways to create thread - 3 marks each + 1 mark example for each category	10	L3	CO3
2	Explain adapter and inner class with example programs. Adapter class- 5 mark Inner class- 5 mark	10	L3	CO3
3	Explain keyEvent and mouseEvent classes with example. keyEvent - 5 marks mouseEvent - 5 marks	10	L3	CO4
4	Explain how delegation event model is used to handle events with an example. delegation event model – 6 marks Example-4 marks	10	L3	CO4
5	Describe Synchronization. Write an example program for implementing static Synchronization.	10	L3	CO4
6	Explain with syntax and example the following: JLabel and JCheckBox -5 marks each	10	L3	CO4
7	Explain with syntax and example the following: JButton and JRadioButton -5 marks each	10	L3	CO4
8	What are Events, Event listener and Event sources? Events- 3 marks Event listener- 3 marks Event sources- 4 marks	10	L3	CO4

1. **What you mean by Thread? Explain how thread can be created in JAVA?**
   A thread is a path of execution within a process. A process can contain multiple threads.
   Java provides built-in support for multi threaded programming. A multi-threaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. Thus multi threading is a specialized form of multitasking.
   Creating a Thread
   A thread can be created in two ways
   1. we can implement the Runnable interface

2. we can extend the Thread class
1. Creating a thread by Implementing Runnable interface:
To create a thread by implementing Runnable interface, create a class that implements the Runnable interface.
Runnable abstracts a unit of executable code.
To implement Runnable, a class has to implement a single method called run(), which is declared like this -
public void run()
Inside run(), you will define the code that constitutes the new thread.
Run() can call other methods, use other classes and declare variables just like the main thread can.
The only difference is that run() establishes the entry point for another concurrent thread of execution within your program. This thread will end when run() returns.
After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class. Thread defines several constructors. One such constructor is
Thread (Runnable threadOb, String threadName)
In this constructor, threadOb is an instance of a class that implements the Runnable interface. This defines where execution of the thread will begin. The name of the new thread is specified by threadName.

After the new thread is created, it will not start running until you call its start() method, which is declared within Thread. In essence, start executes a call to run(). The start() method is shown here
void start()
Example program that creates a new thread and starts it running:

```
// create a second thread
class NewThread implements Runnable {
Thread t;
NewThread() {
//Create a new, second thread
t = new Thread(this, "Demo Thread");
System.out.println("Child thread :" + t);
t.start(); // start the thread
}
// This is the entry point for the second thread
public void run() {
try {
for (int i=5; i >0;i--) {
System.out.println("child Thread:" + i);
Thread.sleep(500);
}
}
catch (InterruptedException e) {
System.out.println("Child Interrupted.");
}
System.out.println("Exiting child thread.");
}
}
class ThreadDemo {
public static void main(String args[]) {
new NewThread(); // create a new thread
try {
for(int i=5;i>0;i--) {
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}
}
catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting");
}
}
```

```
$ javac ThreadDemo.java
$ java ThreadDemo
Child thread :Thread[Demo Thread,5,main]
Main Thread: 5
child Thread:5
child Thread:4
Main Thread: 4
child Thread:3
child Thread:2
Main Thread: 3
child Thread:1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting
```

Note:

In a multi-threaded program, often the main thread must be the last thread to finish running.

In older JVMs, if the main thread finishes before a child thread has completed, then the Java run-time system may “hang”.

The above program ensures that the main thread finished last, because the main thread sleeps for 1000 milliseconds between iterations, but the child sleeps for only 500 milliseconds.

This causes the child thread to terminate earlier than the main thread.

2. Creating a thread by extending Thread class:

The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class.

The extending class must override the run() method, which is the entry point for the new thread.

It must also call start() to begin execution of the new thread.

The following is the example program -

```
// Create a second thread by extending Thread
class NewThread extends Thread {
NewThread () {
// Create a new, second thread
super("Demo Thread");
System.out.println("Child thread: " + this);
start(); // Start the thread
}
// This is the entry point for the second thread.
public void run() {
try {
for(int i = 5;i >0; i--) {

System.out.println("Child thread :" + i);
Thread.sleep(500);
}
}
catch (InterruptedException e) {
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
}
}
class ExtendThread {
public static void main(String args[]) {
new NewThread(); // Create a new thread
try {
for(int i = 5;i>0;i--) {
System.out.println("Main thread : " + i);
Thread.sleep(1000);
}
}
catch (InterruptedException e) {
System.out.println("Main thread Interrupted.");
}
System.out.println("Main thread exiting.");
}
}
```
Output:
```
$ javac ExtendThread.java
$ java ExtendThread
Child thread: Thread[Demo Thread,5,main]
Child thread :5
Main thread : 5
Child thread :4
Main thread : 4
Child thread :3
Child thread :2
```

```
Main thread : 3
Child thread :1
Exiting child thread.
Main thread : 2
Main thread : 1
Main thread exiting.
```

Program Explanation:

This program generates the same output as the previous version.
The child thread is created by instantiating an object of NewThread, which is derived from Thread.
Notice the call to super() inside NewThread. This invokes the following form of the Thread constructor:
public Thread (String threadName)
threadName specifies the name of the thread.

2. Explain adapter and inner class with example programs.

ADAPTER CLASSES:

Java provides a special feature called an adapter class, which simplifies 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.
We have to 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.
For example, the MouseMotionAdapter class has two methods, mouseDragged() and mouseMoved(), which are the methods defined by the MouseMotionListener interface. If you are interested only in mouse drag events, then you could simply extend MouseMotionAdapter and override mouseDragged(). The empty implementation of mouseMoved() would handle the mouse motion events for you.
Table list the commonly used adapter classes in java.awt.event

Example program to demonstrate an adapter:

The applet displays a message in the status bar when the mouse is clicked.

All other mouse events are silently ignored.

The program has two classes

1. AdapterDemo extends Applet.

Its init() method creates an instance of MyMouseAdapter and registers that object to receive notifications of mouse events.

The constructor takes reference to the applet as an argument.

2. MyMouseAdapter extends MouseAdapter and overrides the mouseClicked() method.

The other mouse events are silently ignored by code inherited from the MouseAdapter class.

```
// demonstrate an Adapter
import java.awt.*; // Contains all AWT based event classes used by Delegation Event Model
import java.awt.event.*; // Contains all Event Listener Interfaces
import java.applet.*; // For Applets
/*
<applet code = "AdapterDemo" width=500 height=300>
</applet>
*/
public class AdapterDemo extends Applet {
public void init() {

addMouseListener(new MyMouseAdapter (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");
}
}
```

INNER CLASSES:

Inner class is a a class defined within another class or even within an expression.

Inner classes can be used to simplify the code when using event adapter classes.

. Example program to illustrate inner class

Program Explanation:

. The goal of the applet is to display the string “Mouse Pressed” in the status bar of the applet viewer or browser when the mouse is pressed.

. InnerClassDemo is a top level class that extends Applet

. MyMouseAdapter is an inner class that extends MouseAdapter.

As, MyMouseAdapter is defined within the scope of InnerClassDemo, it has access to all the variables and methods within the scope of that class.

.Therefore, the mousePressed() method can call the showStatus() method directly.

It no longer needs to do this via a stored reference to the applet.

It is no longer necessary to pass MyMouseAdapter() a reference to the invoking object.

```
// inner class demo
import java.awt.event.*; // Contains all Event Listener Interfaces
import java.applet.*; // For Applets
/*
<applet code = "InnerClassDemo" width=500 height=300>
</applet>
*/
public class InnerClassDemo extends Applet {
public void init() {
addMouseListener(new MyMouseAdapter());
}
class MyMouseAdapter extends MouseAdapter { // class within a class
public void mousePressed(MouseEvent me) {
showStatus("Mouse Pressed");
}

}
}
```

3. Explain keyEvent and mouseEvent classes with example.

The KeyEvent Class

A KeyEvent is generated when keyboard input occurs. There are three types of key events, which are identified by these integer constants: KEY_PRESSED, KEY_RELEASED, and KEY_TYPED. The first two events are generated when any key is pressed or released. The last event occurs only when a character is generated. Remember, not all keypresses result in characters. For example, pressing SHIFT does not generate a character.

There are many other integer constants that are defined by KeyEvent. For example, VK_0 through VK_9 and VK_A through VK_Z define the ASCII equivalents of the numbers and letters. Here are some others:

VK_ALT VK_DOWN VK_LEFT VK_RIGHT

VK_CANCEL VK_ENTER VK_PAGE_DOWN VK_SHIFT

VK_CONTROL VK_ESCAPE VK_PAGE_UP VK_UP

The VK constants specify virtual key codes and are independent of any modifiers, such as control, shift, or alt.

KeyEvent is a subclass of InputEvent. Here is one of its constructors:

KeyEvent(Component src, int type, long when, int modifiers, int code, char ch)

Here, src is a reference to the component that generated the event. The type of the event is specified

by type. The system time at which the key was pressed is passed in when. The modifiers argument indicates which modifiers were pressed when this key event occurred. The virtual key code, such as VK_UP, VK_A, and so forth, is passed in code. The character equivalent (if one exists) is passed in ch. If no valid character exists, then ch contains CHAR_UNDEFINED. For KEY_TYPED

events, code will contain VK_UNDEFINED.

The KeyEvent class defines several methods, but the most commonly used ones are getKeyChar( ), which returns the character that was entered, and getKeyCode( ), which returns the key code. Their general forms are

shown here:

char getKeyChar( ) int

getKeyCode( )

If no valid character is available, then getKeyChar( ) returns CHAR_UNDEFINED.

When a KEY_TYPED event occurs, getKeyCode( ) returns VK_UNDEFINED.

The MouseEvent Class

There are eight types of mouse events. The MouseEvent class defines the following integer constants that can be used to identify them:

MOUSE_CLICKED The user clicked the mouse.

MOUSE_DRAGGED The user dragged the mouse.

MOUSE_ENTERED The mouse entered a component.

MOUSE_EXITED The mouse exited from a component.

MOUSE_MOVED The mouse moved.

MOUSE_PRESSED The mouse was pressed.

MOUSE_RELEASED The mouse was released.

MOUSE_WHEEL The mouse wheel was moved.

MouseEvent is a subclass of InputEvent. Here is one of its constructors:

MouseEvent(Component src, int type, long when, int modifiers,

int x, int y, int clicks, boolean triggersPopup)

Here, src is a reference to the component that generated the event. The type of the event is specified by type. The system time at which the mouse event occurred is passed in when. The modifiers argument indicates which modifiers were pressed when a mouse event occurred. The coordinates of the mouse are passed in x and y. The click count is passed in clicks. The triggersPopup flag indicates if this event causes a pop-up menu to appear on this platform.

Two commonly used methods in this class are getX( ) and getY( ). These return the X

and Y coordinates of the mouse within the component when the event occurred. Their

forms are shown here:

int getX( )

int getY( )

Alternatively, you can use the getPoint( ) method to obtain the coordinates of the mouse.

It is shown here:

Point getPoint( )

It returns a Point object that contains the X,Y coordinates in its integer members: x

and y. The translatePoint( ) method changes the location of the event. Its form is

shown here:

void translatePoint(int x, int y)

Here the arguments x and y are added to the coordinates of the event.

The getClickCount() method obtains the number of mouse clicks for this event.

It signature is -

int getClickCount()

The isPopupTrigger() method tests if this event causes a pop-up menu to appear on this platform. Its

form is shown here

boolean isPopupTrigger()

Also available is the getButton() method, shown here

int getButton()

It returns a value that represents the button that caused the event. The return value will be one of

these constants defined by MouseEvent.

4. Explain how delegation event model is used to handle events with an example.

THE DELEGATION EVENT MODEL:

The modern approach to handle events is based on the delegation event model.

This 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.

The listener simply waits until it receives an event. Once the 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 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 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 process, and it wasted valuable time. The delegation event model eliminates this overhead.

USING THE DELEGATION EVENT MODEL:

To use the delegation event model follow these two steps:

◦ Implement the appropriate interface I the listener so that it will receive the type of event desired.

◦ Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications.

A source may generate several type of events. Each such event is registered separately.

An object may register to receive several types of events, but it must implement all of the interfaces that are required to receive these events.

To see how the delegation model works in practice, we will look at examples that handle two commonly used event generators: the mouse and the keyboard.

HANDLING KEYBOARD EVENTS:

When a key is pressed, a KEY_PRESSED event is generated. This results in a call to the keyPressed() event handler.

When a 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 generated and the KeyTyped() handler is invoked.

Each time a key is pressed, two to three events are generated.

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.*; // Contains all AWT based event classes used by Delegation
Event Model
import java.awt.event.*; // Contains all Event Listener Interfaces
import java.applet.*; // For Applets
/*
<applet code = "SimpleKey" width=500 height=300>
</applet>
*/
public class SimpleKey extends Applet
implements KeyListener {
String msg = "";
int X = 10, Y = 20; // Output Coordinates
// Listener must register with the source. General form -
// public void addTypeListener (TypeListener el) Type i s the name of the event
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);
}
}
```

5. Describe Synchronization. Write an example program for implementing static Synchronization.

Synchronization:

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization.

Key to synchronization is the concept of the monitor (also called a semaphore).

A monitor is an object that is used as a mutually exclusive lock or mutex only one thread can own a monitor at a give time.

When a thread acquires a lock, it is said to have entered the monitor.

All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor.

A thread that owns a monitor can reenter the same monitor if it so desires.

As Java implements synchronization through language elements, most of the complexity associated with synchronization has been eliminated.

Two ways of synchronization:

We can synchronize code in two ways

1. Using synchronized methods
2. using synchronized statement

1. Using Synchronized Methods:

Synchronization is easy in Java, because all objects have their own implicit monitor associated with them.

To enter an object’s monitor, just call a method that has been modified with the synchronized keyword.

While a thread is inside a synchronized method, all other threads that try to call it on the same instance have to wait.

To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

Program Explanation:

The example program below has three classes

The first one, Callme class has a single method named call(). The call() method takes a String parameter called msg. This method tries to print the msg string inside of square brackets. But, after call() prints the opening bracket and the msg string, it calls Thread.sleep(1000), which pauses the current thread for one second.

The second class is Caller. The constructor of this class takes a reference to an instance of the Callme class and a String, which are stored in target and msg, respectively.

The constructor also creates a new thread that will call this object's run() method. The thread started immediately.

The run() method of caller calls the call() method on the target instance of Callme, passing in the msg string.

Finally, the third class, Synch class starts by creating a single instance of Callme, and three instances of Caller, each with a unique message string. The same instance of Callme is passed to each Caller.

we must serialize access to call(). That is, we must restrict its access to only one thread at a time.

To do this, we have to precede call()'s definition with the keyword synchronized as shown

```
class Callme {
synchronized void call (String msg) {
....
```

This prevents other threads from entering call() while another thread is using it.

Program:

```
// This program is synchronized
class Callme {
synchronized void call(String msg) {
System.out.print("[" + msg);
try {
Thread.sleep(1000);
}
catch (InterruptedException e) {
System.out.println("Interrupted");
}
System.out.println("]");
}
}
class Caller implements Runnable {
String msg;
Callme target;
Thread t;
public Caller(Callme targ, String s) {
target = targ;
msg = s;
t = new Thread (this);
t.start();
}
public void run() {
target.call(msg);
}
}
class Synch {
public static void main(String args[]) {

Callme target = new Callme();
Caller ob1 = new Caller (target, "Hello");
Caller ob2 = new Caller(target, "Synchronized");
Caller ob3 = new Caller (target, "World");
// wait for the threads to end
```

```
try {
ob1.t.join();
ob2.t.join();
ob3.t.join();
}
catch (InterruptedException e) {
System.out.println("Interrupted");
}
}
}
```

Output:

```
$ javac Synch.java
$ java Synch
[Hello]
[World]
[Synchronized]
```

2. The synchronized Statement:

Creating synchronized methods within classes that you create is an easy and effective means of achieving synchronization, but, it will not work in all cases.

If you want to synchronize access to objects of a class that was not designed for multi-threaded access, the class does not use synchronized methods.

If the class was not created by you, but by a third party and you do not have access to the source code, you can't add synchronized to the appropriate methods within the class.

The solution to this problem is you put calls to the methods defined by this class inside a synchronized block.

The general form of the synchronized statement

```
synchronized (object) {
// Statements to be synchronized
}
```

The object is a reference to the object being synchronized.

A synchronized block ensures that a call to a method that is a member of object occurs only after the current thread has successfully entered object’s monitor.

Example program which uses synchronized block within run() method

```
// This program uses synchronized block
class Callme {
void call(String msg) {
System.out.print("[" + msg);
try {

Thread.sleep(1000);
}
catch (InterruptedException e) {
System.out.println("Interrupted");
}
System.out.println("]");
}
}
class Caller implements Runnable {
String msg;
Callme target;
Thread t;
public Caller(Callme targ, String s) {
target = targ;
msg = s;
t = new Thread (this);
t.start();
}
// Synchronize calls to call()
public void run() {
synchronized (target) { // synchronized block
target.call(msg);
}
}
}
class Synch1 {
public static void main(String args[]) {
Callme target = new Callme();
Caller ob1 = new Caller (target, "Hello");
Caller ob2 = new Caller(target, "Synchronized");
Caller ob3 = new Caller (target, "World");
// wait for the threads to end
try {
ob1.t.join();
ob2.t.join();
ob3.t.join();
}
catch (InterruptedException e) {
System.out.println("Interrupted");
}
}
}
```

Output:

```
$ javac Synch1.java
$ java Synch1
[Hello]
[World]
[Synchronized]
```

Program Explanation:

In the above program, the call() method is not modified by synchronized. Instead, the synchronized statement is used inside Caller’s run() method.

6. Explain with syntax and example the following: JLabel and JCheckBox

Jlabel

Jlabel is Swing's easiest-to-use component.

It creates a label. It can be used to display text and/or an icon.

It is a passive component, it does not respond to user input.

Jlabel defines several constructors. The three among them are

◦ Jlabel(Icon icon)

◦ Jlabel(String str)

◦ Jlabel(String str, Icon icon, int align)

str and icon are the text and icon used for the label. The align argument specifies the horizontal alignment of the text and/or icon within the dimensions of the label. It must be one of the following values

◦ LEFT

◦ RIGHT

◦ CENTER

◦ LEADING

◦ TRAILING

These are the constants defined in the SwingConstants interface.

JCheckBox

The JCheckBox class provides the functionality of a check box.

Its immediate superclass is JtoggleButton which provides support for two-state buttons.

JCheckBox defines several constructors. One among them is

◦ JCheckBox(String str)

It creates a check box that has the text specified by str as a label.

When the user selects or deselects a check box, an ItemEvent is generated.

We can obtain a reference t the JCheckBox that generated the event by calling getItem() on the ItemEvent passed t the itemStateChanged() method defiend by ItemListener.

The easier way to determine the selected state of a check box is to call isSelected() on the JCheckBox instance.

7. Explain with syntax and example the following: JButton and JRadioButton

JButton

The JButton class provides the functionality of a push button.

Jbutton allows an icon, a string or both to be associated with the push button.

Three of its constructors are -

◦ Jbutton (Icon icon)

◦ Jbutton (String str)

◦ Jbutton (String str, Icon icon)

Here, str and icon are the string and icon used for the button.

When the button is pressed, an ActionEvent is generated.

Using the ActionEvent object passed to the actionPerformed() method of the registered

ActionListener, you can obtain the action command string associated with the button.

By default this is the string displayed inside the button. We can set the action command by

calling setActionCommand() on the button.

We can obtain the action command by calling getActionCommand() on the event object.

Radio buttons are a group of mutually exclusive buttons, in which only one button can be selected at any one time. They are supported by the JRadioButton class, which extends JToggleButton. JRadioButton provides several constructors. The one used in the example is shown here:

JRadioButton(String str)

Here, str is the label for the button. Other constructors let you specify the initial selection state of the button and specify an icon.

In order for their mutually exclusive nature to be activated, radio buttons must be configured into a group. Only one of the buttons in the group can be selected at any time. For example, if a user presses a radio button that is in a group, any previously selected button in that group is automatically deselected. A button group is created by the ButtonGroup class. Its default constructor is invoked for this purpose. Elements are then added to the button group via the following method:

void add(AbstractButton ab)

Here, ab is a reference to the button to be added to the group.

A JRadioButton generates action events, item events, and change events each time the button selection changes. Most often, it is the action event that is handled, which means that you will normally implement the ActionListener interface. Recall that the only method defined by ActionListener is actionPerformed( ). Inside this method, you can use a number of different ways to determine which button was selected. First, you can check its action command by calling getActionCommand( ). By default, the action command is the same as the button label, but you can set the action command to something else by calling setActionCommand( ) on the radio button. Second, you can call getSource( ) on the ActionEvent object and check that reference against the buttons. Finally, you can simply check each radio button to find out which one is currently selected by calling isSelected( ) on each button. Remember, each time an action event occurs, it means that the button being selected has changed and that one and only one button will be selected.

```
// Java program to show JRadioButton Example.
//in java. Importing different Package.
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

class Demo extends JFrame {

        // Declaration of object of JRadioButton class.
        JRadioButton jRadioButton1;

        // Declaration of object of JRadioButton class.
        JRadioButton jRadioButton2;

        // Declaration of object of JButton class.
        JButton jButton;

        // Declaration of object of ButtonGroup class.
        ButtonGroup G1;

        // Declaration of object of JLabel class.
        JLabel L1;

        // Constructor of Demo class.
        public Demo()
        {

                // Setting layout as null of JFrame.
                this.setLayout(null);

                // Initialization of object of "JRadioButton" class.
                jRadioButton1 = new JRadioButton();

                // Initialization of object of "JRadioButton" class.
                jRadioButton2 = new JRadioButton();

                // Initialization of object of "JButton" class.
                jButton = new JButton("Click");

                // Initialization of object of "ButtonGroup" class.
                G1 = new ButtonGroup();

                // Initialization of object of " JLabel" class.
                L1 = new JLabel("Qualification");
```

```
// setText(...) function is used to set text of radio button.
// Setting text of "jRadioButton2".
jRadioButton1.setText("Under-Graduate");

// Setting text of "jRadioButton4".
jRadioButton2.setText("Graduate");

// Setting Bounds of "jRadioButton2".
jRadioButton1.setBounds(120, 30, 120, 50);

// Setting Bounds of "jRadioButton4".
jRadioButton2.setBounds(250, 30, 80, 50);

// Setting Bounds of "jButton".
jButton.setBounds(125, 90, 80, 30);

// Setting Bounds of JLabel "L2".
L1.setBounds(20, 30, 150, 50);

// "this" keyword in java refers to current object.
// Adding "jRadioButton2" on JFrame.
this.add(jRadioButton1);

// Adding "jRadioButton4" on JFrame.
this.add(jRadioButton2);

// Adding "jButton" on JFrame.
this.add(jButton);

// Adding JLabel "L2" on JFrame.
this.add(L1);

// Adding "jRadioButton1" and "jRadioButton3" in a Button Group "G2".
G1.add(jRadioButton1);
G1.add(jRadioButton2);

// Adding Listener to JButton.
jButton.addActionListener(new ActionListener() {
        // Anonymous class.

        public void actionPerformed(ActionEvent e)
        {
                // Override Method

                // Declaration of String class Objects.
                String qual = " ";

                // If condition to check if jRadioButton2 is selected.
                if (jRadioButton1.isSelected()) {
```

```
                                        qual = "Under-Graduate";
                                }

                                else if (jRadioButton2.isSelected()) {

                                        qual = "Graduate";
                                }
                                else {

                                        qual = "NO Button selected";
                                }

                                // MessageDialog to show information selected radion buttons.
                                JOptionPane.showMessageDialog(Demo.this, qual);
                        }
                });
        }
}

public class JRadioButtonDemo {
        // Driver code.
        public static void main(String args[])
        { // Creating object of demo class.
                Demo f = new Demo();

                // Setting Bounds of JFrame.
                f.setBounds(100, 100, 400, 200);

                // Setting Title of frame.
                f.setTitle("RadioButtons");

                // Setting Visible status of frame as true.
                f.setVisible(true);
        }
}
```

8. What are Events, Event listener and Event sources?

EVENTS:

An event is an object that describes a change of state 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 events 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 complete.

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.

The general form is :

public void addTypeListener (TypeListener el)

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 addKeyListner().

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 to listeners that register to receive them.

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

public void addTypeListener (TypeListener el) throws

java.util.TooMayListenersException

Here, Type is the name of the event, and el is a reference to the event listener.

When such an event occurs, the registered listener is notified. This is known as unicasting the event.

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 -

public void removeTypeListener (TypeListener el)

Here, Type is the name of the event, and el is a reference to the event listener.

For example, to remove a keyboard listener, call removeKeyListener().

The methods that add or remove listeners are provided by the source that generates events.

For example, the Component class provides methods to add and remove keyboard and mouse event listener.

SOURCES OF EVENTS:

The table lists some of the user interface components that can be generate the events. In addition to these graphical user interface elements, any class derived from Component, such as Applet, can generate events.

EVENT LISTENERS:

A listener is an object that is notified when an event occurs.

It has two major requirements.

First it must have registered with one or more sources to receive notifications about specific type 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.

Any object may receive and process one or both these events if it provides an implementation of this interface.