**THREAD SYNCHONIZATION**

Write a Java program that shows thread synchronization

## THEORY

When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issues. For example, if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file.

So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is implemented using a concept called **monitors**. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.

Java programming language provides a very handy way of creating threads and synchronizing their task by using **synchronized** blocks.

## Syntax

synchronized(objectidentifier)

{

// Access shared variables and other shared resources

}

Here, the **objectidentifier** is a reference to an object whose lock associates with the

monitor that the synchronized statement represents. Now we are going to see two examples, where we will print a counter using two different threads. When threads are not synchronized, they print counter value which is not in sequence, but when we print counter by putting inside synchronized() block, then it prints counter very much in sequence for both the threads.

## ALGORITHM

Step 1: Create a class first and define a function display in it Step 2: Declare the function display as synchronized

Step 3: Create a class second which is the subclass of first which contains a constructor and run() method

Step 4: Create a class ThreadSynSample which contains the main method Step 5: Create an object fnew for the class first

Step 5: Create three objects for the class second with parameters fnew and welcome, new, programmer

Step 6: Stop

## PROGRAM

import java.lang.\*; class first

{

synchronized public void display(String msg)

{

System.out.print("["+msg); try

{

Thread.sleep(1000);

}

catch(InterruptedException e)

{

e.printStackTrace();

}

System.out.println("]");

}

}

class second extends Thread

{

String msg; first fobj;

second(first fp,String str)

{

fobj=fp; msg=str; start();

}

public void run()

{

fobj.display(msg);

}

}

class ThreadSynSample

{

public static void main(String args[])

{

first fnew=new first();

second ss=new second(fnew,"welcome"); second ss1=new second(fnew,"new");

second ss2=new second(fnew,"programmer");

}

}

## OUTPUT

[welcome] [programmer] [new]

## AIM

**Exercise No: 14 SIMPLE CALCULATOR**

Write a Java program that works as a simple calculator. Arrange Buttons for digits and

the + - \* % operations properly. Add a text field to display the result. Handle any possible exceptions like divide by zero. Use Java Swing.

## THEORY

**Event Handling**

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism has the code which is known as event handler that is executed when an event occurs. Java Uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events.

The Delegation Event Model has the following key participants namely:

* **Source** - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to it's handler. Java provide as with classes for source object.
* **Listener** - It is also known as event handler.Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received , the listener process the event an then returns.

## Steps involved in event handling

* The User clicks the button and the event is generated.
* Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.
* Event object is forwarded to the method of registered listener class.
* The method is now get executed and returns.

## Events

The Events are the objects that define state change in a source. An event can be generated as a reaction of a user while interacting with GUI elements. Some of the event generation activities are moving the mouse pointer, clicking on a button, pressing the keyboard key,

selecting an item from the list, and so on. We can also consider many other user operations as events.

The Events may also occur that may be not related to user interaction, such as a timer expires, counter exceeded, system failures, or a task is completed, etc. We can define events for any of the applied actions.

## Event Sources

A source is an object that causes and generates an event. It generates an event when the internal state of the object is changed. The sources are allowed to generate several different types of events.

A source must register a listener to receive notifications for a specific event. Each event contains its registration method. Below is an example:

public void addTypeListener (TypeListener e1)

From the above syntax, the Type is the name of the event, and e1 is a reference to the event listener. For example, for a keyboard event listener, the method will be called

as **addKeyListener()**. For the mouse event listener, the method will be called

as **addMouseMotionListener()**. When an event is triggered using the respected source, all the events will be notified to registered listeners and receive the event object. This process is known as event multicasting. In few cases, the event notification will only be sent to listeners that register to receive them.

Some listeners allow only one listener to register. Below is an example: public void addTypeListener(TypeListener e2)

From the above syntax, the Type is the name of the event, and e2 is the event listener's reference. When the specified event occurs, it will be notified to the registered listener. This process is known as **unicasting** events.

A source should contain a method that unregisters a specific type of event from the listener if not needed. Below is an example of the method that will remove the event from the listener.

public void removeTypeListener(TypeListener e2?)

From the above syntax, the Type is an event name, and e2 is the reference of the listener. For example, to remove the keyboard listener, the **removeKeyListener()** method will be called.

The source provides the methods to add or remove listeners that generate the events. For example, the Component class contains the methods to operate on the different types of events, such as adding or removing them from the listener.

## Event Listeners

An event listener is an object that is invoked when an event triggers. The listeners require two things; first, it must be registered with a source; however, it can be registered with several resources to receive notification about the events. Second, it must implement the methods to receive and process the received notifications.

The methods that deal with the events are defined in a set of interfaces. These interfaces can be found in the java.awt.event package.

For example, the **MouseMotionListener** interface provides two methods when the mouse is dragged and moved. Any object can receive and process these events if it implements the MouseMotionListener interface.

## PROGRAM

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

public class calculator extends JFrame implements ActionListener

{

JButton b10,b11,b12,b13,b14,b15; JButton b[]=new JButton[10];

int i,r,n1,n2; JTextField res; char op;

public calculator()

{

super("Calulator"); setLayout(new BorderLayout()); JPanel p=new JPanel();

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

for(int i=0;i<=9;i++)

{

b[i]=new JButton(i+"");

p.add(b[i]); b[i].addActionListener(this);

}

b10=new JButton("+"); p.add(b10); b10.addActionListener(this);

b11=new JButton("-"); p.add(b11); b11.addActionListener(this);

b12=new JButton("\*"); p.add(b12); b12.addActionListener(this);

b13=new JButton("/"); p.add(b13); b13.addActionListener(this);

b14=new JButton("="); p.add(b14); b14.addActionListener(this);

b15=new JButton("C"); p.add(b15); b15.addActionListener(this);

res=new JTextField(10);

add(p,BorderLayout.CENTER); add(res,BorderLayout.NORTH); setVisible(true); setSize(200,200);

}

public void actionPerformed(ActionEvent ae)

{

JButton pb=(JButton)ae.getSource(); if(pb==b15)

{

r=n1=n2=0; res.setText("");

}

else

if(pb==b14)

{

n2=Integer.parseInt(res.getText()); eval();

res.setText(""+r);

}

else

{

boolean opf=false; if(pb==b10)

{ op='+';

opf=true;

}

if(pb==b11)

{ op='-';opf=true;} if(pb==b12)

{ op='\*';opf=true;} if(pb==b13)

{ op='/';opf=true;}

if(opf==false)

{

for(i=0;i<10;i++)

{

if(pb==b[i])

{

String t=res.getText(); t+=i;

}

int eval()

{

}

}

else

{

}

}

res.setText(t);

}

n1=Integer.parseInt(res.getText()); res.setText("");

switch(op)

{

case '+': r=n1+n2; break; case '-': r=n1-n2; break; case '\*': r=n1\*n2; break; case '/': r=n1/n2; break;

}

}

return 0;

AIM-

public static void main(String arg[])

{

new calculator();

}

}

**Program**

**TRAFFIC LIGHT**

Write a Java program that simulates a traffic light. The program lets the user select one of

three lights: red, yellow, or green. When a radio button is selected, the light is turned on, and only one light can be on at a time. No light is on when the program starts.

**THEORY**

An applet is a special kind of Java program that runs in a Java enabled browser. This is the first Java program that can run over the network using the browser. Applet is typically embedded inside a web page and runs in the browser.

In other words, we can say that Applets are small Java applications that can be accessed on an Internet server, transported over Internet, and can be automatically installed and run as a part of a web document.

After a user receives an applet, the applet can produce a graphical user interface. It has limited access to resources so that it can run complex computations without introducing the risk of viruses or breaching data integrity.

**Lifecycle of Java Applet**

Following are the stages in Applet

1. Applet is initialized.
2. Applet is started
3. Applet is painted.
4. Applet is stopped.
5. Applet is destroyed.

**Circles and Ellipses**

The Graphics class does not contain any method for circles or ellipses. To draw an ellipse, use drawOval(). To fill an ellipse, use fillOval().

**Syntax**

void drawOval(int top, int left, int width, int height) void fillOval(int top, int left, int width, int height)

The ellipse is drawn within a bounding rectangle whose upper-left corner is specified by (top,left) and whose width and height are specified by width and height. To draw a circle, specify a square as the bounding rectangle i.e get height = width.

**Event Handling**

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism has the code which is known as event handler that is executed when an event occurs. Java Uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events.Let's have a brief introduction to this model.

The Delegation Event Model has the following key participants namely:

* **Source** - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to it's handler. Java provide as with classes for source object.
* **Listener** - It is also known as event handler.Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received , the listener process the event an then returns.

**Steps involved in event handling**

* The User clicks the button and the event is generated.
* Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.
* Event object is forwarded to the method of registered listener class.
* The method is now get executed and returns.

**Components of Event Handling**

Event handling has three main components,

* **Events :** An event is a change in state of an object.
* **Events Source :** Event source is an object that generates an event.
* **Listeners :** A listener is an object that listens to the event. A listener gets notified when an event occurs.

**PROGRAM**

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

/\*<applet code="Signals" width=400 height=250></applet>\*/ public class Signals extends Applet implements ItemListener

{

String msg=""; Checkbox stop,ready,go; CheckboxGroup cbg; public void init()

{

cbg = new CheckboxGroup();

stop = new Checkbox("Stop", cbg, false); ready = new Checkbox("Ready", cbg, false); go= new Checkbox("Go", cbg, false); add(stop);

add(ready); add(go);

stop.addItemListener(this); ready.addItemListener(this); go.addItemListener(this);

}

public void itemStateChanged(ItemEvent ie)

{

repaint();

}

public void paint(Graphics g)

{

msg=cbg.getSelectedCheckbox().getLabel();

g.drawOval(165,40,50,50); g.drawOval(165,100,50,50); g.drawOval(165,160,50,50); if(msg.equals("Stop"))

{

g.setColor(Color.red); g.fillOval(165,40,50,50);

}

else if(msg.equals("Ready"))

{

g.setColor(Color.yellow); g.fillOval(165,100,50,50);

}

else

{

g.setColor(Color.green); g.fillOval(165,160,50,50);

}

}

}