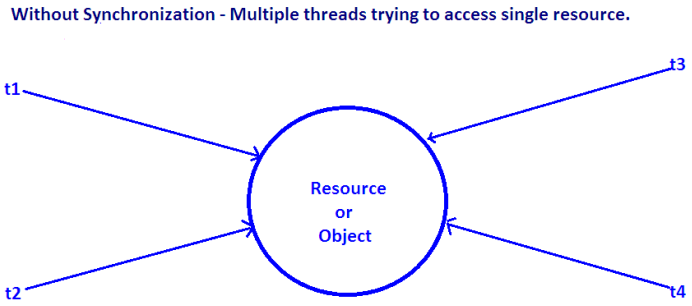
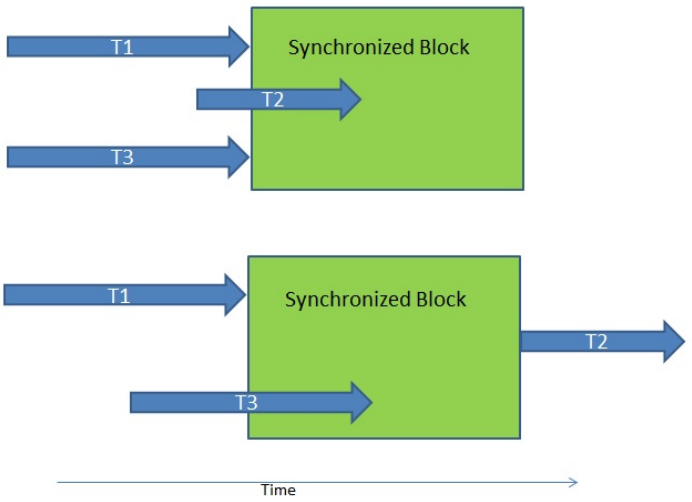
OOC IAT3 Solution 18CS45 2021-22 Even Sem

1. **What is synchronization? when do we use it? How synchronization can be achieved for threads in Java explain with example programs.**

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 given 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.





In Synchronized Block, other threads will have to wait when one thread is in

Two ways of synchronization:
We can synchronize code in two ways. Both involve the use of the **synchronized** keyword.

1. Using synchronized methods
2. using synchronized statement

1. Using Synchronized Methods:

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.

To understand the need of synchronization, first let's see an example that does not use synchronization even though it is required. 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 is 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 here

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

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

```
// Example program to illustrate synchronization using synchronized method
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");
        }
        }
}
```

After synchronized has been added to call(), the output of the program is as follows

Output:

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

If **synchronized** is not added to call(), then we are not serializing access to call(). That is, we are not restricting its access to only one thread at a time. So the output obtained is as shown below

Output:
$ javac Synch.java
$ java Synch
[World[Hello[Synchronized]
]
]

In the above output, by calling sleep(), the call() method allows execution to switch to another thread. This results in the mixed up output of the three message strings.

Nothing exists to stop all three threads from calling the same method, on the same object, at the same time. This is known as race condition, because the three threads are racing each other to complete the method.

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.

1. 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.
2. 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 the **synchronized** modifier 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 the object's monitor.

Example program which uses synchronized block within run() method

```
// This program uses synchronized block
class Callme  {
        void call(String msg)  {  // method is not synchronizzed
        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 - 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 the object's monitor.

            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]
```

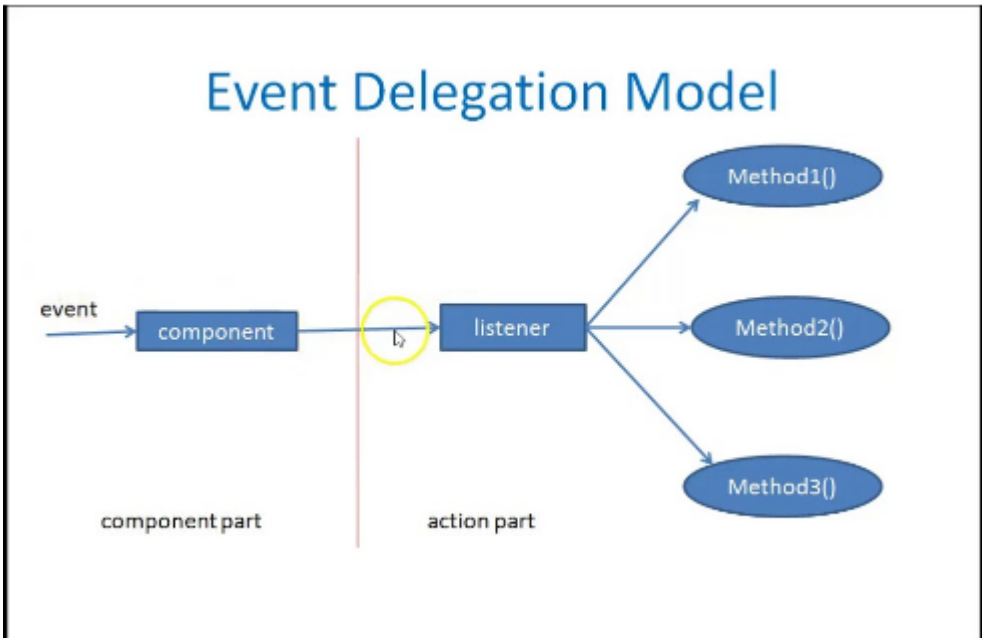
Write a Java program to create two threads one thread displays "CMRIT" and the other thread displays "BENGALURU" on the screen continuously.

```
class MyThread extends Thread
{
  String str;
   public MyThread(String s)
   {
      str=s;
   }
   public void run()
   {
      while(true)
      {   System.out.print(str + " ");
         try
         {
            Thread.sleep(1000);
         }
         catch(InterruptedException ie)
         {
            System.out.println(ie.toString());
         }
      }
   }
}
public class ThreadSetA1
{
   public static void main(String args[])
   {
      MyThread t1= new MyThread("CMRIT");
      MyThread t2= new MyThread("BENGALURU");
      t1.start();
      t2.start();
   }
}
```

```
Output:
java ThreadSetA1
CMRIT BENGALURU CMRIT BENGALURU CMRIT BENGALURU CMRIT
BENGALURU CMRIT BENGALURU CMRIT BENGALURU CMRIT
BENGALURU CMRIT BENGALURU CMRIT BENGALURU CMRIT
BENGALURU CMRIT BENGALURU ^C
```

Explain the event delegation model in Java. Write a Java program to handle mouse events

THE DELEGATION EVENT MODEL:



In Delegation Event Model, 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.

To handle mouse events, we must implement the **MouseListener** and **MouseMotionListener** interfaces. The applet displays the current coordinates of the mouse in the applet's status window. Each time a button is pressed, the word **"Down" is displayed at the location of the mouse pointer.** Each time the button is released, the word "**Up**" is shown. If a button is clicked, the message, "**Mouse clicked" is displayed in the upper-left corner of the applet display area**.

As the mouse enters or exits the applet window, a message is displayed in the upper-left corner of the applet display area. When dragging the mouse, a '*' is shown, which tracks with the mouse pointer as it is dragged. The two variables **mouseX** and **mouseY**, store the location of the mouse when a mouse pressed, released or dragged event occurs. These coordinates are then used by paint() to display output at the point of these occurrences.

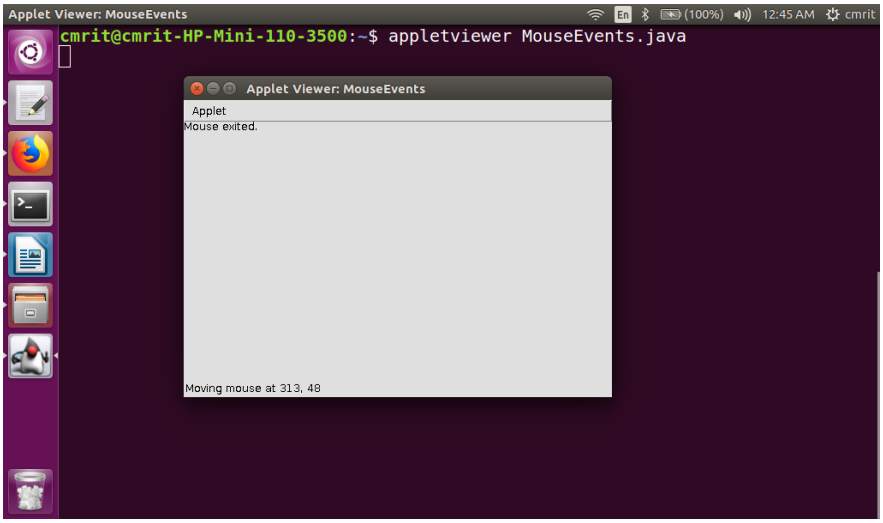
```
// demonstrate the mouse event handlers
import java.awt.*;   // Contains all AWT based event classes used by Delegation Event Model
import java.awt.event.*;      // Contains all Event Listener Interfaces
import java.applet.*;         // For Applets

/*
<applet code = "MouseEvents" width=500 height=300>
</applet>
*/
```

```
public class MouseEvents extends Applet
        implements MouseListener, MouseMotionListener  {
                                // MouseListener defines 5 methods
                                // MouseMotionListener defines 2 methods
                String msg = "";
                int mouseX = 0, mouseY = 0;  // Coordinates of mouse

                // Listener must register with the source. General form -
                // public void addTypeListener (TypeListener el)   Type is the name of
the event

                public void init()  {
                        addMouseListener(this);  //applet registers itself as listener for
mouse events
                        addMouseMotionListener(this);
                }

                // Handle mouse clicked
                public void mouseClicked(MouseEvent me)  {
                        // Save coordinates
                        mouseX = 0;
                        mouseY = 10;
                        msg = "Mouse clicked";
                        repaint();
                }

                // Handle mouse entered
                public void mouseEntered (MouseEvent me)  {
                        // Save coordinates
                        mouseX = 0;
                        mouseY = 10;
                        msg = "Mouse Entered.";
                        repaint();
                }

                // Handle mouse exited.
                public void mouseExited(MouseEvent me)  {
                        // Save coordinates
                        mouseX = 0;
                        mouseY = 10;
                        msg = "Mouse exited.";
                        repaint();
                }

                // Handle button pressed
                public void mousePressed(MouseEvent me)  {
                        // Save coordinates
                        mouseX = me.getX();
                        mouseY = me.getY();
                        msg = "Down";
```

```
                repaint();
        }

        // Handle button released
        public void mouseReleased(MouseEvent me)  {
                // Save coordinates
                mouseX = me.getX();
                mouseY = me.getY();
                msg = "Up";
                repaint();
        }

        // Handle mouse dragged
        public void mouseDragged(MouseEvent me)  {
                // Save coordinates
                mouseX = me.getX();
                mouseY = me.getY();
                msg = "*";
                showStatus("Dragging mouse at  " + mouseX + ", " + mouseY);
                repaint();
        }

        // Handle mouse moved
        public void mouseMoved(MouseEvent me)  {
                // Show status
                showStatus( "Moving mouse at " + me.getX() + ", " +
me.getY());
        }

        // Display message in applet window at current X,Y location
        public void paint(Graphics g)  {
                g.drawString(msg, mouseX, mouseY);
        }
}
```

$ javac MouseEvents.java
$ appletviewer MouseEvents.java

Output:



Program Explanation:

- The **MouseEvent** class extends **Applet** and implements both the **MouseListener** and **MouseMotionListener** interfaces. You may also want to implement MouseWheelListener, but it is not shown in this example program.
- These two interfaces contain methods that receive and process the various types of mouse events.
- The applet is both the source and listener for these events, because **Component** which supplies the **addMouseListener**() and **addMouseMotionListener**() methods, is a superclass of Applet. (Being both the source and the listener for events is a common situation for applets)
- Inside **init**(), the applet registers itself as a listener for mouse events. This is done by using **addMouseListener**() and **addMouseMotionListener**(), which are members of **Component**.
- The applet then implements all of the methods defined by the MouseListener and MouseMotionListener interfaces. These are the event handlers for the various mouse events. Each method handles its event and then returns.

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 dragged 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**

Adapter Class	Listener Interface
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener and (as of JDK 6) MouseMotionListener and MouseWheelListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener, WindowFocusListener, and WindowStateListener

Example program to demonstrate an adapter:

- It displays a message in the status bar of an applet or browser when the mouse is **clicked or dragged. No action is taken when the mouse is moved.**
- All other mouse events are silently ignored.
- The program has three classes
  1. **AdapterDemo extends Applet.**
     - Its init() method creates an instance of MyMouseAdapter and registers that object to receive notifications of mouse events.
     - It also creates an instance of MyMouseMotionAdapter and registers that object to receive notifications of mouse events
     - Both of the constructors take 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.
  3. **MyMouseMotionAdapter extends MouseMotionAdapter and overrides the mouseDragged() method.**
     - The other mouse motion event is silently ignored by code inherited from the MouseMotionAdapter class.

```
// Example program to demonstrate an Adapter class
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));
            addMouseMotionListener(new MyMouseMotionAdapter(this));
    }
}

class MyMouseAdapter extends MouseAdapter {
    AdapterDemo adapterDemo;
    public MyMouseAdapter (AdapterDemo adapterDemo)  {
            this.adapterDemo = adapterDemo;
    }

    // Handle mouse clicked
    public void mouseClicked(MouseEvent me)  {
            adapterDemo.showStatus("Mouse Clicked");
    }
}

class MyMouseMotionAdapter extends MouseMotionAdapter {
    AdapterDemo adapterDemo;
    public  MyMouseMotionAdapter (AdapterDemo adapterDemo)  {
            this.adapterDemo = adapterDemo;
    }

    // Handle mouse dragged
    public void mouseDragged(MouseEvent me)  {
            adapterDemo.showStatus("Mouse Dragged");
    }
}
```

INNER CLASSES:

- **inner class is 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.**
- To understand the benefit of inner classes, consider the applet that does not use an inner class.
- 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.
- There are two top level classes in this programming
  1. **MousePressedDemo extends Applet**
     - Its init() method creates an instance of MyMouseAdapter and registers that object to receive notifications of mouse events.
     -

  2. **MyMouseAdapter extends MouseAdapter**
     - Reference to the applet is supplied as an argument to the **MyMouseAdapter** constructor.

- This reference is stored in an instance variable for later use by **mousePressed**() method.
- When the mouse is pressed, it invokes the **showStatus**() method of the applet through the stored applet reference.

```
// This applet does not use an inner class
import java.awt.event.*;
import java.applet.*;
/*
<applet code = "MousePressedDemo" width=500 height=300>
</applet>
*/

public class MousePressedDemo extends Applet {
    public void init()  {
        addMouseListener(new MyMouseAdapter(this));
    }
}

class MyMouseAdapter extends MouseAdapter {
    MousePressedDemo mousePressedDemo;
    public MyMouseAdapter (MousePressedDemo  mousePressedDemo)  {
        this.mousePressedDemo = mousePressedDemo;
    }

    public void mousePressed(MouseEvent me)  {
        mousePressedDemo.showStatus("Mouse Pressed");
    }
}
```

The above program can be improved by using an inner class.
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");
                }
        }
}
```

What is Swing? Differentiate between Swing and AWT Applet. Explain a simple Swing application with program.

Swing is a set of classes that provide a more powerful and flexible GUI component than does the AWT. Swing provides the look and feel of the modern Java GUI.

Difference between swings and AWT Applet

SWING	AWT
Swing is a set of classes that provides more powerful and flexible GUI components.	AWT defines a basic set of controls, windows, and dialog boxes that support a usable, but limited graphical interface. It translates its various visual components into their corresponding, platform-specific equivalents.
Swing supports a pluggable look and feel of the modern Java GUI. Hence it is platform independent.	Look and feel of the component is defined by the platform, not by Java. Hence it is platform dependent.
Swing components are lightweight.	AWT components use native code resources hence they are referred to as heavy weight.
Swing has the main method to execute the program.	AWT need HTML code to run the applet
Swing uses Model-View-Controller (MVC)	Doesn't use MVC

A Simple Swing Application

There are two types of Java programs in which Swing is typically used.

The first is a desktop application.
The second is the applet.

The following program shows one way to write a Swing application. It demonstrates several key features of Swing.

It uses two Swing components: JFrame and JLabel.

JFrame is the top-level container that is commonly used for Swing applications.

JLabel is the Swing component that creates a label, which is a component that displays information. The label is Swing's simplest component because it is passive. That is, a label does not respond to user input. It just displays output.

The program uses a JFrame container to hold an instance of a JLabel. The label displays a short text message.

```
// A simple Swing application.
import javax.swing.*;
class SwingDemo
{
        SwingDemo()
        {
                        // Create a new JFrame container.
                        JFrame jfrm = new JFrame("A Simple Swing Application");

                        // Give the frame an initial size.
                        jfrm.setSize(300, 200);

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

                        // Create a text-based label.
                        JLabel jlab = new JLabel(" Swing means powerful GUIs.");
                        // 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 SwingDemo();
                                }
                        });
        }
}
```

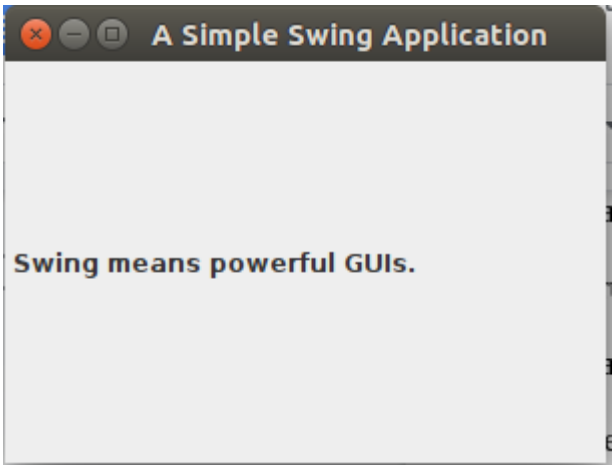
Swing programs are compiled and run in the same way as other Java applications. Thus, to compile this program, you can use this command line:

javac SwingDemo.java

To run the program, use this command line:

java SwingDemo

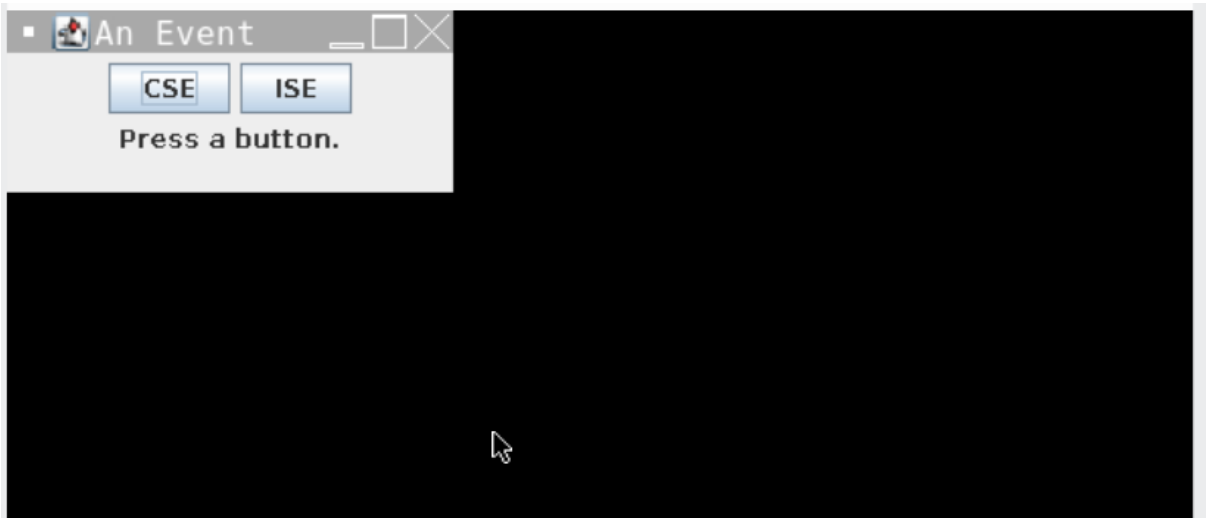
When the program is run, it will produce the window shown in Figure below.

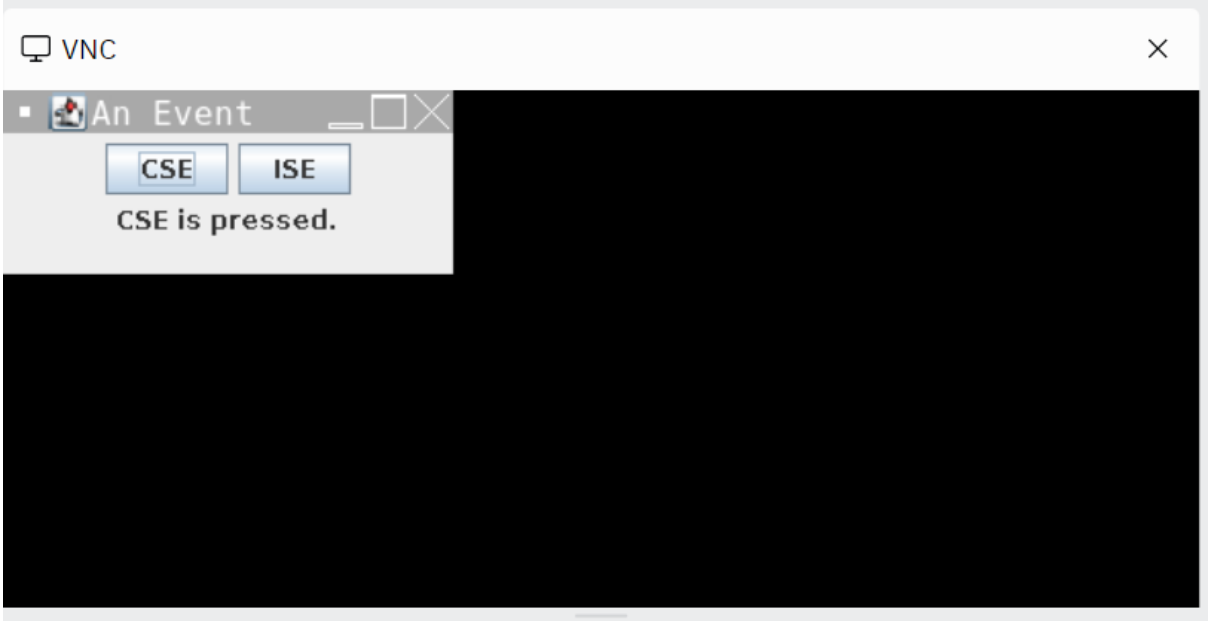


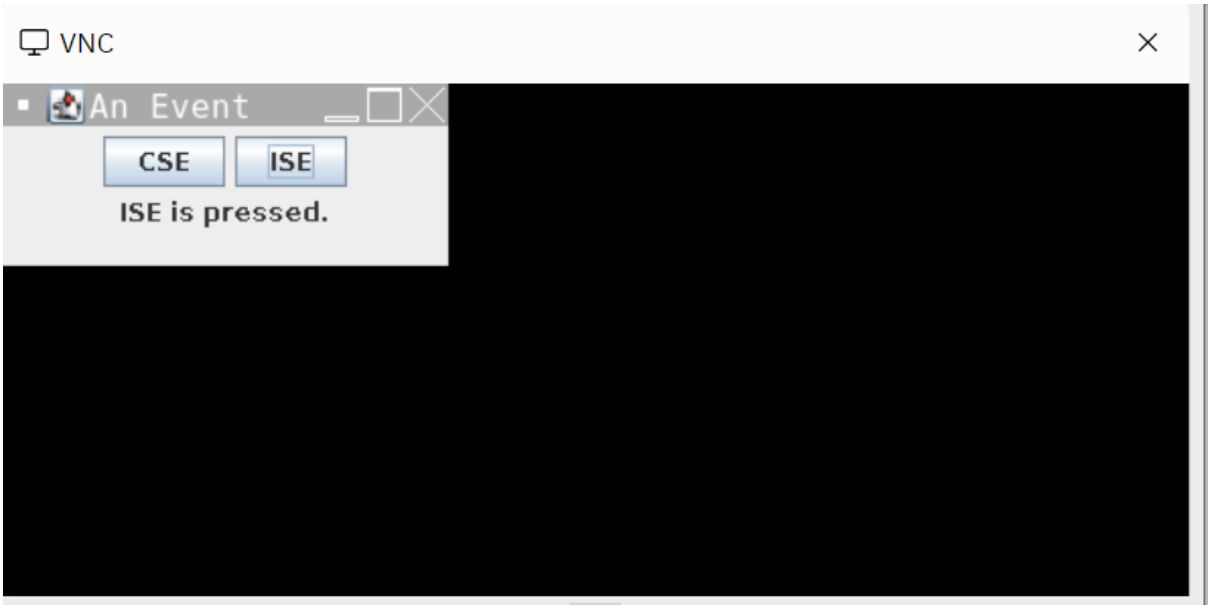
Create a Swing Applet with two buttons named "CSE" and "ISE". When either of the buttons is pressed it should display "CSE" is pressed and "ISE" is pressed respectively

```
// Handle an event in a Swing program.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class cseise
{
    JLabel jlab;
    cseise()
    {
                    // 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 jbtnCse = new JButton("CSE");
                JButton jbtnIse = new JButton("ISE");

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

                // Add action listener for Beta.
                jbtnIse.addActionListener(new ActionListener()
                {
                        public void actionPerformed(ActionEvent ae)
                        {
                                jlab.setText("ISE is pressed.");
                        }
                });

                // Add the buttons to the content pane.
                jfrm.add(jbtnCse);
                jfrm.add(jbtnIse);

                // 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 cseise();
                        }
                });
        }
}
```


An Event
CSE
ISE
Press a button.


VNC
An Event
CSE
ISE
CSE is pressed.


VNC
An Event
CSE
ISE
ISE is pressed.