IAT-3 Solution

1(a). Explain the role of Synchronization with producer and consumer problem.

Producer consumer problem is an important design concept involving multi-threading and inter thread communication. The concept is that, the producer has to produce an element and notify the consumer to consume it. Before producing the next element the producer has to wait until the previously produced element is consumed by the consumer. Similarly before consuming, the consumer has to wait until an element is being produced. It has to consume the element as soon as it is available and notify the producer to produce the next element. In this post we will see Java program for producer consumer problem.

In computer science the producer-consumer problem (also known as the **bounded-buffer problem)** is a classical example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer.

The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time the consumer is consuming the data (i.e. removing it from the buffer) one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer. This is the code for solving the above stated:

```
    class BufferItem {

    public volatile double value = 0; // multiple threads access public volatile boolean occupied
    = false; // so make these `volatile' }
    class BoundedBuffer { // designed for a single producer thread and // a single consumer
    thread
    private int numSlots = 0;
    private BufferItem[] buffer = null;
    private int putIn = 0, takeOut = 0;
    // private int count = 0;
    public BoundedBuffer(int numSlots) {
    if (numSlots <= 0) throw new IllegalArgumentException("numSlots<=0"); this.numSlots
    = numSlots;
    buffer = new BufferItem[numSlots];
    for (int i = 0; i < numSlots; i++) buffer[i] = new
    BufferItem(); putIn = (putIn + 1) % numSlots;
    // count++; // race
condition!!! } public double
fetch() {

    double value;
    while (!buffer[takeOut].occupied) // busy wait
    Thread.currentThread().yield(); value = buffer[takeOut].value; // C


    buffer [takeOut] .occupied = false;          // D takeOut = (takeOut + 1) % numSlots;
    //          count--; // race condition!!! return value;
    }
    }
```

2(a). Explain how one can bridge two classes using friend function. Write C++ to find the sum of two numbers using bridge friend function add().

Friend function: It is a function by which it is possible to grant a nonmember function access to the private members of a class by using it. A friend function has access to all private and protected members of the class for which it is a friend.

The program to find sum of two numbers using friend function is as follows: #include<iostream>

```
using namespace
std; class A {
private:
  int x,
  y;
public:

 void set_xy ( int i, int j
  ); friend int add ( A
  ob);

add (
); };

void A : : set_xy ( int i, int j)
{
  x = i;
  y =j;
}
int add ( A ob)
{
  return ob.x + ob.y;
}
void main ( )
{
  A ob1;
  ob1.set_xy (10, 20);

  cout << add ( ob1);
}
```

2(b). How do namespaces help in preventing pollution of global name space.

Namespaces enable the C++ programmer to prevent pollution of the global namespace that leads to name clashes.

```
/*Beginning of A1.h*/
namespace A1 //beginning of a namespace A1
{
class A
{
};
} //end of a namespace A1
/*End of A1.h*/
/*Beginning of A2.h*/
namespace A2 //beginning of a namespace A2
{
class A
{
};
} //end of a namespace A2
/*End of A2.h*/
```

Now, the two de□ nitions of the class are enveloped in two different namespaces. The corresponding namespace, followed by the scope resolution operator, must be pre□ xed to the name of the class while referring to it anywhere in the source code. Thus, the ambiguity encountered in the above listing can be overcome

```
/*Beginning of multiDef02.cpp*/
#include"A1.h"
#include"A2.h"
void main()
{
A1::A AObj1; //OK: AObj1 is an object of the class
//defined in A1.h
A2::A AObj2; //OK: AObj2 is an object of the class
//defined in A2.h
}
/*End of multiDef02.cpp*/
```

3(a). Define Event Handling. Explain Delegation event model with suitable example.

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism have the code which is known as event handler that is executed when an event occurs.

Delegation Event Model

The advanced versions of Java ruled out the limitations of Java 1.0 event model. This model is referred to as the **Delegation Event Model** which defines a logical approach to handle events. It is based on the concept of source and listener. A **source** generates an event and sends it to one or more listeners. On receiving the event, **listener** processes the event and returns it. The notable feature of this model is that the source has a registered list of listeners which will receive the events as they occur. Only the listeners that have been registered actually receive the notification when a specific event is generated.

For example, as shown in Figure, when the mouse is clicked on *Button*, an event is generated. If the *Button* has a registered listener to handle the event, this event is sent to *Button*, processed and the output is returned to the user. However, if it has no registered listener the event will not be propagated upwards to *Panel* or *Frame*.

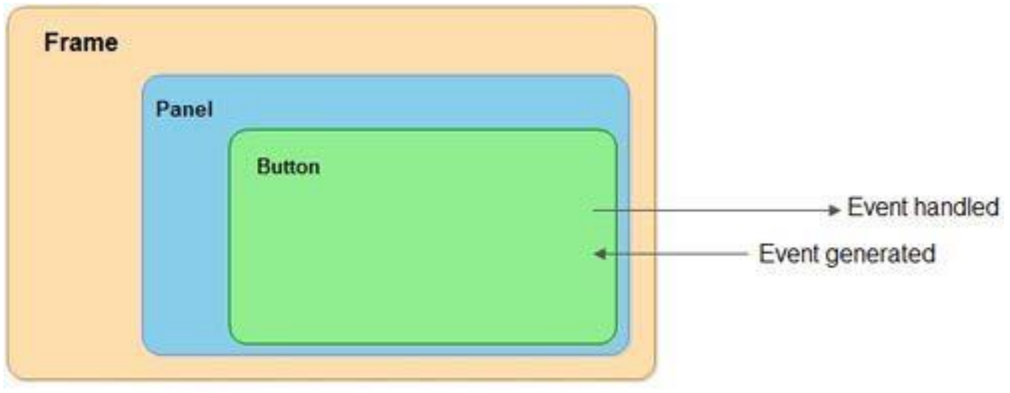


Figure Java 1.1 Event Handling Mechanism

Now we discuss Event Source and Event Listener in detail.

• **Event source:** An event source is an object that generates a particular kind of event. An event is generated when the internal state of the event source is changed. A source may generate more than one type of event. Every source must register a list of listeners that are interested to receive the notifications regarding the type of event. Event source provides methods to add or remove listeners.

The general form of method to register (add) a listener is:

public void addTypeListener(TypeListener eventlistener)

Similarly, the general form of method to unregister (remove) a listener is:

public void removeTypeListener(TypeListener eventlistener)

where, **Type** is the name of the event

eventlistener is a reference to the event listener

• **Event listener:** An event listener is an object which receives notification when an event occurs. As already mentioned, only registered listeners can receive notifications from sources about specific types of events. The role of event listener is to receive these notifications and process them.

4(a). Define an Applet. Explain the skeleton of the applet.

An applet is a small Internet-based program written in Java, a programming language for the Web, which can be downloaded by any computer. The applet is also able to run in HTML. The applet is usually embedded in an HTML page on a Web site and can be executed from within a browser.

Life Cycle of an Applet *//indirect question*

Four methods in the Applet class gives you the framework on which you build any serious applet −

- **init** − This method is intended for whatever initialization is needed for your applet. It is called after the param tags inside the applet tag have been processed.
- **start** − This method is automatically called after the browser calls the init method. It is also called whenever the user returns to the page containing the applet after having gone off to other pages.
- **stop** − This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet.
- **destroy** − This method is only called when the browser shuts down normally. Because applets are meant to live on an HTML page, you should not normally leave resources behind after a user leaves the page that contains the applet.
- **paint** − Invoked immediately after the start() method, and also any time the applet needs to repaint itself in the browser. The paint() method is actually inherited from the java.awt.

These five methods can be assembled into the skeleton shown here:

```
// An Applet skeleton.
import java.awt.*;
import java.applet.*;
/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/
public class AppletSkel extends Applet {
// Called first.
public void init() {
// initialization
}
/* Called second, after init(). Also called whenever
the applet is restarted. */
public void start() {
// start or resume execution
}
// Called when the applet is stopped.
public void stop() {
// suspends execution
}
/* Called when applet is terminated. This is the last
method executed. */
public void destroy() {
// perform shutdown activities
}
// Called when an applet's window must be restored.
public void paint(Graphics g) {
// redisplay contents of window
```

}
}
Helloworld Applet program. //Extra for this question

```
import java.applet.*;

import java.awt.*;

public class HelloWorldApplet extends Applet {

   public void paint (Graphics g) {

      g.drawString ("Hello World", 25, 50);

   }

}
```

4(b). Differentiate between swings and AWT applets.

Swing	**Applet**
Swing is light weight Component.	Applet is heavy weight Component.
Swing have look and feel according to user view you can change look and feel using UIManager.	Applet Does not provide this facility.
Swing uses for standalone Applications, Swing have main method to execute the program.	Applet need HTML code for Run the Applet.
Swing uses MVC Model view Controller.	Applet not.
Swings have its own Layout like most popular Box Layout.	Applet uses AWT Layouts like flow layout.
Swings have some Thread rules.	Applet doesn't have any rule.
To execute Swing no need any browser By	To execute Applet program we should need

which we can create stand alone application But Here we have to add container and maintain all action control within frame container.	any one browser like Applet viewer, web browser. Because Applet using browser container to run and all action control within browser container.

5(a). Explain how to pass parameters for Fontsize and Fontname in applets.

```
// Use Parameters
import java.awt.*;
import java.applet.*;
/*
<applet code="ParamDemo" width=300 height=80>
<param name=fontName value=Courier>
<param name=fontSize value=14>
</applet>
*/
public class ParamDemo extends Applet{
String fontName;
int fontSize;
// Initialize the string to be displayed.
public void start() {
String param;
fontName = getParameter("fontName");
if(fontName == null)
fontName = "Not Found";
param = getParameter("fontSize");
try {
if(param != null) // if not found
fontSize = Integer.parseInt(param);
else
fontSize = 0;
} catch(NumberFormatException e) {
fontSize = -1;
}
// Display parameters.
public void paint(Graphics g) {
g.drawString("Font name: " + fontName, 0, 10);
g.drawString("Font size: " + fontSize, 0, 26);
}
}
```

As the program shows, you should test the return values from **getParameter( )**. If a parameter isn’t available, **getParameter( )** will return **null**.

5(b). Explain different forms of repaint method.

The **repaint( )** method is defined by the AWT. It causes the AWT run-time system to execute a call to your applet's **update( )** method, which, in its default implementation, calls **paint( )**.
The **repaint( )** method has four forms.
The simplest version of **repaint( )** is shown here:
void repaint( )
This version causes the entire window to be repainted. The following version specifies a region that will be repainted:

void repaint(int *left*, int *top*, int *width*, int *height*)

Here, the coordinates of the upper-left corner of the region are specified by *left* and *top,* and the width and height of the region are passed in *width* and *height.*
void repaint(long *maxDelay*)
void repaint(long *maxDelay*, int *x*, int *y*, int *width*, int *height*)
Here, *maxDelay* specifies the maximum number of milliseconds that can elapse before **update( )** is called.

6(a). Explain getDocumentbase() and getCodebase() in an applet class.
We will create applets that will need to explicitly load media and text.
Java will allow the applet to load data from the directory holding the HTML file that started the applet (the document base) and
The directory from which the applet's class file was loaded(the code base).

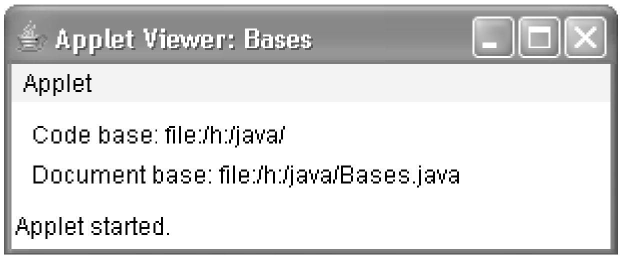
These directories are returned as URL objects by getDocumentBase( ) and getCodeBase( ).

They can be concatenated with a string that names the file you want to load.

```
// Display code and document
bases. import java.awt.*;

import
java.applet.*;
import java.net.*;
/*
<applet code="Bases" width=300
height=50> </applet>
*/
public class Bases extends Applet

{ // Display code and document bases.
    public void paint(Graphics g)
    {
            String msg;
```

```
                        URL url = getCodeBase(); // get code
                        base msg = "Code base: " + url.toString();
                        g.drawString(msg, 10, 20);

                        url = getDocumentBase(); // get document
                        base msg = "Document base: " +
                        url.toString(); g.drawString(msg, 10, 40);

                }
        }
```

Sample output from this program is shown here:



```
Code base: file:/h:/java/
Document base: file:/h:/java/Bases.java
Applet started.
```

7(a). Explain the following Event classes:

i) ActionEvent class ii) KeyEvent class

<u>The ActionEvent Class</u>

An **ActionEvent** is generated when a button is pressed, a list item is double-clicked, or a menu item is selected. The **ActionEvent** class defines four integer constants that can be used to identify any modifiers associated with an action event: **ALT_MASK**, **CTRL_MASK**, **META_MASK**, and **SHIFT_MASK**. In addition, there is an integer constant, **ACTION_ PERFORMED**, which can be used to identify action events.

ActionEvent has these three constructors:

ActionEvent(Object *src*, int *type*, String *cmd*)
ActionEvent(Object *src*, int *type*, String *cmd*, int *modifiers*)
ActionEvent(Object *src*, int *type*, String *cmd*, long *when*, int *modifiers*)

You can obtain the command name for the invoking **ActionEvent** object by using the **getActionCommand( )** method, shown here:

String getActionCommand( )

For example, when a button is pressed The **getModifiers( )** method returns a value that indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. Its form is shown here:

int getModifiers( )

The method **getWhen( )** returns the time at which the event took place. This is called the event's *timestamp*. The **getWhen( )** method is shown here:

long getWhen( )

<u>The KeyEvent Class</u>

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.
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*)
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**.

7(b). Explain JLabel and JButton with the program.
JLabel
JLabel can be used to display text and/or an icon. It is a passive component in that it does not respond to user input. **JLabel** defines several constructors. Here are three of them:
JLabel(Icon *icon*)
JLabel(String *str*)
JLabel(String *str*, Icon *icon*, int *align*)

JButton
The **JButton** class provides the functionality of a push button. You have already seen a simple form of it in the preceding chapter. **JButton** allows an icon, a string, or both to be associated with the push button. Three of its constructors are shown here:

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

```
// Demonstrate an JLable and JButton.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JButtonDemo" width=250 height=450>
</applet>
*/
public class JButtonDemo extends JApplet
implements ActionListener {
JLabel jlab;
public void init() {
```

```
try {
SwingUtilities.invokeAndWait(
new Runnable() {
public void run() {
makeGUI();
}
});
} catch (Exception exc) {
System.out.println("Can't create because of " + exc);
}
}
private void makeGUI() {
// Change to flow layout.
setLayout(new FlowLayout());
// Add buttons to content pane.
ImageIcon france = new ImageIcon("India.gif");
JButton jb = new JButton(france);
jb.setActionCommand("India");
jb.addActionListener(this);
add(jb);
// Create and add the label to content pane.
jlab = new JLabel("Indian Flag");
add(jlab);
}
// Handle button events.
public void actionPerformed(ActionEvent ae) {
jlab.setText("You selected " + ae.getActionCommand());
}
}
```