UNIT - I

SWING

Introducing Swing [2 Marks]

- Swing is used *to create window-based applications*.
- It is built on the top of AWT (Abstract Windowing Toolkit) API.
- The AWT defines a basic set of controls, windows, and dialog boxes that support a usable, but limited graphical interface.
- One reason for the limited nature of the AWT is that it translates its various visual components into their corresponding, platform-specific equivalents.
- This means that the look and feel of a component is defined by the platform, not by Java.
- Because the AWT components use native code resources, they are referred to as *heavyweight*.
- To overcome on these limitations and restrictions they introduced Swing in 1997.
- **Java Swing** is a part of Java Foundation Classes (JFC)

What is JFC?
The Java Foundation Classes (JFC) is a set of GUI components which simplify the development of desktop applications.

Difference between AWT and Swing [5 Marks]

There are many differences between java AWT and swing that are given below.

No.	Java AWT	Java Swing
1)	AWT components are **platform-dependent**.	Java swing components are **platform-independent**.
2)	AWT components are **heavyweight**.	Swing components are **lightweight**.
3)	AWT **doesn't support pluggable look and feel**.	Swing **supports pluggable look and feel**.
4)	AWT provides **less components** than Swing.	Swing provides **more powerful components** such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
5)	AWT **doesn't follows MVC**(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing **follows MVC**.

Features of Swing **[5 Marks]**

1 Swing components are lightweight –

- Swing APIs are written in entirely in java.
- Lightweight components are rendered using graphics primitives, they can be transparent, which enables nonrectangular shapes.
- Thus, lightweight components are more efficient and more flexible.
- Because lightweight components do not translate into native peers, the look and feel of each component is determined by Swing, not by the underlying operating system.
- This means that each component will work in a consistent manner across all platforms.

2 Swings Supports a Pluggable Look and Feel

- Swing supports a *pluggable look and feel* (PLAF).
- Because each Swing component is rendered by Java code rather than by native peers.
- The look and feel of a component is under the control of Swing. This fact means that it is possible to separate the look and feel of a component from the logic of the component, and this is what Swing does.
- Separating out the look and feel provides a significant advantage: it becomes possible to change the way that a component is rendered without affecting any of its other aspects.
- In other words, it is possible to "plug in" a new look and feel for any given component without creating any side effects in the code that uses that component.
- Moreover, it becomes possible to define entire sets of look-and-feels that represent different GUI styles.
- To use a specific style, its look and feel is simply "plugged in." Once this is done, all components are automatically rendered using that style.
- Pluggable look-and-feels offer several important advantages.
- It is possible to define a look and feel that is consistent across all platforms.
- Conversely, it is possible to create a look and feel that acts like a specific platform.
- For example, if you know that an application will be running only in a Windows environment, it is possible to specify the Windows look and feel.
- It is also possible to design a custom look and feel.
- Finally, the look and feel can be changed dynamically at run time.
- Java SE 6 provides look-and-feels, such as metal and Motif, that are available to all Swing users.
- The metal look and feel is also called the *Java look and feel*. It is platform-independent and available in all Java execution environments.
- Windows environments also have access to the Windows and Windows Classic look and feel.

The MVC Connection **[2 Marks]**

MVC –the MVC is popularly called as Software design pattern that separates an application into three main logical components.

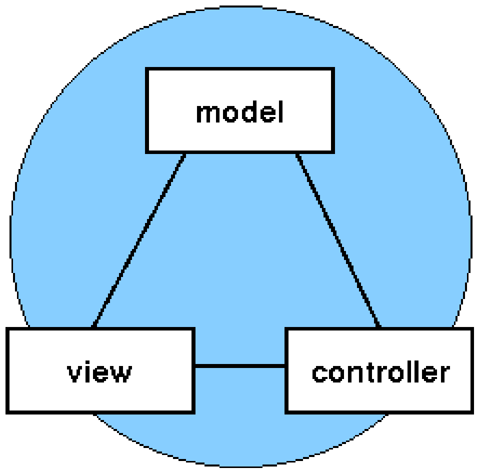
1 The Model
2 The View
3 The Controller

In MVC terminology -

1 **The *model*** corresponds to the state information associated with the component.
   - It responds to the request from the view and it is also responds to instructions from the controller to update itself.
   - For example, in the case of a check box, the model contains a field that indicates if the box is checked or unchecked.

2 **The *view*** determines how the component is displayed on the screen, including any aspects of the view that are affected by the current state of the model.

3 **The *controller*** determines how the component reacts to the user.
   - For example, when the user clicks a check box, the controller reacts by changing the model to reflect the user's choice (checked or unchecked). This then results in the view being updated. By separating a component into a model, a view, and a controller, the specific implementation of each can be changed without affecting the other two.



Components and Containers **[5 Marks]**

Components

- Swing Framework contains a large set of components which provide rich functionalities and allow high level customization.
- All these components are lightweight components.
- They all derived from the JComponent class. It supports the pluggable look and feel.

All of Swing's components are represented by classes defined within the package javax.swing. The following table shows the class names for Swing components.

JApplet	JButton	JCheckBox	JCheckBoxMenuItem
JColorChooser	JComboBox	JComponent	JDesktopPane
JDialog	JEditorPane	JFileChooser	JFormattedTextField
JFrame	JInternalFrame	JLabel	JLayeredPane
JList	JMenu	JMenuBar	JMenuItem
JOptionPane	JPanel	JPasswordField	JPopupMenu
JProgressBar	JRadioButton	JRadioButtonMenuItem	JRootPane
JScrollBar	JScrollPane	JSeparator	JSlider
JSpinner	JSplitPane	JTabbedPane	JTable
JTextArea	JTextField	JTextPane	JTogglebutton
JToolBar	JToolTip	JTree	JViewport
JWindow			

Containers

Swing defines two types of containers.

The first are top-level containers: **JFrame**, **JApplet**, **JWindow**, and **JDialog**. These containers do not inherit **JComponent**. They do, however,inherit the AWT classes **Component** and **Container**. As the name implies, a top-level container must be at the top of a containment hierarchy. A top-level container is not contained within any other container. Furthermore, every containment hierarchy must begin with a top-level container. The one most commonly usedfor applications is **JFrame**. The one used for applets is **JApplet**.

The second type of containers supported by Swing is lightweight containers. Lightweight containers *do* inherit **JComponent**. An example of a lightweight container is **JPanel**, which is a general-purpose container. Lightweight containers are often used to organize and manage groups of related components because a lightweight container can be contained within another container. Thus, you can use lightweight containers such as **JPanel** to create subgroups of related controls that are contained within an outer container.

The Swing Packages **[2 Marks]**

Swing is a very large subsystem and makes use of many packages. These are the packages Used by Swing that are defined by Java SE 6.

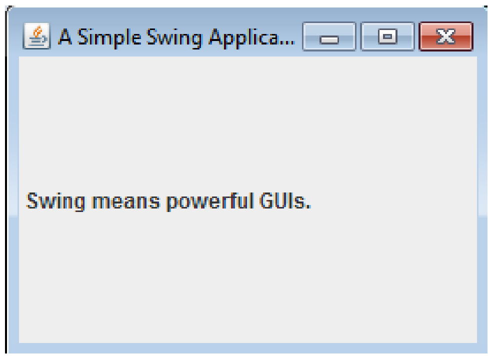
javax.swing	javax.swing.border	javax.swing.colorchooser
javax.swing.event	javax.swing.filechooser	javax.swing.plaf
javax.swing.plaf.basic	javax.swing.plaf.metal	javax.swing.plaf.multi
javax.swing.plaf.synth	javax.swing.table	javax.swing.text
javax.swing.text.html	javax.swing.text.html.parser	javax.swing.text.rtf
javax.swing.tree	javax.swing.undo	

The main package is **javax.swing**. This package must be imported into any program that uses Swing. It contains the classes that implement the basic Swing components, such as push buttons, labels, and check boxes.

A Simple Swing Application **[5 Marks]**

```
import javax.swing.*;

class SwingDemo
{
        public static void main(String args[])
        {
                JFrame jfrm = new JFrame("A Simple Swing Application");
                jfrm.setSize(275, 200);
                JLabel jlab = new JLabel(" Swing means powerful GUIs.");
                jfrm.add(jlab);
               jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                jfrm.setVisible(true);
        }
}
```

OUTPUT



```
Swing means powerful GUIs.
```

- The above swing program 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.

- 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
```

- The program begins by importing **javax.swing**. As mentioned, this package contains the components and models defined by Swing. For example, **javax.swing** defines classes that implement labels, buttons, text controls, and menus. It will be included in all programs that use Swing.

- Next, the program declares the **SwingDemo** class.

- It begins by creating a **JFrame**, using this line of code:

- JFrame jfrm = new JFrame("A Simple Swing Application");

  This creates a container called **jfrm** that defines a rectangular window complete with a title bar; close, minimize, maximize, and restore buttons; and a system menu. Thus, it creates a standard, top-level window. The title of the window is passed to the constructor.

- Next, the window is sized using this statement:

  jfrm.setSize(275, 100);

  The **setSize( )** method (which is inherited by **JFrame** from the AWT class **Component**) sets the dimensions of the window, which are specified in pixels. Its general form is shown here:

  void setSize(int *width*, int *height*)

  In this example, the width of the window is set to 275 and the height is set to 100.

- The next line of code creates a Swing **JLabel** component:
  JLabel jlab = new JLabel(" Swing means powerful GUIs.");

  JLabel is the simplest and easiest-to-use component because it does not accept user input. It simply displays information.

  The next line of code adds the label to the content pane of the frame:
  jfrm.add(jlab);

As explained earlier, all top-level containers have a content pane in which components are stored. Thus, to add a component to a frame, you must add it to the frame's content pane. This is accomplished by calling **add( )** on the **JFrame** reference (**jfrm** in this case). The general form of **add( )** is shown here:
Component add(Component *comp*)

- ❖ By default, when a top-level window is closed (such as when the user clicks the close box), the window is removed from the screen, but the application is not terminated. While this default behavior is useful in some situations, it is not what is needed for most applications. Instead, you will usually want the entire application to terminate when its top-level window is closed. There are a couple of ways to achieve this. The easiest way is to call

  setDefaultCloseOperation( ), as the program does:

  jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

  After this call executes, closing the window causes the entire application to terminate. The general form of **setDefaultCloseOperation( )**

- ❖ The last statement in the **SwingDemo** constructor causes the window to become visible:
  jfrm.setVisible(true);

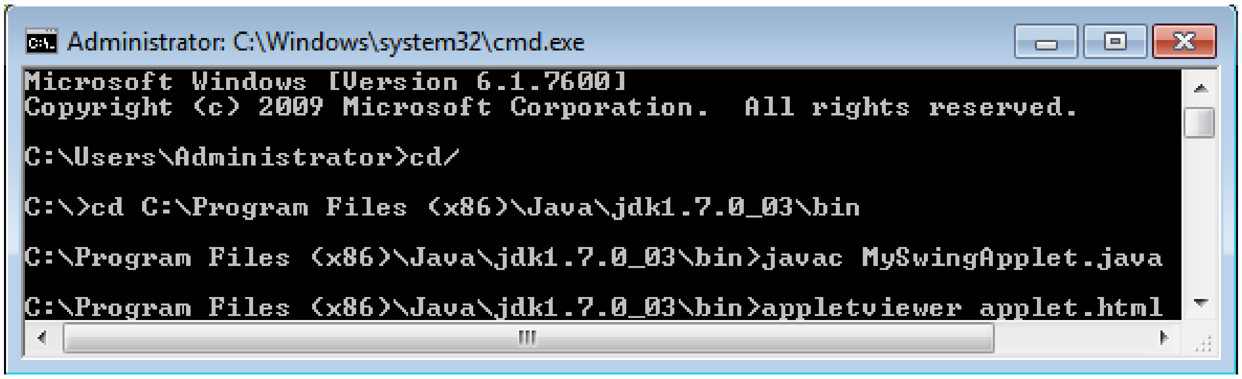
  The **setVisible( )** method is inherited from the AWT **Component** class. If its argument is **true**, the window will be displayed. Otherwise, it will be hidden. By default, a **JFrame** is invisible, so **setVisible(true)** must be called to show it.

Create a Swing Applet

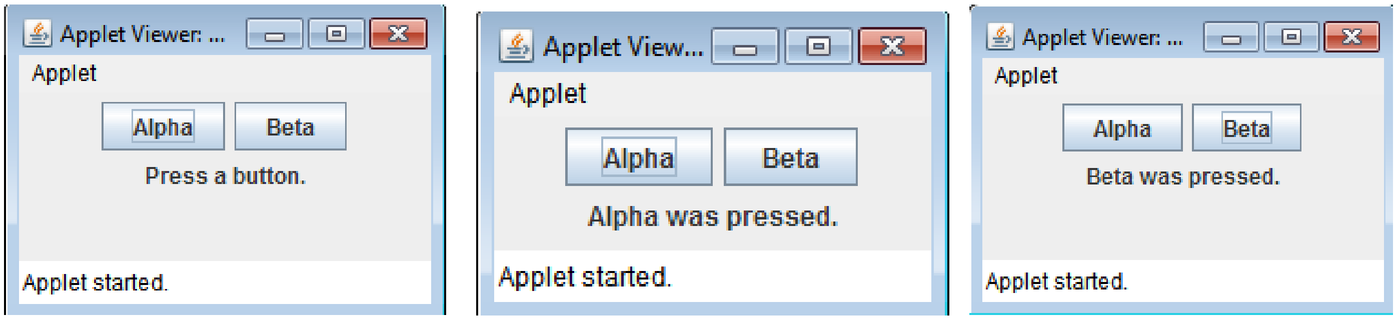
- The second type of program that commonly uses Swing is the applet. Swing-based applets are similar to AWT-based applets, but with an important difference:
- A Swing applet extends **JApplet** rather than **Applet**. **JApplet** is derived from **Applet**. Thus, **JApplet** includes all of the functionality found in **Applet** and adds support for Swing.
- **JApplet** is a top-level Swing container, which means that it is *not* derived from **JComponent**
- Swing applets use the same four lifecycle methods **init( )**, **start( )**, **stop( )**, and **destroy( )**.

```
// A simple Swing-based applet
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class MySwingApplet extends JApplet
{
        JButton jbtnAlpha;
        JButton jbtnBeta;
        JLabel jlab;
        // Initialize the applet.
        public void init()
        {
                // Set the applet to use flow layout.
                setLayout(new FlowLayout());
                // Make two buttons.
                jbtnAlpha = new JButton("Alpha");
                jbtnBeta = new JButton("Beta");
                // Add action listener for Alpha.
                jbtnAlpha.addActionListener(new ActionListener()
                {
                        public void actionPerformed(ActionEvent le)
                        {
                                jlab.setText("Alpha was pressed.");
                        }
                });
                jbtnBeta.addActionListener(new ActionListener()
                {
                        public void actionPerformed(ActionEvent le)
                        {
                                jlab.setText("Beta was pressed.");
                        }
                });
                // Add the buttons to the content pane.
                add(jbtnAlpha);
                add(jbtnBeta);
                // Create a text-based label.
                jlab = new JLabel("Press a button.");
                // Add the label to the content pane.
                add(jlab);
        }
}
```

```
<html>
<body>
<applet code="MySwingApplet.class" width="220" height="90">
</applet>
</body>
</html>
```



```
Administrator: C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Administrator>cd/

C:\>cd C:\Program Files (x86)\Java\jdk1.7.0_03\bin

C:\Program Files (x86)\Java\jdk1.7.0_03\bin>javac MySwingApplet.java

C:\Program Files (x86)\Java\jdk1.7.0_03\bin>appletviewer applet.html
```



JLabel and ImageIcon **[5 Marks]**

JLabel is Swing's easiest-to-use component. **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. **[2 Marks]**

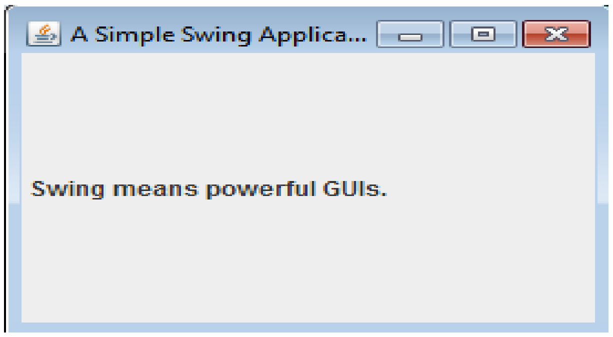
- JLabel(Icon *icon*)
- JLabel(String *str*)
- JLabel(String *tr*, Icon *icon*, int *align*)

Here, *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**, or **TRAILING**.

Example

```
import javax.swing.*;

class SwingDemo
{
          public static void main(String args[])
          {
                    JFrame jfrm = new JFrame("A Simple Swing Application");
                    jfrm.setSize(275, 200);
                    JLabel jlab = new JLabel(" Swing means powerful GUIs.");
                    jfrm.add(jlab);
                  jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    jfrm.setVisible(true);
          }
}
```

OUTPUT



```
Swing means powerful GUIs.
```

JTextField **[5 Marks]**

JTextField is the simplest Swing text component. **JTextField** allows you to edit one line of text. It is derived from **JTextComponent**, which provides the basic functionality common to Swing text components. **JTextField** uses the **Document** interface for its model.

Three of JTextField's constructors are shown here: **[2 Marks]**

- JTextField(int *cols*)
- JTextField(String *str*, int *cols*)
- JTextField(String *str*)

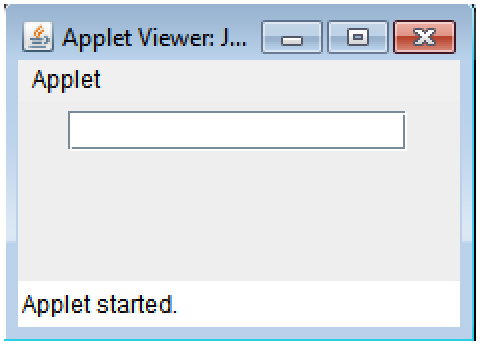
str is the string to be initially presented, and *cols* is the number of columns in the text field.
If no string is specified, the text field is initially empty.
If the number of columns is not specified, the text field is sized to fit the specified string.

JTextField generates events in response to user interaction. For example, an **ActionEvent** is fired when the user presses ENTER. A **CaretEvent** is fired each time the caret (i.e., the cursor) changes position. (**CaretEvent** is packaged in **javax.swing.event**.) Other events are

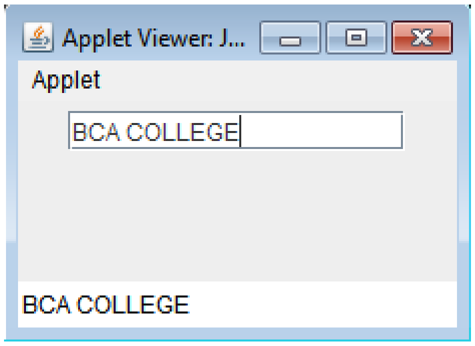
```
// Demonstrate JTextField.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JTextFieldDemo extends JApplet
{
        JTextField jtf;
        public void init()
        {
                // Change to flow layout.
                setLayout(new FlowLayout());
                // Add text field to content pane.
                jtf = new JTextField(15);
                add(jtf);
                jtf.addActionListener(new ActionListener()
                {
                        public void actionPerformed(ActionEvent ae)
                        {
                                // Show text when user presses ENTER.
                                showStatus(jtf.getText());
                        }
                });
        }
}
```

```
<html>
<body>
<applet code="JTextFieldDemo.class" width="220" height="90">
</applet>
</body>
</html>
```

OUTPUT



Applet started.



BCA COLLEGE

BCA COLLEGE

The Swing Buttons [5 or 10 Marks]

Swing defines four types of buttons: **JButton**, **JToggleButton**, **JCheckBox**, and **JRadioButton**. All are subclasses of the **AbstractButton** class, which extends **JComponent**. Thus, all buttons share a set of common traits. **AbstractButton** contains many methods that allow you to control the behavior of buttons.

JButton [5 Marks]

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: [2 Marks]

- 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. However, you can set the action command by calling **setActionCommand( )** on the button. You can obtain the action command by calling **getActionCommand( )** on the event object. It is declared like this:

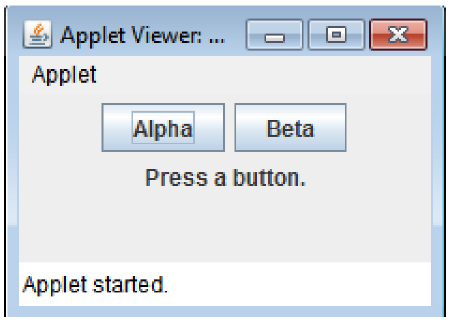
String getActionCommand( )

The action command identifies the button. Thus, when using two or more buttons within the same application, the action command gives you an easy way to determine which button was pressed.

```
// A simple JButton example
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class MySwingApplet extends JApplet
{
        JButton jbtnAlpha;
        JButton jbtnBeta;
        JLabel jlab;
        // Initialize the applet.
        public void init()
        {
                // Set the applet to use flow layout.
                setLayout(new FlowLayout());
                // Make two buttons.
```

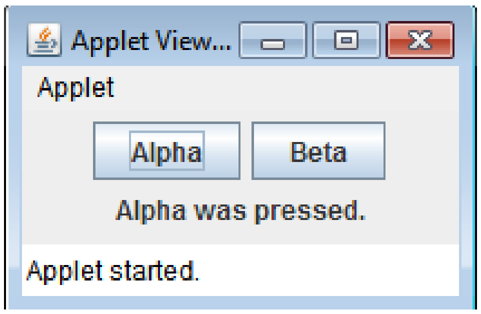
```
                        jbtnAlpha = new JButton("Alpha");
                        jbtnBeta = new JButton("Beta");
                        // Add action listener for Alpha.
                        jbtnAlpha.addActionListener(new ActionListener()
                        {
                                    public void actionPerformed(ActionEvent le)
                                    {
                                                jlab.setText("Alpha was pressed.");
                                    }
                        });
                        jbtnBeta.addActionListener(new ActionListener()
                        {
                                    public void actionPerformed(ActionEvent le)
                                    {
                                                jlab.setText("Beta was pressed.");
                                    }
                        });
                        // Add the buttons to the content pane.
                        add(jbtnAlpha);
                        add(jbtnBeta);
                        // Create a text-based label.
                        jlab = new JLabel("Press a button.");
                        // Add the label to the content pane.
                        add(jlab);
            }
}
```

```
<html>
<body>
<applet code="MySwingApplet.class" width="220" height="90">
</applet>
</body>
</html>
```



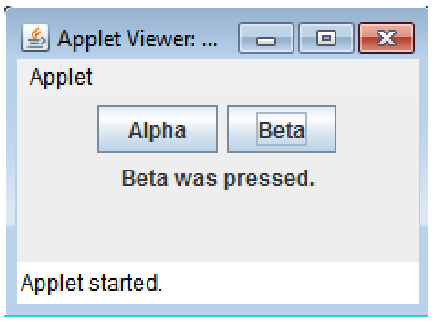
Press a button.

Applet started.



Alpha was pressed.

Applet started.



Beta was pressed.

Applet started.

JToggleButton [5 Marks]

- A useful variation on the push button is called a *toggle button*.
- A toggle button looks just like a push button, but it acts differently because it has two states: pushed and released.
- That is, when you press a toggle button, it stays pressed rather than popping back up as a regular push button does.
- When you press the toggle button a second time, it releases (pops up). Therefore, each time a toggle button is pushed, it toggles between its two states.
- Toggle buttons are objects of the **JToggleButton** class. **JToggleButton** implements **AbstractButton**. **JToggleButton** defines several constructors.

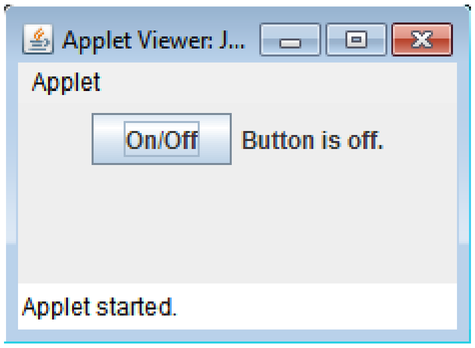
  The one used by the example in this section is shown here:

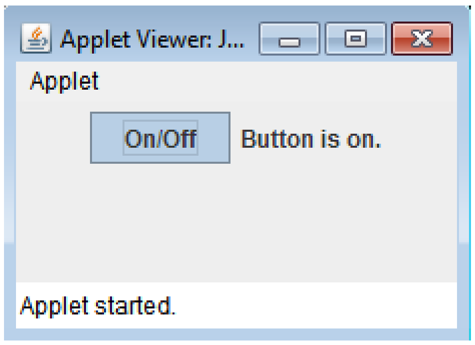
JToggleButton(String *str*)

This creates a toggle button that contains the text passed in *str*. By default, the button is in the off position. Other constructors enable you to create toggle buttons that contain images, or images and text.

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

public class JToggleButtonDemo extends JApplet
{
        JLabel jlab;
        JToggleButton jtbn;
        public void init()
        {
                // Change to flow layout.
                setLayout(new FlowLayout());
                // Create a label.
                jlab = new JLabel("Button is off.");
                // Make a toggle button.
                jtbn = new JToggleButton("On/Off");
                // Add an item listener for the toggle button.
                jtbn.addItemListener(new ItemListener()
                {
                        public void itemStateChanged(ItemEvent ie)
                        {
                                if(jtbn.isSelected())
                                        jlab.setText("Button is on.");
                                else
                                        jlab.setText("Button is off.");
                        }
                });
                // Add the toggle button and label to the content pane.
                add(jtbn);
```

```
                    add(jlab);
          }
}
<html>
<body>
<applet code="JToggleButtonDemo.class" width="220" height="90">
</applet>
</body>
</html>
```



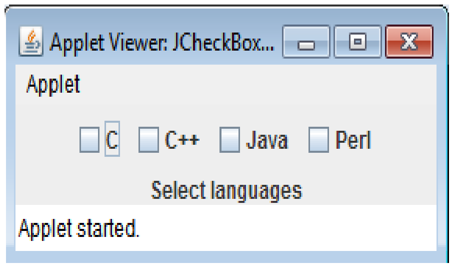


Check Boxes **[5 Marks]**

- 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.
- The one used here 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.
- You can obtain a reference to the **JCheckBox** that generated the event by calling **getItem( )** on the **ItemEvent** passed to the **itemStateChanged( )** method defined by **ItemListener**.
- The easiest way to determine the selected state of a check box is to call **isSelected( )** on the **JCheckBox** instance.
- The following example illustrates check boxes. It displays four check boxes and a label.
- When the user clicks a check box, an **ItemEvent** is generated. Inside the **itemStateChanged( )** method, **getItem( )** is called to obtain a reference to the **JCheckBox** object that generated the event.
- Next, a call to **isSelected( )** determines if the box was selected or cleared.
- The **getText( )** method sets the text for that check box and uses it to set the text inside the label.

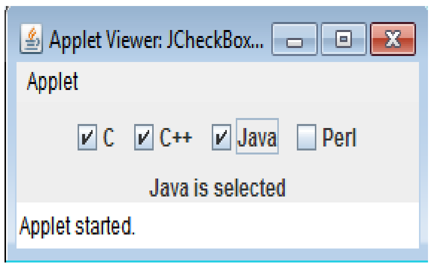
```
// Demonstrate JCheckbox.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JCheckBoxDemo extends JApplet implements ItemListener
{
        JLabel jlab;
        public void init()
        {

                // Change to flow layout.
                setLayout(new FlowLayout());
                // Add check boxes to the content pane.
                JCheckBox cb = new JCheckBox("C");
                cb.addItemListener(this);
                add(cb);
                cb = new JCheckBox("C++");
                cb.addItemListener(this);
                add(cb);
                cb = new JCheckBox("Java");
                cb.addItemListener(this);
                add(cb);
                cb = new JCheckBox("Perl");
                cb.addItemListener(this);
                add(cb);
                // Create the label and add it to the content pane.
                jlab = new JLabel("Select languages");
                add(jlab);
        }
        // Handle item events for the check boxes.
        public void itemStateChanged(ItemEvent ie)
        {
                JCheckBox cb = (JCheckBox)ie.getItem();
                if(cb.isSelected())
                {
                        jlab.setText(cb.getText() + " is selected");
                }
                else
                {
                        jlab.setText(cb.getText() + " is cleared");
                }
        }
}
```

```
<html>
<body>
<applet code=" JCheckBoxDemo.class" width="220" height="90">
</applet>
</body>
</html>
```



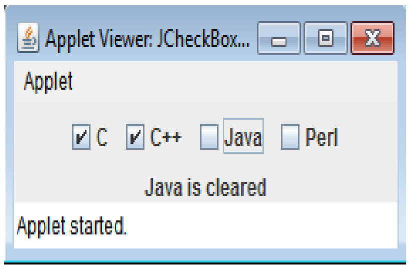
Select languages

Applet started.



Java is selected

Applet started.



Java is cleared

Applet started.

Radio Buttons **[5 Marks]**

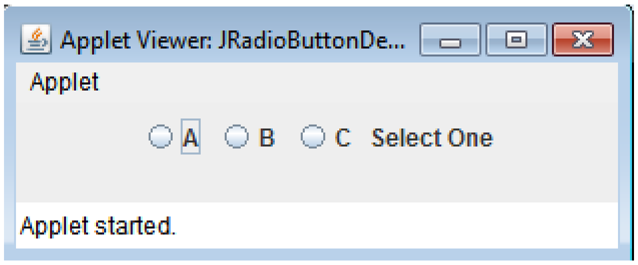
- Radio buttons are a group of mutually exclusive buttons, in which only one button can be selected at any one time.
- They are supported by the **JRadioButton** class, which extends **JToggleButton**.
- **JRadioButton** provides several constructors.

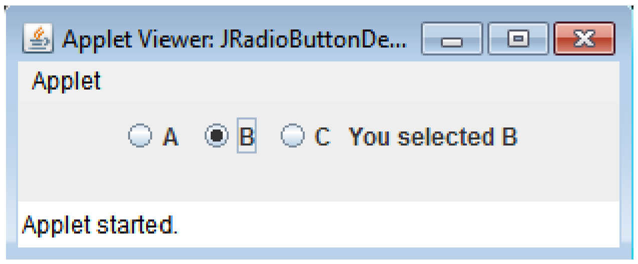
  JRadioButton(String *str*)
- Here, *str* is the label for the button.
- In order for their mutually exclusive nature to be activated, radio buttons must be configured into a group.
- Only one of the buttons in the group can be selected at any time.
- For example, if a user presses a radio button that is in a group, any previously selected button in that group is automatically deselected.
- A button group is created by the **ButtonGroup** class.
- 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 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 caliing **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.
- The following example illustrates how to use radio buttons. Three radio buttons are created. The buttons are then added to a button group.
- As explained, this is necessary to cause their mutually exclusive behavior.
- Pressing a radio button generates an action event, which is handled by **actionPerformed().**

- Within that handler, the **getActionCommand( )** method gets the text that is associated with the radio button and uses it to set the text within a label.

```
// Demonstrate JRadioButton
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JRadioButtonDemo extends JApplet implements ActionListener
{
        JLabel jlab;
        public void init()
        {

                // Change to flow layout.
                setLayout(new FlowLayout());
                // Create radio buttons and add them to content pane.
                JRadioButton b1 = new JRadioButton("A");
                b1.addActionListener(this);
                add(b1);
                JRadioButton b2 = new JRadioButton("B");
                b2.addActionListener(this);
                add(b2);
                JRadioButton b3 = new JRadioButton("C");
                b3.addActionListener(this);
                add(b3);
                // Define a button group.
                ButtonGroup bg = new ButtonGroup();
                bg.add(b1);
                bg.add(b2);
                bg.add(b3);
                // Create a label and add it to the content pane.
                jlab = new JLabel("Select One");
                add(jlab);
        }
        // Handle button selection.
        public void actionPerformed(ActionEvent ae)
        {
                jlab.setText("You selected " + ae.getActionCommand());
        }
}
```

```
<html>
<body>
<applet code="JRadioButtonDemo" width="220" height="90">
</applet>
</body> </html>
```





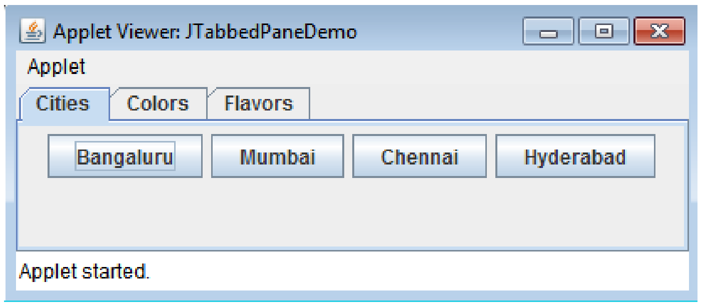
JTabbedPane **[5 or 10 Marks]**

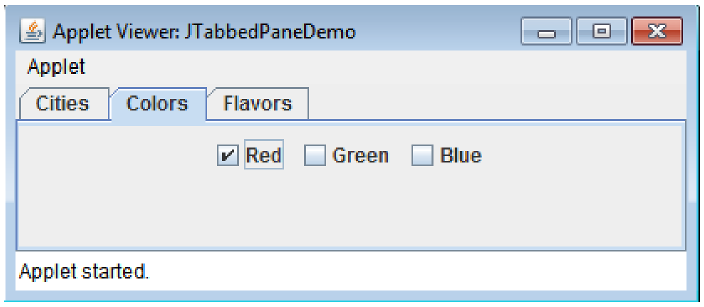
- **JTabbedPane** encapsulates a *tabbed pane*. It manages a set of components by linking them with tabs.
- Selecting a tab causes the component associated with that tab to come to the forefront.
- Tabbed panes are very common in the modern GUI, and you have no doubt used them many times.
- **JTabbedPane** uses the **SingleSelectionModel** model.
- Tabs are added by calling **addTab( )**. Here is one of its forms:
- void addTab(String *name*, Component *comp*)
- Here, *name* is the name for the tab, and *comp* is the component that should be added to the tab.
- Often, the component added to a tab is a **JPanel** that contains a group of related components. This technique allows a tab to hold a set of components. The general procedure to use a tabbed pane is outlined here:
    1. Create an instance of **JTabbedPane**.
    2. Add each tab by calling **addTab( )**.
    3. Add the tabbed pane to the content pane.

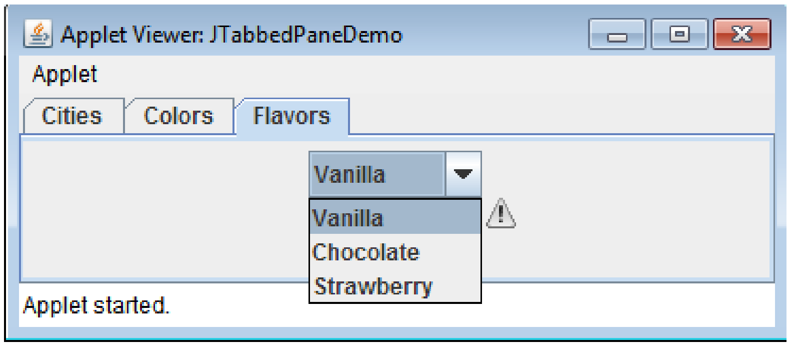
- The following example illustrates a tabbed pane. The first tab is titled "Cities" and contains four buttons. Each button displays the name of a city.
- The second tab is titled "Colors" and contains three check boxes. Each check box displays the name of a color.
- The third tab is titled "Flavors" and contains one combo box. This enables the user to select oneof three flavors.

```
<html>
<body>
<applet code="JTabbedPaneDemo" width="220" height="90">
</applet>
</body> </html>
```

```
import javax.swing.*;
public class JTabbedPaneDemo extends JApplet
{
        public void init()
        {
                JTabbedPane jtp = new JTabbedPane();
                jtp.addTab("Cities", new CitiesPanel());
                jtp.addTab("Colors", new ColorsPanel());
                jtp.addTab("Flavors", new FlavorsPanel());
                add(jtp);
        }
}
// Make the panels that will be added to the tabbed pane.
class CitiesPanel extends JPanel
{
        public CitiesPanel()
        {
                JButton b1 = new JButton("Bangaluru");
                add(b1);
                JButton b2 = new JButton("Mumbai");
                add(b2);
                JButton b3 = new JButton("Chennai");
                add(b3);
                JButton b4 = new JButton("Hyderabad");
                add(b4);
        }
}
class ColorsPanel extends JPanel
{
        public ColorsPanel()
        {
                        JCheckBox cb1 = new JCheckBox("Red");
                        add(cb1);
                        JCheckBox cb2 = new JCheckBox("Green");
                        add(cb2);
                        JCheckBox cb3 = new JCheckBox("Blue");
                        add(cb3);
        }
}
class FlavorsPanel extends JPanel
{
        public FlavorsPanel()
        {
                JComboBox jcb = new JComboBox();
                jcb.addItem("Vanilla");
                jcb.addItem("Chocolate");
                jcb.addItem("Strawberry");
                add(jcb);
        }
}
```







JScrollPane [5 Marks]

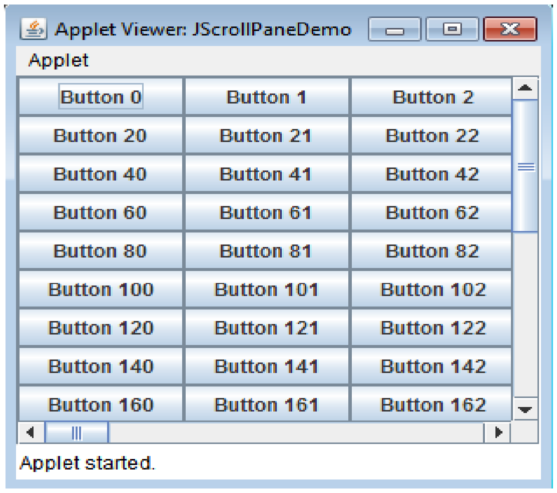
- **JScrollPane** is a lightweight container that automatically handles the scrolling of another component.
- The component being scrolled can either be an individual component, such as a table, or a group of components contained within another lightweight container, such as a **JPanel**.
- In case, if the object being scrolled is larger than the viewable area, horizontal and/or vertical scroll bars are automatically provided, and the component can be scrolled through the pane.
- Because **JScrollPane** automates scrolling, it usually eliminates the need to manage individual scroll bars. The viewable area of a scroll pane is called the *viewport*.
- It is a window in which the component being scrolled is displayed. Thus, the viewport displays the visible portion of the component being scrolled.
- The scroll bars scroll the component through the viewport. In its default behavior, a **JScrollPane** will dynamically add or remove a scroll bar as needed.
- For example, if the component is taller than the viewport, a vertical scroll bar is added. If the component will completely fit within the viewport, the scroll bars are removed. **JScrollPane** defines several constructors.
- The one used in this chapter is shown here:
- JScrollPane(Component *comp*)
- The component to be scrolled is specified by *comp*. Scroll bars are automatically displayed when the content of the pane exceeds the dimensions of the viewport.
- Here are the steps to follow to use a scroll pane:

1. Create the component to be scrolled.
2. Create an instance of **JScrollPane**, passing to it the object to scroll.
3. Add the scroll pane to the content pane.

- The following example illustrates a scroll pane. First, a **JPanel** object is created, and 400 buttons are added to it, arranged into 20 columns.
- This panel is then added to a scroll pane, and the scroll pane is added to the content pane.
- Because the panel is larger than the viewport, vertical and horizontal scroll bars appear automatically. You can use the scroll bars to scroll the buttons into view.

```
// Demonstrate JScrollPane.
import java.awt.*;
import javax.swing.*;

public class JScrollPaneDemo extends JApplet
{
        public void init()
        {

                // Add 400 buttons to a panel.
                JPanel jp = new JPanel();
                jp.setLayout(new GridLayout(20, 20));
                int b = 0;
                for(int i = 0; i < 20; i++)
                {
                        for(int j = 0; j < 20; j++)
                        {
                                jp.add(new JButton("Button " + b));
                                ++b;
                        }
                }
                // Create the scroll pane.
                JScrollPane jsp = new JScrollPane(jp);
                // Add the scroll pane to the content pane.
                // Because the default border layout is used,
                // the scroll pane will be added to the center.
                add(jsp, BorderLayout.CENTER);
        }
}
```

```
<html>
<body>
<applet code="JScrollPaneDemo" width="220" height="90">
</applet>
</body>
 </html>
```



JList **[5 Marks]**

- In Swing, the basic list class is called **JList**.
- It supports the selection of one or more items from a list.
- Although the list often consists of strings, it is possible to create a list of just about any object that can be displayed.
- **JList** is so widely used in Java that it is highly unlikely that you have not seen one before.
- **JList** provides several constructors. The one used here is JList(Object[ ] *items*)
- This creates a **JList** that contains the items in the array specified by *items*.
- **JList** is based on two models. The first is **ListModel**. This interface defines how access to the list data is achieved.
- The second model is the **ListSelectionModel** interface, which defines methods that determine what list item or items are selected.
- Although a **JList** will work properly by itself, most of the time you will wrap a **JList** inside a **JScrollPane**
- A **JList** generates a **ListSelectionEvent** when the user makes or changes a selection. This event is also generated when the user deselects an item. It is handled by implementing **ListSelectionListener**. This listener specifies only one method, called **valueChanged( )**, which is shown here:
- void valueChanged(ListSelectionEvent *le*)
- Here, *le* is a reference to the object that generated the event.
- Although **ListSelectionEvent** does provide some methods of its own, normally you will interrogate the **JList** object itself to determine what has occurred. Both **ListSelectionEvent** and **ListSelectionListener** are packaged in **javax.swing.event**. By default, a **JList** allows the user to select multiple ranges of items within the list, but you can change this behavior by calling **setSelectionMode( )**, which is defined by **JList**. It is shown here:

void setSelectionMode(int *mode*)
Here, *mode* specifies the selection mode. It must be one of these values defined by

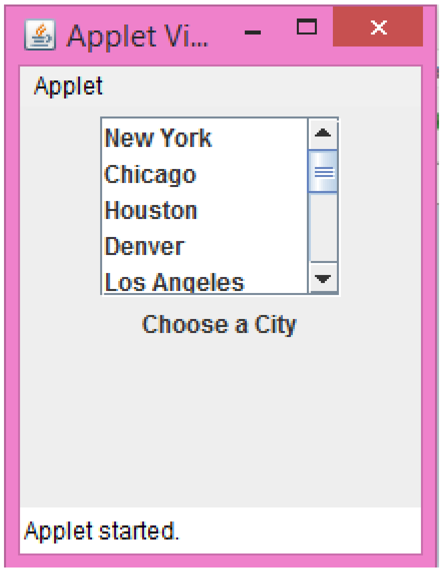
ListSelectionModel:
SINGLE_SELECTION
SINGLE_INTERVAL_SELECTION
MULTIPLE_INTERVAL_SELECTION

The default, multiple-interval selection, lets the user select multiple ranges of items within a list.With single-interval selection, the user can select one range of items.With single selection, the user can select only a single item. Of course, a single item can be selected in the other two modes, too. It’s just that they also allow a range to be selected. You can obtain the index of the first item selected, which will also be the index of the only selected item when using single-selection mode, by calling **getSelectedIndex( )**, shown here:

```
int getSelectedIndex( )
```

Indexing begins at zero. So, if the first item is selected, this method will return 0. If no item is selected, –1 is returned.

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

public class JListDemo extends JApplet
{
        JList jlst;
        JLabel jlab;
        JScrollPane jscrlp;

        String Cities[] = { "New York", "Chicago", "Houston",
        "Denver", "Los Angeles", "Seattle",
        "London", "Paris", "New Delhi",
        "Hong Kong", "Tokyo", "Sydney" };
        public void init()
        {

                setLayout(new FlowLayout());
                jlst = new JList(Cities);
                jlst.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
                jscrlp = new JScrollPane(jlst);
                jscrlp.setPreferredSize(new Dimension(120, 90));
                jlab = new JLabel("Choose a City");
                jlst.addListSelectionListener(new ListSelectionListener()
                {
                        public void valueChanged(ListSelectionEvent le)
                        {
                                int idx = jlst.getSelectedIndex();
                                if(idx != -1)
                                        jlab.setText("Current selection: " + Cities[idx]);
                                else
                                        jlab.setText("Choose a City");
                        }
                });

        add(jscrlp);
        add(jlab);
        }
}
```



JComboBox **[5 Marks]**

Swing provides a *combo box* (a combination of a text field and a drop-down list) through the ComboBox class. A combo box normally displays one entry, but it will also display a drop-down list that allows a user to select a different entry.

You can also create a combo box the example is shown here:

JComboBox(Object[ ] *items*)

Here, *items* is an array that initializes the combo box. Other constructors are available.

JComboBox uses the **ComboBoxModel**. In addition to passing an array of items to be displayed in the drop-down list, items can be dynamically added to the list of choices via the **addItem( )** method, shown here:

void addItem(Object *obj*)

Here, *obj* is the object to be added to the combo box. This method must be used only with mutable combo boxes.

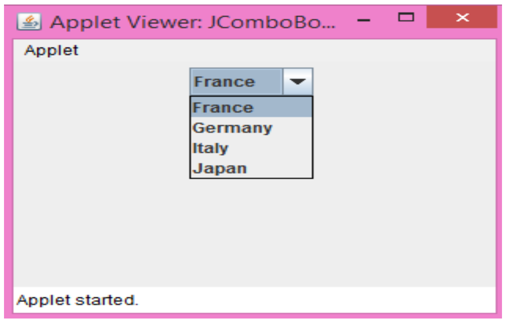
JComboBox generates an action event when the user selects an item from the list. **JComboBox** also generates an item event when the state of selection changes, which occurs when an item is selected or deselected.
One way to obtain the item selected in the list is to call **getSelectedItem( )** on the combo box. It is shown here:

Object getSelectedItem( )
You will need to cast the returned value into the type of object stored in the list.

The following example demonstrates the combo box. The combo box contains entries for “France,” “Germany,” “Italy,” and “Japan.” When a country is selected, an icon-based label is updated to display the flag for that country. You can see how little code is required to use this powerful component.

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

public class JComboBoxDemo extends JApplet
 {
        JLabel jlab;
        JComboBox jcb;
        String flags[] = { "France", "Germany", "Italy", "Japan" };
        public void init()
        {

                setLayout(new FlowLayout());
                // Instantiate a combo box and add it to the content pane.
                jcb = new JComboBox(flags);
                add(jcb);
                // Handle selections.
                jcb.addActionListener(new ActionListener()
                {
                        public void actionPerformed(ActionEvent ae)
                        {
                                String s = (String) jcb.getSelectedItem();

                        }
                });
                // Create a label and add it to the content pane.
                jlab = new JLabel(new ImageIcon("france.gif"));
                add(jlab);
        }
}
```

Output from the combo box example is shown here:



Applet started.

JTable [5 Marks]

JTable is a component that displays rows and columns of data. You can drag the cursor on column boundaries to resize columns. You can also drag a column to a new position. Depending on its configuration, it is also possible to select a row, column, or cell within the table, and to change the data within a cell.

JTable supplies several constructors. The one used here is
JTable(Object *data*[ ][ ], Object *colHeads*[ ])

Here, *data* is a two-dimensional array of the information to be presented, and *colHeads* is a one-dimensional array with the column headings.

JTable relies on three models. The first is the table model, which is defined by the **TableModel** interface. This model defines those things related to displaying data in a two-dimensional format.

The second is the table column model, which is represented by **TableColumnModel**. **JTable** is defined in terms of columns, and it is **TableColumnModel** that specifies the characteristics of a column. These two models are packaged in **javax.swing.table**.

The third model determines how items are selected, and it is specified by the **ListSelectionModel**, which was described when **JList** was discussed.

A**JTable** can generate several different events. The two most fundamental to a table's operation are **ListSelectionEvent** and **TableModelEvent**. A **ListSelectionEvent** is generated when the user selects something in the table.

By default, **JTable** allows you to select one or more complete rows, but you can change this behavior to allow one or more columns, or one or more individual cells to be selected. A **TableModelEvent** is fired when that table's data changes in some way. Handling these events requires a bit more work than it does to handle the events generated by the previously described components and is beyond the scope of this book. However, if you simply want to use **JTable** to display data (as the following example does), then you don't need to handle any events.

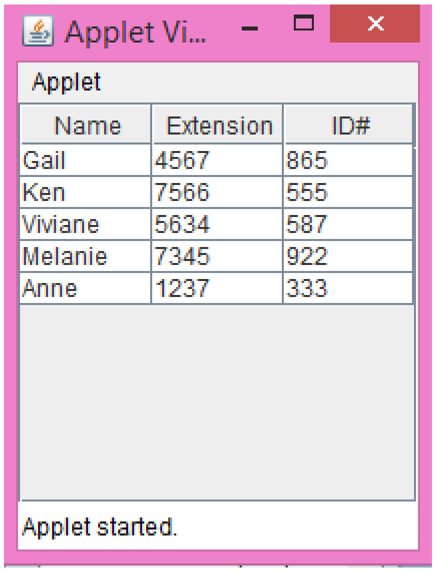
Here are the steps required to set up a simple **JTable** that can be used to display data:

1. Create an instance of **JTable**.
2. Create a **JScrollPane** object, specifying the table as the object to scroll.
3. Add the table to the scroll pane.
4. Add the scroll pane to the content pane.

The following example illustrates how to create and use a simple table. A one-dimensional array of strings called **colHeads** is created for the column headings. A two-dimensional array of strings called **data** is created for the table cells. You can see that each element in the array is an array of three strings. These arrays are passed to the **JTable** constructor. The table is added to a scroll pane, and then the scroll pane is added to the content pane. The table displays the data in the **data** array. The default table configuration also allows the contents of a cell to be edited. Changes affect the underlying array, which is **data** in this case.

```
// Demonstrate JTable.
import java.awt.*;
import javax.swing.*;

public class JTableDemo extends JApplet
{
        public void init()
        {

                // Initialize column headings.
                String[] colHeads = { "Name", "Extension", "ID#" };
                // Initialize data.
                Object[][] data = {
                { "Gail", "4567", "865" },
                { "Ken", "7566", "555" },
                { "Viviane", "5634", "587" },
                { "Melanie", "7345", "922" },
                { "Anne", "1237", "333" },};

                // Create the table.
                JTable table = new JTable(data, colHeads);
                // Add the table to a scroll pane.
                JScrollPane jsp = new JScrollPane(table);
                // Add the scroll pane to the content pane.
                add(jsp);
        }
}
```

Output from this example is shown here:



Name	Extension	ID#
Gail	4567	865
Ken	7566	555
Viviane	5634	587
Melanie	7345	922
Anne	1237	333

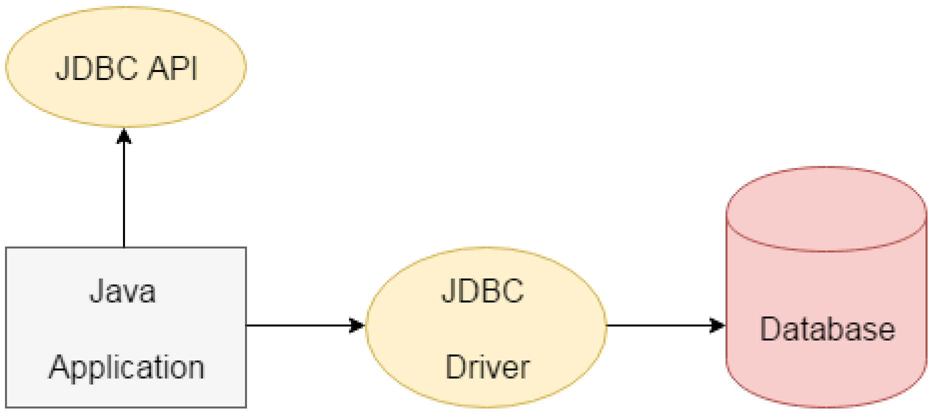
Applet started.

UNIT - II

JAVA 2 ENTERPRISE EDITION OVERVIEW, DATABASE ACCESS

Java JDBC is a java API to connect and execute query with the database. JDBC API uses jdbc drivers to connect with the database.

Java Database Connectivity (JDBC) is an **Application Programming Interface (API)** used to connect Java application with Database. JDBC is used to interact with various type of Database such as Oracle, MS Access, My SQL and SQL Server. JDBC can also be defined as the platform-independent interface between a relational database and Java programming. It allows java program to execute SQL statement and retrieve result from database. **[2 Marks]**



Why use JDBC

Before JDBC, ODBC API was the database API to connect and execute query with the database. But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).

What is API **[2 Marks]**

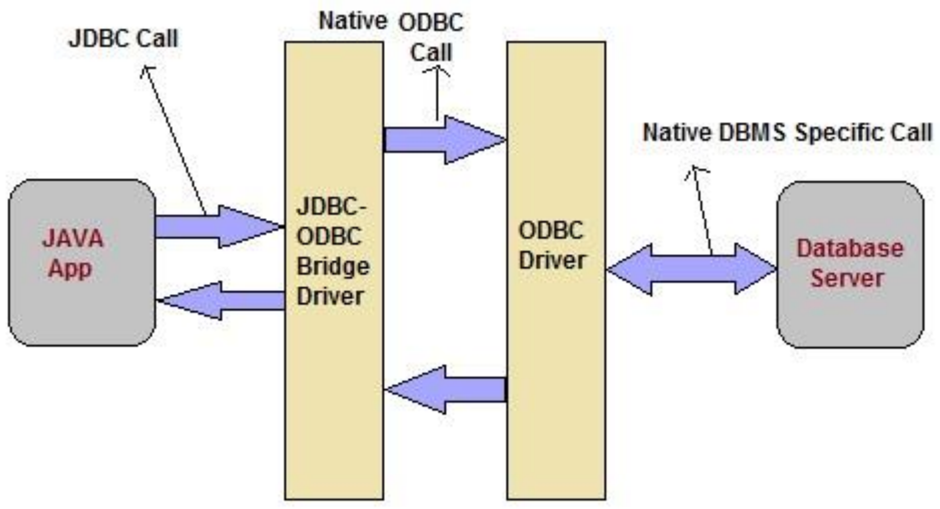
API (Application programming interface) is a document that contains description of all the features of a product or software. It represents classes and interfaces that software programs can follow to communicate with each other. An API can be created for applications, libraries, operating systems, etc

JDBC Driver **[2 or 5 Marks]**

JDBC Driver is a software component that enables java application to interact with the database. There are 4 types of JDBC drivers:

1. Type-1 Driver or JDBC-ODBC bridge
2. Type-2 Driver or Native API Partly Java Driver
3. Type-3 Driver or Network Protocol Driver
4. Type-4 Driver or Thin Driver

Type-1 Driver acts as a bridge between JDBC and other database connectivity mechanism (ODBC). This driver converts JDBC calls into ODBC calls and redirects the request to the ODBC driver.

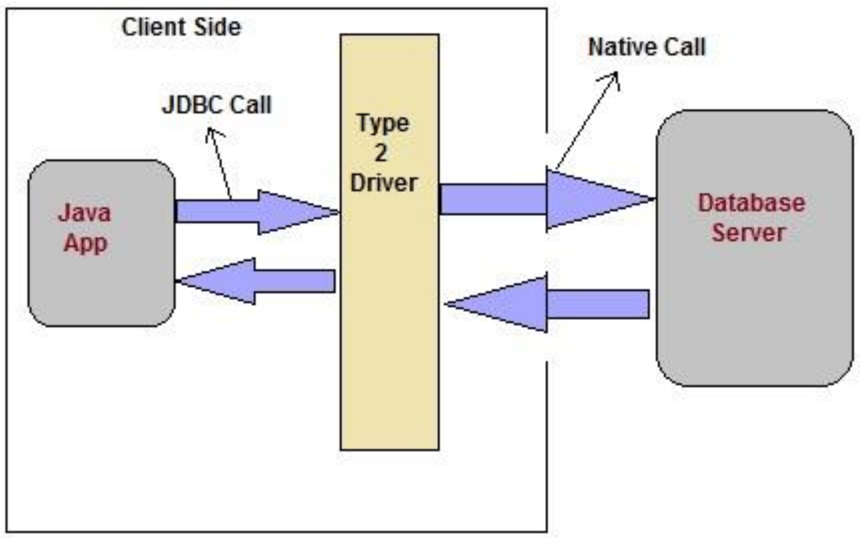


Advantage
- Easy to use
- Allow easy connectivity to all database supported by the ODBC Driver.

Disadvantage
- Slow execution time
- Dependent on ODBC Driver.
- Uses Java Native Interface(JNI) to make ODBC call.

2. Native API Driver

The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java. These native clients API are usually written in C and C++.



Advantage

- faster as compared to **Type-1 Driver**
- Contains additional features.

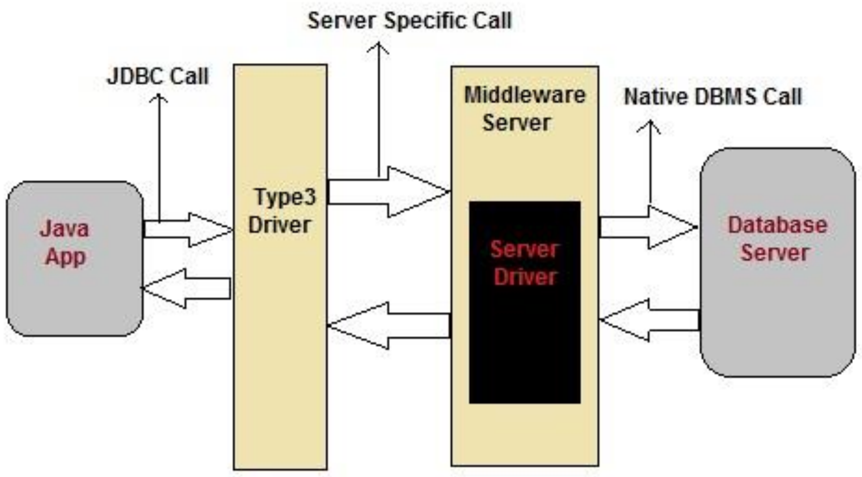
Disadvantage

- Requires native library
- Increased cost of Application

---

3 Network Protocol Driver

This driver translates the JDBC calls into a database server independent and Middleware server-specific calls. Middleware server further translates JDBC calls into database specific calls.



Advantage

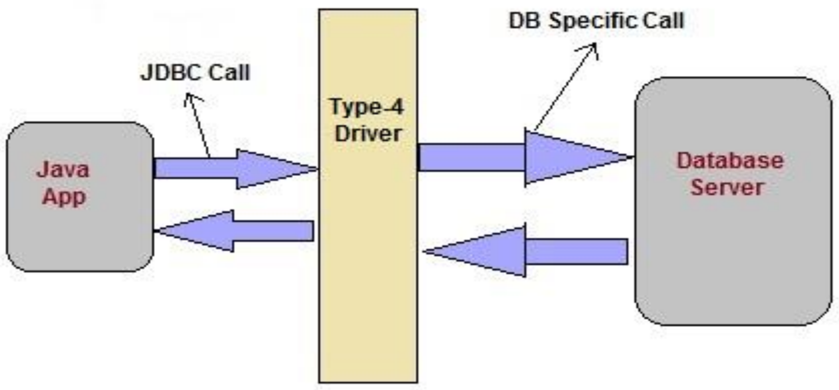
- Does not require any native library to be installed.
- Database Independency.
- Provide facility to switch over from one database to another database.

Disadvantage

- Slow due to increase number of network call.

4 Thin Driver

This is Driver called Pure Java Driver because. This driver interact directly with database. It does not require any native database library that is why it is also known as Thin Driver.



Advantage

- Does not require any native library.
- Does not require any Middleware server.
- Better Performance than other driver.

Disadvantage

- Slow due to increase number of network call.

5 Steps to connect to the database in java **[5 Marks]**

5 Steps to connect to the database in java

1. Register the driver class
2. Create the connection object
3. Create the Statement object
4. Execute the query
5. Close the connection object

1) Register the driver class

The forName() method of Class class is used to register the driver class. This method is used to dynamically load the driver class.

Syntax of forName() method

public static void forName(String className)throws ClassNotFoundException

Example to register the OracleDriver class

Class.forName("oracle.jdbc.driver.OracleDriver");

2) Create the connection object

The getConnection() method of DriverManager class is used to establish connection with the database.

Syntax of getConnection() method

1) public static Connection getConnection(String url)throws SQLException
2) public static Connection getConnection(String url,String name,String password)
throws SQLException

Example to establish connection with the Oracle database

Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","password");

3) Create the Statement object

The createStatement() method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database.

Syntax of createStatement() method

public Statement createStatement()throws SQLException

Example to create the statement object

Statement stmt=con.createStatement();

4) Execute the query

The executeQuery() method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table.

Syntax of executeQuery() method

```
public ResultSet executeQuery(String sql)throws SQLException
```

Example to execute query

```
ResultSet rs=stmt.executeQuery("select * from emp");

while(rs.next())
{
	System.out.println(rs.getInt(1)+" "+rs.getString(2));
}
```

5) Close the connection object

By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

Syntax of close() method

```
public void close()throws SQLException
```

Example to close connection

```
con.close();
```

Example to Connect Java Application with Oracle database [5 Marks]

In this example, system is the username and oracle is the password of the Oracle database.

```
import java.sql.*;
class OracleCon
{
	public static void main(String args[])
	{
		try
		{
			//step1 load the driver class
			Class.forName("oracle.jdbc.driver.OracleDriver");

			//step2 create  the connection object
			Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","root1");

			//step3 create the statement object
			Statement stmt=con.createStatement();

			//step4 execute query
			ResultSet rs=stmt.executeQuery("select * from emp");
			while(rs.next())
			{
				System.out.println(rs.getInt(1)+""+rs.getString(2)+" "+rs.getString(3));
			}

			//step5 close the connection object
			con.close();

		}
		catch(Exception e)
		{
			 System.out.println(e);
		}

	}
}
```

Types of Statement Object **[2 or 5 Marks]**

1. **Statement Object**
2. **Prepared Statement Objetc**
3. **Callable Statement Object**

1 Statement interface **[5 Marks]**

The Statement interface provides methods to execute queries with the database. The statement interface is a factory of ResultSet i.e. it provides factory method to get the object of ResultSet.

Commonly used methods of Statement interface:

The important methods of Statement interface are as follows:

1) public ResultSet executeQuery(String sql): is used to execute SELECT query. It returns the object of ResultSet.
2) public int executeUpdate(String sql): is used to execute specified query, it may be create, drop, insert, update, delete etc.
3) public boolean execute(String sql): is used to execute queries that may return multiple results.
4) public int[] executeBatch(): is used to execute batch of commands.

Example of Statement interface

Let's see the simple example of Statement interface to insert, update and delete the record.

```
import java.sql.*;
class FetchRecord
{
    public static void main(String args[])throws Exception
   {
        Class.forName("oracle.jdbc.driver.OracleDriver");
        Connection
        con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
        Statement stmt=con.createStatement();

        int result=stmt.executeUpdate("delete from emp765 where id=33");
        System.out.println(result+" records affected");
        con.close();
   }
}
```

2 PreparedStatement interface [5 Marks]

The PreparedStatement interface is a subinterface of Statement. It is used to execute parameterized query.
Let's see the example of parameterized query:
String sql="insert into emp values(?,?,?)";
As you can see, we are passing parameter (?) for the values. Its value will be set by calling the setter methods of PreparedStatement.

Why use PreparedStatement?

Improves performance: The performance of the application will be faster if you use PreparedStatement interface because query is compiled only once.

How to get the instance of PreparedStatement?

The prepareStatement() method of Connection interface is used to return the object of PreparedStatement. Syntax:

public PreparedStatement prepareStatement(String query)throws SQLException{}
Methods of PreparedStatement interface

The important methods of PreparedStatement interface are given below:

public void setInt(int paramIndex, int value) sets the integer value to the given parameter index.

- public void setString(int paramIndex, String value) sets the String value to the given parameter index.
- public void setFloat(int paramIndex, float value) sets the float value to the given parameter index.
- public void setDouble(int paramIndex, double value) sets the double value to the given parameter index.
- public int executeUpdate() executes the query. It is used for create, drop, insert, update, delete etc.
- public ResultSet executeQuery() executes the select query. It returns an instance of ResultSet.

Example of PreparedStatement interface that inserts the record

First of all create table as given below:

create table emp(id number(10),name varchar2(50));
Now insert records in this table by the code given below:

```
import java.sql.*;
class InsertPrepared
{
	public static void main(String args[])
	{
		try
		{
			Class.forName("oracle.jdbc.driver.OracleDriver");
			Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
			PreparedStatement stmt=con.prepareStatement("insert into Emp values(?,?)");
			stmt.setInt(1,101);//1 specifies the first parameter in the query
			stmt.setString(2,"Ratan");

			int i=stmt.executeUpdate();
			System.out.println(i+" records inserted");

			con.close();

		}
		catch(Exception e)
		{
			System.out.println(e);
		}

	}
}
```

Example of PreparedStatement interface that updates the record

```
PreparedStatement stmt=con.prepareStatement("update emp set name=? where id=?");
stmt.setString(1,"Sonoo");//1 specifies the first parameter in the query i.e. name
stmt.setInt(2,101);

int i=stmt.executeUpdate();
System.out.println(i+" records updated");
```

download this example
Example of PreparedStatement interface that deletes the record

```
PreparedStatement stmt=con.prepareStatement("delete from emp where id=?");
stmt.setInt(1,101);

int i=stmt.executeUpdate();
System.out.println(i+" records deleted");
```

Example of PreparedStatement interface that retrieve the records of a table

```
PreparedStatement stmt=con.prepareStatement("select * from emp");
ResultSet rs=stmt.executeQuery();
while(rs.next()){
System.out.println(rs.getInt(1)+" "+rs.getString(2));
}
```

3 CallableStatement Interface **[5 Marks]**

CallableStatement interface is used to call the stored procedures and functions.

We can have business logic on the database by the use of stored procedures and functions that will make the performance better because these are precompiled.

Suppose you need the get the age of the employee based on the date of birth, you may create a function that receives date as the input and returns age of the employee as the output.

What is the difference between stored procedures and functions.

The differences between stored procedures and functions are given below:

Stored Procedure	Function
is used to perform business logic.	is used to perform calculation.
must not have the return type.	must have the return type.
may return 0 or more values.	may return only one values.
We can call functions from the procedure.	Procedure cannot be called from function.
Procedure supports input and output parameters.	Function supports only input parameter.
Exception handling used	Exception handling not used

How to get the instance of CallableStatement?

The prepareCall() method of Connection interface returns the instance of CallableStatement. Syntax is given below:

```
public CallableStatement prepareCall("{ call procedurename(?,?...?)}");
```

The example to get the instance of CallableStatement is given below:

```
CallableStatement stmt=con.prepareCall("{call myprocedure(?,?)}");
```

It calls the procedure myprocedure that receives 2 arguments.

Full example to call the stored procedure using JDBC

To call the stored procedure, you need to create it in the database. Here, we are assuming that stored procedure looks like this.

```
create or replace procedure "INSERTR"
(id IN NUMBER,
name IN VARCHAR2)
```

```
is
begin
insert into user420 values(id,name);
end;
/
```

The table structure is given below:

```
create table user420(id number(10), name varchar2(200));
```

In this example, we are going to call the stored procedure INSERTR that receives id and name as the parameter and inserts it into the table user420. Note that you need to create the user420 table as well to run this application.

```
import java.sql.*;
public class Proc
{
        public static void main(String[] args) throws Exception
        {

                Class.forName("oracle.jdbc.driver.OracleDriver");
                Connection con=DriverManager.getConnection(
                        "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

                CallableStatement stmt=con.prepareCall("{call insertR(?,?)}");
                stmt.setInt(1,1011);
                stmt.setString(2,"Amit");
                stmt.execute();

                System.out.println("success");
        }
}
```

executeQuery() Vs executeUpdate() Vs execute() In JDBC : **[2 Marks]**

executeQuery()	executeUpdate()	execute()
This method is used to execute the SQL statements which retrieve some data from the database.	This method is used to execute the SQL statements which update or modify the database.	This method can be used for any kind of SQL statements.
This method returns a ResultSet object which contains the results returned by the query.	This method returns an int value which represents the number of rows affected by the query. This value will be the 0 for the statements which return nothing.	This method returns a boolean value. TRUE indicates that query returned a ResultSet object and FALSE indicates that query returned an int value or returned nothing.
This method is used to execute only select queries.	This method is used to execute only non-select queries.	This method can be used for both select and non-select queries.
Ex : SELECT	Ex : DML → INSERT, UPDATE and DELETE DDL → CREATE, ALTER	This method can be used for any type of SQL statements.

ResultSet interface **[5 Marks]**

The object of ResultSet maintains a cursor pointing to a row of a table. Initially, cursor points to before the first row.

By default, ResultSet object can be moved forward only and it is not updatable.

But we can make this object to move forward and backward direction by passing either TYPE_SCROLL_INSENSITIVE or TYPE_SCROLL_SENSITIVE in createStatement(int,int) method as well as we can make this object as updatable by:

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                ResultSet.CONCUR_UPDATABLE);
```

Example of Scrollable ResultSet

Let's see the simple example of ResultSet interface to retrieve the data of 3rd row.

```
import java.sql.*;
class FetchRecord
{
public static void main(String args[])throws Exception
{

Class.forName("oracle.jdbc.driver.OracleDriver");
Connection
con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
Statement
stmt=con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_UPDATABLE);
ResultSet rs=stmt.executeQuery("select * from emp765");

//getting the record of 3rd row
rs.absolute(3);
System.out.println(rs.getString(1)+" "+rs.getString(2)+" "+rs.getString(3));

con.close();
}
}
```

Transaction Management in JDBC [5 Marks]

Transaction represents a single unit of work.

The ACID properties describes the transaction management well. ACID stands for Atomicity, Consistency, isolation and durability.

Atomicity means either all successful or none.

Consistency ensures bringing the database from one consistent state to another consistent state.

Isolation ensures that transaction is isolated from other transaction.

Durability means once a transaction has been committed, it will remain so, even in the event of errors, power loss etc.

Advantage of Transaction Mangaement

fast performance It makes the performance fast because database is hit at the time of commit.

In JDBC, Connection interface provides methods to manage transaction.

- void setAutoCommit(boolean status) It is true bydefault means each transaction is committed bydefault.
- void commit() commits the transaction.
- void rollback() cancels the transaction.

Simple example of transaction management in jdbc using Statement

Let's see the simple example of transaction management using Statement.

```
import java.sql.*;
class FetchRecords
{
        public static void main(String args[])throws Exception
        {
                Class.forName("oracle.jdbc.driver.OracleDriver");
                Connection
                con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system",
                "oracle");
                con.setAutoCommit(false);

                Statement stmt=con.createStatement();
                stmt.executeUpdate("insert into user420 values(190,'abhi',40000)");
                stmt.executeUpdate("insert into user420 values(191,'umesh',50000)");

                con.commit();
                con.close();
        }
}
```

If you see the table emp400, you will see that 2 records has been added.

Example of transaction management in jdbc using PreparedStatement

```
import java.sql.*;
import java.io.*;
class TM
{
public static void main(String args[])
{
try
{

Class.forName("oracle.jdbc.driver.OracleDriver");
Connection
con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
con.setAutoCommit(false);

PreparedStatement ps=con.prepareStatement("insert into user420 values(?,?,?)");

BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
while(true){

System.out.println("enter id");
String s1=br.readLine();
int id=Integer.parseInt(s1);

System.out.println("enter name");
String name=br.readLine();

System.out.println("enter salary");
String s3=br.readLine();
int salary=Integer.parseInt(s3);

ps.setInt(1,id);
ps.setString(2,name);
ps.setInt(3,salary);
ps.executeUpdate();

System.out.println("commit/rollback");
String answer=br.readLine();
if(answer.equals("commit")){
con.commit();
}
if(answer.equals("rollback")){
con.rollback();
}
System.out.println("Want to add more records y/n");
String ans=br.readLine();
if(ans.equals("n"))
{
break;
}

}
con.commit();
System.out.println("record successfully saved");

con.close();//before closing connection commit() is called
}
catch(Exception e)
{
System.out.println(e);
}
}
}
```

It will ask to add more records until you press n. If you press n, transaction is committed.

MetaData – Data about data is called metadata **[2 Marks]**

DatabaseMetaData interface provides methods to get meta data of a database such as database product name, database product version, driver name, name of total number of tables, name of total number of views etc.

Commonly used methods of DatabaseMetaData interface

- public String getDriverName()throws SQLException: it returns the name of the JDBC driver.
- public String getDriverVersion()throws SQLException: it returns the version number of the JDBC driver.
- public String getUserName()throws SQLException: it returns the username of the database.
- public String getDatabaseProductName()throws SQLException: it returns the product name of the database.
- public String getDatabaseProductVersion()throws SQLException: it returns the product version of the database.
- public ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types)throws SQLException: it returns the description of the tables of the specified catalog. The table type can be TABLE, VIEW, ALIAS, SYSTEM TABLE, SYNONYM etc.

How to get the object of DatabaseMetaData:

The getMetaData() method of Connection interface returns the object of DatabaseMetaData.

Syntax:
public DatabaseMetaData getMetaData()throws SQLException
Simple Example of DatabaseMetaData interface :

Java ResultSetMetaData Interface **[2 Marks]**

The metadata means data about data i.e. we can get further information from the data.

If you have to get metadata of a table like total number of column, column name, column type etc. , ResultSetMetaData interface is useful because it provides methods to get metadata from the ResultSet object.

Commonly used methods of ResultSetMetaData interface

- public int getColumnCount()throws SQLException it returns the total number of columns in the ResultSet object.
- public String getColumnName(int index)throws SQLException it returns the column name of the specified column index.
- public String getColumnTypeName(int index)throws SQLException it returns the column type name for the specified index.
- public String getTableName(int index)throws SQLException it returns the table name for the specified column index.

How to get the object of ResultSetMetaData:

The getMetaData() method of ResultSet interface returns the object of ResultSetMetaData. Syntax:
public ResultSetMetaData getMetaData()throws SQLException
Example of ResultSetMetaData interface :

UNIT - III

SERVLET, JSP

Introduction to Servlet

Consider a request for a static web page. Auser enters a Uniform Resource Locator (URL) into a browser. The browser generates an HTTPrequest to the appropriate web server. The web server maps this request to a specific file. That file is returned in an HTTPresponse to the browser. The HTTPheader in the response indicates the type of the content.

Now consider dynamic content. Assume that an online store uses a database to store information about its business. This would include items for sale, prices, availability, orders, and so forth. It wishes to make this information accessible to customers via web pages. The contents of those web pages must be dynamically generated to reflect the latest information in the database.

In the early days of the Web, a server could dynamically construct a page by creating a separate process to handle each client request. The process would open connections to one or more databases in order to obtain the necessary information. It communicated with the web server via an interface known as the Common Gateway Interface (CGI).

CGI allowed the separate process to read data from the HTTP request and write data to the HTTP response. A variety of different languages were used to build CGI programs. These included C, C++, and Perl. However, CGI suffered serious performance problems. It was expensive in terms of processor and memory resources to create a separate process for each client request. It was also expensive to open and close database connections for each client request. In addition,

Servlets offer several advantages in comparison with CGI. First, performance is significantly better. Servlets execute within the address space of a web server. It is not necessary to create a separate process to handle each client request. Second, servlets are platform-independent because they are written in Java. Third, the Java security manager on the server enforces a set of restrictions to protect the resources on a server machine. Finally, the full functionality of the Java class libraries is available to a servlet. It can communicate with applets, databases, or other software via the sockets and RMI mechanisms that you have seen already.

Difference between CGI and Servlet [2 Marks]

CGI	Servlet
CGI suffered serious performance problems. It was expensive in terms of processor and memory resources to create a separate process for each client request.	Performance is significantly better. Servlets execute within the address space of a web server. It is not necessary to create a separate process to handle each client request.
Platform dependent	servlets are platform-independent because they are written in Java.
Not secured compared to servlet	Java security manager on the server enforces a set of restrictions to protect the resources on a server machine.

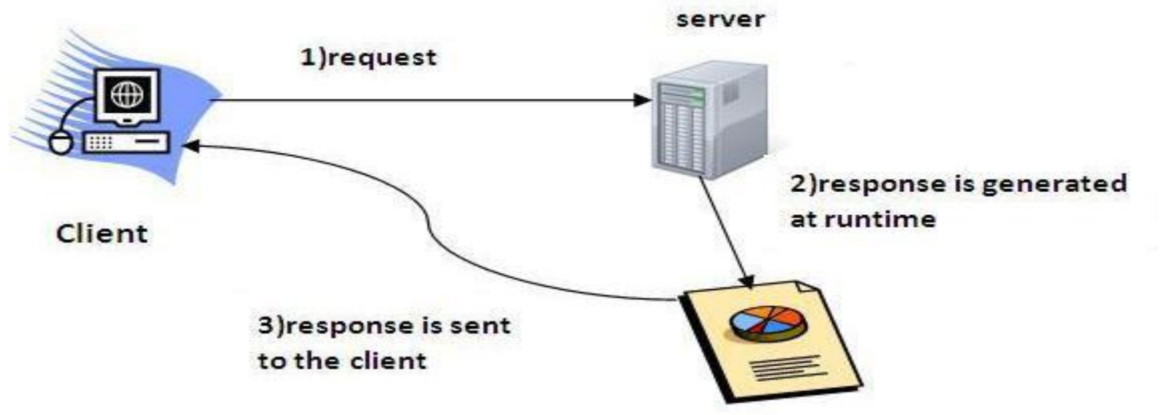
Servlet

- Servlet technology is used to create web application (resides at server side and generates dynamic web page).
- Servlet technology is robust and scalable because of java language. Before Servlet, CGI (Common Gateway Interface) scripting language was popular as a server-side programming language. But there was many disadvantages of this technology.
- There are many interfaces and classes in the servlet API such as Servlet, GenericServlet, HttpServlet, ServletRequest, ServletResponse etc.

What is a Servlet? **[2 Marks]**

Servlet can be described in many ways, depending on the context.

- Servlet is a technology i.e. used to create web application.
- Servlet is an API that provides many interfaces and classes including documentations.
- Servlet is an interface that must be implemented for creating any servlet.
- Servlet is a class that extend the capabilities of the servers and respond to the incoming request. It can respond to any type of requests.
- Servlet is a web component that is deployed on the server to create dynamic web page.



Advantage of Servlet **[2 Marks]**

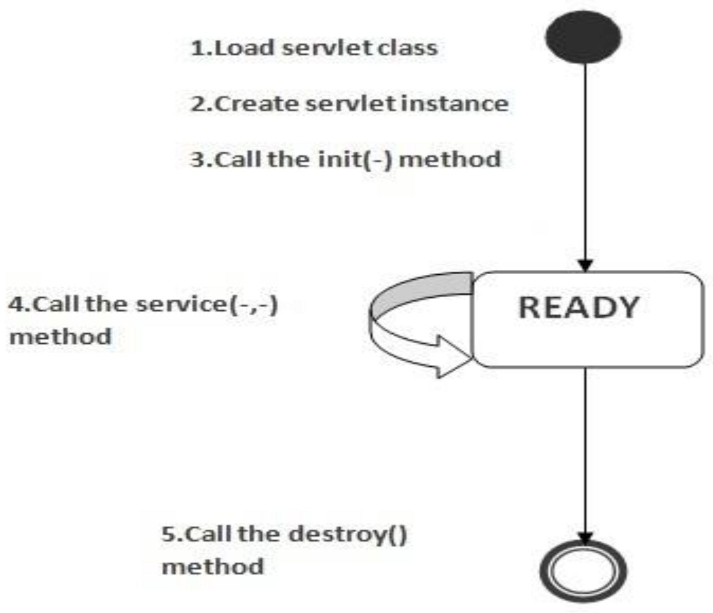
There are many advantages of servlet over CGI. The web container creates threads for handling the multiple requests to the servlet. Threads have a lot of benefits over the Processes such as they share a common memory area, lightweight, cost of communication between the threads are low. The basic benefits of servlet are as follows:

- Better performance: because it creates a thread for each request not process.
- Portability: because it uses java language.
- Robust: Servlets are managed by JVM so we don't need to worry about memory leak, garbage collection etc.
- Secure: because it uses java language..

Life Cycle of a Servlet (Servlet Life Cycle) [5 Marks]

The web container maintains the life cycle of a servlet instance. Let's see the life cycle of the servlet:

1. Servlet class is loaded.
2. Servlet instance is created.
3. init method is invoked.
4. service method is invoked.
5. destroy method is invoked.


1.Load servlet class
2.Create servlet instance
3.Call the init(-) method
4.Call the service(-,-) method
READY
5.Call the destroy() method

As displayed in the above diagram, there are three states of a servlet: new, ready and end. The servlet is in new state if servlet instance is created. After invoking the init() method, Servlet comes in the ready state. In the ready state, servlet performs all the tasks. When the web container invokes the destroy() method, it shifts to the end state.

1) Servlet class is loaded

The classloader is responsible to load the servlet class. The servlet class is loaded when the first request for the servlet is received by the web container.

2) Servlet instance is created

The web container creates the instance of a servlet after loading the servlet class. The servlet instance is created only once in the servlet life cycle.

3) init method is invoked

Third, the server invokes the init( ) method of the servlet. This method is invoked only when the servlet is first loaded into memory. It is possible to pass initialization parameters to the servlet so it may configure itself.

public void init(ServletConfig config) throws ServletException

4) service method is invoked

Fourth, the server invokes the service( ) method of the servlet. This method is called to process the HTTPrequest. You will see that it is possible for the servlet to read data that has been provided in the HTTPrequest. It may also formulate an HTTPresponse for the client. The servlet remains in the server's address space and is available to process any other HTTPrequests received from clients. The service( ) method is called for each HTTPrequest.

```
public void service(ServletRequest request, ServletResponse response)
  throws ServletException, IOException
```

5) destroy method is invoked

Finally, the server may decide to unload the servlet from its memory. The web container calls the destroy method before removing the servlet instance from the service. It gives the servlet an opportunity to clean up any resource for example memory, thread etc. The syntax of the destroy method of the Servlet interface is given below:

```
public void destroy()
```

A Simple Servlet **[5 Marks]**

program:

```
import java.io.*;
 import javax.servlet.*;

public class HelloServlet extends GenericServlet
{
        public void service(ServletRequest request, ServletResponse response)
                throws ServletException, IOException
        {
                response.setContentType("text/html");
                PrintWriter pw = response.getWriter();
                pw.println("<B>Hello!");
                pw.close();
        }
}
```

- Let's look closely at this program. First, note that it imports the javax.servlet package. This package contains the classes and interfaces required to build servlets.
- Next,the program defines HelloServlet as a subclassof GenericServlet.
- The GenericServlet class provides functionality that simplifies the creation of a servlet.
- For example, it provides versions of init( ) and destroy( ), which may be used as is.
- You need supply only the service( ) method. Inside HelloServlet, the service( ) method (which is inherited from GenericServlet) is overridden. This method handles requests from a client.
- Notice that the first argument is a ServletRequest object. This enables the servlet to read data that is provided via the client request.
- The second argument is a ServletResponse object. This enables the servlet to formulate a response for the client.
- The call to setContentType( ) establishes the MIME type of the HTTP response.

- In this program, the MIME type is text/html. This indicates that the browser should interpret the content as HTMLsource code.
- Next, the getWriter( ) method obtains a PrintWriter.
- Anything written to this stream is sent to the client as part of the HTTP response. Then println( ) is used to write some simple HTMLsource code as the HTTPresponse.
- Compile this source code and place the HelloServlet.class file in the proper Tomcat directory as described in the previous section. Also, add HelloServlet to the web.xml file, as described earlier.

```
<servlet>
  <servlet-name>HelloWorld</servlet-name>
  <servlet-class>HelloWorld</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>HelloWorld</servlet-name>
  <url-pattern>/HelloWorld</url-pattern>
</servlet-mapping>
```

Above entries to be created inside <web-app>...</web-app> tags available in web.xml file.

The Servlet API **[2 or 5 Marks]**

Two packages contain the classes and interfaces that are required to build servlets.
These are

1. javax.servlet
2. javax.servlet.http.

The javax.servlet and javax.servlet.http packages represent interfaces and classes for servlet api.

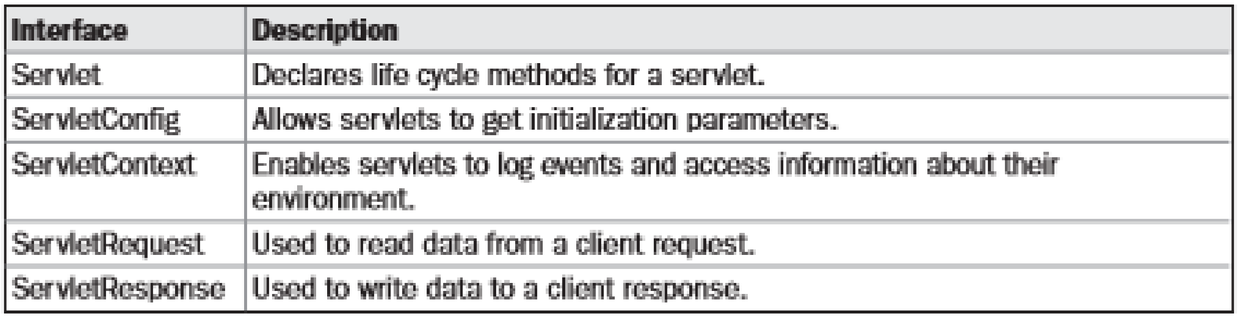
The javax.servlet package contains many interfaces and classes that are used by the servlet or web container. These are not specific to any protocol.

The javax.servlet.http package contains interfaces and classes that are responsible for http requests only.

1 The javax.servlet Package **[2 or 5 Marks]**

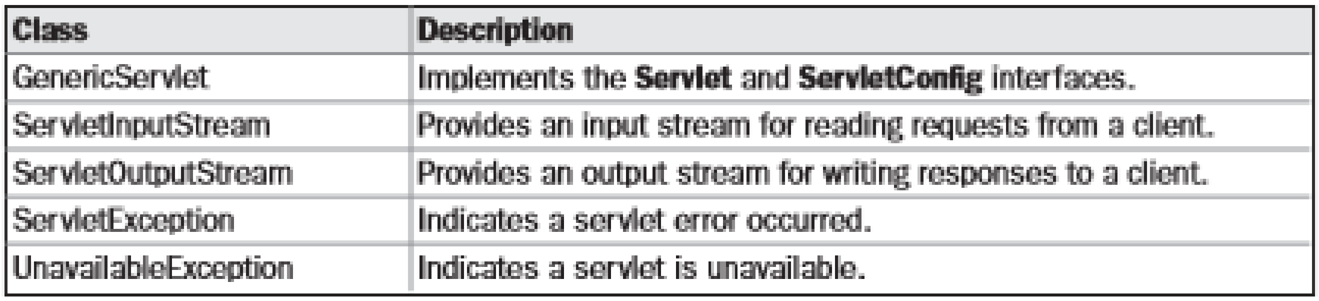
The javax.servlet package contains a number of interfaces and classes that establish the framework in which servlets operate.

The following table summarizes the core interfaces that are provided in this package. [2 Marks]



Interface	Description
Servlet	Declares life cycle methods for a servlet.
ServletConfig	Allows servlets to get initialization parameters.
ServletContext	Enables servlets to log events and access information about their environment.
ServletRequest	Used to read data from a client request.
ServletResponse	Used to write data to a client response.

The following table summarizes the core classes that are provided in the javax.servlet package: [2 Marks]



Class	Description
GenericServlet	Implements the **Servlet** and **ServletConfig** interfaces.
ServletInputStream	Provides an input stream for reading requests from a client.
ServletOutputStream	Provides an output stream for writing responses to a client.
ServletException	Indicates a servlet error occurred.
UnavailableException	Indicates a servlet is unavailable.

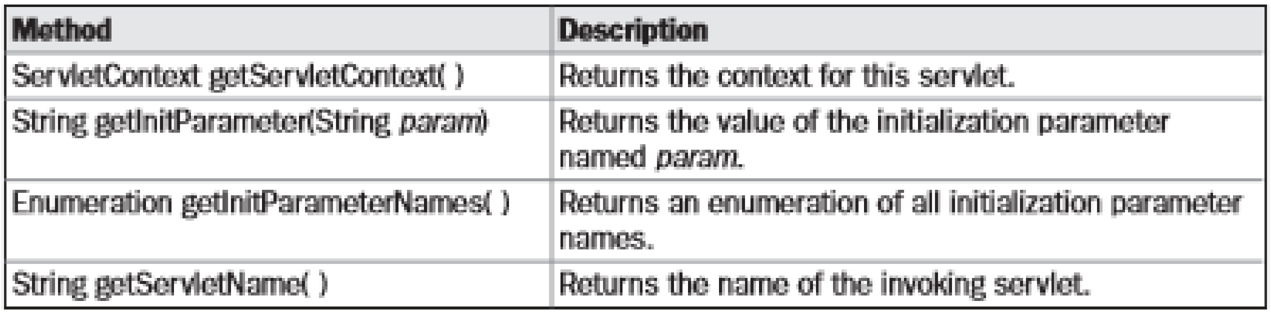
The Servlet Interface

All servlets must implement the Servlet interface. It declares the init ( ), service (), and destroy () methods that are called by the server during the life cycle of a servlet. A method is also provided that allows a servlet to obtain any initialization parameters. The methods defined by Servlet.

The getServletConfig( )

The getServletConfig( ) method is called by the servlet to obtain initialization parameters. A servlet developer overrides the get ServletInfo( ) method to provide a string with useful information (for example, author, version, date, copyright). This method is also invoked by the server. The ServletConfig Interface The ServletConfig interface allows a servlet to obtain configuration data when it is loaded.

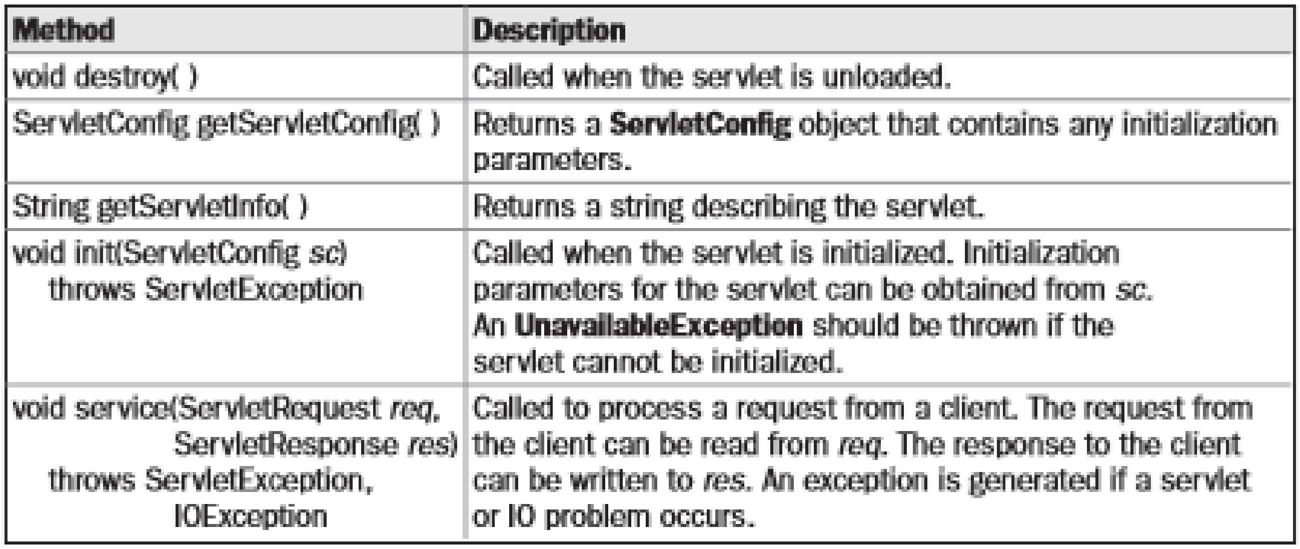
The methods declared by this interface are summarized here: [2 Marks]



Method	Description
ServletContext getServletContext( )	Returns the context for this servlet.
String getInitParameter(String *param*)	Returns the value of the initialization parameter named *param*.
Enumeration getInitParameterNames( )	Returns an enumeration of all initialization parameter names.
String getServletName( )	Returns the name of the invoking servlet.

The ServletContext [2 Marks]

The ServletContext Interface The ServletContext interface enables servlets to obtain information about their environment. Several of Its methods are summarized in Table



Method	Description
void destroy( )	Called when the servlet is unloaded.
ServletConfig getServletConfig( )	Returns a **ServletConfig** object that contains any initialization parameters.
String getServletInfo( )	Returns a string describing the servlet.
void init(ServletConfig *sc*) throws ServletException	Called when the servlet is initialized. Initialization parameters for the servlet can be obtained from *sc*. An **UnavailableException** should be thrown if the servlet cannot be initialized.
void service(ServletRequest *req*, ServletResponse *res*) throws ServletException, IOException	Called to process a request from a client. The request from the client can be read from *req*. The response to the client can be written to *res*. An exception is generated if a servlet or IO problem occurs.

The ServletRequest Interface The ServletRequest interface enables a servlet to obtain information about a client request. **[5 Marks]**

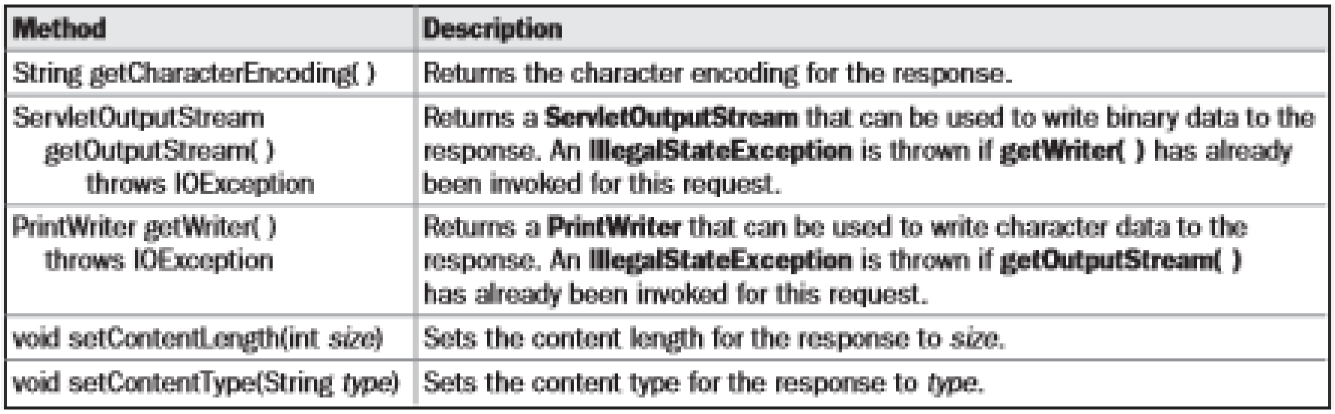
Method	**Description**
Object getAttribute(String *attr*)	Returns the value of the attribute named *attr*.
String getCharacterEncoding( )	Returns the character encoding of the request.
int getContentLength( )	Returns the size of the request. The value –1 is returned if the size is unavailable.
String getContentType( )	Returns the type of the request. A **null** value is returned if the type cannot be determined.
ServletInputStream getInputStream( ) throws IOException	Returns a **ServletInputStream** that can be used to read binary data from the request. An **IllegalStateException** is thrown if **getReader( )** has already been invoked for this request.
String getParameter(String *pname*)	Returns the value of the parameter named *pname*.
Enumeration getParameterNames( )	Returns an enumeration of the parameter names for this request.
String[ ] getParameterValues(String *name*)	Returns an array containing values associated with the parameter specified by *name*.
String getProtocol( )	Returns a description of the protocol.
BufferedReader getReader( ) throws IOException	Returns a buffered reader that can be used to read text from the request. An **IllegalStateException** is thrown if **getInputStream( )** has already been invoked for this request.
String getRemoteAddr( )	Returns the string equivalent of the client IP address.
String getRemoteHost( )	Returns the string equivalent of the client host name.
String getScheme( )	Returns the transmission scheme of the URL used for the request (for example, "http", "ftp").
String getServerName( )	Returns the name of the server.
int getServerPort( )	Returns the port number.

The ServletResponse Interface The ServletResponse interface enables a servlet to formulate a response for a client. **[2 Marks]**

Method	**Description**
Object getAttribute(String *attr*)	Returns the value of the server attribute named *attr*.
String getMimeType(String *file*)	Returns the MIME type of *file*.
String getRealPath(String *vpath*)	Returns the real path that corresponds to the virtual path *vpath*.
String getServerInfo( )	Returns information about the server.
void log(String *s*)	Writes *s* to the servlet log.
void log(String *s*, Throwable *e*)	Writes *s* and the stack trace for *e* to the servlet log.
void setAttribute(String *attr*, Object *val*)	Sets the attribute specified by *attr* to the value passed in *val*.

The GenericServlet Class **[2 Marks]**

The Generic Servlet class provides implementations of the basic life cycle methods for a servlet. Generic Servlet implements the Servlet and ServletConfig interfaces. In addition, a method to append a string to the server logfile is available. The signatures of this method are shown here: void log(String s) void log(String s, Throwable e) Here, s is the string to be appended to the log, and e is an exception that occurred.



Method	**Description**
String getCharacterEncoding( )	Returns the character encoding for the response.
ServletOutputStream getOutputStream( ) throws IOException	Returns a **ServletOutputStream** that can be used to write binary data to the response. An **IllegalStateException** is thrown if **getWriter( )** has already been invoked for this request.
PrintWriter getWriter( ) throws IOException	Returns a **PrintWriter** that can be used to write character data to the response. An **IllegalStateException** is thrown if **getOutputStream( )** has already been invoked for this request.
void setContentLength(int *size*)	Sets the content length for the response to *size*.
void setContentType(String *type*)	Sets the content type for the response to *type*.

The ServletInputStream Class

The ServletInputStream class extends InputStream. It is implemented by the servlet container and provides an input stream that a servlet developer can use to read the data from a client request. It defines the default constructor. In addition, a method is provided to read bytes from the stream. It is shown here:

int readLine(byte[ ] buffer, int offset, int size) throws IOException

Here, buffer is the array into which size bytes are placed starting at offset. The method returns the actual number of bytes read or –1 if an end-of-stream condition is encountered.

The ServletOutputStream Class

The ServletOutputStream class extends OutputStream. It is implemented by the servlet container and provides an output stream that a servlet developer can use to write data to a client response. A default constructor is defined. It also defines the print( ) and println( ) methods, which output data to the stream. The Servlet Exception Classes javax.servlet defines two exceptions. The first is ServletException, which indicates that a servlet problem has occurred. The second is UnavailableException, which extends ServletException. It indicates that a servlet is unavailable.

2 The javax.servlet.http Package [2 or 5 Marks]

The **javax.servlet.http** package contains a number of interfaces and classes that are commonly used by servlet developers.

The following table summarizes the core interfaces that are provided in this package:

Interface	Description
HttpServletRequest	Enables servlets to read data from an HTTP request.
HttpServletResponse	Enables servlets to write data to an HTTP response.
HttpSession	Allows session data to be read and written.
HttpSessionBindingListener	Informs an object that it is bound to or unbound from a session.

The following table summarizes the core classes that are provided in this package.
The most important of these is **HttpServlet**. Servlet developers typically extend this class in Order to process HTTP requests.

Class	Description
Cookie	Allows state information to be stored on a client machine.
HttpServlet	Provides methods to handle HTTP requests and responses.
HttpSessionEvent	Encapsulates a session-changed event.
HttpSessionBindingEvent	Indicates when a listener is bound to or unbound from a session value, or that a session attribute changed.

The HttpServletRequest Interface **[5 Marks]**

The **HttpServletRequest** interface enables a servlet to obtain information about a client

Method	Description
String getAuthType( )	Returns authentication scheme.
Cookie[ ] getCookies( )	Returns an array of the cookies in this request.
long getDateHeader(String *field*)	Returns the value of the date header field named *field*.
String getHeader(String *field*)	Returns the value of the header field named *field*.
Enumeration getHeaderNames( )	Returns an enumeration of the header names.
int getIntHeader(String *field*)	Returns the **int** equivalent of the header field named *field*.
String getMethod( )	Returns the HTTP method for this request.
String getPathInfo( )	Returns any path information that is located after the servlet path and before a query string of the URL.
String getPathTranslated( )	Returns any path information that is located after the servlet path and before a query string of the URL after translating it to a real path.
String getQueryString( )	Returns any query string in the URL.
String getRemoteUser( )	Returns the name of the user who issued this request.
String getRequestedSessionId( )	Returns the ID of the session.
String getRequestURI( )	Returns the URI.
StringBuffer getRequestURL( )	Returns the URL.
String getServletPath( )	Returns that part of the URL that identifies the servlet.
HttpSession getSession( )	Returns the session for this request. If a session does not exist, one is created and then returned.
HttpSession getSession(boolean *new*)	If *new* is **true** and no session exists, creates and returns a session for this request. Otherwise, returns the existing session for this request.
boolean isRequestedSessionIdFromCookie( )	Returns **true** if a cookie contains the session ID. Otherwise, returns **false**.
boolean isRequestedSessionIdFromURL( )	Returns **true** if the URL contains the session ID. Otherwise, returns **false**.
boolean isRequestedSessionIdValid( )	Returns **true** if the requested session ID is valid in the current session context.

The HttpServletResponse Interface [5 Marks]

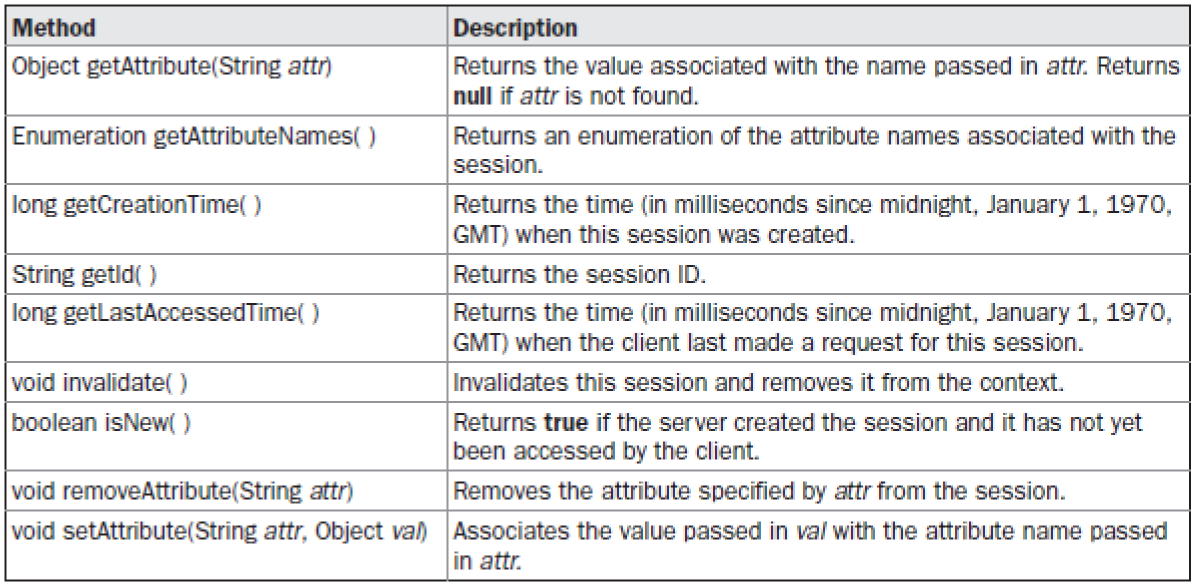
The **HttpServletResponse** interface enables a servlet to formulate an HTTP response to a client. Several constants are defined. These correspond to the different status codes that can be assigned to an HTTP response. For example, **SC_OK** indicates that the HTTP request succeeded, and **SC_NOT_FOUND** indicates that the requested resource is not available.

Several methods of this interface are summarized in Table

Method	**Description**
void addCookie(Cookie *cookie*)	Adds *cookie* to the HTTP response.
boolean containsHeader(String *field*)	Returns **true** if the HTTP response header contains a field named *field*.
String encodeURL(String *url*)	Determines if the session ID must be encoded in the URL identified as *url*. If so, returns the modified version of *url*. Otherwise, returns *url*. All URLs generated by a servlet should be processed by this method.
String encodeRedirectURL(String *url*)	Determines if the session ID must be encoded in the URL identified as *url*. If so, returns the modified version of *url*. Otherwise, returns *url*. All URLs passed to **sendRedirect( )** should be processed by this method.
void sendError(int *c*) throws IOException	Sends the error code *c* to the client.
void sendError(int *c*, String *s*) throws IOException	Sends the error code *c* and message *s* to the client.
void sendRedirect(String *url*) throws IOException	Redirects the client to *url*.
void setDateHeader(String *field*, long *msec*)	Adds *field* to the header with date value equal to *msec* (milliseconds since midnight, January 1, 1970, GMT).
void setHeader(String *field*, String *value*)	Adds *field* to the header with value equal to *value*.
void setIntHeader(String *field*, int *value*)	Adds *field* to the header with value equal to *value*.
void setStatus(int *code*)	Sets the status code for this response to *code*.

The HttpSession Interface **[5 Marks]**

The **HttpSession** interface enables a servlet to read and write the state information that is associated with an HTTP session.



Method	Description
Object getAttribute(String *attr*)	Returns the value associated with the name passed in *attr*. Returns **null** if *attr* is not found.
Enumeration getAttributeNames( )	Returns an enumeration of the attribute names associated with the session.
long getCreationTime( )	Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when this session was created.
String getId( )	Returns the session ID.
long getLastAccessedTime( )	Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when the client last made a request for this session.
void invalidate( )	Invalidates this session and removes it from the context.
boolean isNew( )	Returns **true** if the server created the session and it has not yet been accessed by the client.
void removeAttribute(String *attr*)	Removes the attribute specified by *attr* from the session.
void setAttribute(String *attr*, Object *val*)	Associates the value passed in *val* with the attribute name passed in *attr*.

The HttpSessionBindingListener Interface

The **HttpSessionBindingListener** interface is implemented by objects that need to be notified when they are bound to or unbound from an HTTP session. The methods that are invoked when an object is bound or unbound are

void valueBound(HttpSessionBindingEvent *e*)
void valueUnbound(HttpSessionBindingEvent *e*)
Here, *e* is the event object that describes the binding.

The Cookie Class [5 Marks]

The **Cookie** class encapsulates a cookie. A *cookie* is stored on a client and contains state information. Cookies are valuable for tracking user activities.

For example, assume that a The user does not need to enter this data each time he or she visits the store. A servlet can write a cookie to a user's machine via the **addCookie( )** method of the **HttpServletResponse** interface. The data for that cookie is then included in the header of the HTTP response that is sent to the browser. The names and values of cookies are stored on the user's machine. Some of the information that is saved for each cookie includes the following:

• The name of the cookie
• The value of the cookie
• The expiration date of the cookie
• The domain and path of the cookie

The expiration date determines when this cookie is deleted from the user's machine. If an expiration date is not explicitly assigned to a cookie, it is deleted when the current browser Session ends. Otherwise, the cookie is saved in a file on the user's machine.
The domain and path of the cookie determine when it is included in the header of an HTTP request. If the user enters a URL whose domain and path match these values, the cookie is then supplied to the Web server. Otherwise, it is not.

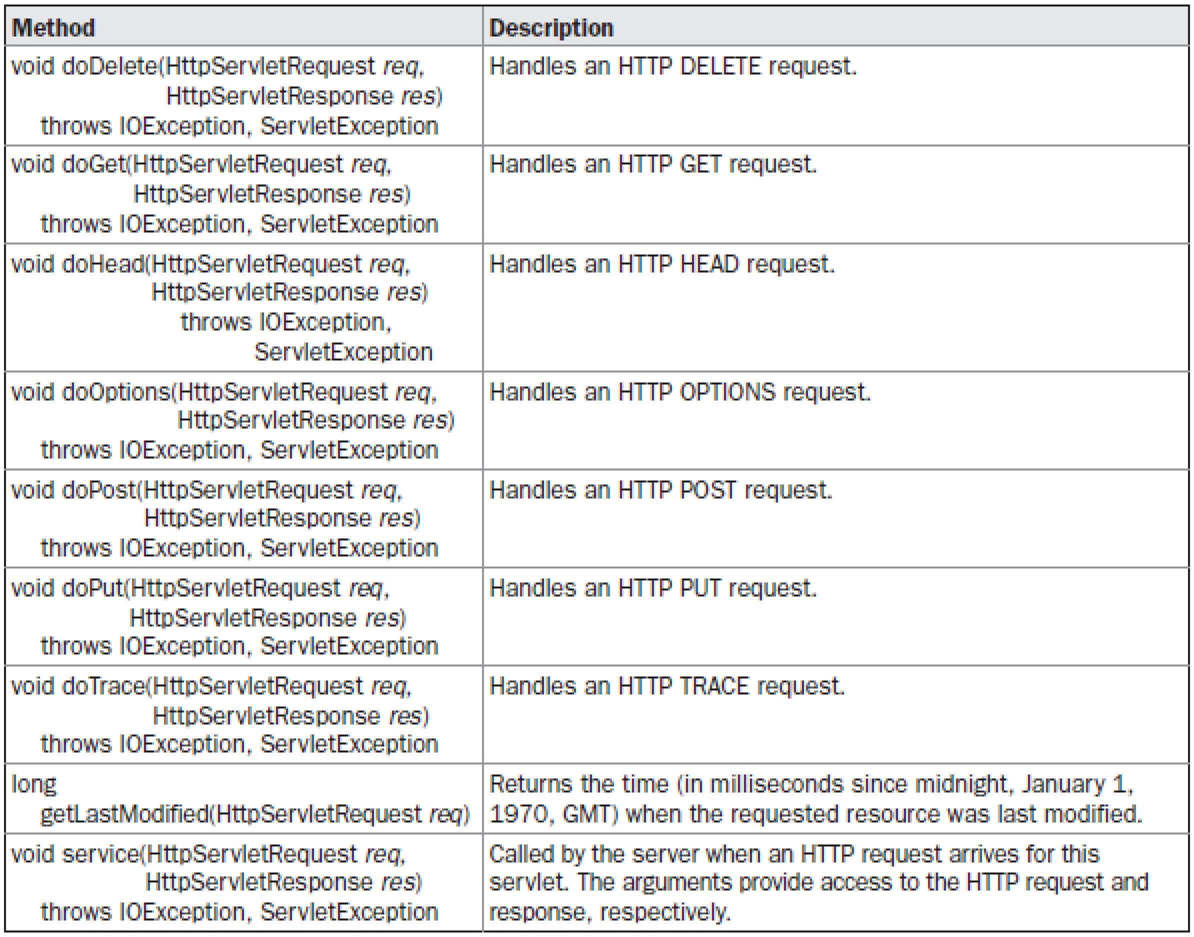
There is one constructor for Cookie. It has the signature shown here:

Cookie(String *name*, String *value*)
Here, the name and value of the cookie are supplied as arguments to the constructor.
The methods of the Cookie class are summarized in Table.

Method	Description
Object clone( )	Returns a copy of this object.
String getComment( )	Returns the comment.
String getDomain( )	Returns the domain.
int getMaxAge( )	Returns the maximum age (in seconds).
String getName( )	Returns the name.
String getPath( )	Returns the path.
boolean getSecure( )	Returns **true** if the cookie is secure. Otherwise, returns **false**.
String getValue( )	Returns the value.
int getVersion( )	Returns the version.
void setComment(String *c*)	Sets the comment to *c*.
void setDomain(String *d*)	Sets the domain to *d*.
void setMaxAge(int *secs*)	Sets the maximum age of the cookie to *secs*. This is the number of seconds after which the cookie is deleted.
void setPath(String *p*)	Sets the path to *p*.
void setSecure(boolean *secure*)	Sets the security flag to *secure*.
void setValue(String *v*)	Sets the value to *v*.
void setVersion(int *v*)	Sets the version to *v*.

The HttpServlet Class **[5 Marks]**

The **HttpServlet** class extends **GenericServlet**. It is commonly used when developing servlets that receive and process HTTP requests.



Method	Description
void doDelete(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP DELETE request.
void doGet(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP GET request.
void doHead(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP HEAD request.
void doOptions(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP OPTIONS request.
void doPost(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP POST request.
void doPut(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP PUT request.
void doTrace(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP TRACE request.
long getLastModified(HttpServletRequest *req*)	Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when the requested resource was last modified.
void service(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Called by the server when an HTTP request arrives for this servlet. The arguments provide access to the HTTP request and response, respectively.

The HttpSessionEvent Class

HttpSessionEvent encapsulates session events. It extends **EventObject** and is generated when a change occurs to the session.

It defines this constructor:

HttpSessionEvent(HttpSession *session*)

Here, *session* is the source of the event.

The HttpSessionBindingEvent Class

The **HttpSessionBindingEvent** class extends **HttpSessionEvent**. It is generated when a listener is bound to or unbound from a value in an **HttpSession** object. It is also generated when an attribute is bound or unbound.

Here are its constructors:

HttpSessionBindingEvent(HttpSession *session*, String *name*)

HttpSessionBindingEvent(HttpSession *session*, String *name*, Object *val*)
Here, *session* is the source of the event, and *name* is the name associated with the object that is being bound or unbound.

Handling HTTP Requests and Responses

The **HttpServlet** class provides specialized methods that handle the various types of HTTP requests. A servlet developer typically overrides one of these methods.

These methods are
doDelete( ), **doGet( )**, **doHead( )**, **doOptions( )**, **doPost( )**, **doPut( )**, and **doTrace( )**.

Handling HTTP GET Requests **[5 Marks]**

Here we will develop a servlet that handles an HTTP GET request. The servlet is invoked when a form on a web page is submitted. The example contains two files. A web page is defined in **ColorGet.htm**, and a servlet is defined in **ColorGetServlet.java**.

The URL identifies a servlet to process the HTTP GET request.

```
<html>
        <body>
        <center>
        <form name="Form1" action="http://localhost:8080/examples/ColorGetServlet" method="get">
                <B>Color:</B>
                <select name="color" size="1">

                        <option value="Red">Red</option>
                        <option value="Green">Green</option>
                        <option value="Blue">Blue</option>
                </select>
                <br><br>
                <input type=submit value="Submit">
                </form>
        </body>
</html>
```

The **doGet( )** method is overridden to process any HTTP GET requests that are sent to **ColorGetServlet.java** servlet. It uses the **getParameter( )** method of **HttpServletRequest** to obtain the selection that was made by the user. A response is then formulated.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class ColorGetServlet extends HttpServlet
{
        public void doGet(HttpServletRequest request, HttpServletResponse response)
                throws ServletException, IOException
        {
                String color = request.getParameter("color");
                response.setContentType("text/html");
                PrintWriter pw = response.getWriter();
                pw.println("<B>The selected color is: ");
                pw.println(color);
                pw.close();
        }
}
```

Compile the servlet. Next, copy it to the appropriate directory, and update the **web.xml** file. Then, perform these steps to test this example:

1. Start Tomcat, if it is not already running.
2. Display the web page in a browser.
3. Select a color.
4. Submit the web page.

After completing these steps, the browser will display the response that is dynamically generated by the servlet.

One other point: Parameters for an HTTP GET request are included as part of the URL that is sent to the web server. Assume that the user selects the red option and submits the form.

The URL sent from the browser to the server is
http://localhost:8080/examples/ColorGetServlet?color=Red
The characters to the right of the question mark are known as the *query string*

Handling HTTP POST Requests [5 Marks]

Here we will develop a servlet that handles an HTTP POST request. The servlet is invoked when a form on a web page is submitted. The example contains two files. A web page is defined in **ColorPost.htm**, and a servlet is defined in **ColorPostServlet.java**.

```
<html>
        <body>
        <center>
        <form name="Form1" action="http://localhost:8080/examples/ColorGetServlet" method="post">
                <B>Color:</B>
                <select name="color" size="1">

                        <option value="Red">Red</option>
                        <option value="Green">Green</option>
                        <option value="Blue">Blue</option>
                </select>
                <br><br>
                <input type=submit value="Submit">
                </form>
        </body>
</html>
```

The **doPost( )** method is overridden to process any HTTP POST requests that are sent to **ColorPostServlet.java** servlet. It uses the **getParameter( )** method of **HttpServletRequest** to obtain the selection that was made by the user. A response is then formulated.

```
import java.io.*;

import javax.servlet.*;
import javax.servlet.http.*;
public class ColorGetServlet extends HttpServlet
{
	public void doPost(HttpServletRequest request, HttpServletResponse response)
		throws ServletException, IOException
	 {
		String color = request.getParameter("color");
		response.setContentType("text/html");
		PrintWriter pw = response.getWriter();
		pw.println("<B>The selected color is: ");
		pw.println(color);
		pw.close();
	}
}
```

Get vs. Post

There are many differences between the Get and Post request. [5 Marks]

GET	POST
1) In case of Get request, only **limited amount of data** can be sent because data is sent in header.	In case of post request, **large amount of data** can be sent because data is sent in body.
2) Get request is **not secured** because data is exposed in URL bar.	Post request is **secured** because data is not exposed in URL bar.
3) Get request **can be bookmarked.**	Post request **cannot be bookmarked.**
4) Get request is **idempotent** . It means second request will be ignored until response of first request is delivered	Post request is **non-idempotent.**
5) Get request is **more efficient** and used more than Post.	Post request is **less efficient** and used less than get.

Using Cookies **[5 Marks]**

The servlet is invoked when a form on a web page is submitted. The example contains three files as summarized here:

File	Description
AddCookie.htm	Allows a user to specify a value for the cookie named **MyCookie**.
AddCookieServlet.java	Processes the submission of **AddCookie.htm**.
GetCookiesServlet.java	Displays cookie values.

The HTML source code for **AddCookie.htm** is shown in the following listing. This page contains a text field in which a value can be entered. There is also a submit button on the page. When this button is pressed, the value in the text field is sent to **AddCookieServlet** via an HTTP POST request.

```
<html>
<body>
        <center>
        <form name="Form1" method="post" action="http://localhost:8080/examples/AddCookieServlet">
                <B>Enter a value for MyCookie:</B>
                <input type=textbox name="data" size=25 value="">
                <input type=submit value="Submit">
        </form>
</body>
</html>
```

The source code for **AddCookieServlet.java** is shown in the following listing. It gets the value of the parameter named "data". It then creates a **Cookie** object that has the name "MyCookie" and contains the value of the "data" parameter. The cookie is then added to the header of the HTTP response via the **addCookie( )** method. A feedback message is then written to the browser.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class AddCookieServlet extends HttpServlet
{
        public void doPost(HttpServletRequest request,HttpServletResponse response)
        throws ServletException, IOException
        {
                String data = request.getParameter("data"); // Get parameter from HTTP request.
                Cookie cookie = new Cookie("MyCookie", data); // Create cookie.
                // Add cookie to HTTP response.
                response.addCookie(cookie);
                // Write output to browser.
                response.setContentType("text/html");
                PrintWriter pw = response.getWriter();
                pw.println("<B>MyCookie has been set to");
                pw.println(data);
                pw.close();
        }
}
```

The source code for **GetCookiesServlet.java** is shown in the following listing. It invokes the **getCookies( )** method to read any cookies that are included in the HTTP GET request. The names and values of these cookies are then written to the HTTP response. Observe that the **getName( )** and **getValue( )** methods are called to obtain this information.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class GetCookiesServlet extends HttpServlet
{
        public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
        {
                // Get cookies from header of HTTP request.
                Cookie[] cookies = request.getCookies();
                // Display these cookies.
                response.setContentType("text/html");
                PrintWriter pw = response.getWriter();
                pw.println("<B>");
                for(int i = 0; i < cookies.length; i++) {
                String name = cookies[i].getName();
                String value = cookies[i].getValue();
                pw.println("name = " + name +"; value = " + value);
        }
}
```

Session Tracking [5 Marks]

HTTP is a stateless protocol. Each request is independent of the previous one. However, in some applications, it is necessary to save state information so that information can be collected from several interactions between a browser and a server. Sessions provide such a mechanism.

A session can be created via the **getSession( )** method of **HttpServletRequest**. An **HttpSession** object is returned. This object can store a set of bindings that associate names with objects. The **setAttribute( )**, **getAttribute( )**, **getAttributeNames( )**, and **remoeAttribute( )** methods of **HttpSession** manage these bindings. It is important to note that session state is shared among all the servlets that are associated with a particular client.

The following servlet illustrates how to use session state.

The **getSession( )** method gets the current session. A new session is created if one does not already exist. The **getAttribute( )** method is called to obtain the object that is bound to the name "date". That object is a **Date** object that encapsulates the date and time when this page was last accessed. (Of course, there is no such binding when the page is first accessed.) A **Date** object encapsulating the current date and time is then created. The **setAttribute( )** method is called to bind the name "date" to this object.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class DateServlet extends HttpServlet
 {
            public void doGet(HttpServletRequest request, HttpServletResponse response)
                        throws ServletException, IOException

             {

                        // Get the HttpSession object.
                        HttpSession hs = request.getSession(true);
                        // Get writer.
                        response.setContentType("text/html");
                        PrintWriter pw = response.getWriter();
                        pw.print("<B>");
                        // Display date/time of last access.
                        Date date = (Date)hs.getAttribute("date");
                        if(date != null)
                         {
                                    pw.print("Last access: " + date + "<br>");
                        }
                        // Display current date/time.
                        date = new Date();
                        hs.setAttribute("date", date);
                        pw.println("Current date: " + date);
            }
}
```

When you first request this servlet, the browser displays one line with the current date and time information. On subsequent invocations, two lines are displayed. The first line shows the date and time when the servlet was last accessed. The second line shows the current date and time.

Introduction to JSP

JSP technology is used to create dynamic web applications. JSP pages are easier to maintain then a Servlet. JSP pages are opposite of Servlets as a servlet adds HTML code inside Java code, while JSP adds Java code inside HTML using JSP tags. Everything a Servlet can do, a JSP page can also do it.

JSP enables us to write HTML pages containing tags, inside which we can include powerful Java programs. Using JSP, one can easily separate Presentation and Business logic as a web designer can design and update JSP pages creating the presentation layer and java developer can write server side complex computational code without concerning the web design. And both the layers can easily interact over HTTP requests.

Why JSP is preferred over servlets? [2 Marks]

- JSP provides an easier way to code dynamic web pages.
- JSP does not require additional files like, java class files, web.xml etc
- Any change in the JSP code is handled by Web Container (Application server like tomcat), and doesn't require re-compilation.
- JSP pages can be directly accessed, and web.xml mapping is not required like in servlets.

Advantage of JSP over Servlet [2 Marks]

1) **Extension to Servlet-** JSP technology is the extension to servlet technology. We can use all the features of servlet in JSP. In addition to, we can use implicit objects, predefined tags, expression language and Custom tags in JSP, that makes JSP development easy.

2) **Easy to maintain**- JSP can be easily managed because we can easily separate our business logic with presentation logic. In servlet technology, we mix our business logic with the presentation logic.
3) **Fast Development: No need to recompile and redeploy**- If JSP page is modified, we don't need to recompile and redeploy the project. The servlet code needs to be updated and recompiled if we have to change the look and feel of the application.

4) **Less code than Servlet -In** JSP, we can use a lot of tags such as action tags, jstl, custom tags etc. that reduces the code. Moreover, we can use EL, implicit objects etc.

Difference between Servlet and JSP **[5 Marks]**

	Servlet	**JSP**
1	Servlet is faster than jsp	JSP is slower than Servlet because it first translate into java code then compile.
2	In Servlet, if we modify the code then we need recompilation, reloading, restarting the server> It means it is time consuming process.	In JSP, if we do any modifications then just we need to click on refresh button and recompilation, reloading, restart the server is not required.
3	Servlet is a java code.	JSP is tag based approach.
4	In Servlet, there is no such method for running JavaScript at client side.	In JSP, we can use the client side validations using running the JavaScript at client side.
5	To run a Servlet you have to make an entry of Servlet mapping into the deployment descriptor file i.e. web.xml file externally.	For running a JSP there is no need to make an entry of Servlet mapping into the web.xml file externally, you may or not make an entry for JSP file as welcome file list.
6	Coding of Servlet is harden than jsp.	Coding of jsp is easier than Servlet because it is tag based.
7	In MVC pattern, Servlet plays a controller role.	In MVC pattern, JSP is used for showing output data i.e. in MVC it is a view.
8	Servlet accept all protocol request.	JSP will accept only http protocol request.
9	In Servlet, aervice() method need to override.	In JSP no need to override service() method.
10	In Servlet, by default session management is not enabled we need to enable explicitly.	In JSP, session management is automatically enabled.
11	In Servlet we do not have implicit object. It means if we want to use an object then we need to get object explicitly form the servlet.	In JSP, we have implicit object support.
12	In Servlet, we need to implement business logic, presentation logic combined.	In JSP, we can separate the business logic from the presentation logic by uses javaBean technology.
13	In Servlet, all package must be imported on top of the servlet.	In JSP, package imported anywhere top, middle and bottom.

JSP Tags [2 or 5 or 10 Marks]

JSP Scripting element are written inside <% %> tags. These code inside <% %> tags are processed by the JSP engine during translation of the JSP page. Any other text in the JSP page is considered as HTML code or plain text.

There are five different types of JSP Tags

1.	Comment	<%-- comment --%>
2.	Directive	<%@ directive %>
3.	Declaration	<%! declarations %>
4.	Scriptlet	<% scriplets %>
5.	Expression	<%= expression %>

JSP Comment

JSP Comment is used when you are creating a JSP page and want to put in comments about what you are doing. JSP comments are only seen in the JSP page. These comments are not included in servlet source code during translation phase, nor they appear in the HTTP response.

Syntax of JSP comment is as follows :

<%-- JSP comment --%>
Simple Example of JSP Comment

```
<html>
   <head>
      <title>My First JSP Page</title>
   </head>
  <%
     int count = 0;
  %>
 <body>
      <%-- Code to show page count  --%>
      Page Count is  <% out.println(++count); %>
 </body>
```

JSP Declaration Tag [2 Marks]

This is used to declare variable and methods in jsp page will be translate and define as class scope declaration in .java file.

The variable and methods will become global to that jsp. It means in that jsp we can use those variables anywhere and we can call those method anywhere.

At the time of translation container inserts the declaration tag into the class. So the variables become instants variables and methods will become instants methods.

The code written inside the jsp declaration tag is placed outside the service() method of auto-generated Servlet, so it does not get memory at each request.

Syntax
<%! variable declaration or method declaration %> <%! declaration %>

Example of Declaration Tag

```
<html>
   <head>
      <title>My First JSP Page</title>
   </head>
  <%!
     int count = 0;
  %>
 <body>
      Page Count is:
      <% out.println(++count); %>
 </body>
</html>
```
In the above code, we have used the declaration tag to declare variable count.

Directive Tag **[2 Marks]**

Directive Tag gives special instruction to Web Container at the time of page translation. Directive tags are of three types: **page**, **include** and **taglib**.

Directive	**Description**
<%@ page ... %>	defines page dependent properties such as language, session, errorPage etc.
<%@ include ... %>	defines file to be included.
<%@ taglib ... %>	declares tag library used in the page

Expression Tag **[2 Marks]**

Expression Tag is used to print out java language expression that is put between the tags. An expression tag can hold any java language expression that can be used as an argument to the out.print() method.

Syntax of Expression Tag

<%= JavaExpression %>
When the Container sees this
<%= (2*5) %>
It turns it into this:

```
out.print((2*5));
```

Note: Never end an expression with semicolon inside Expression Tag. Like this:

```
<%= (2*5); %>
```

Example of Expression Tag

```
<html>
   <head>
      <title>My First JSP Page</title>
   </head>
  <%
     int count = 0;
  %>
 <body>
      Page Count is  <%= ++count %>
 </body>
```

Scriptlet Tag **[2 Marks]**

Scriptlet Tag allows you to write java code inside JSP page. Scriptlet tag implements the _jspService method functionality by writing script/java code. Syntax of Scriptlet Tag is as follows :

```
<% java code %>
```

Example of Scriptlet

In this example, we will show number of page visit.

```
<html>
   <head>
      <title>My First JSP Page</title>
   </head>
  <%
     int count = 0;
  %>
 <body>
      Page Count is  <% out.println(++count); %>
 </body>
</html>
```

Run your first JSP program in Apache Tomcat Server **[10 Marks]**

This post will give you description about how you can run your project using Apache Tomcat Server.
I would like to define these terms before proceeding:

- **Apache Tomcat Server(Jakarta Tomcat)**: It is an open source web server and servlet container developed by the Apache Software Foundation (ASF). It implements the Java Servlet and the JavaServer Pages (JSP) specifications and provides a pure Java HTTP web server environment for java code to run.
- **JavaServerPages(JSP)**: It is a technology that helps in creating dynamically generated web pages.

Step1

Install **Java.**

Step2

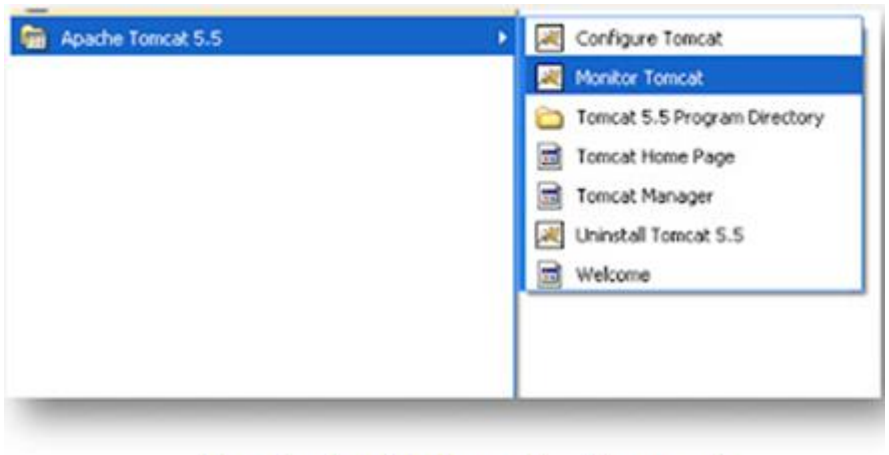
Install **Apache Tomcat**
At the time of installation, it will by-default recognize JRE path.

(under installed java location directory)

Step3

Now Go-To:

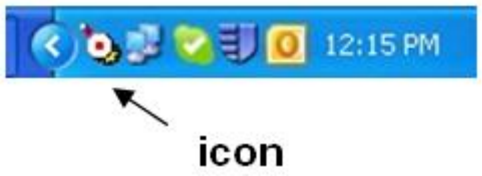
- Start
- Programs
- APACHE TOMCAT
- MONITOR TOMCAT



To start with Apache Tomcat

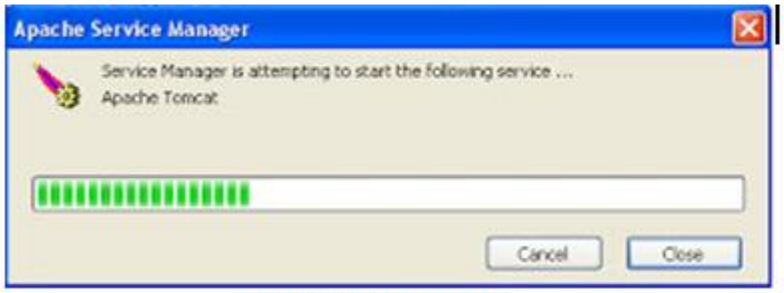
Step4

An **icon** will appear on the taskbar, this **icon** will automatically appear after following above step:



Step5

Click on that **icon** and **START TOMCAT**, you can see the following dialog box:



Service Manager is attempting to start the following service ...
Apache Tomcat

Start Tomcat

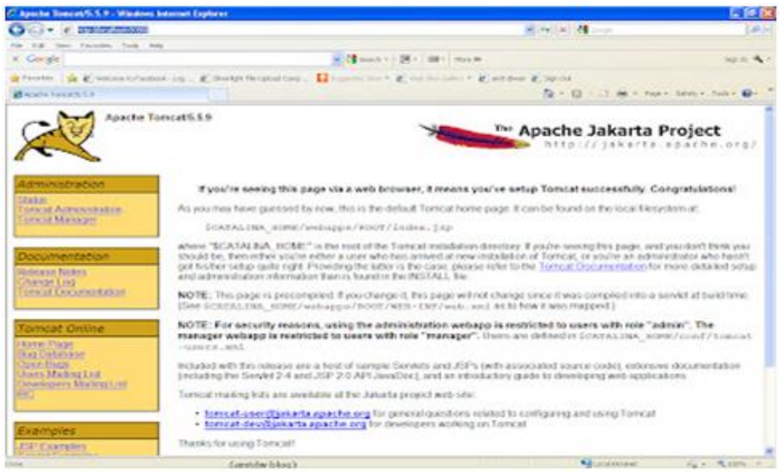
Step6

Now open **Mozilla Firefox**(or any other browser)

Step7

Type **http://localhost:8080/** on address bar and press enter.

The same can be seen here:



Apache Tomcat/5.5.9

The Apache Jakarta Project
http://jakarta.apache.org/

If you're seeing this page via a web browser, it means you've setup Tomcat successfully. Congratulations!

As you may have guessed by now, this is the default Tomcat home page. It can be found on the local filesystem at:

```
$CATALINA_HOME/webapps/ROOT/index.jsp
```

where "$CATALINA_HOME" is the root of the Tomcat installation directory. If you're seeing this page, and you don't think you should be, then either you're either a user who has arrived at new installation of Tomcat, or you're an administrator who hasn't got his/her setup quite right. Providing the latter is the case, please refer to the Tomcat Documentation for more detailed setup and administration information than is found in the INSTALL file.

NOTE: This page is precompiled. If you change it, this page will not change since it was compiled into a servlet at build time. (See $CATALINA_HOME/webapps/ROOT/WEB-INF/web.xml as to how it was mapped.)

NOTE: For security reasons, using the administration webapp is restricted to users with role "admin". The manager webapp is restricted to users with role "manager". Users are defined in $CATALINA_HOME/conf/tomcat-users.xml.

Included with this release are a host of sample Servlets and JSP's (with associated source code), extensive documentation (including the Servlet 2.4 and JSP 2.0 API JavaDoc), and an introductory guide to developing web applications.

Tomcat mailing lists are available at the Jakarta project web site:

- tomcat-user@jakarta.apache.org for general questions related to configuring and using Tomcat
- tomcat-dev@jakarta.apache.org for developers working on Tomcat

Thanks for using Tomcat!

Step8

It will show tomcat, as shown in above window.

(if not, then try again, may be a problem in installation or you're not following above steps correctly

Step9

Now, go to:

- C:drive
- Programs Files
- Apache Software Foundation
- tomcat
- web-apps

(or navigate where you have installed APACHE TOMCAT)

Step10

Open **web-apps** and "copy your project" or "make new folder", which you want to run in JSP. Example: **amit2012PROJECT**

Now, go back :

- Tomcat
- Root
- Copy **Web-inf** from root
- Paste this "web-inf" in your project folder i.e. **amit2012PROJECT**

Step11

Create a text file and name it as **first.jsp**, use the code shown below:

```
<html>
<head>
<title>blog post:ApacheTomcatServer</title>
</head>
<body>

<%-- START --%>
<%
   out.println("UserName = amit2012, ");
   out.println("Running first program in JSP.");
%>
<%-- END --%>

</body>
</html>
```

It includes HTML tags and encloses a **JSP scriptlet** which is a fragment of Java code that is run when the user requests the page.

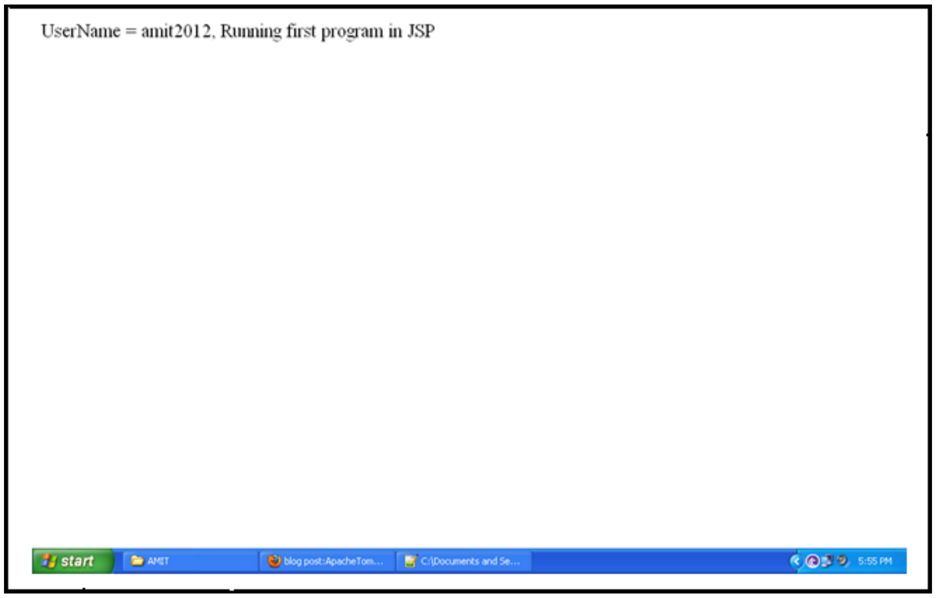
Step12

Now for running your folder [Eg. **amit2012PROJECT** as shown above]

http://localhost:8080/foldername.extension in any WebBrowser i.e:

http://localhost:8080/amit2012PROJECT/first.jsp

The Project will run successfully, as shown below:



```
UserName = amit2012, Running first program in JSP
```

Now, you can successfully try running **JSP** with **ApacheTomcatServer.**

JSP request implicit object(Request String)

The JSP request is an implicit object of type HttpServletRequest i.e. created for each jsp request by the web container. It can be used to get request information such as parameter, header information, remote address, server name, server port, content type, character encoding etc.
It can also be used to set, get and remove attributes from the jsp request scope.
Let's see the simple example of request implicit object where we are printing the name of the user with welcome message.

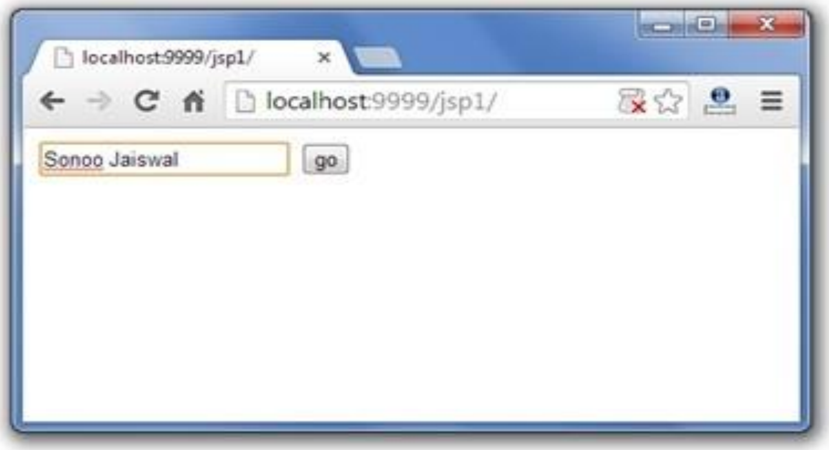
Example of JSP request implicit object
index.html

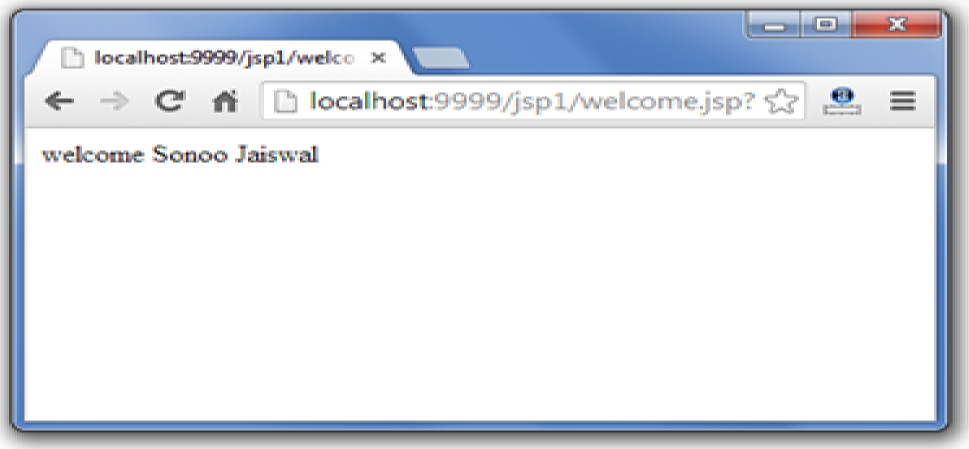
```
<form action="welcome.jsp">
<input type="text" name="uname">
<input type="submit" value="go"><br/>
</form>
```

welcome.jsp

```
<%
String name=request.getParameter("uname");
out.print("welcome "+name);
%>
```

Output





```
welcome Sonoo Jaiswal
```

JSP - Session Tracking

HTTP is a "stateless" protocol which means each time a client retrieves a Web page, the client opens a separate connection to the Web server and the server automatically does not keep any record of previous client request. Still there are following three ways to maintain session between web client and web server:

Cookies:

A webserver can assign a unique session ID as a cookie to each web client and for subsequent requests from the client they can be recognized using the received cookie.

This may not be an effective way because many time browser does not support a cookie, so I would not recommend to use this procedure to maintain the sessions.

Hidden Form Fields:
A web server can send a hidden HTML form field along with a unique session ID as follows:
<input type="hidden" name="sessionid" value="12345">
This entry means that, when the form is submitted, the specified name and value are automatically included in the GET or POST data. Each time when web browser sends request back, then session_id value can be used to keep the track of different web browsers.
This could be an effective way of keeping track of the session but clicking on a regular (<A HREF...>) hypertext link does not result in a form submission, so hidden form fields also cannot support general session tracking.

URL Rewriting:
You can append some extra data on the end of each URL that identifies the session, and the server can associate that session identifier with data it has stored about that session.
For example, with http://ligandsolutions.com/file.htm;sessionid=12345, the session identifier is attached as sessionid=12345 which can be accessed at the web server to identify the client.
URL rewriting is a better way to maintain sessions and works for the browsers when they don't support cookies but here drawback is that you would have generate every URL dynamically to assign a session ID though page is simple static HTML page.

The session Object: [5 Marks]

Apart from the above mentioned three ways, JSP makes use of servlet provided HttpSession Interface which provides a way to identify a user across more than one page request or visit to a Web site and to store information about that user.
By default, JSPs have session tracking enabled and a new HttpSession object is instantiated for each new client automatically. Disabling session tracking requires explicitly turning it off by setting the page directive session attribute to false as follows:

<%@ page session="false" %>

The JSP engine exposes the HttpSession object to the JSP author through the implicit session object. Since session object is already provided to the JSP programmer, the programmer can immediately begin storing and retrieving data from the object without any initialization or getSession().

In JSP, session is an implicit object of type HttpSession.The Java developer can use this object to set,get or remove attribute or to get session information.
Example of session implicit object

index.html

```
<html>
<body>
<form action="welcome.jsp">
<input type="text" name="uname">
<input type="submit" value="go"><br/>
</form>
</body>
</html>
```
welcome.jsp

```
<html>
<body>
<%

String name=request.getParameter("uname");
out.print("Welcome "+name);

session.setAttribute("user",name);

<a href="second.jsp">second jsp page</a>

%>
</body>
</html>
```

second.jsp

```
<html>
<body>
<%

String name=(String)session.getAttribute("user");
out.print("Hello "+name);

%>
</body>
</html>
```

Output

jsp session implicit object output 1 jsp session implicit object output 2 jsp session implicit object output 3

UNIT – IV

NETWORKING AND RMI

The Networking Classes and Interfaces [5 Marks]

Java supports TCP/IP both by extending the already established stream I/O interface introduced in Chapter 19 and by adding the features required to build I/O objects across the network. Java supports both the TCP and UDP protocol families. TCP is used for reliable stream-based I/O across the network. UDP supports a simpler, hence faster, point-to-point datagram-oriented model. The classes contained in the java.net package are shown here:

Authenticator	Inet6Address	ServerSocket
CacheRequest	InetAddress	Socket
CacheResponse	InetSocketAddress	SocketAddress
ContentHandler	InterfaceAddress (Added by Java SE 6.)	SocketImpl
CookieHandler	JarURLConnection	SocketPermission
CookieManager (Added by Java SE 6.)	MulticastSocket	URI
DatagramPacket	NetPermission	URL
DatagramSocket	NetworkInterface	URLClassLoader
DatagramSocketImpl	PasswordAuthentication	URLConnection
HttpCookie (Added by Java SE 6.)	Proxy	URLDecoder
HttpURLConnection	ProxySelector	URLEncoder
IDN (Added by Java SE 6.)	ResponseCache	URLStreamHandler
Inet4Address	SecureCacheResponse	

The **java.net** package's interfaces are listed here:

ContentHandlerFactory	DatagramSocketImplFactory	SocketOptions
CookiePolicy (Added by Java SE 6.)	FileNameMap	URLStreamHandlerFactory
CookieStore (Added by Java SE 6.)	SocketImplFactory	

InetAddress [2 Marks]

The InetAddress class is used to encapsulate both the numerical IP address and the domain name for that address. You interact with this class by using the name of an IP host, which is more convenient and understandable than its IP address. The InetAddress class hides the number inside. InetAddress can handle both IPv4 and IPv6 addresses.

Factory Methods
The InetAddress class has no visible constructors. To create an InetAddress object, you have to use one of the available factory methods. *Factory methods* are merely a convention whereby static methods in a class return an instance of that class. This is done in lieu of overloading a constructor with various parameter lists when having unique method names makes the results much clearer.

Three commonly used InetAddress factory methods are shown here: [2 Marks]

static InetAddress getLocalHost( ) throws UnknownHostException
static InetAddress getByName(String *hostName*) throws UnknownHostException
static InetAddress[ ] getAllByName(String *hostName*) throws UnknownHostException

The getLocalHost( ) method simply returns the InetAddress object that represents the local host. The getByName( ) method returns an InetAddress for a host name passed to it. If these methods are unable to resolve the host name, they throw an UnknownHostException.

Java Socket Programming [2 Marks]

Java Socket programming is used for communication between the applications running on different JRE.

Java Socket programming can be connection-oriented or connection-less.

Socket and ServerSocket classes are used for connection-oriented socket programming and DatagramSocket and DatagramPacket classes are used for connection-less socket programming.

The client in socket programming must know two information:
IP Address of Server, and
Port number.

Socket class
A socket is simply an endpoint for communications between the machines. The Socket class can be used to create a socket.
Important methods [2 Marks]

Method	Description
1) public InputStream getInputStream()	returns the InputStream attached with this socket.
2) public OutputStream getOutputStream()	returns the OutputStream attached with this socket.
3) public synchronized void close()	closes this socket

ServerSocket class **[2 Marks]**

The ServerSocket class can be used to create a server socket. This object is used to establish communication with the clients.

Important methods

Method	Description
1) public Socket accept()	returns the socket and establish a connection between server and client.
2) public synchronized void close()	closes the server socket.

Example of Java Socket Programming **[5 Marks]**

Let's see a simple of java socket programming In which client sends a text and server receives it.

File: MyServer.java

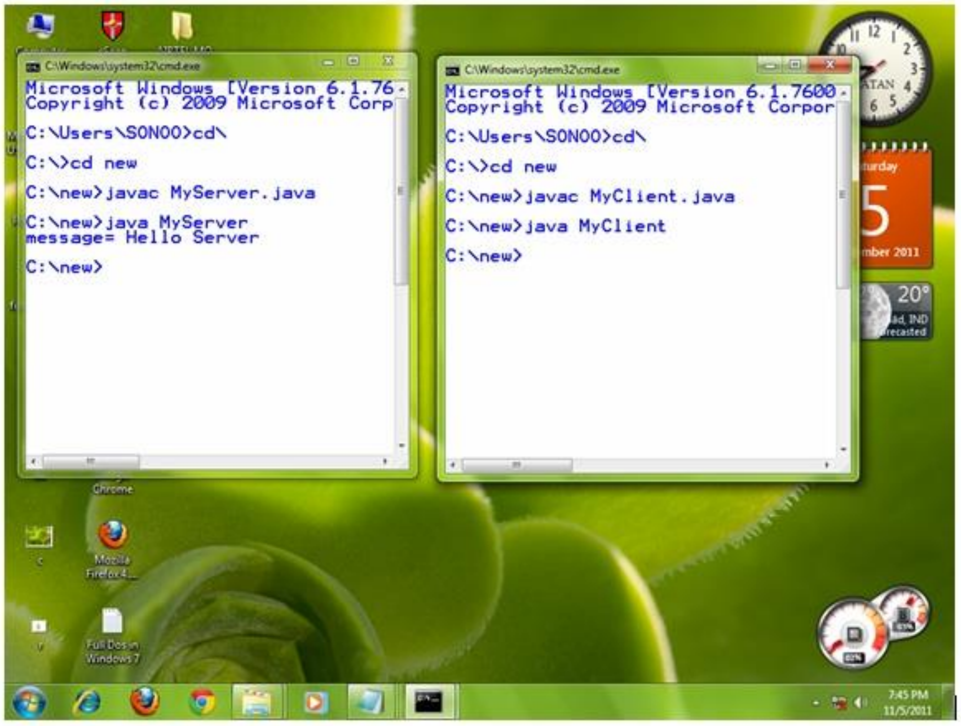
```
import java.io.*;
import java.net.*;
public class MyServer
{
	public static void main(String[] args)
	{
		try
		{
			ServerSocket ss=new ServerSocket(6666);
			Socket s=ss.accept();//establishes connection
			DataInputStream dis=new DataInputStream(s.getInputStream());
			String  str=(String)dis.readUTF();
			System.out.println("message= "+str);
			ss.close();
		}
		catch(Exception e)
		{
			System.out.println(e);
		}
	}
}
```

File: MyClient.java **[5 Marks]**

```
import java.io.*;
import java.net.*;
public class MyClient
{
public static void main(String[] args)
{
Try
{
        Socket s=new Socket("localhost",6666);
        DataOutputStream dout=new DataOutputStream(s.getOutputStream());
        dout.writeUTF("Hello Server");
        dout.flush();
        dout.close();
        s.close();
}
catch(Exception e)
{
        System.out.println(e);}
}
}
```

To execute this program open two command prompts and execute each program at each command prompt as displayed in the below figure.

After running the client application, a message will be displayed on the server console.



```
Microsoft Windows [Version 6.1.76
Copyright (c) 2009 Microsoft Corp

C:\Users\SONOO>cd\

C:\>cd new

C:\new>javac MyServer.java

C:\new>java MyServer
message= Hello Server

C:\new>
```

```
Microsoft Windows [Version 6.1.7600
Copyright (c) 2009 Microsoft Corpor

C:\Users\SONOO>cd\

C:\>cd new

C:\new>javac MyClient.java

C:\new>java MyClient

C:\new>
```

Java URL **[5 Marks]**

The Java URL class represents an URL. URL is an acronym for Uniform Resource Locator. It points to a resource on the World Wide Web. For example:

http://www.javatpoint.com/java-tutorial

A URL contains many information:

Protocol: In this case, http is the protocol.

Server name or IP Address: In this case, www.javatpoint.com is the server name.

Port Number: It is an optional attribute. If we write http//ww.javatpoint.com:80/sonoojaiswal/ , 80 is the port number. If port number is not mentioned in the URL, it returns -1.

File Name or directory name: In this case, index.jsp is the file name.

Commonly used methods of Java URL class

The java.net.URL class provides many methods. The important methods of URL class are given below.

Method	Description
public String getProtocol()	it returns the protocol of the URL.
public String getHost()	it returns the host name of the URL.
public String getPort()	it returns the Port Number of the URL.
public String getFile()	it returns the file name of the URL.
public URLConnection openConnection()	it returns the instance of URLConnection i.e. associated with this URL.

Example of Java URL class

```
//URLDemo.java
import java.io.*;
import java.net.*;
public class URLDemo
{
        public static void main(String[] args)
        {
                Try
                {
                        URL url=new URL("http://www.javatpoint.com/java-tutorial");
                        System.out.println("Protocol: "+url.getProtocol());
                        System.out.println("Host Name: "+url.getHost());
                        System.out.println("Port Number: "+url.getPort());
                        System.out.println("File Name: "+url.getFile());

                }
                catch(Exception e)
                {
                        System.out.println(e);
                }
        }
}
```

Output:

```
Protocol: http
Host Name: www.javatpoint.com
Port Number: -1
File Name: /java-tutorial
```

Java URLConnection class **[5 Marks]**

The Java URLConnection class represents a communication link between the URL and the application. This class can be used to read and write data to the specified resource referred by the URL.

How to get the object of URLConnection class

The openConnection() method of URL class returns the object of URLConnection class.

Syntax:

public URLConnection openConnection()throws IOException{}

Displaying source code of a webpage by URLConnecton class

The URLConnection class provides many methods, we can display all the data of a webpage by using the getInputStream() method. The getInputStream() method returns all the data of the specified URL in the stream that can be read and displayed.

The Methods defined in URL Connection

int getContentLength( )	Returns the size in bytes of the content associated with the resource. If the length is unavailable, –1 is returned.
String getContentType( )	Returns the type of content found in the resource. This is the value of the **content-type** header field. Returns null if the content type is not available.
long getDate( )	Returns the time and date of the response represented in terms of milliseconds since January 1, 1970 GMT.
long getExpiration( )	Returns the expiration time and date of the resource represented in terms of milliseconds since January 1, 1970 GMT. Zero is returned if the expiration date is unavailable.
String getHeaderField(int *idx*)	Returns the value of the header field at index *idx*. (Header field indexes begin at 0.) Returns null if the value of *idx* exceeds the number of fields.
String getHeaderField(String *fieldName*)	Returns the value of header field whose name is specified by *fieldName*. Returns null if the specified name is not found.
String getHeaderFieldKey(int *idx*)	Returns the header field key at index *idx*. (Header field indexes begin at 0.) Returns null if the value of *idx* exceeds the number of fields.
Map<String, List<String>> getHeaderFields( )	Returns a map that contains all of the header fields and values.
long getLastModified( )	Returns the time and date, represented in terms of milliseconds since January 1, 1970 GMT, of the last modification of the resource. Zero is returned if the last-modified date is unavailable.
InputStream getInputStream( ) throws IOException	Returns an **InputStream** that is linked to the resource. This stream can be used to obtain the content of the resource.

Example of Java URLConnecton class

```
import java.io.*;
import java.net.*;
public class URLConnectionExample
 {
        public static void main(String[] args)
        {
                Try
                {
                        URL url=new URL("http://www.javatpoint.com/java-tutorial");
                        URLConnection urlcon=url.openConnection();
                        InputStream stream=urlcon.getInputStream();
                        int i;
                        while((i=stream.read())!=-1)
                        {
                                System.out.print((char)i);
                        }
                }
                catch(Exception e)
                {
                        System.out.println(e);
                }
        }
}
```

Java HttpURLConnection class **[5 Marks]**

The Java HttpURLConnection class is http specific URLConnection. It works for HTTP protocol only.

By the help of HttpURLConnection class, you can information of any HTTP URL such as header information, status code, response code etc.

The java.net.HttpURLConnection is subclass of URLConnection class.

How to get the object of HttpURLConnection class

The openConnection() method of URL class returns the object of URLConnection class.

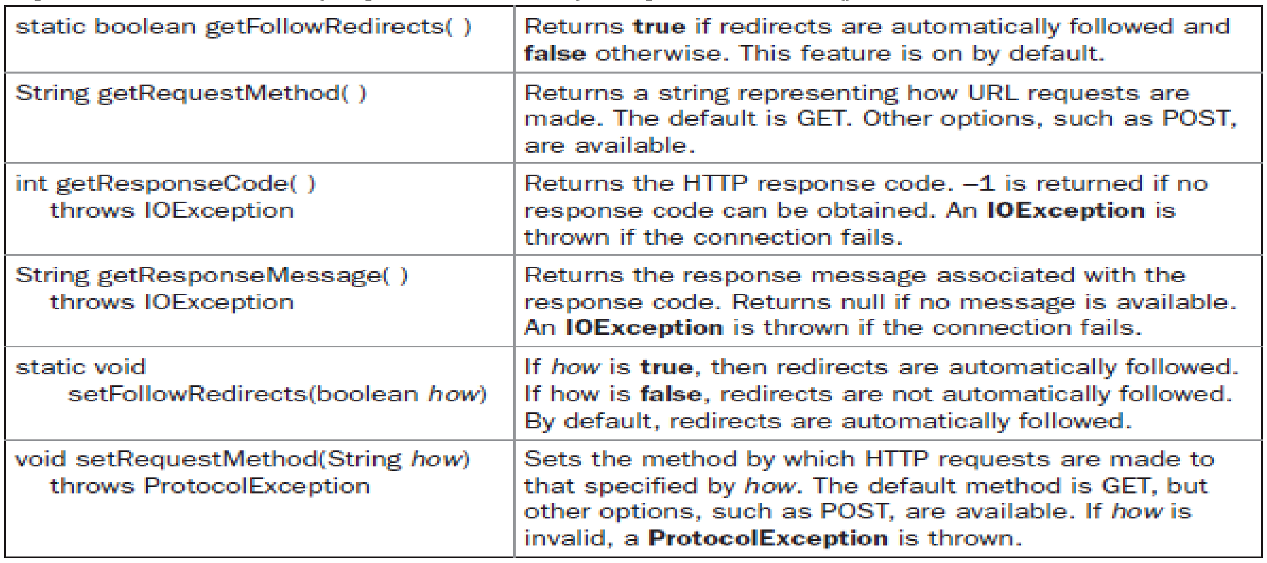
Syntax:

public URLConnection openConnection()throws IOException{}

You can typecast it to HttpURLConnection type as given below.

URL url=new URL("http://www.javatpoint.com/java-tutorial");

HttpURLConnection huc=(HttpURLConnection)url.openConnection();



static boolean getFollowRedirects( )	Returns **true** if redirects are automatically followed and **false** otherwise. This feature is on by default.
String getRequestMethod( )	Returns a string representing how URL requests are made. The default is GET. Other options, such as POST, are available.
int getResponseCode( ) throws IOException	Returns the HTTP response code. –1 is returned if no response code can be obtained. An **IOException** is thrown if the connection fails.
String getResponseMessage( ) throws IOException	Returns the response message associated with the response code. Returns null if no message is available. An **IOException** is thrown if the connection fails.
static void setFollowRedirects(boolean *how*)	If *how* is **true**, then redirects are automatically followed. If how is **false**, redirects are not automatically followed. By default, redirects are automatically followed.
void setRequestMethod(String *how*) throws ProtocolException	Sets the method by which HTTP requests are made to that specified by *how*. The default method is GET, but other options, such as POST, are available. If *how* is invalid, a **ProtocolException** is thrown.

Java HttpURLConnecton Example

```
import java.io.*;
import java.net.*;
public class HttpURLConnectionDemo
{
        public static void main(String[] args)
        {
                Try
                {
                        URL url=new URL("http://www.javatpoint.com/java-tutorial");
                        HttpURLConnection huc=(HttpURLConnection)url.openConnection();
                        for(int i=1;i<=8;i++)
                        {
                                System.out.println(huc.getHeaderFieldKey(i)+" = "+huc.getHeaderField(i));
                        }
                        huc.disconnect();
                }
                catch(Exception e)
                {
                        System.out.println(e);
                }
        }
}
```

Output:

```
Date = Wed, 10 Dec 2014 19:31:14 GMT

Set-Cookie = JSESSIONID=D70B87DBB832820CACA5998C90939D48; Path=/

Content-Type = text/html

Cache-Control = max-age=2592000

Expires = Fri, 09 Jan 2015 19:31:14 GMT

Vary = Accept-Encoding,User-Agent

Connection = close

Transfer-Encoding = chunked
```

Java DatagramSocket and DatagramPacket [2 or 5 Marks]

Java DatagramSocket and DatagramPacket classes are used for connection-less socket programming.

Java DatagramSocket class [2 Marks]

Java DatagramSocket class represents a connection-less socket for sending and receiving datagram packets.

A datagram is basically information but there is no guarantee of its content, arrival or arrival time.

Commonly used Constructors of DatagramSocket class

DatagramSocket() throws SocketEeption: it creates a datagram socket and binds it with the available Port Number on the localhost machine.

DatagramSocket(int port) throws SocketEeption: it creates a datagram socket and binds it with the given Port Number.

DatagramSocket(int port, InetAddress address) throws SocketEeption: it creates a datagram socket and binds it with the specified port number and host address.

Java DatagramPacket class [2 Marks]

Java DatagramPacket is a message that can be sent or received. If you send multiple packet, it may arrive in any order. Additionally, packet delivery is not guaranteed.

Commonly used Constructors of DatagramPacket class

DatagramPacket(byte[] barr, int length): it creates a datagram packet. This constructor is used to receive the packets.

DatagramPacket(byte[] barr, int length, InetAddress address, int port): it creates a datagram packet. This constructor is used to send the packets.

Example of Sending DatagramPacket by DatagramSocket

```
import java.net.*;
public class DSender
{
        public static void main(String[] args) throws Exception
        {
                DatagramSocket ds = new DatagramSocket();
                String str = "Welcome java";
                InetAddress ip = InetAddress.getByName("127.0.0.1");
                DatagramPacket dp = new DatagramPacket(str.getBytes(), str.length(), ip, 3000);
                ds.send(dp);
                ds.close();
         }}
```

Example of Receiving DatagramPacket by DatagramSocket

```
//DReceiver.java
import java.net.*;
public class DReceiver
{
        public static void main(String[] args) throws Exception
        {
                DatagramSocket ds = new DatagramSocket(3000);
                 byte[] buf = new byte[1024];
                 DatagramPacket dp = new DatagramPacket(buf, 1024);
                 ds.receive(dp);
                 String str = new String(dp.getData(), 0, dp.getLength());
                System.out.println(str);
                ds.close();
        }
}
```

RMI (Remote Method Invocation) [5 Marks]

The RMI (Remote Method Invocation) is an API that provides a mechanism to create distributed application in java. The RMI allows an object to invoke methods on an object running in another JVM.

The RMI provides remote communication between the applications using two objects *stub* and *skeleton*.

Understanding stub and skeleton

RMI uses stub and skeleton object for communication with the remote object.

A remote object is an object whose method can be invoked from another JVM. Let's understand the stub and skeleton objects:

stub

The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:

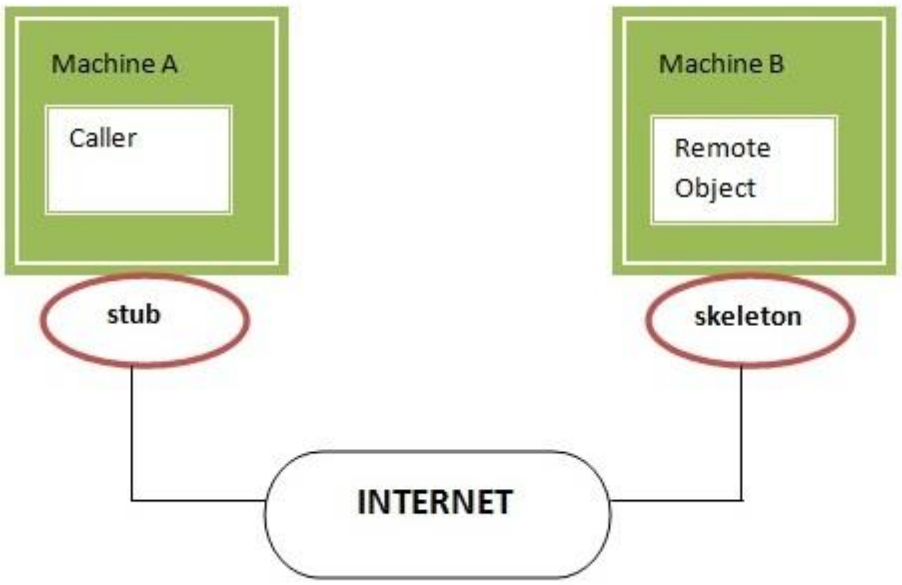
1. It initiates a connection with remote Virtual Machine (JVM),
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads (unmarshals) the return value or exception, and
5. It finally, returns the value to the caller.

skeleton

The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:

1. It reads the parameter for the remote method
2. It invokes the method on the actual remote object, and
3. It writes and transmits (marshals) the result to the caller.

In the Java 2 SDK, an stub protocol was introduced that eliminates the need for skeletons.



Understanding requirements for the distributed applications

If any application performs these tasks, it can be distributed application.

The application need to locate the remote method

It need to provide the communication with the remote objects, and

The application need to load the class definitions for the objects.

The RMI application have all these features, so it is called the distributed application.

Java RMI Example

The is given the 6 steps to write the RMI program. **[5 or 10 Marks]**

1. Create the remote interface
2. Provide the implementation of the remote interface
3. Compile the implementation class and create the stub and skeleton objects using the rmic tool
4. Start the registry service by rmiregistry tool
5. Create and start the remote application
6. Create and start the client application

RMI Example

In this example, we have followed all the 6 steps to create and run the rmi application. The client application need only two files, remote interface and client application. In the rmi application, both client and server interacts with the remote interface. The client application invokes methods on the proxy object, RMI sends the request to the remote JVM. The return value is sent back to the proxy object and then to the client application..

1) create the remote interface

For creating the remote interface, extend the Remote interface and declare the RemoteException with all the methods of the remote interface. Here, we are creating a remote interface that extends the Remote interface. There is only one method named add() and it declares RemoteException.

```
import java.rmi.*;
public interface Adder extends Remote{
public int add(int x,int y)throws RemoteException;
}
```

2) Provide the implementation of the remote interface

Now provide the implementation of the remote interface. For providing the implementation of the Remote interface, we need to

Either extend the UnicastRemoteObject class,

or use the exportObject() method of the UnicastRemoteObject class

In case, you extend the UnicastRemoteObject class, you must define a constructor that declares RemoteException.

```
import java.rmi.*;
import java.rmi.server.*;
public class AdderRemote extends UnicastRemoteObject implements Adder{
AdderRemote()throws RemoteException{
super();
}
public int add(int x,int y){return x+y;}
}
```

3) create the stub and skeleton objects using the rmic tool.

Next step is to create stub and skeleton objects using the rmi compiler. The rmic tool invokes the RMI compiler and creates stub and skeleton objects.

```
rmic AdderRemote
```

4) Start the registry service by the rmiregistry tool

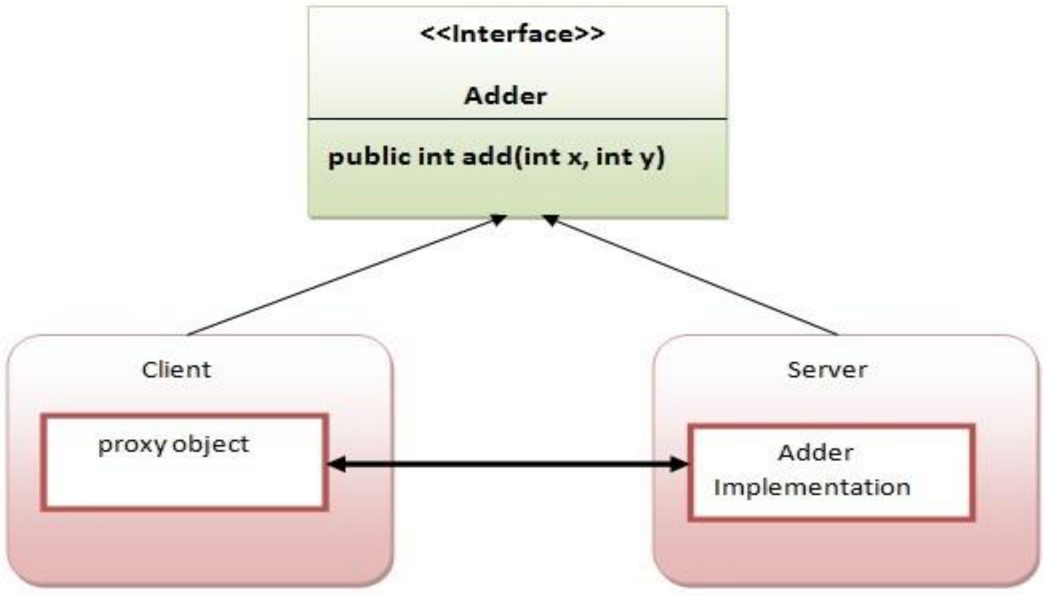
Now start the registry service by using the rmiregistry tool. If you don't specify the port number, it uses a default port number. In this example, we are using the port number 5000.

```
rmiregistry 5000
```

5) Create and run the server application

Now rmi services need to be hosted in a server process. The Naming class provides methods to get and store the remote object. The Naming class provides 5 methods.

public static java.rmi.Remote lookup(java.lang.String) throws java.rmi.NotBoundException, java.net.MalformedURLException, java.rmi.RemoteException;	It returns the reference of the remote object.
public static void bind(java.lang.String, java.rmi.Remote) throws java.rmi.AlreadyBoundException, java.net.MalformedURLException, java.rmi.RemoteException;	It binds the remote object with the given name.
public static void unbind(java.lang.String) throws java.rmi.RemoteException, java.rmi.NotBoundException, java.net.MalformedURLException;	It destroys the remote object which is bound with the given name.
public static void rebind(java.lang.String, java.rmi.Remote) throws java.rmi.RemoteException, java.net.MalformedURLException;	It binds the remote object to the new name.
public static java.lang.String[] list(java.lang.String) throws java.rmi.RemoteException, java.net.MalformedURLException;	It returns an array of the names of the remote objects bound in the registry.



UNIT – V

ENTERPRISE JAVA BEAN

EJB (*Enterprise Java Bean*) **[5 Marks]**

EJB (*Enterprise Java Bean*) is used to develop scalable, robust and secured enterprise applications in java.

Unlike RMI, middleware services such as security, transaction management etc. are provided by **EJB Container** to all EJB applications.

The current version of EJB is EJB 3.2. The development of EJB 3 is faster than EJB 2 because of simplicity and annotations such as @EJB, @Stateless, @Stateful, @ModelDriven, @PreDestroy,

What is EJB **[5 Marks]**

EJB is an acronym for *enterprise java bean*. It is a specification provided by Sun Microsystems to develop secured, robust and scalable distributed applications.

To run EJB application, you need an *application server* (EJB Container) such as Jboss, Glassfish, Web logic, Web sphere etc. It performs:

a. life cycle management,
b. security,
c. transaction management, and
d. Object pooling.

EJB application is deployed on the server, so it is called server side component also.

EJB is like COM (*Component Object Model*) provided by Microsoft. But, it is different from Java Bean, RMI and Web Services.

When use Enterprise Java Bean?

1. **Application needs Remote Access**. In other words, it is distributed.
2. **Application needs to be scalable**. EJB applications supports load balancing, clustering and fail-over.
3. **Application needs encapsulated business logic**. EJB application is separated from presentation and persistent layer.

Difference between RMI and EJB [5 Marks]

Both RMI and EJB, provides services to access an object running in another JVM (known as remote object) from another JVM. The differences between RMI and EJB are given below:

RMI	EJB
In RMI, middleware services such as security, transaction management, object pooling etc. need to be done by the java programmer.	In EJB, middleware services are provided by EJB Container automatically.
RMI is not a server-side component. It is not required to be deployed on the server.	EJB is a server-side component, it is required to be deployed on the server.
RMI is built on the top of socket programming.	EJB technology is built on the top of RMI.

Types of Enterprise Java Bean [5 Marks]

There are 3 types of enterprise bean in java.

Session Bean

Session bean contains business logic that can be invoked by local, remote or web service client.

Message Driven Bean

Like Session Bean, it contains the business logic but it is invoked by passing message.

Entity Bean

It encapsulates the state that can be persisted in the database. It is deprecated. Now, it is replaced with JPA (Java Persistent API).

1 Session Bean **[2 Marks]**

Session bean encapsulates business logic only, it can be invoked by local, remote and web service client.

It can be used for calculations, database access etc.

The life cycle of session bean is maintained by the application server (EJB Container).

Types of Session Bean

There are 3 types of session bean.

1) Stateless Session Bean: It doesn't maintain state of a client between multiple method calls.

2) Stateful Session Bean: It maintains state of a client across multiple requests.

3) Singleton Session Bean: One instance per application, it is shared between clients and supports concurrent access.

2 Entity Bean in EJB **[2 Marks]**

Entity bean represents the persistent data stored in the database. It is a server-side component.

In EJB 2.x, there was two types of entity beans: **bean managed persistence** (BMP) and container managed persistence (CMP).

Since EJB 3.x, it is deprecated and replaced by JPA (Java Persistence API) that is covered in the hibernate tutorial.

In hibernate tutorial, there are given hibernate with annotation examples where we are using JPA annotations. The JPA with Hibernate is widely used today.

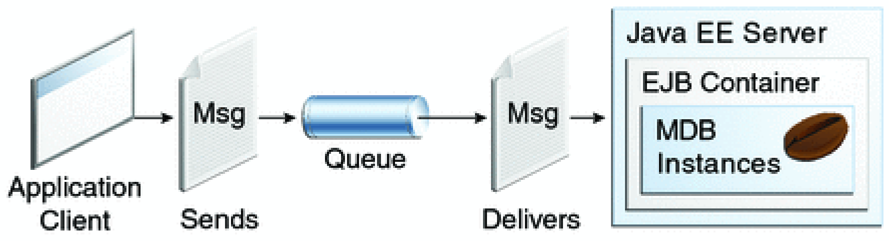
3 Message Driven Bean **[2 Marks]**

A message driven bean (MDB) is a bean that contains business logic. But, it is invoked by passing the message. So, it is like JMS Receiver.

MDB asynchronously receives the message and processes it.

A message driven bean receives message from queue or topic, so you must have the knowledge of JMS API.

A message driven bean is like stateless session bean that encapsulates the business logic and doesn't maintain state.



EJB Interfaces **[5 Marks]**

<–Prior Index Next–>

Interface in java means a group of related methods with empty bodies. EJB have generally 4 interfaces. These are as follows

1) Remote interface:- Remote interface are the interface that has the methods that relate to a Particular bean instance. In the Remote interface we have all get methods as given below in the program. This is the interface where all of the business method go.**javax.ejb.Remote** package is used for creating Remote interface.

```
package ejb;

import javax.ejb.Remote;

@Remote
public interface Bean30Remote
 {

String getMessage();

String getAddress();

String getCompanyname();
}
```

2)Local Interface:-Local interface are the type of interface that are used for making local connections to EJB.**@Local** annotation is used for declaring interface as Local. **javax.ejb.Local** package is used for creating Local interface.

```
package ejb;
import javax.ejb.Local;
@Local
public interface NewSessionLocal
{
}
```

3)Home Interface:-Home interface is the interface that has methods that relate to all EJB of a certain class as a whole. The methods which are defined in this Interface are create() methods and find() methods . The create() method allows us to create beans. find() method is used in Entity beans.

```
import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import java.rmi.RemoteException;

public interface Bean30RemoteHome extends JBHome
{
public Bean30Remote create() throws CreateException, RemoteException;
}
```

4)Localhome Interface:-The local interfaces extend the following interfaces. These interfaces are generally for use by clients
javax.ejb.EJBLocalObject - for the Object interface
javax.ejb.EJBLocalHome - for the Home interface

```
package ejb;

import javax.ejb.EJBLocalHome;
import javax.ejb.CreateException;

public interface NewSessionLocalHome extends EJBLocalHome
 {
public NewSessionLocal create() throws CreateException;
}
```

Using JAR Files: The Basics **[5 Marks]**

JAR files are packaged with the ZIP file format, so you can use them for tasks such as lossless data compression, archiving, decompression, and archive unpacking.

JAR file is the compressed file format. You can store many files in a JAR file. JAR stands for the Java Archive. This file format is used to distribute a set of java classes. This file helps you to reduce the file size and collect many file in one by compressing files. Downloading the files are become completed in very short duration of time because of reducing the file size. You can make the jar file executable by collecting many class file of your java application in it. The jar file can execute from the javaw (Java Web Start).

The JAR file format is based on the popular ZIP file format. Usually these file format is not only used for archiving and distribution the files, these are also used for implementing various libraries, components and plug-ins in java applications. Compiler and JVMs (Java Virtual Machine) can understand and implement these formats for java application.

For mentioning the product information like vendor name, product version, date of creation of the product and many other things related to the product are mentioned in the manifest file. Such type of files are special which are mentioned in the jar file for making it executable for the application. This file format is to be used for collecting auxiliary files associated with the components.

To perform basic operations for the jar file there has to be used the Java Archive Tool (jar tool). It is provided by the jdk (Java Development Kit). Following are some jar command which are invoked by the jar tool:

Functions	Command
creation a jar file	jar cf jar-file-name file-name(s)_or_directory-name
viewing contents of a jar file	jar tf jar-file-name
viewing contents with detail of a jar file	jar tvf jar-file-name
extract all the files of a jar file	jar xf jar-file-name
extract specific files from the jar file	jar xf jar-file-name file-name(s)_from_jar-file
update jar files	jar uf jar-file-name file-name(s)_from_jar-file
running a executable packaged jar file	java -jar jar-file-name