T.Y.B.Sc.(cs)
Paper II
Advanced Java Syllabus

UNIT I

Introduction to JFC and Swing, Features of the Java Foundation Classes, Swing API Components, JComponent Class, Windows, Dialog Boxes, and Panels, Labels, Buttons, Check Boxes, Menus, Toolbars, Implementing Action interface, Pane, JScrollPane, Desktop pane, Scrollbars, Lists and Combo Boxes, Text-Entry Components, Colors and File Choosers, Tables and Trees, Printing with 2D API and Java Print Service API.

JDBC Introduction, JDBC Architecture, Types of JDBC Drivers, The Connectivity Model, The java.sql package, Navigating the ResultSet object's contents, Manipulating records of a ResultSet object through User Interface , The JDBC Exception classes, Database Connectivity, Data Manipulation (using Prepared Statements, Joins, Transactions, Stored Procedures), Data navigation.

UNIT II

Threads and Multithreading, The Lifecycle of a thread, Creating and running threads, Creating the Service Threads, Schedules Tasks using JVM, Thread-safe variables, Synchronizing threads, Communication between threads.

Overview of Networking, Working with URL, Connecting to a Server, Implementing Servers, Serving multiple Clients, Sending E-Mail, Socket Programming, Internet Addresses, URL Connections, Accessing Network interface parameters, Posting Form Data, Cookies, Overview of Understanding the Sockets Direct Protocol.

Introduction to distributed object system, Distributed Object Technologies, RMI for distributed computing, RMI Architecture, RMI Registry Service, Parameter Passing in Remote Methods, Creating RMI application, Steps involved in running the RMI application, Using RMI with Applets.

Unit III

What Is a Servlet? The Example Servlets, Servlet Life Cycle, Sharing Information, Initializing a Servlet, Writing Service Methods, Filtering Requests and Responses, Invoking Other Web Resources, Accessing the Web Context, Maintaining Client State, Finalizing a Servlet.

What Is a JSP Page?, The Example JSP Pages, The Life Cycle of a JSP Page, Creating Static Content, Creating Dynamic Content, Unified Expression Language, JavaBeans Components, JavaBeans Concepts, Using NetBeans GUI Builder Writing a Simple Bean, Properties: Simple Properties, Using Custom tags, Reusing content in JSP Pages, Transferring Control to Another Web Component, Including an Applet.

Unit IV

Introduction to EJB, Benefits of EJB, Types of EJB, Session Bean: State Management Modes; Message-Driven Bean, Differences between Session Beans and Message- Driven Beans, Defining Client Access with Interfaces: Remote Access, Local Access, Local Interfaces and Container-Managed Relationships, Deciding on Remote or Local Access, Web Service Clients, Method Parameters and Access, The Contents of an Enterprise Bean, Naming Conventions for Enterprise Beans, The Life Cycles of Enterprise Beans, The Life Cycle of a Stateful Session Bean, The Life Cycle of a Stateless Session Bean, The Life Cycle of a Message-Driven Bean

Building Web Services with JAX-WS: Setting the Port, Creating a Simple Web Service and Client with JAX-WS.

❖❖❖❖

1

INTRODUCTION TO SWING

Unit Structure:

1.0 Objectives
1.1 Introduction to JFC and Swing
1.2 Swing Features and Concepts
1.3 Heavy Weight Containers
1.4 Top-Level Containers
1.5 Intermediate Swing Containers
1.6 Internal Frames
1.7 Summary
1.8 Unit end exercise
1.9 Further Reading

1.0 OBJECTIVES

The objective of this chapter is to learn the basics of how Swing containers can be used to create good Graphical User Interfaces. Here we will start with the heavy weight containers and some very important intermediate containers.

1.1 INTRODUCTION TO JFC AND SWING

JFC is short for Java Foundation Classes, which encompass a group of features for building graphical user interfaces (GUIs) and adding rich graphics functionality and interactivity to Java applications. It is defined as containing the features shown in the table below.

Features of the Java Foundation Classes	
Feature	**Description**
Swing GUI Components	Includes everything from buttons to split panes to tables. Many components are capable of sorting, printing, and drag and drop, to name a few of the supported features.

Pluggable Look-and-Feel Support	The look and feel of Swing applications is pluggable, allowing a choice of look and feel. For example, the same program can use either the Java or the Windows look and feel. Additionally, the Java platform supports the GTK+ look and feel, which makes hundreds of existing look and feels available to Swing programs. Many more look-and-feel packages are available from various sources.
Accessibility API	Enables assistive technologies, such as screen readers and Braille displays, to get information from the user interface.
Java 2D API	Enables developers to easily incorporate high-quality 2D graphics, text, and images in applications and applets. Java 2D includes extensive APIs for generating and sending high-quality output to printing devices.
Internationalization	Allows developers to build applications that can interact with users worldwide in their own languages and cultural conventions. With the input method framework developers can build applications that accept text in languages that use thousands of different characters, such as Japanese, Chinese, or Korean.

How Are Swing Components Different from AWT Components?

The AWT components are those provided by the JDK 1.0 and 1.1 platforms. Although the Java 2 Platform still supports the AWT components. You can identify Swing components because their names start with J. The AWT button class, for example, is named Button, while the Swing button class is named JButton. Additionally, the AWT components are in the java.awt package, while the Swing components are in the javax.swing package.

The biggest difference between the AWT components and Swing components is that the Swing components are implemented with absolutely no native code. Since Swing components aren't restricted to the least common denominator -- the features that are present on every platform -- they can have more functionality than AWT components. Because the Swing components have no native code, they can be be shipped as an add-on to JDK 1.1, in addition to being part of the Java 2 Platform.

Swing lets you specify which look and feel your program's GUI uses. By contrast, AWT components always have the look and feel of the native platform.

Example: Write a progam to create a Login Screen.

```
//Step 1 – import all the required packages
import javax.swing.*;
import java.awt.*;
//Step 2 – Decide the class name & Container class
//(JFrame/JApplet) to be used
public class Login extends JFrame
{
        //Step 3 – Create all the instances required
        JTextField txtName,txtPass;
        JLabel lblName, lblPass;
        JButton cmdOk,cmdCancel;
        public Login()
        {
                //Step 4- Create objects for the declared instances
                txtName=new JTextField();
                txtPass      =new JTextField();
                lblName=new JLabel(“User Name”);
                lblPass      =new JLabel(“Password”);
                cmdOk        =new JButton(“Ok”);
                cmdCancel=new JButton(“Cancel”);
                //Step 5 – Add all the objects in the Content Pane with
                //Layout
                Container con=getContentPane();
                con.setLayout(new FlowLayout());
                con.add(lblName);   con.add(txtName);
                con.add(lblPass);   con.add(txtPass);
                con.add(cmdOk);     con.add(cmdCancel);

        }//constructor
        //Step 6 – Event Handling. (Event handling code will come
        //here)
        public static void main(String args[])
        {
                //Step 7 – Create object of class in main method
                Login l=new Login();
                l.setSize(150,200);
                l.setVisible(true);
        }//main
}//class
```

The code in Login.java accomplishes the following tasks:

- Import all the required packages
- Decide the class name & Container class (JFrame/JApplet) to be used
- Create all the instances required
- Create objects for the declared instances
- Add all the objects in the Content Pane with Layout
- Event Handling.
- Creating object of the class in main method.

Importing required packages

The following line imports the main Swing package:

```
        import javax.swing.*;
```

Most Swing programs also need to import the two main AWT packages:

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

Decide the class name & Container class

Every program that presents a Swing GUI contains at least one top-level Swing container. For most programs, the top-level Swing containers are instances of JFrame, JDialog, or JApplet. Each JFrame object implements a single main window, and each JDialog implements a secondary window. Each JApplet object implements an applet's display area within a browser window. A top-level Swing container provides the support that Swing components need to perform their painting and event handling.

Create all the instances required

In the above example, first instances are created so that they can be accessed from anywhere in the program and then the objects are created inside the constructor.

Adding Components to Containers

Every Container has a default layout manger that places the components inside the container according to the available size of the conent pane.

In swing we cannot add a component directly to the heavy weight, we need to get the object of the content pane and add all the components to the content pane. We use the method getContentPane() of the heavy container to get a Container object. We then add all the components to the Container object.

Creating object of the class in main method

If the heavy weight continer is JFrame the we need to write the main(). The main() will include the object of the class. Two important properties we need to set is the size and visibility. The methods used are setSize() and setVisible().

1.2 SWING FEATURES AND CONCEPTS

- **Swing Components and the Containment Hierarchy** - Swing provides many standard GUI components such as buttons, lists, menus, and text areas, which you combine to create your program's GUI. It also includes containers such as windows and tool bars.
- **Layout Management** - Layout management is the process of determining the size and position of components. By default, each container has a layout manager -- an object that performs layout management for the components within the container. Components can provide size and alignment hints to layout managers, but layout managers have the final say on the size and position of those components.
- **Event Handling** - Event handling is how programs respond to external events, such as the user pressing a mouse button. Swing programs perform all their painting and event handling in the event-dispatching thread. Every time the user types a character or pushes a mouse button, an event occurs. Any object can be notified of the event. All it has to do is implement the appropriate interface and be registered as an event listener on the appropriate event source. Swing components can generate many kinds of events. Here are a few examples:

Act that results in the event	**Listener type**
User clicks a button, presses Return while typing in a text field, or chooses a menu item	ActionListener
User closes a frame (main window)	WindowListener
User presses a mouse button while the cursor is over a component	MouseListener
User moves the mouse over a component	MouseMotionListener
Component becomes visible	ComponentListener
Component gets the keyboard focus	FocusListener
Table or list selection changes	ListSelectionListener

- Each event is represented by an object that gives information about the event and identifies the event source. Event sources are typically components, but other kinds of objects can also be event sources. Each event source can have multiple listeners registered on it. Conversely, a single listener can register with multiple event sources.
- **Painting** - Painting means drawing the component on-screen. Although it's easy to customize a component's painting, most programs don't do anything more complicated than customizing a component's border.
- **Threads and Swing** - If you do something to a visible component that might depend on or affect its state, then you need to do it from the event-dispatching thread. This isn't an issue for many simple programs, which generally refer to components only in event-handling code.
- **More Swing Features and Concepts** - Swing offers many features, many of which rely on support provided by the JComponent class. Some of the interesting features include support for icons, actions, Pluggable Look & Feel technology, assistive technologies, and separate models.

1.3 SWING API COMPONENTS – HEAVY WEIGHT CONTAINERS

1.3.1 JFrames

A frame, implemented as an instance of the JFrame class, is a window that has decorations such as a border, a title, and buttons for closing and iconifying the window. Applications with a GUI typically use at least one frame. By default, when the user closes a frame onscreen, the frame is hidden. Although invisible, the frame still exists and the program can make it visible again. If you want different behavior, then you need to either register a window listener that handles window-closing events, or you need to specify default close behavior using the setDefaultCloseOperation method. You can even do both.

The argument to setDefaultCloseOperation must be one of the following values, which are defined in the WindowConstants interface:

- DO_NOTHING_ON_CLOSE -- Don't do anything when the user's requests that the frame close. Instead, the program should probably use a window listener that performs some other action in its windowClosing method.
- HIDE_ON_CLOSE (the default) -- Hide the frame when the user closes it. This removes the frame from the screen.
- DISPOSE_ON_CLOSE -- Hide and dispose of the frame when the user closes it. This removes the frame from the screen and frees up any resources used by it.

Constructors:

JFrame()
JFrame(String)
	Create a frame that is initially invisible. Call setVisible(true) on the frame to make it visible. The String argument provides a title for the frame. You can also use setTitle to set a frame's title.

Methods:

1	Container getContentPane() - Returns the contentPane object for this frame.
2	JMenuBar getJMenuBar() - Returns the menubar set on this frame.
3	void setDefaultCloseOperation(int operation) - Sets the operation that will happen by default when the user initiates a "close" on this frame.
4	void setJMenuBar(JMenuBar menubar) - Sets the menubar for this frame.
5	void setVisible(boolean b) - Shows or hides this component depending on the value of parameter b.
6	void setLocation(int x,int y) - Moves this component to a new location. The top-left corner of the new location is specified by the x and y parameters in the coordinate space of this component's parent.

7	void pack() - Causes this Window to be sized to fit the preferred size and layouts of its subcomponents. If the window and/or its owner are not yet displayable, both are made displayable before calculating the preferred size. The Window will be validated after the preferredSize is calculated.
8	void setTitle(String title) - Sets the title for this frame to the specified string.

1.3.2 JDialog

The JDialog is the main class for creating a dialog window. You can use this class to create a custom dialog, or invoke the many class methods in JOptionPane to create a variety of standard dialogs. Every dialog is dependent on a frame. When that frame is destroyed, so are its dependent dialogs. When the frame is iconified, its dependent dialogs disappear from the screen. When the frame is deiconified, its dependent dialogs return to the screen.

A dialog can be modal. When a modal dialog is visible, it blocks user input to all other windows in the program. The dialogs that JOptionPane provides are modal. To create a non-modal dialog, you must use the JDialog class directly. To create simple, standard dialogs, you use the JOptionPane class. The ProgressMonitor class can put up a dialog that shows the progress of an operation. Two other classes, JColorChooser and JFileChooser, also supply standard dialogs.

Constructors :

1	JDialog(Frame owner)	Creates a non-modal dialog without a title with the specifed Frame as its owner.
2	JDialog(Frame owner, String title, boolean modal)	Creates a modal or non-modal dialog with the specified title and the specified owner Frame.

Methods:

1	protected void dialogInit() - Called by the constructors to init the JDialog properly.
2	Container getContentPane() - Returns the contentPane object for this dialog.
3	void setDefaultCloseOperation(int operation) - Sets the operation which will happen by default when the user initiates a "close" on this dialog.

4	void setLayout(LayoutManager manager) - By default the layout of this component may not be set, the layout of its contentPane should be set instead.

1.3.3 JApplet

JApplet is an extended version of java.applet.Applet that adds support for the JFC/Swing component architecture. The JApplet class is slightly incompatible with java.applet.Applet. JApplet contains a JRootPane as it's only child. The contentPane should be the parent of any children of the JApplet.

To add the child to the JApplet's contentPane we use the getContentPane() method and add the components to the contentPane. The same is true for setting LayoutManagers, removing components, listing children, etc. All these methods should normally be sent to the contentPane() instead of the JApplet itself. The contentPane() will always be non-null. Attempting to set it to null will cause the JApplet to throw an exception. The default contentPane() will have a BorderLayout manager set on it.

JApplet adds two major features to the functionality that it inherits from java.applet.Applet. First, Swing applets provide support for assistive technologies. Second, because JApplet is a top-level Swing container, each Swing applet has a root pane. The most noticeable results of the root pane's presence are support for adding a menu bar and the need to use a content pane.

Constructors :

JApplet() - Creates a swing applet instance.

Methods:

1	Container getContentPane() - Returns the contentPane object for this applet.
2	void setJMenuBar(JMenuBar menuBar) - Sets the menubar for this applet.
3	void setLayout(LayoutManager manager) - By default the layout of this component may not be set, the layout of its contentPane should be set instead.
4	void update(Graphics g) - Just calls paint(g).

Example: Define a class that enables the drawing of freehand lines on a screen through mouse clicking and dragging. The drawing should be cleared when a key is pressed and the line color should be selectable.

```
import java.awt.event.*;
import java.awt.*;
import javax.swing.*;
import java.applet.JApplet;
/*   <applet   code="FreeHand.class"   width=500   height=500   >
</applet> */
public class FreeHand extends JApplet
{
        int lastx,lasty,newx,newy;
        JButton b1=new JButton("Color Chooser");
        JColorChooser c1= new JColorChooser();
        Graphics g;
        Color ss;
        public void init()
        {
                FreeHandListener fhl=new FreeHandListener(this);
                g=getGraphics();
                JPanel jp=(JPanel)getContentPane();
                jp.setLayout(new FlowLayout());
                b1.addActionListener(fhl);
                jp.add(b1);
                addMouseListener(fhl);
                addMouseMotionListener(fhl);
                addKeyListener(fhl);
        }
}// Class FH
class FreeHandListener implements
ActionListener,MouseMotionListener,MouseListener,KeyListener
{
        FreeHand fh;
        public FreeHandListener(FreeHand fh)
        {
                this.fh=fh;
        }
        public void actionPerformed(ActionEvent e)
        {
                JDialog jd=JColorChooser.createDialog(fh,"Choose
                Color",true,fh.c1,new SetColor(fh),null);
                jd.setVisible(true);
        }
        public class SetColor implements ActionListener
        {
        FreeHand fh;
        public SetColor(FreeHand fh)
        {
                this.fh=fh;
        }
        public void actionPerformed(ActionEvent e)
        {
                fh.ss=fh.c1.getColor();
        }
        }// inner class
        public void mousePressed(MouseEvent e)
        {
                fh.lastx=e.getX();
                fh.lasty=e.getY();
        }
        public void mouseDragged(MouseEvent e)
        {
                fh.g.setColor(fh.ss);
                fh.newx=e.getX();
                fh.newy=e.getY();
                fh.g.drawLine(fh.lastx,fh.lasty,fh.newx,fh.newy);
                fh.lastx=fh.newx;
                fh.lasty=fh.newy;
        }
        public void keyPressed(KeyEvent e)
        {
                if(e.getKeyCode()==KeyEvent.VK_C)
                fh.repaint();
        }
        public void keyTyped(KeyEvent e){}
        public void keyReleased(KeyEvent e){}
        public void mouseReleased(MouseEvent e){}
        public void mouseExited(MouseEvent e) {}
        public void mouseClicked(MouseEvent e) {}
        public void mouseMoved(MouseEvent e) {}
        public void mouseEntered(MouseEvent e) {}
}//class fhl
```

Example: Create a class called “ColouredCanvas” which extends Canvas and whose constructor takes three arguments, its color, width and height. When a “ColouredCanvas” is initialized, it should set its size and background color as per the arguments. Create a class which extents JApplet and adds to the Applet a “ColouredCanvas” of red color with size 50,100.

```
import javax.swing.JApplet;
import java.awt.Container;
import java.awt.GridBagLayout;
import java.awt.GridBagConstraints;
import java.awt.Color;
import java.awt.Canvas;
import java.awt.event.*;
/* <applet code=MyColoredCanvas.class height=500 width=500 >
</applet> */

class ColoredCanvas extends Canvas
{
        public ColoredCanvas(Color c,int w,int h)
        {
                setSize(w,h);
                setBackground(c);
        }
}
public class MyColoredCanvas extends JApplet
{
        public void init()
        {
                ColoredCanvas cc=new ColoredCanvas
                                        (new Color(1,1,250),100,50);

                Container con=getContentPane();
                con.setLayout(new GridBagLayout());

                GridBagConstraints gbc=new GridBagConstraints();

                gbc.gridx=2;
                gbc.gridy=2;
                con.add(cc,gbc);
        }
}
```

1.3.4 JWindow

A JWindow is a container that can be displayed anywhere on the user's desktop. It does not have the title bar, window-management buttons, or other trimmings associated with a JFrame, but it is still a "first-class citizen" of the user's desktop, and can exist anywhere on it. The JWindow component contains a JRootPane as its only child. The contentPane should be the parent of any children of the JWindow. From the older java.awt.Window object you would normally do something like this:

```
window.add(child);
```

However, using JWindow you would code:

```
window.getContentPane().add(child);
```

The same is true of setting LayoutManagers, removing components, listing children, etc. All these methods should normally be sent to the contentPane instead of the JWindow itself. The contentPane will always be non-null. Attempting to set it to null will cause the JWindow to throw an exception. The default contentPane will have a BorderLayout manager set on it.

Constructors:

1	JWindow() - Creates a window with no specified owner.
2	JWindow(Frame owner) - Creates a window with the specified owner frame.

Methods :

1	Conatiner getContentPane() - Returns the contentPane object for this applet.
2	void setLayout(LayoutManager manager) - By default the layout of this component may not be set, the layout of its contentPane should be set instead.
3	void update(Graphics g) - Just calls paint(g).
4	void windowInit() - Called by the constructors to init the JWindow properly.

1.4 SWING API COMPONENTS - TOP-LEVEL CONTAINERS

Swing provides three generally useful top-level container classes: JFrame JDialog, and JApplet. To appear onscreen, every GUI component must be part of a containment hierarchy. Each containment hierarchy has a top-level container as its root. Each top-level container has a content pane that, generally speaking, contains the visible components in that top-level container's GUI. You can optionally add a menu bar to a top-level container. The menu bar is positioned within the top-level container, but outside the content pane.

Top-Level Containers and Containment Hierarchies

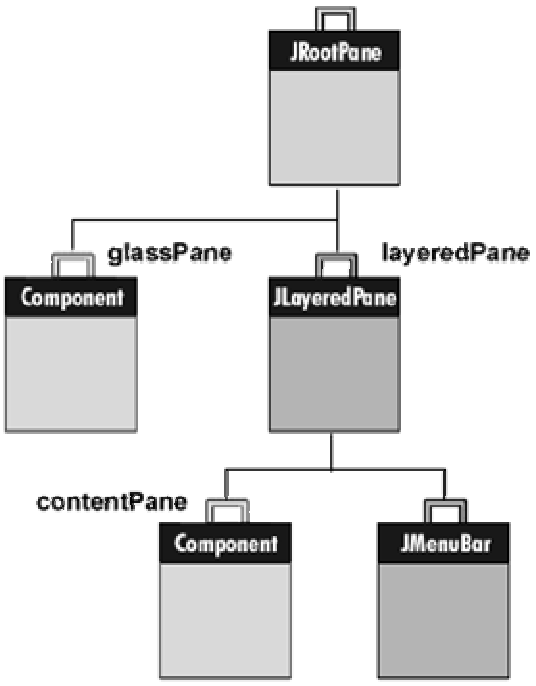
Each program that uses Swing components has at least one top-level container. This top-level container is the root of a containment hierarchy -- the hierarchy that contains all of the Swing components that appear inside the top-level container. As a rule, a standalone application with a Swing-based GUI has at least one containment hierarchy with a JFrame as its root.

Adding Components to the Content Pane

Here's the code that is used to get a frame's content pane and add the yellow label to it:

```
        frame.getContentPane().add(yellowLabel,
BorderLayout.CENTER);
```

As the code shows, you find the content pane of a top-level container by calling the getContentPane method. The default content pane is a simple intermediate container that inherits from JComponent, and that uses a BorderLayout as its layout manager. It's easy to customize the content pane -- setting the layout manager or adding a border, for example. The getContentPane method returns a Container object, not a JComponent object.



Adding a Menu Bar

All top-level containers can, in theory, have a menu bar. In practice, however, menu bars usually appear only in frames and perhaps in applets. To add a menu bar to a frame or applet, you create a JMenuBar object, populate it with menus, and then call setJMenuBar. To adds a menu bar to its frame use this code:

```
frame.setJMenuBar(MenuBar_Name);
```

The Root Pane

Each top-level container relies on a reclusive intermediate container called the root pane. The root pane manages the content pane and the menu bar, along with a couple of other containers. If you need to intercept mouse clicks or paint over multiple components, you should get acquainted with root panes.

We've already discussed about the content pane and the optional menu bar. The two other components that a root pane adds are a layered pane and a glass pane. The layered pane directly contains the menu bar and content pane, and enables Z-ordering of other components you might add. The glass pane is often used to intercept input events occuring over the top-level container, and can also be used to paint over multiple components.

JRootPane's layeredPane

The layeredPane is the parent of all children in the JRootPane. It is an instance of JLayeredPane, which provides the ability to add components at several layers. This capability is very useful when working with popup menus, dialog boxes, and dragging -- situations in which you need to place a component on top of all other components in the pane.

JRootPane's glassPane

The glassPane sits on top of all other components in the JRootPane. This positioning makes it possible to intercept mouse events, which is useful for dragging one component across another. This positioning is also useful for drawing.

1.5 SWING API COMPONENTS - INTERMEDIATE SWING CONTAINERS

1.5.1 JPanel

JPanel is a generic lightweight container. JPanel is the most flexible, frequently used intermediate container. Implemented with the JPanel class, panels add almost no functionality beyond what all JComponent objects have. They are often used to group components, whether because the components are related or just because grouping them makes layout easier. A panel can use any layout manager, and you can easily give it a border. The content panes of top-level containers are often implemented as JPanel instances.

When you add components to a panel, you use the add method. Exactly which arguments you specify to the add method depend on which layout manager the panel uses. When the layout manager is FlowLayout, BoxLayout, GridLayout, or GridBagLayout, you'll typically use the one-argument add method, like this:

```
aFlowPanel.add(aComponent);
aFlowPanel.add(anotherComponent);
```

When the layout manager is BorderLayout, you need to provide a second argument specifying the added component's position within the panel. For example:

```
aBorderPanel.add(aComponent,
BorderLayout.CENTER);
aBorderPanel.add(anotherComponent,
BorderLayout.SOUTH);
```

Constructors :

Constructor	Purpose

JPanel()
JPanel(LayoutManager)
	Create a panel. The LayoutManager parameter provides a layout manager for the new panel. By default, a panel uses a FlowLayout to lay out its components.

Methods :

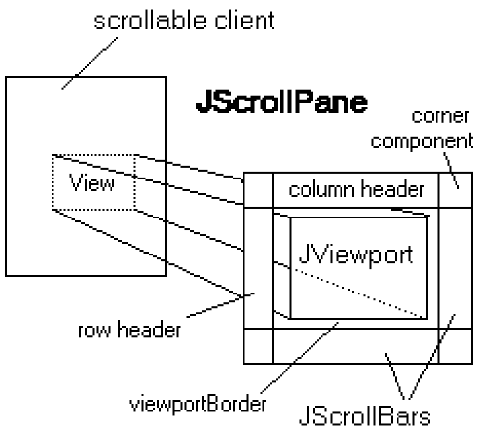
Method	Purpose
void add(Component) void add(Component, int) void add(Component, Object) void add(Component, Object, int)	Add the specified component to the panel. When present, the int parameter is the index of the component within the container. By default, the first component added is at index 0, the second is at index 1, and so on. The Object parameter is layout manager dependent and typically provides information to the layout manager regarding positioning and other layout constraints for the added component.

void remove(Component) void remove(int) void removeAll()	Remove the specified component(s).
void setLayout(LayoutManager) LayoutManager getLayout()	Set or get the layout manager for this panel. The layout manager is responsible for positioning the panel's components within the panel's bounds according to some philosophy.

1.5.2 JSrollPane

JScrollPane provides a scrollable view of a component. A JScrollPane manages a viewport, optional vertical and horizontal scroll bars, and optional row and column heading viewports.

The JViewport provides a window, or "viewport" onto a data source -- for example, a text file. That data source is the "scrollable client" displayed by the JViewport view. A JScrollPane basically consists of JScrollBars, a JViewport, and the wiring between them, as shown in the diagram at right.



In addition to the scroll bars and viewport, a JScrollPane can have a column header and a row header. Each of these is a JViewport object that you specify with setRowHeaderView, and setColumnHeaderView.

To add a border around the main viewport, you can use setViewportBorder. A common operation to want to do is to set the background color that will be used if the main viewport view is smaller than the viewport. This can be accomplished by setting the background color of the viewport, via
scrollPane.getViewport().setBackground().

Constructors :

Constructor	Purpose

JScrollPane()
JScrollPane(Component)
JScrollPane(int, int)
JScrollPane(Component, int, int)
	Create a scroll pane. The Component parameter, when present, sets the scroll pane's client. The two int parameters, when present, set the vertical and horizontal scroll bar policies (respectively).

Methods:

Method	Purpose
void setVerticalScrollBarPolicy(int) int getVerticalScrollBarPolicy() SAME FOR HORIZONTAL	Set or get the vertical scroll policy. ScrollPaneConstants defines three values for specifying this policy: VERTICAL_SCROLLBAR_AS_NEEDED (the default), VERTICAL_SCROLLBAR_ALWAYS, and VERTICAL_SCROLLBAR_NEVER.
void setViewportBorder(Border) Border getViewportBorder()	Set or get the border around the viewport.
void setColumnHeaderView(Component) void setRowHeaderView(Component)	t the column or row ader for the scroll ne.
void setCorner(Component, int) Component getCorner(int)	Set or get the corner specified. The int parameter specifies which corner and must be one of the following constants defined in ScrollPaneConstants: UPPER_LEFT_CORNER, UPPER_RIGHT_CORNER, LOWER_LEFT_CORNER, and LOWER_RIGHT_CORNER.

1.5.3 Tabbed Panes

A component that lets the user switch between a group of components by clicking on a tab with a given title and/or icon. Tabs/components are added to a TabbedPane object by using the addTab and insertTab methods. A tab is represented by an index corresponding to the position it was added in, where the first tab has an index equal to 0 and the last tab has an index equal to the tab count minus 1.

The TabbedPane uses a SingleSelectionModel to represent the set of tab indices and the currently selected index. If the tab count is greater than 0, then there will always be a selected index, which by default will be initialized to the first tab. If the tab count is 0, then the selected index will be -1.

Constructors:

Constructor	Purpose

JTabbedPane()
JTabbedPane(int tabPlacement)
JTabbedPane(int tabPlacement, int tabLayoutPolicy)
	Creates a tabbed pane. The first optional argument specifies where the tabs should appear. By default, the tabs appear at the top of the tabbed pane.You can specify these positions TOP, BOTTOM, LEFT, RIGHT. The second optional argument specifies the tab layout policy.

Methods:

Method	Purpose

JTabbedPane()
JTabbedPane(int)
	Create a tabbed pane. The optional argument specifies where the tabs should appear. By default, the tabs appear at the top of the tabbed pane. You can specify these positions (defined in the SwingConstants interface, which JTabbedPane implements): TOP, BOTTOM, LEFT, RIGHT.

addTab(String, Icon, Component, String)
addTab(String, Icon, Component)
addTab(String, Component)
	Add a new tab to the tabbed pane. The first argument specifies the text on the tab. The optional icon argument specifies the tab's icon. The component argument specifies the component that the tabbed pane should show when the tab is selected. The fourth argument, if present, specifies the tool tip text for the tab.

insertTab(String, Icon, Component, String, int)	Insert a tab at the specified index, where the first tab is at index 0. The arguments are the same as for addTab.
void set SelectedIndex (int) void set Selected Component (Component)	Select the tab that has the specified component or index. Selecting a tab has the effect of displaying its associated component.
void set EnabledAt (int, boolean) boolean is EnabledAt (int)	Set or get the enabled state of the tab at the specified index.

Example: Demo example to use JTabbedPane. Create a tabbed pane and add three tabs using JPanel class.

```
import javax.swing.*;
/*<applet code="TabbedPaneDemo.class" height=500
width=500></applet> */
public class TabbedPaneDemo extends JApplet {

        public void init() {
                JTabbedPane jtp=new JTabbedPane();
                jtp.addTab("Cities",new CitiesPanel());
                jtp.addTab("Color",new ColorPanel());
                jtp.addTab("Flavour",new FlavourPanel());
                getContentPane().add(jtp);
        }
}
class CitiesPanel extends JPanel {

        public CitiesPanel() {
                add(new JButton("Mumbai"));
                add(new JButton("Delhi"));
                add(new JButton("Banglore"));
                add(new JButton("Chennai"));
        }
}
class ColorPanel extends JPanel {

        public ColorPanel() {
                add(new JCheckBox("Red"));
                add(new JCheckBox("Yellow"));
                add(new JCheckBox("Green"));
                add(new JCheckBox("Blue"));
        }
}
class FlavourPanel extends JPanel {

        public FlavourPanel() {
                String item[]={"Vanila","Stroberry","Chocolet"};
                JComboBox jcb=new JComboBox(item);
                add(jcb);
        }
}
```

1.6 SWING API COMPONENTS - INTERNAL FRAMES

A lightweight object that provides many of the features of a native frame, including dragging, closing, becoming an icon, resizing, title display, and support for a menu bar.

Rules of Using Internal Frames

- You must set the size of the internal frame - If you don't set the size of the internal frame, it will have zero size and thus never be visible. You can set the size using one of the following methods: setSize, pack, or setBounds.
- As a rule, you should set the location of the internal frame - If you don't set the location of the internal frame, it will come up at 0,0 (the upper left of its container). You can use the setLocation or setBounds method to specify the upper left point of the internal frame, relative to its container.
- To add components to an internal frame, you add them to the internal frame's content pane.
- You must add an internal frame to a container - If you don't add the internal frame to a container (usually a JDesktopPane), the internal frame won't appear.
- You need to call show or setVisible on internal frames.
- Internal frames fire internal frame events, not window events.

Constructors:

Constructor Summary
JInternalFrame() - Creates a non-resizable, non-closable, non-maximizable, non-iconifiable JInternalFrame with no title.
JInternalFrame(String title, boolean resizable, boolean closable) - Creates a non-maximizable, non-iconifiable JInternalFrame with the specified title, resizability, and closability.

Methods:

Method	Purpose
void setVisible(boolean)	Make the internal frame visible (if true) or invisible (if false). You should invoke setVisible(true) on each JInternalFrame before adding it to its container. (Inherited from Component).

void pack ()	Size the internal frame so that its components are at their preferred sizes.
void setLocation(Point) void setLocation(int, int)	Set the position of the internal frame. (Inherited from Component).
void setBounds(Rectangle) void setBounds(int, int, int, int)	Explicitly set the size and location of the internal frame. (Inherited from Component).
void setSize(Dimension) void setSize(int, int)	Explicitly set the size of the internal frame. (Inherited from Component).
void set Closed (boolean) boolean is Closed()	Set or get whether the internal frame is currently closed. The argument to setClosed must be true. When reopening a closed internal frame, you make it visible and add it to a container (usually the desktop pane you originally added it to).

Example: Demo example to use internal frames. Create three internal frames and add them to the main frame.

```
//First internal frame
import javax.swing.*;
import java.awt.Dimension;
public class CitiesPanel extends JInternalFrame
{
        public CitiesPanel()
        {
                super("Select Cities",true,true);
                JPanel jp=new JPanel();
                jp.add(new JButton("Mumbai"));
                jp.add(new JButton("Pune"));
                jp.add(new JButton("Kolkata"));
                getContentPane().add(jp);
                setPreferredSize(new Dimension(300,300));
                setDefaultCloseOperation(HIDE_ON_CLOSE);
        }
}
//Second internal frame
import javax.swing.*;
import java.awt.Dimension;
public class ColorPanel extends JInternalFrame
```

```
{
        public ColorPanel()
        {
                super("Select Colors",true,true);
                JPanel jp=new JPanel();
                jp.add(new JButton("Red"));
                jp.add(new JButton("Blue"));
                jp.add(new JButton("Green"));
                getContentPane().add(jp);
                setPreferredSize(new Dimension(300,300));
                setDefaultCloseOperation(HIDE_ON_CLOSE);
        }
}
//Third internal frame
import javax.swing.*;
import java.awt.Dimension;
public class FlavourPanel extends JInternalFrame
{
        public FlavourPanel()
        {
                super("Select Flavours",true,true);
                JPanel jp=new JPanel();
                jp.add(new JButton("Vanilla"));
                jp.add(new JButton("Chocolate"));
                jp.add(new JButton("Strawberry"));
                getContentPane().add(jp);
                setPreferredSize(new Dimension(300,300));
                setDefaultCloseOperation(HIDE_ON_CLOSE);
        }
}
//Main Frame
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class IFrameDemo extends JFrame implements
ActionListener
{
        CitiesPanel c1=new CitiesPanel();
        ColorPanel c2=new ColorPanel();
        FlavourPanel c3=new FlavourPanel();
        public IFrameDemo()
        {
                JMenuBar mb=new JMenuBar();
                JMenu select=new JMenu("Select");
                JMenuItem city=new JMenuItem("City");
                JMenuItem color=new JMenuItem("Color");
                JMenuItem flavour=new JMenuItem("Flavour");
                select.add(city);
                select.add(color);
                select.add(flavour);
                mb.add(select);
                setJMenuBar(mb);
                city.addActionListener(this);
                color.addActionListener(this);
                flavour.addActionListener(this);
                JDesktopPane dp=new JDesktopPane();
                dp.setLayout(new FlowLayout());
                dp.add(c1);
                dp.add(c2);
                dp.add(c3);
                getContentPane().add(dp,BorderLayout.CENTER);
        }
public void actionPerformed(ActionEvent e)
{
                String args=e.getActionCommand();
                if(args.equals("City"))
                {
                        c1.setVisible(true);
                        c2.setVisible(false);
                        c3.setVisible(false);
                }
                else if(args.equals("Color"))
                {
                        c1.setVisible(false);
                        c2.setVisible(true);
                        c3.setVisible(false);
                }
                else if(args.equals("Flavour"))
                {
                        c1.setVisible(false);
                        c2.setVisible(false);
```

```
                    c3.setVisible(true);
            }
}
public static void main(String args[])
{
        IFrameDemo f1=new IFrameDemo();
        f1.setVisible(true);
        f1.setSize(500,500);
        f1.setTitle("Internal Frame Demo");
        f1.setDefaultCloseOperation(EXIT_ON_CLOSE);
}
}
```

1.7 SUMMARY

- Swing provides many standard GUI components such as buttons, lists, menus, and text areas, which you combine to create your program's GUI.
- A frame, implemented as an instance of the JFrame class, is a window that has decorations such as a border, a title, and buttons for closing and iconifying the window.
- JApplet is an extended version of java.applet.Applet that adds support for the JFC/Swing component architecture.
- JPanel is a generic lightweight container. JPanel is the most flexible, frequently used intermediate container.
- Tabs/components are added to a TabbedPane object by using the addTab and insertTab methods.
- JInternalFrame is used to create a MDI Form with the help of other Swing Containers

1.8 UNIT END EXERCISE

1) Explain different types of panes used in swing?
2) What is the purpose of using the getContentPane() in swing program ?
3) Write a short note on JFC?
4) How Are Swing Components Different from AWT Components?
5) Explain any 5 Swing features.
6) Write a short note on JFrame?

7) Explain with an example JScrollPane.

8) How can we use Internal Frames?

9) Write a program to create a JFrame containing JDesktopPane which has a single internal frame ?

10) What is JTabbed Pane ? Explain with an example.

1.9 FURTHER READING

- Joe Wigglesworth and Paula McMillan, Java Programming: Advanced Topics, Thomson Course Technology (SPD), Third Edition, 2004
- Cay S. Horstmann, Gary Cornell, Core Java™ 2: Volume II–Advanced Features Prentice Hall PTR, 2001

❖❖❖❖

2

BASIC SWING COMPONENTS

Unit Structure:

2.0 Objectives
2.1 The JComponent Class & Swing Components
2.2 Interface Action
2.3 Printing with 2D API
2.4 Java Print Service API
2.5 Summary
2.6 Unit end exercise
2.7 Further Reading

2.0 OBJECTIVES

The objective of this chapter is to lear how to use the Swing Components to create a user interface. We will start the chapter with a few components and then lear how to incorporate the Java Print API.

2.1 THE JCOMPONENT CLASS

With the exception of top-level containers, all Swing components whose names begin with "J" descend from the JComponent class. For example, JLabel, JButton, JTree, and JTable all inherit from JComponent. However, JFrame doesn't because it implements a top-level container.

The JComponent class extends the Container class, which itself extends Component. The Component class includes everything from providing layout hints to supporting painting and events. The Container class has support for adding components to the container and laying them out.

JComponent Features

The JComponent class provides the following functionality to its descendants:

- **Tool tips** - By specifying a string with the setToolTipText method, you can provide help to users of a component. When the cursor pauses over the component, the specified string is displayed in a small window that appears near the component.
- **Borders** - The setBorder method allows you to specify the border that a component displays around its edges.
- **Keyboard-generated actions** - Using the registerKeyboardAction method, you can enable the user to use the keyboard, instead of the mouse, to operate the GUI. The combination of character and modifier keys that the user must press to start an action is represented by a KeyStroke object. The resulting action event must be handled by an action listener. Each keyboard action works under exactly one of three conditions: only when the actual component has the focus, only when the component or one of its containers has the focus, or any time that anything in the component's window has the focus.
- **Application-wide pluggable look and feel** - Behind the scenes, each JComponent object has a corresponding ComponentUI object that performs all the drawing, event handling, size determination, and so on for that JComponent. Exactly which ComponentUI object is used depends on the current look and feel, which you can set using the UIManager.setLookAndFeel method.
- **Support for layout** - To give you a way to set layout hints, the JComponent class adds setter methods -- setPreferredSize, setMinimumSize, setMaximumSize, setAlignmentX, and setAlignmentY.
- **Double buffering** - Double buffering smooths on-screen painting.
- **Methods to increase efficiency** - JComponent has a few methods that provide more efficient ways to get information than the JDK 1.1 API allowed. The methods include getX and getY, which you can use instead of getLocation; and getWidth and getHeight, which you can use instead of getSize. It also adds one-argument forms of getBounds, getLocation, and getSize for which you specify the object to be modified and returned, letting you avoid unnecessary object creation. These methods have been added to Component for Java 2 (JDK 1.2).

2.1.2 JLabel

A label object is a single line of read only text. A common use of JLabel objects is to position descriptive text above or besides other components. JLabel extends the JComponent class. It can display text and/or icon.

Method or Constructor	Purpose

JLabel(Icon)
JLabel(Icon, int)
JLabel(String)
JLabel(String, Icon, int)
JLabel(String, int)
JLabel()
	Creates a JLabel instance, initializing it to have the specified text/image/alignment. The int argument specifies the horizontal alignment of the label's contents within its drawing area. The horizontal alignment must be one of the following constants defined in the SwingConstants interface (which JLabel implements): LEFT, CENTER, RIGHT, LEADING, or TRAILING. For ease of localization, we strongly recommend using LEADING and TRAILING, rather than LEFT and RIGHT.

void setText(String) String getText()	Sets or gets the text displayed by the label.
void setIcon(Icon) Icon getIcon()	Sets or gets the image displayed by the label.
void setDisplayedMnemonicIndex(int) int getDisplayedMnemonicIndex()	Sets or gets a hint as to which character in the text should be decorated to represent the mnemonic. This is useful when you have two instances of the same character and wish to decorate the second instance. For example, setDisplayedMnemonicIndex(5) decorates the character that is at position 5 (that is, the 6th character in the text). Not all types of look and feel may support this feature.
void setDisabledIcon(Icon) Icon getDisabledIcon()	Sets or gets the image displayed by the label when it is disabled. If you do not specify a disabled image, then the look and feel creates one by manipulating the default image.

2.1.3 JTextField

A text field is a basic text control that enables the user to type a small amount of text. When the user indicates that text entry is complete (usually by pressing Enter), the text field fires an action event. If you need to obtain more than one line of input from the user, use a text area. The horizontal alignment of JTextField can be set to be left justified, leading justified, centered, right justified or trailing justified. Right/trailing justification is useful if the required size of the field text is smaller than the size allocated to it. This is determined by the setHorizontalAlignment and getHorizontalAlignment methods. The default is to be leading justified.

Method or Constructor	Purpose

JTextField()
JTextField(String)
JTextField(String, int)
JTextField(int)
	Creates a text field. When present, the int argument specifies the desired width in columns. The String argument contains the field's initial text.

void setText(String) String getText()	Sets or obtains the text displayed by the text field.
void setEditable(boolean) boolean isEditable()	Sets or indicates whether the user can edit the text in the text field.
void setColumns(int) int getColumns()	Sets or obtains the number of columns displayed by the text field. This is really just a hint for computing the field's preferred width.
void setHorizontalAlignment(int) int getHorizontalAlignment()	Sets or obtains how the text is aligned horizontally within its area. You can use JTextField.LEADING, JTextField.CENTER, and JTextField.TRAILING for arguments.

2.1.4 JButton

A JButton class provides the functionality of a push button. JButton allows an icon, a string or both to be associated with the push button. JButton is a subclass of AbstractButton which extends JComponent.

Method or Constructor	Purpose
JButton(Action)	Create a JButton instance, initializing it to

JButton(String, Icon)
JButton(String)
JButton(Icon)
JButton()
	have the specified text/image/action.

void setAction(Action) Action getAction()	Set or get the button's properties according to values from the Action instance.
void setText(String) String getText()	Set or get the text displayed by the button.
void setIcon(Icon) Icon getIcon()	Set or get the image displayed by the button when the button isn't selected or pressed.
void setDisabledIcon(Icon) Icon getDisabledIcon()	Set or get the image displayed by the button when it is disabled. If you do not specify a disabled image, then the look and feel creates one by manipulating the default image.
void setPressedIcon(Icon) Icon getPressedIcon()	Set or get the image displayed by the button when it is being pressed.

Example: Write a program to create a user interface for students biodata. (Demo example for JLabel, JTextfield and JButton).

```
//Step 1 – import all the required packages
import javax.swing.*;
import java.awt.*;
/* Step 2 – Decide the class name & Container class
(JFrame/JApplet) to be used */
public class StudentBioData01 extends JFrame
{
//Step 3 – Create all the instances required
        JTextField txtName,txtMobNo;
        JLabel lblName, lblMobNo;
        JButton cmdOk,cmdCancel;
        public StudentBioData01(){
                //Step 4- Create objects for the declared instances
                txtName=new JTextField(20);
                txtMobNo=new JTextField(20);
                lblName=new JLabel("Student Name");
                lblMobNo=new JLabel("Mobile No.");
                cmdOk       =new JButton("Ok");
                cmdCancel=new JButton("Cancel");
//Step 5 – Add all the objects in the Content Pane with Layout
                Container con=getContentPane();
                con.setLayout(new FlowLayout());
                con.add(lblName);   con.add(txtName);
                con.add(lblMobNo); con.add(txtMobNo);
                con.add(cmdOk);             con.add(cmdCancel);
        }//constructor
//Step 6 – Event Handling. (Event handling code will come here)
public static void main(String args[])
{
        //Step 7 – Create object of class in main method
        StudentBioData01 sbd=new StudentBioData01();
        sbd.setSize(150,200);
        sbd.setVisible(true);
}//main
}//class
```

2.1.5 JCheckBox

A JCheckBox class provides the functionality of a Check box. Its immediate super class is JToggleButton which provides support for 2 state buttons.

Constructor	Purpose

JCheckBox(String)
JCheckBox(String, boolean)
JCheckBox(Icon)
JCheckBox(Icon, boolean)
JCheckBox(String, Icon)
JCheckBox(String, Icon, boolean)
JCheckBox()
	Create a JCheckBox instance. The string argument specifies the text, if any, that the check box should display. Similarly, the Icon argument specifies the image that should be used instead of the look and feel's default check box image. Specifying the boolean argument as true initializes the check box to be selected. If the boolean argument is absent or false, then the check box is initially unselected.

String getActionCommand()	Returns the action command for this button.

String getText()	Returns the button's text.
boolean isSelected()	Returns the state of the button.
void setEnabled(boolean b)	Enables (or disables) the button.
void setSelected(boolean b)	Sets the state of the button.
void setText(String text)	Sets the button's text.

2.1.6 JRadioButton

A JRadioButton class provides the functionality of a radio button. Its immediate super class is JToggleButton which provides support for 2 state buttons.

Constructor	Purpose

JRadioButton(String)
JRadioButton(String, boolean)
JRadioButton(Icon)
JRadioButton(Icon, boolean)
JRadioButton(String, Icon)
JRadioButton(String, Icon, boolean)
JRadioButton()
	Create a JRadioButton instance. The string argument specifies the text, if any, that the radio button should display. Similarly, the Icon argument specifies the image that should be used instead of the look and feel's default radio button image. Specifying the boolean argument as true initializes the radio button to be selected, subject to the approval of the ButtonGroup object. If the boolean argument is absent or false, then the radio button is initially unselected.

Methods same as JCheckBox	

Example: Write a program to create a user interface for students biodata. (Demo example for JCheckbox, JRadioButton).

```
//Step 1 – import all the required packages
import javax.swing.*;
import java.awt.*;
/* Step 2 – Decide the class name & Container class
(JFrame/JApplet) to be used */
public class StudentBioData02 extends JFrame
{
        //Step 3 – Create all the instances required
        JLabel lbllang,lblstream;
        JCheckBox cbeng,cbhin,cbmar;
        JRadioButton rbart,rbcomm,rbsci;
        ButtonGroup bg;
        public StudentBioData02()
        {
                //Step 4- Create objects for the declared instances
                lbllang=new JLabel("Languages Known");
                lblstream=new JLabel("Stream");
                cbeng=new JCheckBox("English",true);
                cbhin=new JCheckBox("Hindi");
                cbmar=new JCheckBox("Marathi");
                rbart=new JRadioButton("Arts");
                rbcomm=new JRadioButton("Commerce");
                rbsci=new JRadioButton("Science");
                bg=new ButtonGroup();
                bg.add(rbart); bg.add(rbcomm); bg.add(rbsci);
//Step 5 – Add all the objects in the Content Pane with Layout
                Container con=getContentPane();
                con.setLayout(new FlowLayout());
                con.add(lbllang);
                con.add(cbeng); con.add(cbhin); con.add(cbmar);
                con.add(lblstream);
                con.add(rbart);       con.add(rbcomm);con.add(rbsci);

        }//constructor
//Step 6 – Event Handling. (Event handling code will come here)
public static void main(String args[])
{
        //Step 7 – Create object of class in main method
        StudentBioData02 sbd=new StudentBioData02();
        sbd.setSize(150,200);
        sbd.setVisible(true);
}//main
}//class
```

2.1.7 JComboBox

A component that combines a button or editable field and a drop-down list. The user can select a value from the drop-down list, which appears at the user's request. If you make the combo box editable, then the combo box includes an editable field into which the user can type a value.

Constructors & Method	**Purpose**

JComboBox()
JComboBox(Object[])
JComboBox(Vector)
	Create a combo box with the specified items in its menu. A combo box created with the default constructor has no items in the menu initially. Each of the other constructors initializes the menu from its argument: a model object, an array of objects, or a Vector of objects.

void addItem(Object) void insertItemAt(Object, int)	Add or insert the specified object into the combo box's menu. The insert method places the specified object at the specified index, thus inserting it before the object currently at that index. These methods require that the combo box's data model be an instance of MutableComboBoxModel.
Object getItemAt(int) Object getSelectedItem()	Get an item from the combo box's menu.
void removeAllItems() void removeItemAt(int) void removeItem(Object)	Remove one or more items from the combo box's menu. These methods require that the combo box's data model be an instance of MutableComboBoxModel.
int getItemCount()	Get the number of items in the combo box's menu.
void addActionListener(ActionListener)	Add an action listener to the combo box. The listener's actionPerformed method is called when the user selects an item from the combo box's menu or, in an editable combo box,

	when the user presses Enter.
void addItemListener(ItemListener)	Add an item listener to the combo box. The listener's itemStateChanged method is called when the selection state of any of the combo box's items change.

2.1.8 JList

A component that allows the user to select one or more objects from a list. A separate model, ListModel, represents the contents of the list. It's easy to display an array or vector of objects, using a JList constructor that builds a ListModel instance for you.

Method or Constructor	**Purpose**

JList(Object[])
JList(Vector)
JList()
	Create a list with the initial list items specified. The second and third constructors implicitly create an immutable ListModel; you should not subsequently modify the passed-in array or Vector.

void setListData(Object[]) void setListData(Vector)	Set the items in the list. These methods implicitly create an immutable ListModel.
void setVisibleRowCount(int) int getVisibleRowCount()	Set or get the visibleRowCount property. For a VERTICAL layout orientation, this sets or gets the preferred number of rows to display without requiring scrolling. For the HORIZONTAL_WRAP or VERTICAL_WRAP layout orientations, it defines how the cells wrap. The default value of this property is VERTICAL.
void setSelectionMode(int) int getSelectionMode()	Set or get the selection mode. Acceptable values are: SINGLE_SELECTION, SINGLE_INTERVAL_SELECTION, or MULTIPLE_INTERVAL_SELECTION (the default), which are defined in ListSelectionModel.

int getAnchorSelectionIndex() int getLeadSelectionIndex() int getSelectedIndex() int getMinSelectionIndex() int getMaxSelectionIndex() int[] getSelectedIndices() Object getSelectedValue() Object[] getSelectedValues()	Get information about the current selection as indicated.

Example: Write a program to create a user interface for students biodata. (Demo example for JComboBox, JList).

```
//Step 1 – import all the required packages
import javax.swing.*;
import java.awt.*;
/* Step 2 – Decide the class name & Container class
(JFrame/JApplet) to be used */
public class StudentBioData03 extends JFrame
{
        //Step 3 – Create all the instances required
        JLabel lblplang,lblyear;
        JList lst;
        JComboBox jcb;
        public StudentBioData03()
        {
                //Step 4- Create objects for the declared instances
                lblplang=new JLabel("Programming Lang.");
                lblyear=new JLabel("Academic Year");
                Object obj[]={"C","C++","C#","Java"};
                lst=new JList(obj);
                jcb=new JComboBox();
                jcb.addItem("First Year");
                jcb.addItem("Second Year");
                jcb.addItem("Third Year");
//Step 5 – Add all the objects in the Content Pane with Layout
                Container con=getContentPane();
                con.setLayout(new FlowLayout());
                con.add(lblplang);
                con.add(lst);
                con.add(lblyear);
                con.add(jcb);
        }//constructor
//Step 6 – Event Handling. (Event handling code will come here)
public static void main(String args[])
{
        //Step 7 – Create object of class in main method
        StudentBioData03 sbd=new StudentBioData03();
        sbd.setSize(150,200);
        sbd.setVisible(true);
}//main
}//class
```

2.1.9 Menus

JMenuBar

An implementation of a menu bar. You add JMenu objects to the menu bar to construct a menu. When the user selects a JMenu object, its associated JPopupMenu is displayed, allowing the user to select one of the JMenuItems on it.

JMenu

An implementation of a menu -- a popup window containing JMenuItems that is displayed when the user selects an item on the JMenuBar. In addition to JMenuItems, a JMenu can also contain JSeparators.

In essence, a menu is a button with an associated JPopupMenu. When the "button" is pressed, the JPopupMenu appears. If the "button" is on the JMenuBar, the menu is a top-level window. If the "button" is another menu item, then the JPopupMenu is "pull-right" menu.

JMenuItem

An implementation of an item in a menu. A menu item is essentially a button sitting in a list. When the user selects the "button", the action associated with the menu item is performed. A JMenuItem contained in a JPopupMenu performs exactly that function.

Constructor or Method	**Purpose**
JMenuBar()	Creates a menu bar.

JMenu()
JMenu(String)
	Creates a menu. The string specifies the text to display for

	the menu.

JMenuItem()
JMenuItem(String)
JMenuItem(Icon)
JMenuItem(String, Icon)
JMenuItem(String, int)
	Creates an ordinary menu item. The icon argument, if present, specifies the icon that the menu item should display. Similarly, the string argument specifies the text that the menu item should display. The integer argument specifies the keyboard mnemonic to use. You can specify any of the relevant VK constants defined in the KeyEvent class. For example, to specify the A key, use KeyEvent.VK_A.

JMenuItem add(JMenuItem) JMenuItem add(String)	Adds a menu item to the current end of the popup menu. If the argument is a string, then the menu automatically creates a JMenuItem object that displays the specified text.
JMenu add(JMenu)	Creates a menu bar.
void setJMenuBar(JMenuBar) JMenuBar getJMenuBar()	Sets or gets the menu bar of an applet, dialog, frame, internal frame, or root pane.
void setEnabled(boolean)	If the argument is true, enable the menu item. Otherwise, disable the menu item.
void setMnemonic(int)	Set the mnemonic that enables keyboard navigation to the menu or menu item. Use one of the VK constants defined in the KeyEvent class.
void setAccelerator(KeyStroke)	Set the accelerator that activates the menu item.
void addActionListener(ActionListener) void addItemListener(ItemListener)	Add an event listener to the menu item. See Handling Events from Menu Items for details.

Example:Create an animation with a single line, which changes its position in a clock-wise direction. This line should

produce an effect of a spinning line. In the same example give option for clock-wise or anti-clock-wise spinning. Also provide options to start and stop the animation. Demonstrate the animation on the screen.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class LineAnimation extends JFrame implements
ActionListener
{
        Timer t;
        JLabel lblText;
        JMenuBar mb;
        JMenu m1,m2;
        JMenuItem mi1,mi2,mi3,mi4;
        double theta=0.0;
        int x,y,incr=-1;
        //Graphics g;
        Container con;
        public LineAnimation()
        {
                super("Line Animation");
                lblText=new JLabel("Line Animation");
                mb=new JMenuBar();
                m1=new JMenu("Motion");
                m2=new JMenu("Direction");
                mi1=new JMenuItem("Start");
                mi2=new JMenuItem("Stop");
                mi3=new JMenuItem("Clock-wise");
                mi4=new JMenuItem("Anti-Clock-wise");
                t=new Timer(100,this);

                mi1.addActionListener(this);
                mi2.addActionListener(this);
                mi3.addActionListener(this);
                mi4.addActionListener(this);

                con=getContentPane();
                con.setLayout(new FlowLayout());
                con.add(lblText);
```

```
		m1.add(mi1); m1.add(mi2);
		m2.add(mi3); m2.add(mi4);
		mb.add(m1); mb.add(m2);
		setJMenuBar(mb);
		//g=getGraphics();
		setDefaultCloseOperation(EXIT_ON_CLOSE);
	}
	public void actionPerformed(ActionEvent ae)
	{
		if(ae.getSource()==t)
			repaint();

		if(ae.getSource()==mi1)
			t.start();

		if(ae.getSource()==mi2)
			t.stop();

		if(ae.getSource()==mi3)
			incr=1;

		if(ae.getSource()==mi4)
			incr=-1;
	}
	public void paint(Graphics g)
	{
		g.clearRect(0,0,300,300);
		x=(int)(100*Math.cos(theta*Math.PI/180));
		y=(int)(100*Math.sin(theta*Math.PI/180));
		g.drawLine(150+x,150+y,150-x,150-y);
		theta+=incr;
	}
	public static void main(String args[])
	{
		LineAnimation la=new LineAnimation();
		la.setVisible(true);
		la.setSize(300,300);
	}//main
}//class
```

2.1.10 JTable

The JTable is used to display and edit regular two-dimensional tables of cells. The JTable has many facilities that make it possible to customize its rendering and editing but provides defaults for these features so that simple tables can be set up easily.

The JTable uses integers exclusively to refer to both the rows and the columns of the model that it displays. The JTable simply takes a tabular range of cells and uses getValueAt(int, int) to retrieve the values from the model during painting. By default, columns may be rearranged in the JTable so that the view's columns appear in a different order to the columns in the model. This does not affect the implementation of the model at all: when the columns are reordered, the JTable maintains the new order of the columns internally and converts its column indices before querying the model.

Constructors	
JTable()	Constructs a default JTable that is initialized with a default data model, a default column model, and a default selection model.
JTable(int numRows, int numColumns)	Constructs a JTable with numRows and numColumns of empty cells using DefaultTableModel.
JTable(Object[][] rowData, Object[] columnNames)	Constructs a JTable to display the values in the two dimensional array, rowData, with column names, columnNames.
JTable(Vector rowData, Vector columnNames)	Constructs a JTable to display the values in the Vector of Vectors, rowData, with column names, columnNames.
Methods	
void clearSelection()	Deselects all selected columns and rows.
int getColumnCount()	Returns the number of columns in the column model.

String getColumnName(int column)	Returns the name of the column appearing in the view at column position column.
int getRowCount()	Returns the number of rows in this table's model.
int getRowHeight()	Returns the height of a table row, in pixels.
int getSelectedColumn()	Returns the index of the first selected column, -1 if no column is selected.
int getSelectedRow()	Returns the index of the first selected row, -1 if no row is selected.
Object getValueAt(int row, int column)	Returns the cell value at row and column.
void selectAll()	Selects all rows, columns, and cells in the table.
void setValueAt(Object aValue, int row, int column)	Sets the value for the cell in the table model at row and column.

Example: Write a program to create a user interface for students biodata. (Demo example for JTable, JScrollPane).

```
//Step 1 – import all the required packages
import javax.swing.*;
import java.awt.*;
/* Step 2 – Decide the class name & Container class (JFrame/JApplet) to be used */
public class StudentBioData04 extends JFrame
{
        //Step 3 – Create all the instances required
        JLabel lbldata;
        JTable tbldata;
        JScrollPane jsp;
        public StudentBioData04()
        {
                //Step 4- Create objects for the declared instances
                lbldata=new JLabel("Educational Details");
                Object header[]={"Year","Degree","Class"};
                Object rowdata[][]={
                        {"2005","SSC","First"},
                        {"2007","HSC","Second"},
                        {"2011","BSc","Distinction"}
                     };
                tbldata=new JTable(rowdata,header);
                jsp=new JScrollPane(tbldata);
//Step 5 – Add all the objects in the Content Pane with Layout
                Container con=getContentPane();
                con.setLayout(new FlowLayout());
                con.add(lbldata);
                con.add(jsp);
        }//constructor
//Step 6 – Event Handling. (Event handling code will come here)
        public static void main(String args[])
        {
        //Step 7 – Create object of class in main method
        StudentBioData04 sbd=new StudentBioData04();
        sbd.setSize(150,200);
        sbd.setVisible(true);
        }//main
}//class
```

Note: If you combine all the four student bio-data program, you would end up with the final program with all the components included. Just remove the common code from all the files.

2.1.11 JTree

A JTree is a component that displays information in a hierarchical format. Steps for creating a JTree are as follows:

- Create an Instance of JTree.
- Create objects for all the nodes required with the help of DefaultMutableTreeNode, which implemets Mutable TreeNode interface which extends the TreeNode interface. The TreeNode interface declares methods that obtains information about a TreeNode. To create a heirarchy of TreeNodes the add() of the DefaultMutableTreeNode can be used.

Methods of TreeNode interface	
TreeNode getChildAt(int childIndex)	Returns the child TreeNode at index childIndex.
int getChildCount()	Returns the number of children TreeNodes the receiver contains.
int getIndex(TreeNode node)	Returns the index of node in the receivers children.
TreeNode getParent()	Returns the parent TreeNode of the receiver.

Methods of MutuableTreeNode	
void insert(MutableTreeNode child, int index)	Adds child to the receiver at index.
void remove(int index)	Removes the child at index from the receiver.
void remove (MutableTreeNode node)	Removes node from the receiver.
void setParent (MutableTreeNode newParent)	Sets the parent of the receiver to newParent.

Methods of DefaultMutableTreeNode	
void add (MutableTreeNode newChild)	Removes newChild from its parent and makes it a child of this node by adding it to the end of this node's child array.
int getChildCount()	Returns the number of children of this node.
TreeNode[] getPath()	Returns the path from the root, to get to this node.
TreeNode getRoot()	Returns the root of the tree that contains this node.
boolean isRoot()	Returns true if this node is the root of the tree.

- Create the object of the tree with the top most node as an argument.
- Use the add method to create the heirarchy.
- Create an object of JScrollpane and add the JTree to it. Add the scroll pane to the content pane.

Event Handling

The JTree generates a TreeExpansionEvent which is in the package javax.swing.event. The getPath() of this class returns a Tree Path object that describes the path to the changed node. The addTreeExpansionListener and removeTreeExpansionListener methods allows listeners to register and unregister for the notifications. The TreeExpansionListener interface provides two methods:

void treeCollapsed (TreeExpansionEvent event)	Called whenever an item in the tree has been collapsed.
void treeExpanded (TreeExpansionEvent event)	Called whenever an item in the tree has been expanded.

Example: Demo example for tree.

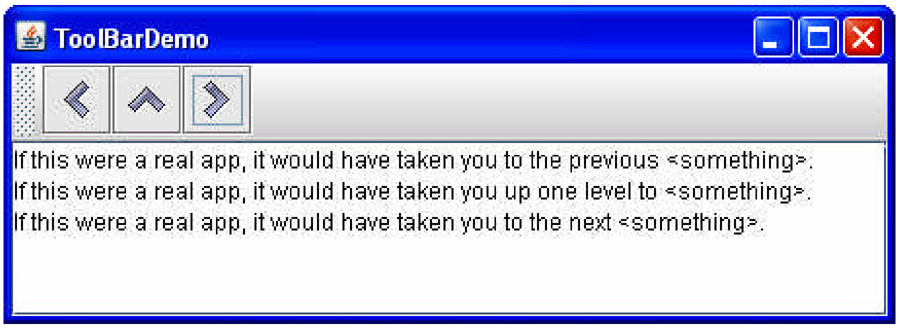
```
import java.awt.*;
import javax.swing.*;
import javax.swing.tree.*;
import java.awt.event.*;

//<applet code="TreeDemo" height=150 width=120></applet>

public class TreeDemo extends JApplet
{
JTree tree;
JTextField txt1;
public void init()
{
        Container con=getContentPane();
        con.setLayout(new BorderLayout());

        DefaultMutableTreeNode top=new
                                DefaultMutableTreeNode("Option");

        DefaultMutableTreeNode stream=new
                                DefaultMutableTreeNode("Stream");
```

```
DefaultMutableTreeNode arts=new
                        DefaultMutableTreeNode("Arts");
DefaultMutableTreeNode science=new
                        DefaultMutableTreeNode("Science");
DefaultMutableTreeNode comm=new
                        DefaultMutableTreeNode("Commerce");

DefaultMutableTreeNode year=new
                        DefaultMutableTreeNode("Year");

DefaultMutableTreeNode fy=new
                        DefaultMutableTreeNode("FY");
DefaultMutableTreeNode sy=new
                        DefaultMutableTreeNode("SY");
DefaultMutableTreeNode ty=new
                        DefaultMutableTreeNode("TY");

top.add(stream);
stream.add(arts); stream.add(comm); stream.add(science);

top.add(year);
year.add(fy); year.add(sy); year.add(ty);

tree=new JTree(top);
JScrollPane jsp=new JScrollPane(
tree,
JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,
JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);

con.add(jsp,BorderLayout.CENTER);
txt1=new JTextField("",20);
con.add(txt1,BorderLayout.SOUTH);

tree.addMouseListener(new MouseAdapter()
{
        public void mouseClicked(MouseEvent me)
        {
                        doMouseClicked(me);
        }
});
```

```
}// init()

void doMouseClicked(MouseEvent me)
{
        TreePath tp=tree.getPathForLocation(me.getX(),me.getY());
        if(tp!=null)
        {
                txt1.setText(tp.toString());
        }
        else
        {
                txt1.setText("");
        }
}//doMouse
}//class
```

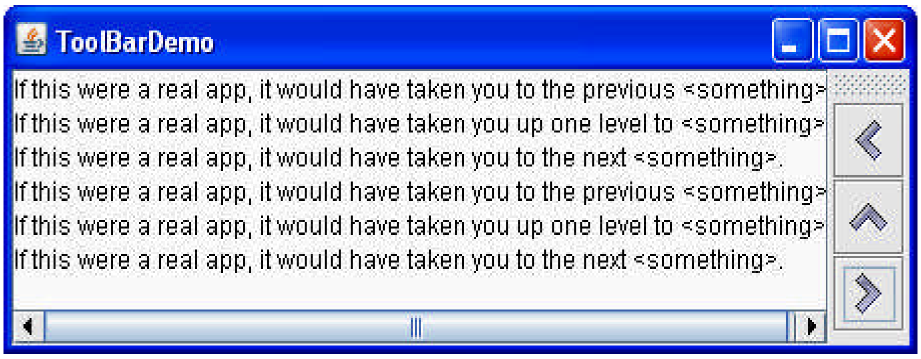
2.1.12 JToolBar

A JToolBar is a container that groups several components — usually buttons with icons — into a row or column. Often, tool bars provide easy access to functionality that is also in menus. The following images show an application named ToolBarDemo that contains a tool bar above a text area.



```
If this were a real app, it would have taken you to the previous <something>.
If this were a real app, it would have taken you up one level to <something>.
If this were a real app, it would have taken you to the next <something>.
```

By default, the user can drag the tool bar to another edge of its container or out into a window of its own. The next figure shows how the application looks after the user has dragged the tool bar to the right edge of its container.



```
If this were a real app, it would have taken you to the previous <something>
If this were a real app, it would have taken you up one level to <something>
If this were a real app, it would have taken you to the next <something>.
If this were a real app, it would have taken you to the previous <something>
If this were a real app, it would have taken you up one level to <something>
If this were a real app, it would have taken you to the next <something>.
```

For the drag behavior to work correctly, the tool bar must be in a container that uses the BorderLayout layout manager. The component that the tool bar affects is generally in the center of the container. The tool bar must be the only other component in the container, and it must not be in the center.

Method or Constructor	**Purpose**

JToolBar()
JToolBar(int)
JToolBar(String)
JToolBar(String, int)
	Creates a tool bar. The optional int parameter lets you specify the orientation; the default is HORIZONTAL. The optional String parameter allows you to specify the title of the tool bar's window if it is dragged outside of its container.

Component add(Component)	Adds a component to the tool bar. You can associate a button with an Action using the setAction(Action) method defined by the AbstractButton.
void addSeparator()	Adds a separator to the end of the tool bar.
void setFloatable(boolean) boolean isFloatable()	The floatable property is true by default, and indicates that the user can drag the tool bar out into a separate window. To turn off tool bar dragging, use toolBar.setFloatable(false). Some types of look and feel might ignore this property.
void setRollover(boolean) boolean isRollover()	The rollover property is false by default. To make tool bar buttons be indicated visually when the user passes over them with the cursor, set this property to true. Some types of look and feel might ignore this property.

Example: Demo example for JToolbar

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

public class ToolBarDemo extends JFrame implements ActionListener
{
JToolBar toolBar;
JButton cmdPrev,cmdUp,cmdNext;
JTextArea textArea;
JScrollPane scrollPane;
String newline = "\n";

public ToolBarDemo()
{
super("Tool bar Demo");
toolBar = new JToolBar("Still draggable");

cmdPrev=new JButton("Prev",new ImageIcon("Back24.gif"));
cmdUp=new JButton("Up",new ImageIcon("Up24.gif"));
cmdNext=new JButton("Next",new ImageIcon("Forward24.gif"));

toolBar.add(cmdPrev);
toolBar.add(cmdUp);
toolBar.add(cmdNext);

textArea = new JTextArea(5, 30);
textArea.setEditable(false);
scrollPane = new JScrollPane(textArea);

cmdPrev.addActionListener(this);
cmdUp.addActionListener(this);
cmdNext.addActionListener(this);

Container con=getContentPane();
con.setLayout(new BorderLayout());
con.add(toolBar, BorderLayout.NORTH);
con.add(scrollPane, BorderLayout.CENTER);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
public void actionPerformed(ActionEvent e)
{
String cmd = e.getActionCommand();
String description = null;
if(cmd.equals("Prev"))
{description = "taken you to the previous <something>.";}
if(cmd.equals("Up"))
{description = "taken you up one level to <something>.";}
if(cmd.equals("Next"))
{description = "taken you to the next <something>.";
}
textArea.append("If this were a real app, it would have "+description + newline);
textArea.setCaretPosition(textArea.getDocument().getLength());
}

public static void main(String[] args)
{
ToolBarDemo tb=new ToolBarDemo();
tb.setSize(300,300);
tb.setVisible(true);
}
}
```

2.1.13 JColorChooser

JColorChooser provides a pane of controls designed to allow a user to manipulate and select a color. This class provides three levels of API:

1. A static convenience method which shows a modal color-chooser dialog and returns the color selected by the user.
2. A static convenience method for creating a color-chooser dialog where ActionListeners can be specified to be invoked when the user presses one of the dialog buttons.
3. The ability to create instances of JColorChooser panes directly (within any container). PropertyChange listeners can be added to detect when the current "color" property changes.

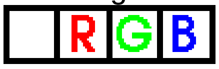
Creating and Displaying the Color Chooser

Method or Constructor	**Purpose**

JColorChooser()
JColorChooser(Color)
JColorChooser(ColorSelectionModel)
	Create a color chooser. The default constructor creates a color chooser with an initial color of Color.white. Use the second constructor to specify a different initial color. The ColorSelectionModel argument, when present, provides the color chooser with a color selection model.

Color showDialog(Component, String, Color)	Create and show a color chooser in a modal dialog. The Component argument is the parent of the dialog, the String argument specifies the dialog title, and the Color argument specifies the chooser's initial color.
JDialog createDialog(Component, String, boolean, JColorChooser, ActionListener, ActionListener)	Create a dialog for the specified color chooser. As with showDialog, the Component argument is the parent of the dialog and the String argument specifies the dialog title. The other arguments are as follows: the boolean specifies whether the dialog is modal, the JColorChooser is the color chooser to display in the dialog, the first ActionListener is for the **OK** button, and the second is for the **Cancel** button.

Setting or Getting the Current Color

Method	Purpose
void setColor(Color) void setColor(int, int, int) void setColor(int) Color getColor()	Set or get the currently selected color. The three integer version of the setColor method interprets the three integers together as an RGB color. The single integer version of the setColor method divides the integer into four 8-bit bytes and interprets the integer as an RGB color as follows: 

2.1.14 JFileChooser

File choosers provide a GUI for navigating the file system, and then either choosing a file or directory from a list, or entering the name of a file or directory. To display a file chooser, you usually use the JFileChooser API to show a modal dialog containing the file chooser. Another way to present a file chooser is to add an instance of JFileChooser to a container.

The JFileChooser API makes it easy to bring up open and save dialogs. The type of look and feel determines what these standard dialogs look like and how they differ. In the Java look and feel, the save dialog looks the same as the open dialog, except for the title on the dialog's window and the text on the button that approves the operation.

Creating and Showing the File Chooser

Method or Constructor	Purpose

JFileChooser()
JFileChooser(File)
JFileChooser(String)
	Creates a file chooser instance. The File and String arguments, when present, provide the initial directory.

int showOpenDialog(Component) int showSaveDialog(Component) int showDialog(Component, String)	Shows a modal dialog containing the file chooser. These methods return APPROVE_OPTION if the user approved the operation and CANCEL_OPTION if the user cancelled it. Another possible return value is ERROR_OPTION, which means an unanticipated error occurred.

Selecting Files and Directories

Method	Purpose
void setSelectedFile(File) File getSelectedFile()	Sets or obtains the currently selected file or (if directory selection has been enabled) directory.
void setSelectedFiles(File[]) File[] getSelectedFiles()	Sets or obtains the currently selected files if the file chooser is set to allow multiple selection.

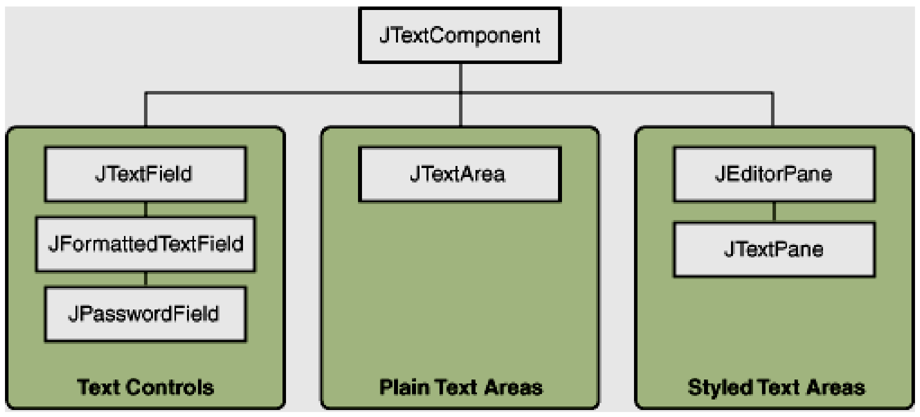
void setFileSelectionMode(int) void getFileSelectionMode() boolean is Directory Selection Enabled() boolean is File Selection Enabled()	Sets or obtains the file selection mode. Acceptable values are FILES_ONLY (the default), DIRECTORIES_ONLY, and FILES_AND_DIRECTORIES. Interprets whether directories or files are selectable according to the current selection mode.
void setMultiSelectionEnabled(boolean) boolean isMultiSelectionEnabled()	Sets or interprets whether multiple files can be selected at once. By default, a user can choose only one file.

2.1.15 Using Text Components

Swing text components display text and optionally allow the user to edit the text. Programs need text components for tasks ranging from the straightforward (enter a word and press Enter) to the complex (display and edit styled text with embedded images in an Asian language).

Swing provides six text components, along with supporting classes and interfaces that meet even the most complex text requirements. In spite of their different uses and capabilities, all Swing text components inherit from the same superclass, JTextComponent, which provides a highly-configurable and powerful foundation for text manipulation.

The following figure shows the JTextComponent hierarchy.



The following table tells you more about what you can do with each kind of text component.

Group	Description	Swing Classes
Text Controls	Also known simply as text fields, text controls can display only one line of editable text. Like buttons, they generate action events. Use them to get a small amount of textual information from the user and perform an action after the text entry is complete.	JTextField and its sub classes JPassword Field and JFormattedTextField
Plain Text Areas	JTextArea can display multiple lines of editable text. Although a text area can display text in any font, all of the text is in the same font. Use a text area to allow the user to enter unformatted text of any length or to display unformatted help information.	JTextArea
Styled Text Areas	A styled text component can display editable text using more than one font. Some styled text components allow embedded images and even embedded components. Styled text components are powerful and multi-faceted components suitable for high-end needs, and offer more avenues for customization than the other text components. Because they are so powerful and flexible, styled text components typically require more initial programming to set up and use. One exception is that editor panes can be easily loaded with formatted text from a URL, which makes them useful for displaying uneditable help information.	JEditorPane and its subclass JTextPane

2.2 INTERFACE ACTION

The Action interface provides a useful extension to the ActionListener interface in cases where the same functionality may be accessed by several controls.

public interface Action extends ActionListener

In addition to the actionPerformed method defined by the ActionListener interface, this interface allows the application to define, in a single place:

- One or more text strings that describe the function. These strings can be used, for example, to display the flyover text for a button or to set the text in a menu item.
- One or more icons that depict the function. These icons can be used for the images in a menu control, or for composite entries in a more sophisticated user interface.
- The enabled/disabled state of the functionality. Instead of having to separately disable the menu item and the toolbar button, the application can disable the function that implements this interface. All components which are registered as listeners for the state change then know to disable event generation for that item and to modify the display accordingly.

Certain containers, including menus and tool bars, know how to add an Action object. When an Action object is added to such a container, the container:

- Creates a component that is appropriate for that container (a tool bar creates a button component, for example).
- Gets the appropriate property(s) from the Action object to customize the component (for example, the icon image and flyover text).
- Checks the intial state of the Action object to determine if it is enabled or disabled, and renders the component in the appropriate fashion.
- Registers a listener with the Action object so that is notified of state changes. When the Action object changes from enabled to disabled, or back, the container makes the appropriate revisions to the event-generation mechanisms and renders the component accordingly.

For example, both a menu item and a toolbar button could access a Cut action object. The text associated with the object is specified as "Cut", and an image depicting a pair of scissors is specified as its icon. The Cut action-object can then be added to a menu and to a tool bar. Each container does the appropriate things with the object, and invokes its actionPerformed method when the component associated with it is activated. The application can then disable or enable the application object without worrying about what user-interface components are connected to it.

This interface can be added to an existing class or used to create an adapter (typically, by subclassing AbstractActio). The Action object can then be added to multiple action-aware containers and connected to Action-capable components. The GUI controls can then be activated or deactivated all at once by invoking theAction object's setEnabled method.

Note that Action implementations tend to be more expensive in terms of storage than a typical ActionListener, which does not offer the benefits of centralized control of functionality and broadcast of property changes. For th is reason, you should take care to only use Actions where their benefits are desired, and use simple ActionListeners elsewhere.

Method Summary	
void	addPropertyChangeListener(PropertyChangeListener listener) Adds a PropertyChange listener.
Object	getValue(String key) Gets one of this object's properties using the associated key.
boolean	isEnabled() Returns the enabled state of the Action.
void	putValue(String key, Object value) Sets one of this object's properties using the associated key.
void	removePropertyChangeListener(PropertyChangeListener listener) Removes a PropertyChange listener.
void	setEnabled(boolean b) Sets the enabled state of the Action.

Methods inherited from interface java.awt.event.ActionListener
actionPerformed

2.3 PRINTING WITH 2D API

The Java 2D printing API is the java.awt.print package that is part of the Java 2 SE,version 1.2 and later. The Java 2D printing API provides for creating a PrinterJob, displaying a printer dialog to the user, and printing paginated graphics using the same java.awt.Graphics and java.awt.Graphics2D classes that are used to draw to the screen.

Many of the features that are new in the Java Print Service, such as printer discovery and specification of printing attributes, are also very important to users of the Java 2D printing API. To make these features available to users of Java 2D printing, the java.awt.print package has been updated for version 1.4 of the JavaTM 2 SE to allow access to the Java^{TM} Print Service from the Java 2D printing API.

Developers of Java 2D printing applications have four ways of using the Java Print Service with the Java 2D API:

- Print 2D graphics using PrinterJob.
- Stream 2D graphics using PrinterJob
- Print 2D graphics using using DocPrintJob and a service-formatted DocFlavor
- Stream 2D graphics using DocPrintJob and a service-formatted DocFlavor

2.4 JAVA PRINT SERVICES API

The Java Print Service (JPS) is a new Java Print API that is designed to support printing on all Java platforms, including platforms requiring a small footprint, but also supports the current Java 2 Print API. This unified Java Print API includes extensible print attributes based on the standard attributes specified in the Internet Printing Protocol (IPP) 1.1 from the IETF Specification, RFC 2911. With the attributes, client and server applications can discover and select printers that have the capabilities specified by the attributes. In addition to the included StreamPrintService, which allows applications to transcode print data to different formats, a third party can dynamically install their own print services through the Service Provider Interface.

The Java Print Service API unifies and extends printing on the Java platform by addressing many of the common printing needs of both client and server applications that are not met by the current Java printing APIs. In addition to supporting the current Java 2D printing features, the Java Print Service offers many improvements, including:

• Both client and server applications can discover and select printers based on their capabilities and specify the properties of a print job. Thus, the JPS provides the missing component in a printing subsystem: programmatic printer discovery.

• Implementations of standard IPP attributes are included in the JPS API as first-class objects.

• Applications can extend the attributes included with the JPS API.

• Third parties can plug in their own print services with the Service Provider Interface.

How Applications Use the Java Print Service

A typical application using the Java Print Service API performs these steps to process a print request:

1. Obtain a suitable DocFlavor, which is a class that defines the format of the print data.

2. Create and populate an AttributeSet, which encapsulates a set of attributes that describe the desired print service capabilities, such as the ability to print five copies, stapled, and double-sided.

3. Lookup a print service that can handle the print request as specified by the DocFlavor and the attribute set.

4. Create a print job from the print service.

5. Call the print job's print method.

The application performs these steps differently depending on what and how it intends to print. The application can either send print data to a printer or to an output stream. The print data can either be a document in the form of text or images, or a Java object encapsulating 2D Graphics. If the print data is 2D graphics , the print job can be represented by either a DocPrintJob or a PrinterJob. If the print data is a document then a DocPrintJob must be used.

2.5 SUMMARY

- The JComponent class extends the Container class, which itself extends Component.
- A text field is a basic text control that enables the user to type a small amount of text.

- JComboBox is a component that combines a button or editable field and a drop-down list.
- JList is a component that allows the user to select one or more objects from a list.
- JTable is used to display and edit regular two-dimensional tables of cells.
- A JTree is a component that displays information in a hierarchical format.
- A JToolBar is a container that groups several components — usually buttons with icons — into a row or column.
- The Java 2D printing API provides for creating a PrinterJob, displaying a printer dialog to the user, and printing paginated graphics

2.6 UNIT END EXERCISE

1) Explain the importance of JComponent Class.
2) Write a Swing program containing a button with the caption “Now” and a textfield. On click of the button, the current date and time should be displayed in a textfield?
3) State and explain any three classes used to create Menus in Swing?
4) How to create a JMenu and add it to a JMenuBar inside a JFrame?
5) Explain the classes used to create a tree in Swing?
6) List and explain any three Text-Entry Components.
7) How Applications Use the Java Print Service?
8) Write a swing program containing three text fields. The first text field accepts first name, second accepts last name and the third displays full name on click of a button.
9) Define a class that enables the drawing of freehand lines on a screen through mouse clicking and dragging. Use anonymous inner classes to implement event listeners.
10) Define a class that displays circle when key C is typed and rectangle when key R is typed.
11) Write a program containing JMenu. It contains menus such as Circle, Rectangle and Exit. When the menu is clicked, appropriate function should be executed as suggested by the menu.

12) Write a program containing three text fields, out of which third is disabled. It also contains 4 buttons representing +, - , * and / operations. First two text fields should accept two numbers. When any button is clicked, the appropriate result should be displayed in the third text field.

2.7 FURTHER READING

- Joe Wigglesworth and Paula McMillan, Java Programming: Advanced Topics, Thomson Course Technology (SPD), Third Edition, 2004
- Cay S. Horstmann, Gary Cornell, Core Java™ 2: Volume II–Advanced Features Prentice Hall PTR, 2001

❖❖❖❖

3

JAVA DATABASE CONNECTIVITY

Unit Structure:

3.0 Objectives
3.1 Introduction
3.2 Load the JDBC Driver
3.3 Define the Connection URL
3.4 Establish the Connection
3.5 Create a Statement Object
3.6 Execute a Query or Update
3.7 Process the Results
3.8 Close the Connection
3.9 Summary
3.10 Unit end exercise
3.11 Further Reading

3.0 OBJECTIVES

The objective of this chapter is to learn the seven steps to connect to a database and make transactions with the database. The java.sql package has a lot of classes and interfaces which will help us in connecting to a database. We will learn how to use the classes from this package.

3.1 INTRODUCTION

JDBC provides a standard library for accessing relational databases. By using the JDBC API, you can access a wide variety of SQL databases with exactly the same Java syntax. It is important to note that although the JDBC API standardizes the approach for connecting to databases, the syntax for sending queries and committing transactions, and the data structure representing the result, JDBC does not attempt to standardize the SQL syntax. So, you can use any SQL extensions your database vendor supports. However, since most queries follow standard SQL syntax, using

JDBC lets you change database hosts, ports, and even database vendors with minimal changes to your code.

Using JDBC in General

In this section we present the seven standard steps for querying databases. Following is a summary; details are given in the rest of the section.

- ***Load the JDBC driver*** - To load a driver, you specify the class name of the database driver in the Class.forName method. By doing so, you automatically create a driver instance and register it with the JDBC driver manager.
- ***Define the connection URL*** - In JDBC, a connection URL specifies the server host, port, and database name with which to establish a connection.
- ***Establish the connection*** - With the connection URL, username, and password, a network connection to the database can be established. Once the connection is established, database queries can be performed until the connection is closed.
- ***Create a Statement object*** - Creating a Statement object enables you to send queries and commands to the database.
- ***Execute a query or update*** - Given a Statement object, you can send SQL statements to the database by using the execute, executeQuery, executeUpdate, or executeBatch methods.
- ***Process the results*** - When a database query is executed, a ResultSet is returned. The ResultSet represents a set of rows and columns that you can process by calls to next and various getXxx methods.
- ***Close the connection*** - When you are finished performing queries and processing results, you should close the connection, releasing resources to the database.

3.2 LOAD THE JDBC DRIVER

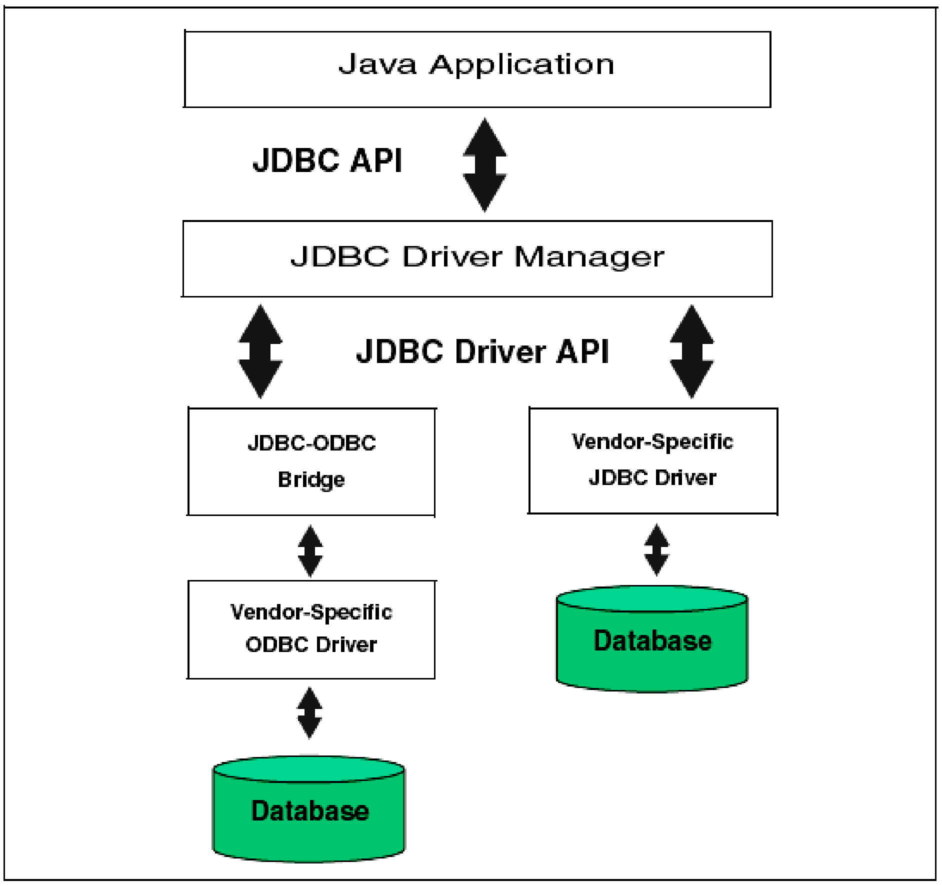
The driver is the piece of software that knows how to talk to the actual database server. To load the driver, you just load the appropriate class; a static block in the driver class itself automatically makes a driver instance and registers it with the JDBC driver manager. To make your code as flexible as possible, avoid hard-coding the reference to the class name. These requirements bring up two interesting questions. First, how do you load a class without making an instance of it? Second, how can you refer to a class whose name isn't known when the code is compiled? The answer to both questions is to use Class.forName.

This method takes a string representing a fully qualified class name (i.e., one that includes package names) and loads the corresponding class. This call could throw a Class Not Found Exception, so it should be inside a try/catch block as shown below.

```
try
{
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        Class.forName("oracle.jdbc.driver.OracleDriver");
        Class.forName("com.sybase.jdbc.SybDriver");
}
catch(ClassNotFoundException cnfe)
{
        System.err.println("Error loading driver: " + cnfe);
}
```

One of the beauties of the JDBC approach is that the database server requires no changes whatsoever. Instead, the JDBC driver (which is on the client) translates calls written in the Java programming language into the native format required by the server. This approach means that you have to obtain a JDBC driver specific to the database you are using and that you will need to check the vendor's documentation for the fully qualified class name to use. In principle, you can use Class.forName for any class in your CLASSPATH. In practice, however, most JDBC driver vendors distribute their drivers inside JAR files. So, during development be sure to include the path to the driver JAR file in your CLASSPATH setting. For deployment on a Web server, put the JAR file in the WEB-INF/lib directory of your Web application. Check with your Web server administrator, though. Often, if multiple Web applications are using the same database drivers, the administrator will place the JAR file in a common directory used by the server. For example, in Apache Tomcat, JAR files common to multiple applications can be placed in install_dir/common/lib.

Figure 1 illustrates two common JDBC driver implementations. The first approach is a JDBC-ODBC bridge, and the second approach is a pure Java implementation. A driver that uses the JDBC-ODBC bridge approach is known as a Type I driver. Since many databases support Open DataBase Connectivity (ODBC) access, the JDK includes a JDBC-ODBC bridge to connect to databases. However, you should use the vendor's pure Java driver, if available, because the JDBC-ODBC driver implementation is slower than a pure Java implementation. Pure Java drivers are known as Type IV. The JDBC specification defines two other driver types, Type II and Type III; however, they are less common. Type 2 - drivers are written partly in the Java programming language and partly in native code. These drivers use a native client library specific to the data source to which they connect. Again, because of the native code, their portability is limited. Type 3 - drivers use a pure Java client and communicate with a middleware server using a database-independent protocol. The middleware server then communicates the client's requests to the data source.



3.3 DEFINE THE CONNECTION URL

Once you have loaded the JDBC driver, you must specify the location of the database server. URLs referring to databases use the jdbc: protocol and embed the server host, port, and database name (or reference) within the URL. The exact format is defined in the documentation that comes with the particular driver, but here are a few representative examples.

```
String host = "dbhost.yourcompany.com";
String dbName = "someName";
int port = 1234;
String oracleURL = "jdbc:oracle:thin:@" + host +":" + port + ":" +
dbName;
String sybaseURL = "jdbc:sybase:Tds:" + host +":" + port + ":" +
"?SERVICENAME=" + dbName;
String msAccessURL = "jdbc:odbc:" + dbName;
```

3.4 ESTABLISH THE CONNECTION

To make the actual network connection, pass the URL, database username, and database password to the getConnection method of the DriverManager class, as illustrated in the following example. Note that getConnection throws an SQLException, so you need to use a try/catch block..

```
String username = "jay_debesee";
String password = "secret";
Connection connection
=DriverManager.getConnection(msAccessURL, username,
password);
```

The Connection class includes other useful methods, which we briefly describe below.

- prepareStatement - Creates precompiled queries for submission to the database.
- prepareCall - Accesses stored procedures in the database.
- rollback/commit - Controls transaction management.
- close - Terminates the open connection.
- Is Closed - Determines whether the connection timed out or was explicitly closed.

3.5 CREATE A STATEMENT OBJECT

A Statement object is used to send queries and commands to the database. It is created from the Connection object using create Statement as follows.

```
Statement statement = connection.createStatement();
```

Most, but not all, database drivers permit multiple concurrent Statement objects to be open on the same connection.

3.6 EXECUTE A QUERY OR UPDATE

Once you have a Statement object, you can use it to send SQL queries by using the executeQuery method, which returns an object of type ResultSet. Here is an example.

```
String query = "SELECT col1, col2, col3 FROM sometable";
ResultSet resultSet = statement.executeQuery(query);
```

The following list summarizes commonly used methods in the Statement class.

- executeQuery - Executes an SQL query and returns the data in a ResultSet. The ResultSet may be empty, but never null.
- executeUpdate - Used for UPDATE, INSERT, or DELETE commands. Returns the number of rows affected, which could be zero. Also provides support for Data Definition Language (DDL) commands, for example, CREATE TABLE, DROP TABLE, and ALTER TABLE.
- executeBatch - Executes a group of commands as a unit, returning an array with the update counts for each command. Use addBatch to add a command to the batch group. Note that vendors are not required to implement this method in their driver to be JDBC compliant.
- setQueryTimeout - Specifies the amount of time a driver waits for the result before throwing an SQLException.
- getMaxRows/setMaxRows - Determines the number of rows a ResultSet may contain. Excess rows are silently dropped. The default is zero for no limit.

In addition to using the methods described here to send arbitrary commands, you can use a Statement object to create parameterized queries by which values are supplied to a precompiled fixed-format query using Prepared Statements.

3.7 PROCESS THE RESULTS

The simplest way to handle the results is to use the next method of ResultSet to move through the table a row at a time. Within a row, ResultSet provides various getXxx methods that take a column name or column index as an argument and return the result in a variety of different Java types. For instance, use getInt if the value should be an integer, getString for a String, and so on for most other data types. If you just want to display the results, you can use getString for most of the column types. However, if you use the version of getXxx that takes a column index (rather than a column name), note that columns are indexed starting at 1 (following the SQL convention), not at 0 as with arrays, vectors, and most other data structures in the Java programming language.

Here is an example that prints the values of the first two columns and the first name and last name, for all rows of a ResultSet.

```
while(resultSet.next())
{
System.out.println(  resultSet.getString(1) + " " +
resultSet.getString(2) + " " +
resultSet.getString("firstname") + " "
resultSet.getString("lastname")              );
}
```

We suggest that when you access the columns of a ResultSet, you use the column name instead of the column index. That way, if the column structure of the table changes, the code interacting with the ResultSet will be less likely to fail. In JDBC 1.0, you can only move forward in the ResultSet; however, in JDBC 2.0, you can move forward (next) and backward (previous) in the ResultSet as well as move to a particular row (relative, absolute).

The following list summarizes useful ResultSet methods.

- next/previous - Moves the cursor to the next (any JDBC version) or previous (JDBC version 2.0 or later) row in the ResultSet, respectively.
- relative/absolute - The relative method moves the cursor a relative number of rows, either positive (up) or negative (down). The absolute method moves the cursor to the given row number. If the absolute value is negative, the cursor is positioned relative to the end of the ResultSet (JDBC 2.0).
- getXxx - Returns the value from the column specified by the column name or column index as an Xxx Java type (see java.sql.Types). Can return 0 or null if the value is an SQL NULL.
- wasNull - Checks whether the last getXxx read was an SQL NULL. This check is important if the column type is a primitive (int, float, etc.) and the value in the database is 0. A zero value would be indistinguishable from a database value of NULL, which is also returned as a 0. If the column type is an object (String, Date, etc.), you can simply compare the return value to null.
- findColumn - Returns the index in the ResultSet corresponding to the specified column name.
- getRow - Returns the current row number, with the first row starting at 1 (JDBC 2.0).
- getMetaData - Returns a ResultSetMetaData object describing the ResultSet. ResultSetMetaData gives the number of columns and the column names.

The getMetaData method is particularly useful. Given only a ResultSet, you have to know the name, number, and type of the columns to be able to process the table properly. For most fixed-format queries, this is a reasonable expectation. For ad hoc queries, however, it is useful to be able to dynamically discover high-level information about the result. That is the role of the ResultSetMetaData class: it lets you determine the number, names, and types of the columns in the ResultSet.

Useful ResultSetMetaData methods are described below.

- getColumnCount - Returns the number of columns in the ResultSet.
- getColumnName - Returns the database name of a column (indexed starting at 1).
- getColumnType - Returns the SQL type, to compare with entries in java.sql.Types.
- isReadOnly - Indicates whether the entry is a read-only value.
- isSearchable - Indicates whether the column can be used in a WHERE clause.
- isNullable - Indicates whether storing NULL is legal for the column.

ResultSetMetaData does not include information about the number of rows; however, if your driver complies with JDBC 2.0, you can call last on the ResultSet to move the cursor to the last row and then call getRow to retrieve the current row number. In JDBC 1.0, the only way to determine the number of rows is to repeatedly call next on the ResultSet until it returns false.

3.8 CLOSE THE CONNECTION

To close the connection explicitly, you would do:

```
        connection.close();
```

Closing the connection also closes the corresponding Statement and ResultSet objects. You should postpone closing the connection if you expect to perform additional database operations, since the overhead of opening a connection is usually large. In fact, reusing existing connections is such an important optimization that the JDBC 2.0 API defines a ConnectionPoolDataSource interface for obtaining pooled connections.

Example - Write a JDBC program to create a table and insert records into it.

```
import java.sql.*;

public class FirstJDBC
```

```
{
        public static void main(String args[])
        {
                try
                {
                Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
                }
                catch(Exception e){
                        System.out.println("Error"+e.getMessage());}
                try
                {
                Connection con =
                DriverManager.getConnection("jdbc:odbc:myDSN");

                Statement st = con.createStatement();

                String str = "create table Login
                                (UserName text,Password text)";

                st.executeUpdate(str);
                System.out.println("Table Created");

                st.executeUpdate("insert into Login
                                                values ('Raj','ghf')");
                st.executeUpdate("insert into Login
                                                values ('Harsh','hhh')");
                st.executeUpdate("insert into Login
                                                values ('Ram','gfg')");
                st.executeUpdate("insert into Login
                                                values ('Raju','ggg')");
                st.executeUpdate("insert into Login
                                                values ('Ramu','mmm')");
                con.commit();

                System.out.println("Values Inserted");
                }
                catch(Exception
e){System.out.println("Error"+e.getMessage());}
        }
}
```

Note: To run the program we need to create a DSN. Here are the steps to create.

1) Click on Start → Settings → Control Panel → Administrative Tools
2) Click on Data Sources (ODBC) which opens a dialog box. Click Add button.
3) Select the driver for your database and click on Finish button.
4) Give name myDSN and select your database which was already created.

Example - Write a JDBC program to fetch values and display them on screen.

```
import java.sql.*;

public class SelDemo
{
        public static void main(String args[])
        {
                try
                {
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

                        Connection con =
                                DriverManager.getConnection("jdbc:odb
                                c: myDSN ");
                        Statement st = con.createStatement();

                        ResultSet sel = st.executeQuery
                                        ("select * from Login");
                        while(sel.next())
                        {
                                String name = sel.getString(1);
                                String pass = sel.getString(2);
                                System.out.println(name+" "+pass);
                        }
                }
                catch(Exception e)
                {
                System.out.println("Errooorrrr"+e.getMessage());
                }
        }
}
```

Example - Write a JDBC program to get the columns names and row data from a table.

```
import java.sql.*;

class FetchData
{
        public static void main(String args[])
        {
                try
                {
                Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
                Connection con=DriverManager.getConnection
                                                ("jdbc:odbc: myDSN");
                System.out.println("Connecting to database....");
                Statement st=con.createStatement();
                ResultSet rs=st.executeQuery
                        ("select * from dept order by deptno asc");
                ResultSetMetaData rsmd = rs.getMetaData();

                System.out.println("Displaying Values....");
                for(int i=1;i<=rsmd.getColumnCount();i++)
                        System.out.print(rsmd.getColumnName(i)+"\t");

                System.out.println("\n");
                con.commit();
                while(rs.next()) {
                System.out.println(rs.getInt(1)+
                                "\t"+rs.getString(2)+"\t"+rs.getString(3));
                }
                st.close();
                }
                catch(Exception a){
                        System.out.println("ERROR"+a.getMessage());
                }
        }
}
```

3.9 SUMMARY

- Load the JDBC driver - To load a driver, you specify the class name of the database driver
- Define the connection URL - In JDBC, a connection URL specifies the server host, port, and database name with which to establish a connection.
- Establish the connection - With the connection URL, username, and password, a network connection to the database can be established.
- Create a Statement object - Creating a Statement object enables you to send queries and commands to the database.
- Execute a query or update - Given a Statement object, you can send SQL statements to the database
- Process the results - When a database query is executed, a ResultSet is returned.
- Close the connection - When you are finished performing queries and processing results, you should close the connection, releasing resources to the database.

3.10 UNIT END EXERCISE

1) What are the components of JDBC?
2) Explain the importance of the following methods:
    a. Class.forName()
    b. DriverManager.getConnection()
    c. Connection.createStatement()
3) Explain any two drivers of JDBC.
4) Explain different types of JDBC Drivers.
5) Outline the steps to access a database using JDBC.
6) Expalin the following methods and state the class/interface to which they belong to:
    a. executeUpdate()
    b. getColumnCount()
    c. getString()
7) Write a JDBC program that accepts a table name and displays total number of records present in it.
8) Write a JDBC program that accepts account number from the user and obtains the name and current balance by checking the appropriate fields from the customer table.

9) Write a JDBC program to accept and table name and to display the total number of columns and total number of records from it.

10) Write a JDBC program to accept name of a student, find the student in the table and based on the date of birth calculate and display the students age.

3.11 FURTHER READING

- Ivan Bayross, Web Enabled Commercial Applications Development Using Java 2, BPB Publications, Revised Edition, 2006
- Joe Wigglesworth and Paula McMillan, Java Programming: Advanced Topics, Thomson Course Technology (SPD), Third Edition, 2004
- Cay S. Horstmann, Gary Cornell, Core Java™ 2: Volume II– Advanced Features Prentice Hall PTR, 2001

❖❖❖❖

4

ADVANCE JDBC

Unit Structure:

4.0 Objectives
4.1 The JDBC Exception classes
4.2 Prepared Statements
4.3 Joins
4.4 Transactions
4.5 Stored Procedures
4.6 Summary
4.7 Unit end exercise
4.8 Further Reading

4.0 OBJECTIVES

The objective of this chapter is to learn the various SQL JDBC Exceptions and advance JDBC features such as joins, stored procedures, prepared statements and transactions.

4.1 THE JDBC EXCEPTION CLASSES

- class java.lang.**Throwable**
    - class java.lang.**Exception**
        - class java.sql.**SQLException**
            - class java.sql.**BatchUpdateException**
            - class java.sql.**SQLWarning**
                - class java.sql.**DataTruncation**

4.1.1 SQLException

An exception that provides information on a database access error or other errors.

Each SQLException provides several kinds of information:

- a string describing the error. This is used as the Java Exception message, available via the method getMesage.

- a "SQLstate" string, which follows either the XOPEN SQLstate conventions or the SQL 99 conventions. The values of the SQLState string are described in the appropriate spec. The DatabaseMetaData method getSQLStateType can be used to discover whether the driver returns the XOPEN type or the SQL 99 type.
- an integer error code that is specific to each vendor. Normally this will be the actual error code returned by the underlying database.
- a chain to a next Exception. This can be used to provide additional error information.

4.1.2 BatchUpdateException

An exception thrown when an error occurs during a batch update operation. In addition to the information provided by SQLException, a BatchUpdateException provides the update counts for all commands that were executed successfully during the batch update, that is, all commands that were executed before the error occurred. The order of elements in an array of update counts corresponds to the order in which commands were added to the batch.

After a command in a batch update fails to execute properly and a BatchUpdateException is thrown, the driver may or may not continue to process the remaining commands in the batch. If the driver continues processing after a failure, the array returned by the method BatchUpdateException.getUpdateCounts will have an element for every command in the batch rather than only elements for the commands that executed successfully before the error. In the case where the driver continues processing commands, the array element for any command that failed is Statement.EXECUTE_FAILED.

4.1.3 SQLWarning

An exception that provides information on database access warnings. Warnings are silently chained to the object whose method caused it to be reported.

Warnings may be retrieved from Connection, Statement, and ResultSet objects. Trying to retrieve a warning on a connection after it has been closed will cause an exception to be thrown. Similarly, trying to retrieve a warning on a statement after it has been closed or on a result set after it has been closed will cause an exception to be thrown. Note that closing a statement also closes a result set that it might have produced.

4.1.4 DataTruncation

An exception that reports a DataTruncation warning (on reads) or throws a DataTruncation exception (on writes) when JDBC unexpectedly truncates a data value.

The SQLstate for a DataTruncation is 01004.

4.2 PREPARED STATEMENTS:

A PreparedStatement object is more convenient for sending SQL statements to the database. This special type of statement is derived from the more general class, Statement. If you want to execute a Statement object many times, use a PreparedStatement object instead which reduces execution time.

The main feature of a Prepared Statement object is that it is given an SQL statement when it is created. The advantage to this is that in most cases, this SQL statement is sent to the DBMS right away, where it is compiled. As a result, the Prepared Statement object contains not just an SQL statement, but an SQL statement that has been precompiled.

This means that when the PreparedStatement is executed, the DBMS can just run the PreparedStatement SQL statement without having to compile it first. Although PreparedStatement objects can be used for SQL statements with no parameters, we use them most often for SQL statements that take parameters. The advantage of using SQL statements that take parameters is that you can use the same statement and supply it with different values each time you execute it.

Creating a PreparedStatement Object

A PreparedStatement objects can be created with a Connection method.

```
E.g. PreparedStatement updateEmp = con.prepareStatement(
                "UPDATE EMPLOYEES SET SALARY  = ?
                WHERE EMP_ID = ?");
```

Supplying Values for PreparedStatement Parameters

You need to supply values to be used in place of the question mark placeholders (if there are any) before you can execute a PreparedStatement object. You do this by calling one of the setXXX methods defined in the PreparedStatement class. If the value you want to substitute for a question mark is a Java int, you call the method setInt. If the value you want to substitute for a question mark is a Java String, you call the method setString, and so on. In general, there is a setXXX method for each primitive type declared in the Java programming language.

```
updateEmp.setInt(1, 7500);
updateEmp.setInt(2,1030);
updateEmp.executeUpdate():
```

The method executeUpdate was used to execute both the Statement stmt and the Prepared Statement update Emp. Notice, however, that no argument is supplied to execute Update when it is used to execute update Emp. This is true because update Emp already contains the SQL statement to be executed.

4.3 JOINS

Sometimes we need to use two or more tables to get the data we want. For example, suppose we want a list of the coffees we buy from Acme, Inc. This involves information in the COFFEES table as well as SUPPLIERS table. This is a case where a join is needed. A join is a database operation that relates two or more tables by means of values that they share in common. In our database, the tables COFFEES and SUPPLIERS both have the column SUP_ID, which can be used to join them.

The table COFFEES, is shown here:

COF_NAME	**SUP_ID**	**PRICE**	**SALES**	**TOTAL**
Colombian	101	7.99	0	0
French_Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colombian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

The following code selects the whole table and lets us see what the table SUPPLIERS looks like:
ResultSet rs = stmt.executeQuery("select * from SUPPLIERS");
The result set will look similar to this:

SUP_ID	SUP_NAME	STREET	CITY	STATE	ZIP
101	Reliance	99 Market Street	New Delhi	Delhi	95199
49	Tata	1 Party Place	Mumbai	Mah	95460
150	Sahara	100 Coffee Lane	Kolkata	WB	93966

The names of the suppliers are in the table SUPPLIERS, and the names of the coffees are in the table COFFEES. Since both tables have the column SUP_ID, this column can be used in a join. It follows that you need some way to distinguish which SUP_ID column you are referring to. This is done by preceding the column name with the table name, as in "COFFEES.SUP_ID" to indicate that you mean the column SUP_ID in the table COFFEES. The following code, in which stmt is a Statement object, selects the coffees bought from Acme, Inc.:

```
String query = "
SELECT COFFEES.COF_NAME " +
  "FROM COFFEES, SUPPLIERS " +
  "WHERE SUPPLIERS.SUP_NAME LIKE 'Reliance' " +
  "and SUPPLIERS.SUP_ID = COFFEES.SUP_ID";

ResultSet rs = stmt.executeQuery(query);
System.out.println("Coffees bought from Reliance.: ");
while (rs.next()) {
   String coffeeName = rs.getString("COF_NAME");
   System.out.println("     " + coffeeName);
}
```

This will produce the following output:

```
Coffees bought from Reliance.:
    Colombian
    Colombian_Decaf
```

4.4 TRANSACTIONS

There are times when you do not want one statement to take effect unless another one completes. For example, when the proprietor of The Coffee Break updates the amount of coffee sold each week, he will also want to update the total amount sold to date. However, he will not want to update one without updating the other; otherwise, the data will be inconsistent. The way to be sure that either both actions occur or neither action occurs is to use a transaction. A transaction is a set of one or more statements that are executed together as a unit, so either all of the statements are executed, or none of the statements is executed.

Disabling Auto-commit Mode

When a connection is created, it is in auto-commit mode. This means that each individual SQL statement is treated as a transaction and is automatically committed right after it is executed.

The way to allow two or more statements to be grouped into a transaction is to disable auto-commit mode. This is demonstrated in the following line of code, where con is an active connection:

```
con.setAutoCommit(false);
```

Committing a Transaction

Once auto-commit mode is disabled, no SQL statements are committed until you call the method commit explicitly. All statements executed after the previous call to the method commit are included in the current transaction and committed together as a unit.

The following code, in which con is an active connection, illustrates a transaction:

```
con.setAutoCommit(false);
PreparedStatement updateSales = con.prepareStatement(
   "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
updateSales.setInt(1, 50);
updateSales.setString(2, "Colombian");
updateSales.executeUpdate();
PreparedStatement updateTotal = con.prepareStatement(
   "UPDATE COFFEES SET TOTAL = TOTAL + ? WHERE COF_NAME LIKE ?");
updateTotal.setInt(1, 50);
updateTotal.setString(2, "Colombian");
updateTotal.executeUpdate();
con.commit();
con.setAutoCommit(true);
```

In this example, auto-commit mode is disabled for the connection con, which means that the two prepared statements updateSales and updateTotal are committed together when the method commit is called. The final line of the previous example enables auto-commit mode, which means that each statement is once again committed automatically when it is completed. Then, you are back to the default state where you do not have to call the method commit yourself. It is advisable to disable auto-commit mode only while you want to be in transaction mode. This way, you avoid holding database locks for multiple statements, which increases the likelihood of conflicts with other users.

Using Transactions to Preserve Data Integrity

In addition to grouping statements together for execution as a unit, transactions can help to preserve the integrity of the data in a table. For instance, suppose that an employee was supposed to enter new coffee prices in the table COFFEES but delayed doing it for a few days. In the meantime, prices rose, and today the owner is in the process of entering the higher prices. The employee finally gets around to entering the now outdated prices at the same time that the owner is trying to update the table. After inserting the outdated prices, the employee realizes that they are no longer valid and calls the Connection method rollback to undo their effects. (The method rollback aborts a transaction and restores values to what they were before the attempted update.) At the same time, the owner is executing a SELECT statement and printing out the new prices. In this situation, it is possible that the owner will print a price that was later rolled back to its previous value, making the printed price incorrect.

This kind of situation can be avoided by using transactions, providing some level of protection against conflicts that arise when two users access data at the same time. To avoid conflicts during a transaction, a DBMS uses locks, mechanisms for blocking access by others to the data that is being accessed by the transaction. Once a lock is set, it remains in force until the transaction is committed or rolled back. For example, a DBMS could lock a row of a table until updates to it have been committed. The effect of this lock would be to prevent a user from getting a dirty read, that is, reading a value before it is made permanent. (Accessing an updated value that has not been committed is considered a dirty read because it is possible for that value to be rolled back to its previous value. If you read a value that is later rolled back, you will have read an invalid value.)

Types of Result Sets

Results sets may have different levels of functionality. For example, they may be scrollable or non-scrollable. A scrollable result set has a cursor that moves both forward and backward and can be moved to a particular row. Also, result sets may be sensitive or insensitive to changes made while they are open; that is, they may or may not reflect changes to column values that are modified in the database. A developer should always keep in mind the fact that adding capabilities to a ResultSet object incurs additional overhead, so it should be done only as necessary.

Based on the capabilities of scrollability and sensitivity to changes, there are three types of result sets available. The following constants, defined in the ResultSet interface, are used to specify these three types of result sets:

1. **TYPE_FORWARD_ONLY**
    - The result set is nonscrollable; its cursor moves forward only, from top to bottom.
    - The view of the data in the result set depends on whether the DBMS materializes results incrementally.
2. **TYPE_SCROLL_INSENSITIVE**
    - The result set is scrollable: Its cursor can move forward or backward and can be moved to a particular row or to a row whose position is relative to its current position.
    - The result set generally does not show changes to the underlying database that are made while it is open. The membership, order, and column values of rows are typically fixed when the result set is created.
3. **TYPE_SCROLL_SENSITIVE**
    - The result set is scrollable; its cursor can move forward or backward and can be moved to a particular row or to a row whose position is relative to its current position.
    - The result set is sensitive to changes made while it is open. If the underlying column values are modified, the new values are visible, thus providing a dynamic view of the underlying data. The membership and ordering of rows in the result set may be fixed or not, depending on the implementation.

Concurrency Types

A result set may have different update capabilities. As with scrollability, making a ResultSet object updatable increases overhead and should be done only when necessary. That said, it is often more convenient to make updates programmatically, and that can only be done if a result set is made updatable. The JDBC 2.0 core API offers two update capabilities, specified by the following constants in the ResultSet interface:

1. **CONCUR_READ_ONLY**
    - Indicates a result set that cannot be updated programmatically
    - Offers the highest level of concurrency (allows the largest number of simultaneous users). When a ResultSet object with read-only concurrency needs to set a lock, it uses a read-only lock. This allow users to read data but not to change it. Because there is no limit to the number of read-only locks that may be held on data at one time, there is no limit to the number of concurrent users unless the DBMS or driver imposes one.

2. CONCUR_UPDATABLE

- Indicates a result set that can be updated programmatically
- Reduces the level on concurrency. Updatable results sets may use write-only locks so that only one user at a time has access to a data item. This eliminates the possibility that two or more users might change the same data, thus ensuring database consistency. However, the price for this consistency is a reduced level of concurrency.

To allow a higher level of concurrency, an updatable result set may be implemented so that it uses an optimistic concurrency control scheme. This implementation assumes that conflicts will be rare and avoids using write-only locks, thereby permitting more users concurrent access to data. Before committing any updates, it determines whether a conflict has occurred by comparing rows either by value or by a version number. If there has been an update conflict between two transactions, one of the transactions will be aborted in order to maintain consistency. Optimistic concurrency control implementations can increase concurrency; however, if there are too many conflicts, they may actually reduce performance.

How locks are set is determined by what is called a transaction isolation level, which can range from not supporting transactions at all to supporting transactions that enforce very strict access rules. One example of a transaction isolation level is TRANSACTION_READ_COMMITTED, which will not allow a value to be accessed until after it has been committed. In other words, if the transaction isolation level is set to TRANSACTION_READ_COMMITTED, the DBMS does not allow dirty reads to occur. The interface Connection includes five values which represent the transaction isolation levels you can use in JDBC.

Normally, you do not need to do anything about the transaction isolation level; you can just use the default one for your DBMS. JDBC allows you to find out what transaction isolation level your DBMS is set to (using the Connection method getTransactionIsolation) and also allows you to set it to another level (using the Connection method setTransactionIsolation). Keep in mind, however, that even though JDBC allows you to set a transaction isolation level, doing so has no effect unless the driver and DBMS you are using support it.

static int	**TRANSACTION_NONE** A constant indicating that transactions are not supported.
static int	**TRANSACTION_READ_COMMITTED** A constant indicating that dirty reads are prevented; non-repeatable reads and phantom reads can occur.
static int	**TRANSACTION_READ_UNCOMMITTED** A constant indicating that dirty reads, non-repeatable reads and phantom reads can occur.
static int	**TRANSACTION_REPEATABLE_READ** A constant indicating that dirty reads and non-repeatable reads are prevented; phantom reads can occur.
static int	**TRANSACTION_SERIALIZABLE** A constant indicating that dirty reads, non-repeatable reads and phantom reads are prevented.

Setting and Rolling Back to a Savepoint

The JDBC 3.0 API adds the method Connection. Set Save point, which sets a savepoint within the current transaction. The Connection.rollback method has been overloaded to take a savepoint argument.

The example below inserts a row into a table, sets the savepoint svpt1, and then inserts a second row. When the transaction is later rolled back tosvpt1, the second insertion is undone, but the first insertion remains intact. In other words, when the transaction is committed, only the row containing ?FIRST? will be added to TAB1:

```
Statement stmt = conn.createStatement();
int rows = stmt.executeUpdate("INSERT INTO TAB1 (COL1)
VALUES " + "(?FIRST?)");
// set savepoint
Savepoint svpt1 = conn.setSavepoint("SAVEPOINT_1");
rows = stmt.executeUpdate("INSERT INTO TAB1 (COL1) " +
                                "VALUES (?SECOND?)");
...
conn.rollback(svpt1);
...
conn.commit();
```

Releasing a Savepoint

The method Connection.rollback() release Savepoint takes a Savepoint object as a parameter and removes it from the current transaction. Once a savepoint has been released, attempting to reference it in a rollback operation causes an SQLException to be thrown. Any savepoints that have been created in a transaction are automatically released and become invalid when the transaction is committed, or when the entire transaction is rolled back. Rolling a transaction back to a savepoint automatically releases and makes invalid any other savepoints that were created after the savepoint in question.

When to Call the Method rollback

As mentioned earlier, calling the method rollback aborts a transaction and returns any values that were modified to their previous values. If you are trying to execute one or more statements in a transaction and get an SQLException, you should call the method rollback to abort the transaction and start the transaction all over again. That is the only way to be sure of what has been committed and what has not been committed. Catching an SQLException tells you that something is wrong, but it does not tell you what was or was not committed. Since you cannot count on the fact that nothing was committed, calling the method rollback is the only way to be sure.

4.5 STORED PROCEDURES

A stored procedure is a group of SQL statements that form a logical unit and perform a particular task, and they are used to encapsulate a set of operations or queries to execute on a database server. For example, operations on an employee database (hire, fire, promote, lookup) could be coded as stored procedures executed by application code. Stored procedures can be compiled and executed with different parameters and results, and they may have any combination of input, output, and input/output parameters.

This simple stored procedure has no parameters. Even though most stored procedures do something more complex than this example, it serves to illustrate some basic points about them. As previously stated, the syntax for defining a stored procedure is different for each DBMS. For example, some use begin . . . end , or other keywords to indicate the beginning and ending of the procedure definition. In some DBMSs, the following SQL statement creates a stored procedure:

```
create procedure SHOW_SUPPLIERS
as
select SUPPLIERS.SUP_NAME, COFFEES.COF_NAME
from SUPPLIERS, COFFEES
where SUPPLIERS.SUP_ID = COFFEES.SUP_ID
order by SUP_NAME
```

The following code puts the SQL statement into a string and assigns it to the variable createProcedure, which we will use later:

```
String createProcedure = "create procedure SHOW_SUPPLIERS "
+ "as " + "select SUPPLIERS.SUP_NAME, COFFEES.COF_NAME
" + "from SUPPLIERS, COFFEES " + "where SUPPLIERS.SUP_ID
= COFFEES.SUP_ID " + "order by SUP_NAME";
```

The following code fragment uses the Connection object con to create a Statement object, which is used to send the SQL statement creating the stored procedure to the database:

```
        Statement stmt = con.createStatement();
        stmt.executeUpdate(createProcedure);
```

The procedure SHOW_SUPPLIERS is compiled and stored in the database as a database object that can be called, similar to the way you would call a method.

Calling a Stored Procedure from JDBC

JDBC allows you to call a database stored procedure from an application written in the Java programming language. The first step is to create a Callable Statement object. As with Statement and Prepared Statement objects, this is done with an open Connection object. A callable Statement object contains a call to a stored procedure; it does not contain the stored procedure itself. The first line of code below creates a call to the stored procedure SHOW_SUPPLIERS using the connection con. The part that is enclosed in curly braces is the escape syntax for stored procedures. When the driver encounters "{call SHOW_SUPPLIERS}", it will translate this escape syntax into the native SQL used by the database to call the stored procedure named SHOW_SUPPLIERS.

```
CallableStatement cs = con.prepareCall
                                        ("{call SHOW_SUPPLIERS}");
ResultSet rs = cs.executeQuery();
```

The ResultSet rs will be similar to the following:

```
SUP_NAME                COF_NAME
----------------        ----------------------
Reliance                Colombian
Reliance                Colombian_Decaf
Tata                    French_Roast
Tata                    French_Roast_Decaf
Sahara                  Espresso
```

Note that the method used to execute cs is execute Query because cs calls a stored procedure that contains one query and thus produces one result set. If the procedure had contained one update or one DDL statement, the method execute

Update would have been the one to use. It is sometimes the case, however, that a stored procedure contains more than one SQL statement, in which case it will produce more than one result set, more than one update count, or some combination of result sets and update counts. In this case, where there are multiple results, the method execute should be used to execute the Callable Statement.

The class Callable Statement is a subclass of Prepared Statement, so a Callable Statement object can take input parameters just as a Prepared Statement object can. In addition, a Callable Statement object can take output parameters, or parameters that are for both input and output. INOUT parameters and the method execute are used rarely.

Example - Write a java code to accept the query from user at command line argument. Then process this query and using ResultSet and ResultSetMetaData class. Display the result on the screen with proper title of the fields and the fields separator should be '|' (pipe) symbol.

```
import java.sql.*;
import java.io.*;
class CLAQuery
{
        public static void main(String args[])throws SQLException
        {
                Connection con;
                int i;
                String str;
                try
                {
                Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
                Connection con=DriverManager.
                                getConnection("jdbc:odbc:sidd");

                Statement st=con.createStatement();
                BufferedReader br=new BufferedReader((
                        new InputStreamReader(System.in)));
                System.out.println("Enter The Query");
                str=br.readLine();

                ResultSet rs=st.executeQuery(str);
                ResultSetMetaData md=rs.getMetaData();
                int num=md.getColumnCount();
                System.out.print("\n");
                for(i=1;i<=num;i++)
                        System.out.print(md.getColumnName(i)+"\t\t");
                System.out.println("");
                while(rs.next())
                {
                        for(i=1;i<=num;i++)
                                System.out.print(rs.getString(i)+"\t\t");
                        System.out.print("\n");
                }
                }
                catch(Exception e)
                {
                System.out.println("Error1:"+e.getMessage());
                }
        }
}
```

Example - Create a JFrame, which will add data of Friends (Name, DOB, Address and Tel. No.) to the database and on click of Show All button displays the records that are there in the database.

```
import java.awt.event.*;
import javax.swing.*;
import java.awt.*;
import java.sql.*;
import java.lang.*;
class FriendDatabase extends JFrame implements ActionListener
{
        JLabel lblDob,lblName,lblDetails,lblPhno,lblHead;
        JTextField txtName,txtDob,txtPhno;
        JTextArea txtDetails;
        JButton cmdAdd,cmdShowAll,cmdExit;
        GridBagLayout gbl=new GridBagLayout();
        GridBagConstraints gbc=new GridBagConstraints();
        Container con=getContentPane();

        public FriendDatabase()
        {
                lblHead=new JLabel("Information");
                lblName=new JLabel("Friend Name");
```

```
lblDob=new JLabel("DOB");
lblDetails=new JLabel("Address");
lblPhno=new JLabel("Phno");

txtName=new JTextField("");
txtDob=new JTextField("");
txtDetails=new JTextArea(2,2);
int v=ScrollPaneConstants.
        VERTICAL_SCROLLBAR_AS_NEEDED;
int h=ScrollPaneConstants.
        HORIZONTAL_SCROLLBAR_AS_NEEDED;
JScrollPane jsp=new JScrollPane(txtDetails,v,h);
txtPhno=new JTextField("");

cmdAdd=new JButton("Add");
cmdAdd.addActionListener(this);
cmdShowAll=new JButton("Show All");
cmdShowAll.addActionListener(this);
cmdExit=new JButton("Exit");
cmdExit.addActionListener(this);

con.setLayout(new GridBagLayout());
gbc.fill=GridBagConstraints.BOTH;
gbc.weightx=1.0;
gbc.insets=new Insets(20,20,20,20);

gbc.gridx=1;gbc.gridy=0;
con.add(lblHead,gbc);
gbc.gridx=0;gbc.gridy=1;
con.add(lblName,gbc);

gbc.gridx=1;gbc.gridy=1;
con.add(txtName,gbc);

gbc.gridx=0;gbc.gridy=2;
con.add(lblDob,gbc);

gbc.gridx=1;gbc.gridy=2;
con.add(txtDob,gbc);
```

```
        gbc.gridx=0;gbc.gridy=3;
        con.add(lblDetails,gbc);

        gbc.gridx=1;gbc.gridy=3;
        con.add(jsp,gbc);

        gbc.gridx=0;gbc.gridy=4;
        con.add(lblPhno,gbc);

        gbc.gridx=1;gbc.gridy=4;
        con.add(txtPhno,gbc);

        gbc.fill=GridBagConstraints.NONE;
        gbc.gridx=0;gbc.gridy=5;
        con.add(cmdAdd,gbc);

        gbc.gridx=1;gbc.gridy=5;
        con.add(cmdShowAll,gbc);

        gbc.gridx=2;gbc.gridy=5;
        con.add(cmdExit,gbc);
}
public void actionPerformed(ActionEvent ae)
{
        Object obj=ae.getSource();
        if(obj==cmdAdd)
        {
        Connection con;
        try
        {
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        Connection con=
        DriverManager.getConnection("jdbc:odbc:sidd");
        String query="insert into
                FriendDatabase values(?,?,?,?)";
        PreparedStatement ps;
        ps=con.prepareStatement(query);
        ps.setString(1,txtName.getText());
        ps.setDate(2,Date.valueOf(txtDob.getText()));
        ps.setString(3,txtDetails.getText());
        String str=txtPhno.getText();
```

```
                int m=Integer.parseInt(str);
                ps.setInt(4,m);
                ps.executeUpdate();
                JOptionPane.showMessageDialog(this,
                        "Record Entered", "FriendsDatabase",
                        JOptionPane.ERROR_MESSAGE);

                txtName.setText("");
                txtDob.setText("");
                txtDetails.setText("");
                txtPhno.setText("");
                ps.close();
                con.close();
                }
                catch(Exception e)
                {
                System.out.println("Error:"+e.getMessage());
                }
                }
                if(obj==cmdExit)
                {
                        System.exit(0);
                }
                if(obj==cmdShowAll)
                {
                        ShowAll as=new ShowAll(this,true);
                }
        }

        public static void main(String args[])throws SQLException
        {
                FriendDatabase b=new FriendDatabase();
                b.setSize(650,650);
                b.setVisible(true);
        }
}
class ShowAll extends JDialog
{
Object[][] rows={ {"","","",""},{"","","",""},{"","","",""},{"","","",""}};
        Object[] cols={"Name","Dob","Details","Phno"};
```

```
int i,j;
JScrollPane p;
JTable t;
Connection con;
public ShowAll(JFrame f,boolean m)
{

        super(f,m);
        try
        {
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        Connection con=DriverManager.
                        getConnection("jdbc:odbc:sidd");
        Statement st=con.createStatement();
        ResultSet rs=st.executeQuery
                        ("Select * from FriendDatabase");
        i=0;
        while(rs.next())
        {
                for(j=0;j<=3;j++)
                {
                        Object obj=rs.getObject(j+1);
                        rows[i][j]=obj.toString();
                        System.out.print(""+obj.toString()+"\t");
                }
                        i++;
        }
        t=new JTable(rows,cols);
        int v1=ScrollPaneConstants.
                VERTICAL_SCROLLBAR_ALWAYS;
        int h1=ScrollPaneConstants.
                HORIZONTAL_SCROLLBAR_ALWAYS;
        p=new JScrollPane(t,v1,h1);
        Container con1=getContentPane();
        con1.add(p,BorderLayout.CENTER);
        setSize(300,300);
        setVisible(true);
        }
        catch(Exception e)
        {
                System.out.println("Error2:"+e.getMessage());
                }
        }
}
```

4.6 SUMMARY

- An exception that provides information on a database access error or other errors.
- A PreparedStatement object is more convenient for sending SQL statements to the database.
- A transaction is a set of one or more statements that are executed together as a unit, so either all of the statements are executed, or none of the statements is executed.
- A stored procedure is a group of SQL statements that form a logical unit and perform a particular task.

4.7 UNIT END EXERCISE

1) State and explain any two exception classes related to JDBC?
2) Explain with an example how to create and use Prepared Statement.
3) How SQL joins are executed using JDBC?
4) Explain how a Stored Procedure can be called from JDBC.

4.8 FURTHER READING

- Ivan Bayross, Web Enabled Commercial Applications Development Using Java 2, BPB Publications, Revised Edition, 2006
- Joe Wigglesworth and Paula McMillan, Java Programming: Advanced Topics, Thomson Course Technology (SPD), Third Edition, 2004
- Cay S. Horstmann, Gary Cornell, Core Java™ 2: Volume II– Advanced Features Prentice Hall PTR, 2001

❖❖❖❖

5

THREADS AND MULTITHREADING

Unit Structure:

5.0 Objectives
5.1 Introduction
5.2 Life Cycle of a Thread
5.3 Thread Priorities
5.4 Creating a Thread
5.5 Using Multithreading
5.6 Thread Deadlock
5.7 Summary
5.8 Unit end exercise
5.9 Further Reading

5.0 OBJECTIVES

The objective of this chapter is to learn how threads are created and used in Java. In this chapter we will learn the lifecycle of the thread and how various properties can be set of the thread.

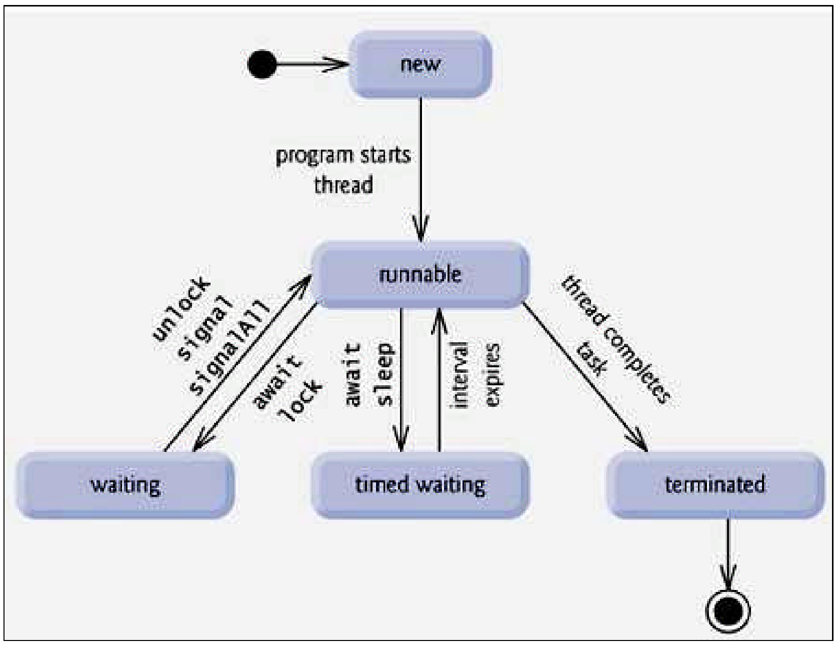
5.1. INTRODUCTION

Java provides built-in support for *multithreaded programming*. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. A multithreading is a specialized form of multitasking. Multitasking threads require less overhead than multitasking processes.

Another term related to threads is: **process:** A process consists of the memory space allocated by the operating system that can contain one or more threads. A thread cannot exist on its own; it must be a part of a process. A process remains running until all of the non-daemon threads are done executing. Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

5.1 LIFE CYCLE OF A THREAD:

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. Following diagram shows complete life cycle of a thread.



Above mentioned stages are explained here:

- **New:** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
- **Runnable:** After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- **Waiting:** Sometimes a thread transitions to the waiting state while the thread waits for another thread to perform a task.A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed waiting:** A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transition back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated:** A runnable thread enters the terminated state when it completes its task or otherwise terminates.

5.2 THREAD PRIORITIES:

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependentant.

5.3 CREATING A THREAD:

Java defines two ways in which this can be accomplished:

- You can implement the Runnable interface.
- You can extend the Thread class, itself.

5.3.1 Create Thread by Implementing Runnable:

The easiest way to create a thread is to create a class that implements the **Runnable** interface. To implement Runnable, a class need only implement a single method called **run( )**, which is declared like this:

```
public void run( )
```

You will define the code that constitutes the new thread inside run() method. It is important to understand that run() can call other methods, use other classes, and declare variables, just like the main thread can.

After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class. Thread defines several constructors. The one that we will use is shown here:

```
Thread(Runnable threadOb, String threadName);
```

Here *threadOb* is an instance of a class that implements the Runnable interface and the name of the new thread is specified by *threadName*. After the new thread is created, it will not start running until you call its **start( )** method, which is declared within Thread. The start( ) method is shown here:

```
void start( );
```

Example:

```
public class RunnableThread implements Runnable
{
        private int countDown = 5;
        public String toString()
        {
                return "#" + Thread.currentThread().getName()
                                        +": " + countDown;
        }
        public void run()
        {
                while(true)
                {
                        System.out.println(this);
                        if(--countDown == 0) return;
                }
        }
        public static void main(String[] args)
        {
                for(int i = 1; i <= 5; i++)
                        new Thread(new RunnableThread(), "" +
                i).start();
        }
}
```

5.3.2 Create Thread by Extending Thread:

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

The extending class must override the **run( )** method, which is the entry point for the new thread. It must also call **start( )** to begin execution of the new thread.

Example:
Here is the preceding program rewritten to extend Thread:

```
public class SimpleThread extends Thread
{
        private int countDown = 5;
        private static int threadCount = 0;
        public SimpleThread()
        {
```

```
		super("" + ++threadCount); // Store the thread name
		start();
	}
	public String toString()
	{
		return "#" + getName() + ": " + countDown;
	}
	public void run()
	{
		while(true)
		{
			System.out.println(this);
			if(--countDown == 0) return;
		}
	}
	public static void main(String[] args)
	{
		for(int i = 0; i < 5; i++)
			new SimpleThread();
	}
}
```

Thread Methods: Following is the list of important medthods available in the Thread class.

SN	Methods
1	**public void start() -** Starts the thread in a separate path of execution, then invokes the run() method on this Thread object.
2	**public void run() -** If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.
3	**public final void setName(String name) -** Changes the name of the Thread object. There is also a getName() method for retrieving the name.
4	**public final void setPriority(int priority) -** Sets the priority of this Thread object. The possible values are between 1 and 10.

5	**public final void setDaemon(boolean on) -** A parameter of true denotes this Thread as a daemon thread.
6	**public final void join(long millisec) -** The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.
7	**public void interrupt() -** Interrupts this thread, causing it to continue execution if it was blocked for any reason.
8	**public final boolean isAlive() -** Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.

The previous methods are invoked on a particular Thread object. The following methods in the Thread class are static. Invoking one of the static methods performs the operation on the currently running thread

SN	Methods
1	**public static void yield() -** Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled
2	**public static void sleep(long millisec)** - Causes the currently running thread to block for at least the specified number of milliseconds
3	**public static boolean holdsLock(Object x)** - Returns true if the current thread holds the lock on the given Object.
4	**public static Thread currentThread() -** Returns a reference to the currently running thread, which is the thread that invokes this method.
5	**public static void dumpStack() -** Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application.

5.4 USING MULTITHREADING:

The key to utilizing multithreading support effectively is to think concurrently rather than serially. For example, when you have two subsystems within a program that can execute concurrently, make them individual threads.

With the careful use of multithreading, you can create very efficient programs. A word of caution is in order, however: If you create too many threads, you can actually degrade the performance of your program rather than enhance it.

Remember, some overhead is associated with context switching. If you create too many threads, more CPU time will be spent changing contexts than executing your program!

5.4.1 Thread Synchronization

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

The synchronized keyword in Java creates a block of code referred to as a critical section. Every Java object with a critical section of code gets a lock associated with the object. To enter a critical section, a thread needs to obtain the corresponding object's lock.

This is the general form of the synchronized statement:

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

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

5.4.2 Interthread Communication

Consider the classic queuing problem, where one thread is producing some data and another is consuming it. To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data.

In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce. Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on. Clearly, this situation is undesirable.

To avoid polling, Java includes an elegant interprocess communication mechanism via the following methods:

- **wait( ):** This method tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ).
- **notify( ):** This method wakes up the first thread that called wait( ) on the same object.
- **notifyAll( ):** This method wakes up all the threads that called wait( ) on the same object.c The highest priority thread will run first.

These methods are implemented as **final** methods in Object, so all classes have them. All three methods can be called only from within a **synchronized** context.

These methods are declared within Object. Various forms of wait( ) exist that allow you to specify a period of time to wait.

5.5 THREAD DEADLOCK

A special type of error that you need to avoid that relates specifically to multitasking is deadlock, which occurs when two threads have a circular dependency on a pair of synchronized objects.

For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete.

Example:

```
class A
{
        synchronized void foo(B b)
        {
                String name = Thread.currentThread().getName();
```

```
		System.out.println(name + " entered A.foo");
		try{
			Thread.sleep(1000);
		}
		catch(Exception e) {
			System.out.println("A Interrupted");
		}
		System.out.println(name + " trying to call B.last()");
		b.last();
	}
	synchronized void last()
	{
		System.out.println("Inside A.last");
	}
}
class B
{
	synchronized void bar(A a)
	{
		String name = Thread.currentThread().getName();
		System.out.println(name + " entered B.bar");
		try{
			Thread.sleep(1000);
		}
		catch(Exception e) {
			System.out.println("B Interrupted");
		}
		System.out.println(name + " trying to call A.last()");
		a.last();
	}
	synchronized void last()
	{
		System.out.println("Inside A.last");
	}
}
class Deadlock implements Runnable
{
	A a = new A();
	B b = new B();
	Deadlock()
```

```
        {
                Thread.currentThread().setName("MainThread");
                Thread t = new Thread(this, "RacingThread");
                t.start();
                a.foo(b); // get lock on a in this thread.
                System.out.println("Back in main thread");
        }
        public void run()
        {
                b.bar(a); // get lock on b in other thread.
                System.out.println("Back in other thread");
        }
        public static void main(String args[])
        {
                new Deadlock();
        }
}
```

Because the program has deadlocked, you need to press CTRL-C to end the program. You can see a full thread and monitor cache dump by pressing CTRL-BREAK on a PC. You will see that RacingThread owns the monitor on **b**, while it is waiting for the monitor on **a**. At the same time, MainThread owns **a** and is waiting to get **b**. This program will never complete. As this example illustrates, if your multithreaded program locks up occasionally, deadlock is one of the first conditions that you should check for.

Ordering Locks:

Acommon threading trick to avoid the deadlock is to order the locks. By ordering the locks, it gives threads a specific order to obtain multiple locks.

5.7 SUMMARY

- A multithreaded program contains two or more parts that can run concurrently.
- Each part of such a program is called a thread, and each thread defines a separate path of execution.
- A thread goes through various stages in its life cycle - New, Runnable, Waiting, Timed waiting and Terminated.
- Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

- Java defines two ways in which this can be accomplished:
    - You can implement the Runnable interface.
    - You can extend the Thread class, itself.
- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.

5.8 UNIT END EXERCISE

1) Write a short note on Life Cycle of a Thread?
2) Explain with an example how a Thread can be created using Runnable Class.
3) State and explain the methods used for Thread Synchronization?
4) Explain with an example how a Thread can be created using Thread Class.
5) Write a program to display rotating line. It should start and stop rotating on click of buttons. The direction should be controlled by 2 radiobuttons – ClockWise & AntiClockWise.
6) Write a program to show bouncing ball. The ball should bounce when you click on button.
7) Create a class FileCopy to copy contents of 1 file to other file. The names of files should be accepted from user. Implement threading so that many files can be copied simultaneously.

5.9 FURTHER READING

- The Java Tutorials of Sun Microsystems Inc.
- Joe Wigglesworth and Paula McMillan, Java Programming: Advanced Topics, Thomson Course Technology (SPD), Third Edition, 2004
- Cay S. Horstmann, Gary Cornell, Core Java™ 2: Volume II–Advanced Features Prentice Hall PTR, 2001

❖❖❖❖

6

NETWORKING BASICS

Unit Structure:

6.0 Objectives
6.1 Introduction to Networking
6.2 Working with URL’s
6.3 URLConnection
6.4 TCP/IP Sockets
6.5 UDP Sockets
6.6 Summary
6.7 Unit end exercise
6.8 Further Reading

6.1 INTRODUCTION TO NETWORKING

1) Sockets: Different programs can communicate through communication channels called sockets. Sockets can be considered as the end points in the communication link.

2) Client/Server: A server is a system that has some resource that can be shared. There are compute servers, which provide computing power, print servers which manage a collection of printers and webservers which stores web pages. A client is simply any other system that wants to gain access in a particular server.

3) Reserved Sockets: TCP/IP reserves the lower 1024 codes for specific protocols. Port number 21 is used for FTP (File Transfer Protocol), 23 is used for Telnet, 25 is used for E-mail, 80 is used for HTTP etc.

4) Proxy Servers: A proxy server speaks the client side of a protocol to another server. This is often required when clients have contained restrictions on which servers they can connect to. Thus a client would connect to a proxyserver and the proxyserver would in turn communicate with the client.

5) Internet Addressing: An internet address is a number that uniquely identifies each computer on the net. Every computer on the internet has an address. There are 32-bits in an IP address and we often refer to them as a sequence of 4 numbers between 0 and 255 separated by dots.

6) Domain Naming Service (DNS): A domain name describes a machine location is a namespace from right to left. E.g. www.yahoo.com is in the com domain, it is called yahoo after the company name and www is the name of specific computer that is yahoo's web server www corresponds to the right most no. in the equivalent IP address.

6.2 WORKING WITH URL'S

The class URL represents a Uniform Resource Locator, a pointer to a resource on the World Wide Web. A resource can be something as simple as a file or a directory, or it can be a reference to a more complicated object such as a query to a database or to a search engine. URL's begin with a protocol specification such as HTTP of FTP, followed be "://" and the host name along with optional file and port no. Java's URL class is used to create a URL object, which has several constructors and each throws a MalformedURLException.

Constructor Summary
URL(String spec) Creates a URL object from the String representation.
URL(String protocol, String host, int port, String file) Creates a URL object from the specified protocol, host, port number, and file.
URL(String protocol, String host, String file) Creates a URL from the specified protocol name, host name, and file name.

Method Summary	
String getAuthority()	Gets the authority part of this URL.
Object getContent()	Gets the contents of this URL.
String getFile()	Gets the file name of this URL.
String getHost()	Gets the host name of this URL, if applicable.

String getPath()	Gets the path part of this URL.
int getPort()	Gets the port number of this URL.
String getProtocol()	Gets the protocol name of this URL.
String getQuery()	Gets the query part of this URL.
String getRef()	Gets the anchor (also known as the "reference") of this URL.
String getUserInfo()	Gets the userInfo part of this URL.
URLConnection openConnection()	Returns a URLConnection object that represents a connection to the remote object referred to by the URL.
InputStream openStream()	Opens a connection to this URL and returns an InputStream for reading from that connection.

6.3 URLCONNECTION:

URL connection is a general purpose class for accessing the attributes of a remote resource. Once a connection is established with the remote server, we can inspect the properties of the remote object before actually transporting it locally. We can obtain an object of URL connection with the help of openConnection() of the URL class.

Constructor Summary
URLConnection(URL url) Constructs a URL connection to the specified URL.

Method Summary	
abstract void connect()	Opens a communications link to the resource referenced by this URL, if such a connection has not already been established.
String getContentEncoding()	Returns the value of the content-encoding header field.
int getContentLength()	Returns the value of the content-length header field.

String getContentType()	Returns the value of the content-type header field.
long getDate()	Returns the value of the date header field.
long getExpiration()	Returns the value of the expires header field.
String getHeaderField(int n)	Returns the value for the nth header field.
String getHeaderField(String name)	Returns the value of the named header field.
InputStream getInputStream()	Returns an input stream that reads from this open connection.
long getLastModified()	Returns the value of the last-modified header field.
OutputStream getOutputStream()	Returns an output stream that writes to this connection.
URL getURL()	Returns the value of this URLConnection's URL field.

Example: Define a class that displays information about a file URL like its type, encoding, length, dates of creation, last modification and expiry. Additionally the class should display the request method, response message and the response code for a Web URL.

```
import java.net.URL;
import java.net.URLConnection;
import java.net.HttpURLConnection;
import java.util.Date;
class URLDemo
{
        long d;
        public static void main(String args[])throws Exception
        {
                URL u=new URL("http://localhost:8080/index.html");
                URLConnection uc=u.openConnection();
                HttpURLConnection huc=(HttpURLConnection)uc;
                Date d=new Date(uc.getDate());
                System.out.println("File Name
                                ="+u.getFile());
                System.out.println("Host Name
                                ="+u.getHost());
                System.out.println("Path Name
                                ="+u.getPath());
                System.out.println("Port Name
                                ="+u.getPort());
                System.out.println("Protocol Name
                                ="+u.getProtocol());
                System.out.println("Reference Name
                                ="+u.getRef());
                System.out.println("User Info
                                ="+u.getUserInfo());
                System.out.println("Content Name
                                ="+u.getContent());
                System.out.println("Authority Name
                                ="+u.getAuthority());
                System.out.println("Content Type
                                ="+uc.getContentType());
                System.out.println("Length
                                ="+uc.getContentLength());
                System.out.println("Expiration Date
                                ="+uc.getExpiration());
                System.out.println("Encoding Type
                                ="+uc.getContentEncoding());
                System.out.println("Last Modified Date
                                ="+uc.getLastModified());
                System.out.println("Date
                                ="+d.toString());
                System.out.println("Request Method
                                ="+huc.getRequestMethod());
                 System.out.println("Response Message
                                ="+huc.getResponseMessage());
                System.out.println("Response Code
                                ="+huc.getResponseCode());
        }
}
```

Example: Define a class to download a file from the Internet and either copy it as a file on the local machine, or output it to the screen.

```
import java.net.URL;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.IOException;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;

class Download
{
        public static void main(String args[])throws Exception
        {
        int b;
        char c;
        if(args.length==0)
                throw new Exception("Invalid Number of argument");
        URL u=new URL(args[0]);
        InputStream is=u.openStream();
        OutputStream os;
        if(args.length==1)
        {
                while ((b=is.read())!=-1)
                                System.out.print((char)b);
        }
        else
        {       File f2=new File(args[1]);
                os=new FileOutputStream(f2);
                if(f2.exists()==true)
                {
                        System.out.println("This file exists");
                        System.exit(0);
                }
                else
                {
                        while ((b=is.read())!=-1)
                                os.write(b);
                }
        }
        }//main
}//class
```

6.4 TCP/IP SOCKETS:

These are used to implement reliable, persistent, point to point, stream based connections between hosts on the internet. The program at the 2 ends of the socket can write to and read from the sockets.

Advantages:

- Easy to use.
- Used for 2 way communication.
- No parts of the transmission are lost.

Disadvantages:

- It requires setup time and shutdown time.
- Delivery is slower than datagram sockets.

There are two types of TCP/IP sockets. One is used for Servers and other is used for clients.

Socket Class:

The socket class is designed to connect to a server socket. The creation of to a socket object implicitly establishes a connection between client and server.

Constructor Summary
Socket(InetAddress address, int port) Creates a stream socket and connects it to the specified port number at the specified IP address.
Socket(InetAddress address, int port, InetAddress localAddr, int localPort) Creates a socket and connects it to the specified remote address on the specified remote port.
Socket(String host, int port) Creates a stream socket and connects it to the specified port number on the named host.
Socket(String host, int port, InetAddress localAddr, int localPort) Creates a socket and connects it to the specified remote host on the specified remote port.

Method Summary	
void close()	Closes this socket.
void connect (SocketAddress endpoint)	Connects this socket to the server.
void connect (SocketAddress endpoint, int timeout)	Connects this socket to the server with a specified timeout value.
InetAddress getInetAddress()	Returns the address to which the socket is connected.
InputStream getInputStream()	Returns an input stream for this socket.
InetAddress getLocalAddress()	Gets the local address to which the socket is bound.
int getLocalPort()	Returns the local port to which this socket is bound.
OutputStream getOutputStream()	Returns an output stream for this socket.
int getPort()	Returns the remote port to which this socket is connected.
int getReceiveBufferSize()	Gets the value of the SO_RCVBUF option for this Socket, that is the buffer size used by the platform for input on this Socket.
boolean isClosed()	Returns the closed state of the socket.
boolean isConnected()	Returns the connection state of the socket.

ServerSocket Class:

The ServerSocket class is designed to be a listener which waits for the clients to connect. When you create a ServerSocket it will register itself with the system as having as interest in client connection.

Constructor Summary
ServerSocket(int port) Creates a server socket, bound to the specified port.
ServerSocket(int port, int backlog) Creates a server socket and binds it to the specified local port number, with the specified backlog.

Method Summary	
Socket accept()	Listens for a connection to be made to this socket and accepts it.
void close()	Closes this socket.
InetAddress getInetAddress()	Returns the local address of this server socket.
int getLocalPort()	Returns the port on which this socket is listening.
int getReceiveBufferSize()	Gets the value of the SO_RCVBUF option for this ServerSocket, that is the proposed buffer size that will be used for Sockets accepted from this ServerSocket.
boolean isClosed()	Returns the closed state of the ServerSocket.

Example: Write a simple server that reports the current time (in textual form) to any client that connects. This server should simply output the current time and close the connection, without reading anything from the client. You need to choose a port number that your service listens on.

```
import java.net.*;
import java.io.*;
import java.util.*;
class TimeServer
{
        public static void main(String args[])throws Exception
        {
                ServerSocket s=new ServerSocket(1234);
                Socket c=s.accept();
                Calendar calendar = new GregorianCalendar();
```

```
            PrintWriter out=new PrintWriter(c.getOutputStream());
            out.println(new Date());
            out.println("Time :");
            out.print(
                    calendar.get(Calendar.HOUR)+"HRS."
                    + calendar.get(Calendar.MINUTE)+"MIN."
                    + calendar.get(Calendar.SECOND)+"SEC");
            out.flush();
            s.close();
            c.close();
        }
}

import java.net.*;
import java.io.*;
class TimeClient
{
        public static void main(String args[])throws Exception
        {
        Socket c=new Socket(InetAddress.getLocalHost(),1234);
         BufferedReader br=new BufferedReader(new

        InputStreamReader(c.getInputStream()));
        String userInput;

        while((userInput=br.readLine())!=null)
            {
                    System.out.println(userInput);
            }
        c.close();
        }
}
```

6.5 UDP SOCKETS:

Datagram sockets are not connection oriented. Here we send self contained packets of data. Each packet contains information that identifies in addition to the content of the message.

Advantages:

- Does not require setup time and shutdown time.
- Delivery is faster than TCP/IP sockets.

Disadvantages:

- Datagram sockets cannot be used for two way communication.
- Parts of the transmission are lost.

Datagram Socket Class:

A datagram socket is the sending or receiving point for a packet delivery service. Each packet send or receive on a datagramsocket is individually addressed or routed. Multiple packets send from 1 machine may be routed differently and may arrive in any order.

Constructor Summary
DatagramSocket() Constructs a datagram socket and binds it to any available port on the local host machine.
DatagramSocket(int port) Constructs a datagram socket and binds it to the specified port on the local host machine.
DatagramSocket(int port, InetAddress laddr) Creates a datagram socket, bound to the specified local address.

Method Summary	
void close()	Closes this datagram socket.
void connect(InetAddress address, int port)	Connects the socket to a remote address for this socket.
InetAddress getInetAddress()	Returns the address to which this socket is connected.
InetAddress getLocalAddress()	Gets the local address to which the socket is bound.
int getLocalPort()	Returns the port number on the local host to which this socket is bound.
int getPort()	Returns the port for this socket.

boolean isClosed()	Returns wether the socket is closed or not.
boolean isConnected()	Returns the connection state of the socket.
void receive(DatagramPacket p)	Receives a datagram packet from this socket.
void send(DatagramPacket p)	Sends a datagram packet from this socket.

DatagramPacket:

DatagramPackets are use to implement a connection less packet delivery service. Each message is routed from one machine to another based on the information contained within that packet.

Constructor Summary
DatagramPacket(byte[] buf, int length) Constructs a DatagramPacket for receiving packets of length length.
DatagramPacket(byte[] buf, int length, InetAddress address, int port) Constructs a datagram packet for sending packets of length length to the specified port number on the specified host.
DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port) Constructs a datagram packet for sending packets of length length with offset set to the specified port number on the specified host.

Method Summary	
InetAddress getAddress()	Returns the IP address of the machine to which this datagram is being sent or from which the datagram was received.
byte[]getData()	Returns the data buffer.
int getLength()	Returns the length of the data to be sent or the length of the data received.

Int getOffset()	Returns the offset of the data to be sent or the offset of the data received.
int getPort()	Returns the port number on the remote host to which this datagram is being sent or from which the datagram was received.
void setAddress (InetAddress iaddr)	Sets the IP address of the machine to which this datagram is being sent.
void setData(byte[] buf)	Set the data buffer for this packet.
void setData(byte[] buf, int offset, int length)	Set the data buffer for this packet.
void setLength(int length)	Set the length for this packet.
void setPort(int iport)	Sets the port number on the remote host to which this datagram is being sent.

Example: Write a Java program for datagram communication between two client machines using Unreliable Datagram Protocol.

```
import java.net.*;
import java.io.*;
class DataGramServer{
        public static DatagramSocket ds;
        public static byte buffer[]=new byte[1024];
        public static int cp=1510,sp=1511;
        public static void main(String args[])throws Exception
        {
                ds=new DatagramSocket(sp);
                BufferedReader br=new BufferedReader(
                        new InputStreamReader(System.in));
                InetAddress ia=InetAddress.getByName(args[0]);
                while(true)
                {
                        String str=br.readLine();
                        buffer=str.getBytes();
                        ds.send(new DatagramPacket
                                        (buffer,str.length(),ia,cp));
                        if(str.length()==0)
                        {
                                ds.close();
                                break;
                        }
                }
        }//main
}//class

import java.net.*;
import java.io.*;
class DataGramClient
{
        public static DatagramSocket ds;
        public static byte buffer[]=new byte[1024];
        public static int cp=1510,sp=1511;
        public static void main(String args[])throws Exception
        {
                ds=new DatagramSocket(cp);
                System.out.print("Client is waiting for
                                        server to send data....");
                while(true)
                {
                        DatagramPacket dp=new
                                DatagramPacket(buffer,buffer.length);
                        ds.receive(dp);
                        String str=new
                                String(dp.getData(),0,dp.getLength());
                        if(dp.getLength()==0)
                                        break;
                        System.out.println(str);
                }
        }//main
}//class
```

6.6 SUMMARY

- URL represents a Uniform Resource Locator, a pointer to a resource on the World Wide Web. A resource can be something as simple as a file or a directory.
- URLConnection is a general purpose class for accessing the attributes of a remote resource.
- Socket class is designed to connect to a server socket.
- ServerSocket class is designed to be a listener which waits for the clients to connect.
- A datagram socket is the sending or receiving point for a packet delivery service.
- DatagramPackets are use to implement a connection less packet delivery service.

6.7 UNIT END EXERCISE

1) How can you get the IP Address of a machine from its hostname?
2) What does an object of type URL represent and how is it used?
3) What is a ServerSocket and how it is used?
4) Explain the class URLConnection.
5) State the useof the following methods with their parameters:
    a. InetAddress.getByName()
    b. DatagramSocket.send()
6) Explain with an example the procedure to connect a client to the server for communication using DatagramSocket and DatagramPacket.
7) Write a java program to accept a URL and display its length of contents and date of creation.
8) Write a java program to accept a URL and display its contents only if the file type is HTML/ASP/JSP. (Hint : Use string function to check whether URL ends with html/asp/jsp)
9) Write a Java program for datagram communication between two client machines using Unreliable Datagram Protocol such that when client passes any string, the server returns the length.
10) Write only TCP Server that listens to port 2345 to accept client connections.

6.8 FURTHER READING

- Ivan Bayross, Web Enabled Commercial Applications Development Using Java 2, BPB Publications, Revised Edition, 2006
- Joe Wigglesworth and Paula McMillan, Java Programming: Advanced Topics, Thomson Course Technology (SPD), Third Edition, 2004
- Cay S. Horstmann, Gary Cornell, Core Java™ 2: Volume II– Advanced Features Prentice Hall PTR, 2001

❖❖❖❖

7

ADVANCE NETWORKING

Unit Structure:

7.0 Objectives
7.1 Programmatic Access to Network Parameters
7.2 What Is a Network Interface?
7.3 Retrieving Network Interfaces
7.4 Listing Network Interface Addresses
7.5 Network Interface Parameters
7.6 Summary
7.7 Unit end exercise
7.8 Further Reading

7.0 OBJECTIVES

The objective of this chapter is to learn the advance concepts of Networking. Here we will learn how to access network parameters and how to use the network interface for programming.

7.1 PROGRAMMATIC ACCESS TO NETWORK PARAMETERS

Systems often run with multiple active network connections, such as wired Ethernet, 802.11 b/g (wireless), and bluetooth. Some applications might need to access this information to perform the particular network activity on a specific connection.

The java.net.NetworkInterface class provides access to this information.This chapter guides you through some of the more common uses of this class and provides examples that list all the network interfaces on a machine as well as their IP addresses and status.

7.2 WHAT IS A NETWORK INTERFACE?

A network interface is the point of interconnection between a computer and a private or public network. A network interface is generally a network interface card (NIC), but does not have to have a physical form. Instead, the network interface can be implemented in software. For example, the loopback interface (127.0.0.1 for IPv4 and ::1 for IPv6) is not a physical device but a piece of software simulating a network interface. The loopback interface is commonly used in test environments.

The java.net.NetworkInterface class represents both types of interfaces. NetworkInterface is useful for a multi-homed system, which is a system with multiple NICs. Using NetworkInterface, you can specify which NIC to use for a particular network activity.

For example, assume you have a machine with two configured NICs, and you want to send data to a server. You create a socket like this:

```
Socket soc = new java.net.Socket();
soc.connect(new InetSocketAddress(address, port));
```

To send the data, the system determines which interface is used. However, if you have a preference or otherwise need to specify which NIC to use, you can query the system for the appropriate interfaces and find an address on the interface you want to use. When you create the socket and bind it to that address, the system uses the associated interface. For example:

```
NetworkInterface nif = NetworkInterface.getByName("bge0");
Enumeration<InetAddress> nifAddresses = nif.getInetAddresses();
Socket soc = new java.net.Socket();
soc.bind(new InetSocketAddress(nifAddresses.nextElement(), 0));
soc.connect(new InetSocketAddress(address, port));
```

You can also use NetworkInterface to identify the local interface on which a multicast group is to be joined. For example:

```
NetworkInterface nif = NetworkInterface.getByName("bge0");
MulticastSocket ms = new MulticastSocket();
ms.joinGroup(new InetSocketAddress(hostname, port), nif);
```

7.3 RETRIEVING NETWORK INTERFACES

The NetworkInterface class has no public constructor. Therefore, you cannot just create a new instance of this class with the new operator. Instead, the following static methods are available so that you can retrieve the interface details from the system: getByInetAddress(), getByName(), and get NetworkInterfaces(). The first two methods are used when you already know the IP address or the name of the particular interface. The third method, getNetworkInterfaces() returns the complete list of interfaces on the machine.

Network interfaces can be hierarchically organized. The NetworkInterface class includes two methods, getParent() and getSubInterfaces(), that are pertinent to a network interface hierarchy. The getParent() method returns the parent NetworkInterface of an interface. If a network interface is a subinterface, getParent() returns a non-null value. The getSubInterfaces() method returns all the subinterfaces of a network interface.

7.4 NETWORK INTERFACE PARAMETERS

You can access network parameters about a network interface beyond the name and IP addresses assigned to it. You can discover if a network interface is “up” (that is, running) with the isUP() method. The following methods indicate the network interface type:

- isLoopback() indicates if the network interface is a loopback interface.
- isPointToPoint() indicates if the interface is a point-to-point interface.
- isVirtual() indicates if the interface is a virtual interface.

The supportsMulticast() method indicates whether the network interface supports multicasting. The getHardwareAddress() method returns the network interface's physical hardware address, usually called MAC address, when it is available. The getMTU() method returns the Maximum Transmission Unit (MTU), which is the largest packet size.

The following example expands on the example in Listing Network Interface Addresses by adding the additional network parameters described on this page:

```
import java.io.*;
import java.net.*;
import java.util.*;
import static java.lang.System.out;

public class ListNetsEx
{
```

```
public static void main(String args[]) throws SocketException
{
Enumeration<NetworkInterface> nets
=NetworkInterface.getNetworkInterfaces();
for (NetworkInterface netint : Collections.list(nets))
        displayInterfaceInformation(netint);
}

static void displayInterfaceInformation(NetworkInterface netint)
throws SocketException
{
out.printf("Display name: %s\n", netint.getDisplayName());
out.printf("Name: %s\n", netint.getName());
Enumeration<InetAddress> inetAddresses =
                netint.getInetAddresses();
for (InetAddress inetAddress : Collections.list(inetAddresses))
{
        out.printf("InetAddress: %s\n", inetAddress);
}
out.printf("Up? %s\n", netint.isUp());
out.printf("Loopback? %s\n", netint.isLoopback());
out.printf("PointToPoint? %s\n", netint.isPointToPoint());
out.printf("Supports multicast? %s\n", netint.supportsMulticast());
out.printf("Virtual? %s\n", netint.isVirtual());
out.printf("Hardware address: %s\n",
Arrays.toString(netint.getHardwareAddress()));
out.printf("MTU: %s\n", netint.getMTU());
out.printf("\n");
}
}
```

The following is sample output from the example program:

```
Display name: bge0
Name: bge0
InetAddress: /fe80:0:0:0:203:baff:fef2:e99d%2
InetAddress: /129.156.225.59
Up? true
Loopback? false
PointToPoint? false
Supports multicast? false
Virtual? false
Hardware address: [0, 3, 4, 5, 6, 7]
MTU: 1500

Display name: lo0
Name: lo0
InetAddress: /0:0:0:0:0:0:0:1%1
InetAddress: /127.0.0.1
Up? true
```

```
Loopback? true
PointToPoint? false
Supports multicast? false
Virtual? false
Hardware address: null
MTU: 8232
```

7.6 SUMMARY

- The java.net.NetworkInterface class provides access to information to perform the particular network activity on a specific connection.
- A network interface is the point of interconnection between a computer and a private or public network.
- The NetworkInterface class includes two methods, getParent() and getSubInterfaces(), that are pertinent to a network interface hierarchy.

7.7 UNIT END EXERCISE

1) What Is a Network Interface?
2) Write a short note on Network Interface Parameters?

7.8 FURTHER READING

- Ivan Bayross, Web Enabled Commercial Applications Development Using Java 2, BPB Publications, Revised Edition, 2006
- Joe Wigglesworth and Paula McMillan, Java Programming: Advanced Topics, Thomson Course Technology (SPD), Third Edition, 2004
- Cay S. Horstmann, Gary Cornell, Core Java™ 2: Volume II– Advanced Features Prentice Hall PTR, 2001

❖❖❖❖

8

REMOTE METHOD INVOCATIONS

Unit Structure:

8.0 Objectives
8.1 Introduction to Distributed Computing with RMI
8.2 Java RMI Architecture
8.3 RMI Architecture Layers
8.4 Naming Remote Objects
8.5 Using RMI
8.6 Summary
8.7 Unit end exercise
8.8 Further Reading

8.0 OBJECTIVES

The objectives of this chapter are to understand what is RMI and the different layers of the RMI architecture. We will study various classes and interface used to achieve distributed architecture.

8.1 INTRODUCTION TO DISTRIBUTED COMPUTING WITH RMI

Remote Method Invocation (RMI) technology, first introduced in JDK 1.1, elevates network programming to a higher plane. Although RMI is relatively easy to use, it is a remarkably powerful technology and exposes the average Java developer to an entirely new paradigm--the world of distributed object computing.

This chapter provides you with an in-depth introduction to this versatile technology. RMI has evolved considerably since JDK 1.1, and has been significantly upgraded under the Java 2 SDK. Where applicable, the differences between the two releases will be indicated.

8.1.1 Goals

A primary goal for the RMI designers was to allow programmers to develop distributed Java programs with the same syntax and semantics used for non-distributed programs. To do this, they had to carefully map how Java classes and objects work in a single Java Virtual Machine (JVM) to a new model of how classes and objects would work in a distributed (multiple JVM) computing environment.

This section introduces the RMI architecture from the perspective of the distributed or remote Java objects, and explores their differences through the behavior of local Java objects. The RMI architecture defines how objects behave, how and when exceptions can occur, how memory is managed, and how parameters are passed to, and returned from, remote methods.

8.1.2 Comparison of Distributed and Nondistributed Java Programs

The RMI architects tried to make the use of distributed Java objects similar to using local Java objects. While they succeeded, some important differences are listed in the table below. You can use this table as a reference as you learn about RMI.

	Local Object	Remote Object
Object Definition	A local object is defined by a Java class.	A remote object's exported behavior is defined by an interface that must extend the Remote interface.
Object Implementation	A local object is implemented by its Java class.	A remote object's behavior is executed by a Java class that implements the remote interface.
Object Creation	A new instance of a local object is created by the newoperator.	A new instance of a remote object is created on the host computer with the new operator. A client cannot directly create a new remote object (unless using Java 2 Remote Object Activation).
Object Access	A local object is accessed directly via an object reference variable.	A remote object is accessed via an object reference variable which points to a proxy stub implementation of the remote interface.

References	In a single JVM, an object reference points directly at an object in the heap.	A "remote reference" is a pointer to a proxy object (a "stub") in the local heap. That stub contains information that allows it to connect to a remote object, which contains the implementation of the methods.
Active References	In a single JVM, an object is considered "alive" if there is at least one reference to it.	In a distributed environment, remote JVMs may crash, and network connections may be lost. A remote object is considered to have an active remote reference to it if it has been accessed within a certain time period (the lease period). If all remote references have been explicitly dropped, or if all remote references have expired leases, then a remote object is available for distributed garbage collection.
Finalization	If an object implements the finalize() method , it is called before an object is reclaimed by the garbage collector.	If a remote object implements the Unreferenced interface, the unreferenced method of that interface is called when all remote references have been dropped.
Garbage Collection	When all local references to an object have been dropped, an object becomes a candidate for garbage collection.	The distributed garbage collector works with the local garbage collector. If there are no remote references and all local references to a remote object have been dropped, then it becomes a candidate for garbage collection through the normal means.

Exceptions	Exceptions are either Runtime exceptions or Exceptions. The Java compiler forces a program to handle all Exceptions.	RMI forces programs to deal with any possible Remote Exception objects that may be thrown. This was done to ensure the robustness of distributed applications.

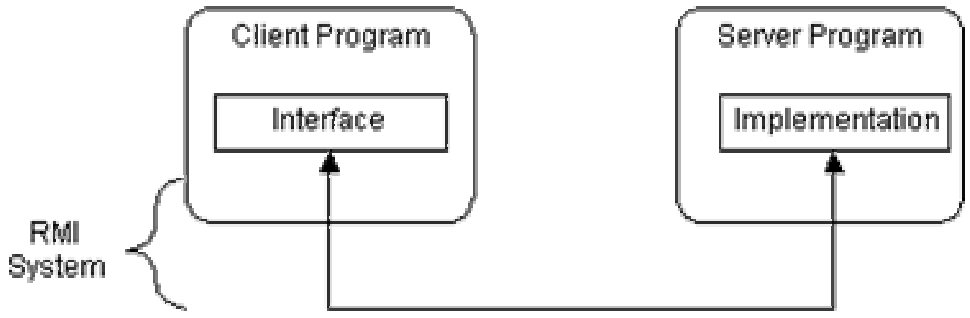
8.2 JAVA RMI ARCHITECTURE

The design goal for the RMI architecture was to create a Java distributed object model that integrates naturally into the Java programming language and the local object model. RMI architects have succeeded; creating a system that extends the safety and robustness of the Java architecture to the distributed computing world.

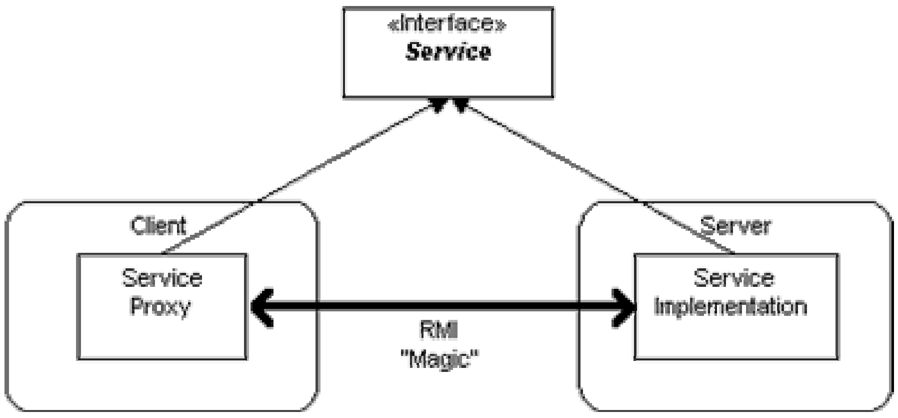
Interfaces: The Heart of RMI

The RMI architecture is based on one important principle: the definition of behavior and the implementation of that behavior are separate concepts. RMI allows the code that defines the behavior and the code that implements the behavior to remain separate and to run on separate JVMs.

This fits nicely with the needs of a distributed system where clients are concerned about the definition of a service and servers are focused on providing the service. Specifically, in RMI, the definition of a remote service is coded using a Java interface. The implementation of the remote service is coded in a class. Therefore, the key to understanding RMI is to remember that interfaces define behavior and classes define implementation. While the following diagram illustrates this separation,



remember that a Java interface does not contain executable code. RMI supports two classes that implement the same interface. The first class is the implementation of the behavior, and it runs on the server. The second class acts as a proxy for the remote service and it runs on the client. This is shown in the following diagram.



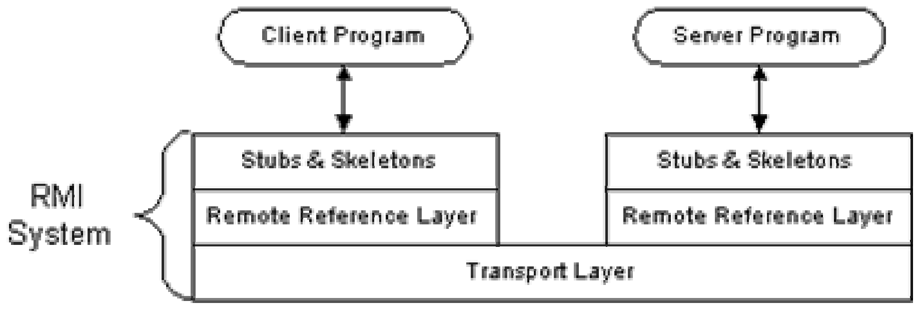
A client program makes method calls on the proxy object, RMI sends the request to the remote JVM, and forwards it to the implementation. Any return values provided by the implementation are sent back to the proxy and then to the client's program.

8.3 RMI ARCHITECTURE LAYERS

The RMI implementation is essentially built from three abstraction layers. The first is the Stub and Skeleton layer, which lies just beneath the view of the developer. This layer intercepts method calls made by the client to the interface reference variable and redirects these calls to a remote RMI service.

The next layer is the Remote Reference Layer. This layer understands how to interpret and manage references made from clients to the remote service objects. In JDK 1.1, this layer connects clients to remote service objects that are running and exported on a server. The connection is a one-to-one (unicast) link. In the Java 2 SDK, this layer was enhanced to support the activation of dormant remote service objects via Remote Object Activation.

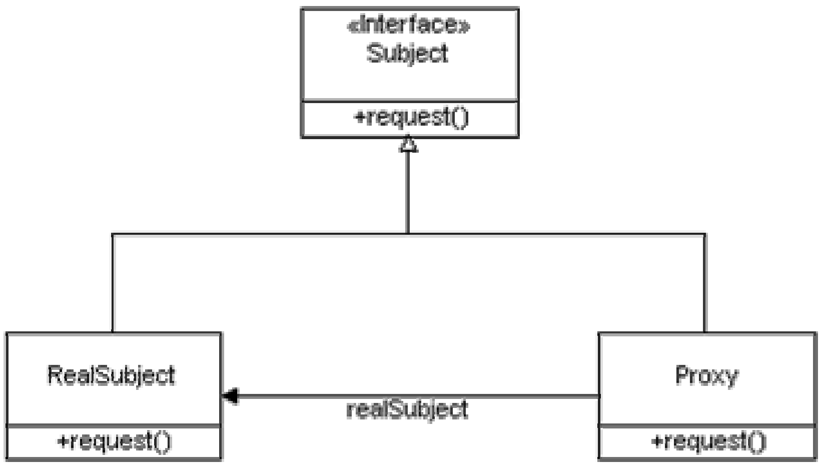
The transport layer is based on TCP/IP connections between machines in a network. It provides basic connectivity, as well as some firewall penetration strategies.



By using a layered architecture each of the layers could be enhanced or replaced without affecting the rest of the system. For example, the transport layer could be replaced by a UDP/IP layer without affecting the upper layers.

8.3.1 Stub and Skeleton Layer

The stub and skeleton layer of RMI lie just beneath the view of the Java developer. In this layer, RMI uses the Proxy design pattern. In the Proxy pattern, an object in one context is represented by another (the proxy) in a separate context. The proxy knows how to forward method calls between the participating objects. The following class diagram illustrates the Proxy pattern.



In RMI's use of the Proxy pattern, the stub class plays the role of the proxy, and the remote service implementation class plays the role of the RealSubject.

A skeleton is a helper class that is generated for RMI to use. The skeleton understands how to communicate with the stub across the RMI link. The skeleton carries on a conversation with the stub; it reads the parameters for the method call from the link, makes the call to the remote service implementation object, accepts the return value, and then writes the return value back to the stub.

In the Java 2 SDK implementation of RMI, the new wire protocol has made skeleton classes obsolete. RMI uses reflection to make the connection to the remote service object. You only have to worry about skeleton classes and objects in JDK 1.1 and JDK 1.1 compatible system implementations.

8.3.2 Remote Reference Layer

The Remote Reference Layers defines and supports the invocation semantics of the RMI connection. This layer provides a RemoteRef object that represents the link to the remote service implementation object.

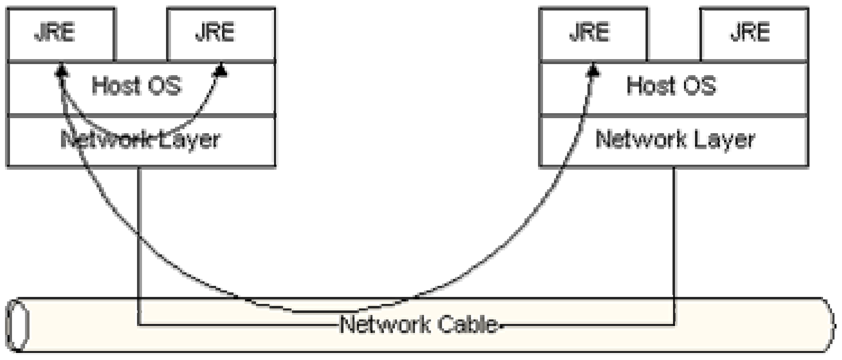
The stub objects use the invoke() method in RemoteRef to forward the method call. The RemoteRef object understands the invocation semantics for remote services. The JDK 1.1 implementation of RMI provides only one way for clients to connect to remote service implementations: a unicast, point-to-point connection. Before a client can use a remote service, the remote service must be instantiated on the server and exported to the RMI system. (If it is the primary service, it must also be named and registered in the RMI Registry).

The Java 2 SDK implementation of RMI adds a new semantic for the client-server connection. In this version, RMI supports activatable remote objects. When a method call is made to the proxy for an activatable object, RMI determines if the remote service implementation object is dormant. If it is dormant, RMI will instantiate the object and restore its state from a disk file. Once an activatable object is in memory, it behaves just like JDK 1.1 remote service implementation objects.

8.3.3 Transport Layer

The Transport Layer makes the connection between JVMs. All connections are stream-based network connections that use TCP/IP.

Even if two JVMs are running on the same physical computer, they connect through their host computer's TCP/IP network protocol stack. (This is why you must have an operational TCP/IP configuration on your computer to run the Exercises in this course). The following diagram shows the unfettered use of TCP/IP connections between JVMs.



As you know, TCP/IP provides a persistent, stream-based connection between two machines based on an IP address and port number at each end. Usually a DNS name is used instead of an IP address; this means you could talk about a TCP/IP connection between flicka. magelang. com:3452 and rosa. jguru.com:4432. In the current release of RMI, TCP/IP connections are used as the foundation for all machine-to-machine connections.

On top of TCP/IP, RMI uses a wire level protocol called Java Remote Method Protocol (JRMP). JRMP is a proprietary, stream-based protocol that is only partially specified is now in two versions. The first version was released with the JDK 1.1 version of RMI and required the use of Skeleton classes on the server. The second version was released with the Java 2 SDK. It has been optimized for performance and does not require skeleton classes. (Note that some alternate implementations, such as BEA Weblogic and NinjaRMI do not use JRMP, but instead use their own wire level protocol. ObjectSpace's Voyager does recognize JRMP and will interoperate with RMI at the wire level.) Some other changes with the Java 2 SDK are that RMI service interfaces are not required to extend from java.rmi.Remoteand their service methods do not necessarily throw RemoteException.

8.4 NAMING REMOTE OBJECTS

During the presentation of the RMI Architecture, one question has been repeatedly postponed: "How does a client find an RMI remote service?" Now you'll find the answer to that question. Clients find remote services by using a naming or directory service. This may seem like circular logic. How can a client locate a service by using a service? In fact, that is exactly the case. A naming or directory service is run on a well-known host and port number.

RMI can use many different directory services, including the Java Naming and Directory Interface (JNDI). RMI itself includes a simple service called the RMI Registry, rmiregistry. The RMI

Registry runs on each machine that hosts remote service objects and accepts queries for services, by default on port 1099.

On a host machine, a server program creates a remote service by first creating a local object that implements that service. Next, it exports that object to RMI. When the object is exported, RMI creates a listening service that waits for clients to connect and request the service. After exporting, the server registers the object in the RMI Registry under a public name.

On the client side, the RMI Registry is accessed through the static class Naming. It provides the method lookup() that a client uses to query a registry. The method lookup() accepts a URL that specifies the server host name and the name of the desired service. The method returns a remote reference to the service object. The URL takes the form:

```
rmi://<host_name>
    [:<name_service_port>]
        /<service_name>
```

where the host_name is a name recognized on the local area network (LAN) or a DNS name on the Internet. The name_service_port only needs to be specified only if the naming service is running on a different port to the default 1099.

8.5 USING RMI

It is now time to build a working RMI system and get hands-on experience. In this section, you will build a simple remote calculator service and use it from a client program.
A working RMI system is composed of several parts.

- Interface definitions for the remote services
- Implementations of the remote services
- Stub and Skeleton files
- A server to host the remote services
- An RMI Naming service that allows clients to find the remote services
- A class file provider (an HTTP or FTP server)
- A client program that needs the remote services

In the next sections, you will build a simple RMI system in a step-by-step fashion. You are encouraged to create a fresh subdirectory on your computer and create these files as you read the text.

To simplify things, you will use a single directory for the client and server code. By running the client and the server out of the same directory, you will not have to set up an HTTP or FTP server to provide the class files. Assuming that the RMI system is already designed, you take the following steps to build a system:

- Write and compile Java code for interfaces
- Write and compile Java code for implementation classes
- Generate Stub and Skeleton class files from the implementation classes
- Write Java code for a remote service host program
- Develop Java code for RMI client program
- Install and run RMI system

Interfaces

The first step is to write and compile the Java code for the service interface. The Calculator interface defines all of the remote features offered by the service:

```
public interface Calculator extends java.rmi.Remote
{
public long add(long a, long b) throws java.rmi.RemoteException;
public long sub(long a, long b) throws java.rmi.RemoteException;
public long mul(long a, long b) throws java.rmi.RemoteException;
public long div(long a, long b) throws java.rmi.RemoteException;
}
```

Notice this interface extends Remote, and each method signature declares that it may throw a RemoteException object.

Copy this file to your directory and compile it with the Java compiler:

```
>javac Calculator.java
```

Implementation

Next, you write the implementation for the remote service. This is the CalculatorImpl class:

```
public class CalculatorImpl extends
java.rmi.server.UnicastRemoteObject implements Calculator
{
// Implementations must have an explicit constructor  in order to
//declare the RemoteException exception
public CalculatorImpl() throws java.rmi.RemoteException {
        super();
}
public long add(long a, long b) throws java.rmi.RemoteException {
        return a + b;
}
public long sub(long a, long b) throws java.rmi.RemoteException {
        return a - b;
}
public long mul(long a, long b) throws java.rmi.RemoteException {
        return a * b;
}
public long div(long a, long b) throws java.rmi.RemoteException {
        return a / b;
}
}
```

Again, copy this code into your directory and compile it. The implementation class uses UnicastRemoteObject to link into the RMI system. In the example the implementation class directly extends UnicastRemoteObject. This is not a requirement. A class that does not extend UnicastRemoteObject may use its exportObject() method to be linked into RMI.

When a class extends UnicastRemoteObject, it must provide a constructor that declares that it may throw a Remote Exception object. When this constructor calls super(), it activates code in Unicast Remote Object that performs the RMI linking and remote object initialization.

Stubs and Skeletons

You next use the RMI compiler, rmic, to generate the stub and skeleton files. The compiler runs on the remote service implementation class file.

```
>rmic CalculatorImpl
```

Try this in your directory. After you run rmic you should find the file Calculator_Stub.class and, if you are running the Java 2 SDK, Calculator_Skel.class.
Options for the JDK 1.1 version of the RMI compiler, rmic, are:

Host Server

Remote RMI services must be hosted in a server process. The class CalculatorServer is a very simple server that provides the bare essentials for hosting.

```
import java.rmi.Naming;
public class CalculatorServer
{
public CalculatorServer() {
try {
```

```
          Calculator c = new CalculatorImpl();
          Naming.rebind("rmi://localhost:1099/CalculatorService", c);
}
catch (Exception e) {
          System.out.println("Trouble: " + e);
}
}
public static void main(String args[]) {
          new CalculatorServer();
}//main
}//class
```

Client

The source code for the client follows:

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.net.MalformedURLException;
import java.rmi.NotBoundException;
public class CalculatorClient
{
public static void main(String[] args)
{
          try {
          Calculator c = (Calculator)
                    Naming.lookup("rmi://localhost/CalculatorService");
          System.out.println( c.sub(4, 3) );
          System.out.println( c.add(4, 5) );
          System.out.println( c.mul(3, 6) );
          System.out.println( c.div(9, 3) );
}
catch (Exception e) {
          System.out.println("Trouble: " + e);
}
}//main
}//class
```

Running the RMI System

You are now ready to run the system! You need to start three consoles, one for the server, one for the client, and one for the RMIRegistry.Start with the Registry. You must be in the directory that contains the classes you have written. From there, enter the following:

```
rmiregistry
```

If all goes well, the registry will start running and you can switch to the next console.
In the second console start the server hosting the CalculatorService, and enter the following:

```
>java CalculatorServer
```

It will start, load the implementation into memory and wait for a client connection.
In the last console, start the client program.

```
>java CalculatorClient
```

If all goes well you will see the following output:

```
1
9
18
3
```

That's it; you have created a working RMI system. Even though you ran the three consoles on the same computer, RMI uses your network stack and TCP/IP to communicate between the three separate JVMs. This is a full-fledged RMI system.

8.6 SUMMARY

- RMI is a powerful technology and exposes the developer to the world of distributed object computing.
- The RMI architecture is based on one important principle: the definition of behavior and the implementation of that behavior are separate concepts
- The RMI Registry runs on each machine that hosts remote service objects and accepts queries for services, by default on port 1099.

8.7 UNIT END EXERCISE

1) Explain RMI Architecture.
2) What is the role of Remote Interface in RMI?
3) What is meant by binding in RMI?

4) What is the difference between using bind() and rebind() methods of Naming Class?

5) What is the use of UnicastRemoteObject in RMI?

6) Write an RMI program that returns date & time of server to the client who requests it.

8.8 FURTHER READING

- Ivan Bayross, Web Enabled Commercial Applications Development Using Java 2, BPB Publications, Revised Edition, 2006
- Joe Wigglesworth and Paula McMillan, Java Programming: Advanced Topics, Thomson Course Technology (SPD), Third Edition, 2004
- The Java Tutorials of Sun Microsystems Inc.
- Herbert Schildt, Java2: The Complete Reference, Tata McGraw-Hill, Fifth edition, 2002
- Cay S. Horstmann, Gary Cornell, Core Java™ 2: Volume II– Advanced Features Prentice Hall PTR, 2001

❖❖❖❖

9

SERVLET BASICS

Unit Structure:

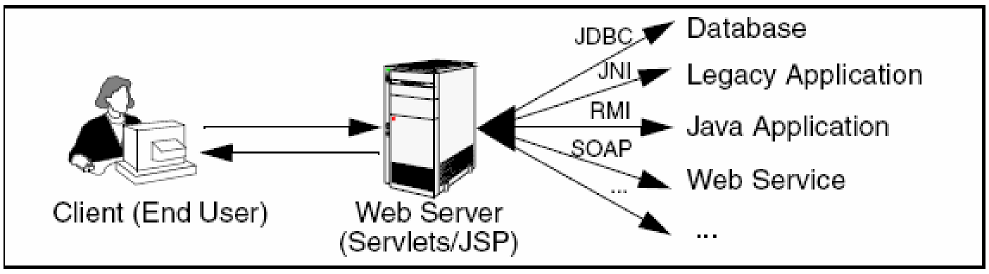
9.0 Objectives
9.1 Introduction to Servlet.
9.2 The Servlet Life Cycle
9.3 Reading Form Data from Servlets
9.4 Response Headers
9.5 Request Headers
9.6 Summary
9.7 Unit end exercise
9.8 Further Reading

9.0 OBJECTIVES

The objective of this chapter is to learn the basics of Servlets, Why is it used, How it is created, Life Cycle of the Servlet, how to read a Request, how a response object is created.

9.1 INTRODUCTION TO SERVLET

Servlets are Java programs that run on Web or application servers, acting as a middle layer between requests coming from Web browsers or other HTTP clients and databases or applications on the HTTP server. Their job is to perform the following tasks, as illustrated in Figure 1.



- **Read the explicit data sent by the client** - The end user normally enters the data in an HTML form on a Web page.

However, the data could also come from an applet or a custom HTTP client program.

- **Read the implicit HTTP request data sent by the browser** - Figure 1 shows a single arrow going from the client to the Web server (the layer where servlets and JSP execute), but there are really two varieties of data: the explicit data that the end user enters in a form and the behind-the-scenes HTTP information. Both varieties are critical. The HTTP information includes cookies, information about media types and compression schemes the browser understands, and so forth.

- **Generate the results** - This process may require talking to a database, executing an RMI or EJB call, invoking a Web service, or computing the response directly. Your real data may be in a relational database. Fine. But your database probably doesn't speak HTTP or return results in HTML, so the Web browser can't talk directly to the database. Even if it could, for security reasons, you probably would not want it to. The same argument applies to most other applications. You need the Web middle layer to extract the incoming data from the HTTP stream, talk to the application, and embed the results inside a document.

- **Send the explicit data (i.e., the document) to the client** - This document can be sent in a variety of formats, including text (HTML or XML), binary (GIF images), or even a compressed format like gzip that is layered on top of some other underlying format. But, HTML is by far the most common format, so an important servlet/JSP task is to wrap the results inside of HTML.

- **Send the implicit HTTP response data** - Figure 1 shows a single arrow going from the Web middle layer (the servlet or JSP page) to the client. But, there are really two varieties of data sent: the document itself and the behind-the-scenes HTTP information. Again, both varieties are critical to effective development. Sending HTTP response data involves telling the browser or other client what type of document is being returned (e.g., HTML), setting cookies and caching parameters, and other such tasks.

Why Build Web Pages Dynamically?

There are a number of reasons why Web pages need to be built on-the-fly:

- **The Web page is based on data sent by the client** - For instance, the results page from search engines and order confirmation pages at online stores are specific to particular user requests. You don't know what to display until you read the data that the user submits. Just remember that the user submits two kinds of data: explicit (i.e., HTML form data) and implicit (i.e., HTTP request headers). Either kind of input can be used to build the output page. In particular, it is quite common to build a user-specific page based on a cookie value.

- **The Web page is derived from data that changes frequently** - If the page changes for every request, then you certainly need to build the response at request time. If it changes only periodically, however, you could do it two ways: you could periodically build a new Web page on the server (independently of client requests), or you could wait and only build the page when the user requests it. The right approach depends on the situation, but sometimes it is more convenient to do the latter: wait for the user request. For example, a weather report or news headlines site might build the pages dynamically, perhaps returning a previously built page if that page is still up to date.
- **The Web page uses information from corporate databases or other server-side sources** - If the information is in a database, you need server-side processing even if the client is using dynamic Web content such as an applet. Imagine using an applet by itself for a search engine site: “Downloading 50 terabyte applet, please wait!” Obviously, that is silly; you need to talk to the database. Going from the client to the Web tier to the database (a three-tier approach) instead of from an applet directly to a database (a two-tier approach) provides increased flexibility and security with little or no performance penalty. After all, the database call is usually the rate-limiting step, so going through the Web server does not slow things down. In fact, a three-tier approach is often faster because the middle tier can perform caching and connection pooling.

The Advantages of Servlets Over “Traditional” CGI

Java servlets are more efficient, easier to use, more powerful, more portable, safer, and cheaper than traditional CGI(Common Gateway Interface) and many alternative CGI-like technologies.

Efficient

With traditional CGI, a new process is started for each HTTP request. If the CGI program itself is relatively short, the overhead of starting the process can dominate the execution time. With servlets, the Java virtual machine stays running and handles each request with a lightweight Java thread, not a heavyweight operating system process. Similarly, in traditional CGI, if there are N requests to the same CGI program, the code for the CGI program is loaded into memory N times. With servlets, however, there would be N threads, but only a single copy of the servlet class would be loaded. This approach reduces server memory requirements and saves time by instantiating fewer objects. Finally, when a CGI program finishes handling a request, the program terminates. This approach makes it difficult to cache computations, keep database connections open, and perform other optimizations that rely on persistent data. Servlets, however, remain in memory even after they complete a response, so it is straightforward to store arbitrarily complex data between client requests.

Convenient

Servlets have an extensive infrastructure for automatically parsing and decoding HTML form data, reading and setting HTTP headers, handling cookies, tracking sessions, and many other such high-level utilities. In CGI, you have to do much of this yourself. Besides, if you already know the Java programming language, why learn Perl too? You're already convinced that Java technology makes for more reliable and reusable code than does Visual Basic, VBScript, or C++. Why go back to those languages for server-side programming?

Powerful

Servlets support several capabilities that are difficult or impossible to accomplish with regular CGI. Servlets can talk directly to the Web server, whereas regular CGI programs cannot, at least not without using a server-specific API. Communicating with the Web server makes it easier to translate relative URLs into concrete path names, for instance. Multiple servlets can also share data, making it easy to implement database connection pooling and similar resource-sharing optimizations. Servlets can also maintain information from request to request, simplifying techniques like session tracking and caching of previous computations.

Portable

Servlets are written in the Java programming language and follow a standard API. Servlets are supported directly or by a plugin on virtually every major Web server. Consequently, servlets written for, say, Macromedia JRun can run virtually unchanged on Apache Tomcat, Microsoft Internet Information Server (with a separate plugin), IBM WebSphere, iPlanet Enterprise Server, Oracle9i AS, or StarNine WebStar. They are part of the Java 2 Platform, Enterprise Edition (J2EE), so industry support for servlets is becoming even more pervasive.

Inexpensive

A number of free or very inexpensive Web servers are good for development use or deployment of low- or medium-volume Web sites. Thus, with servlets and JSP you can start with a free or inexpensive server and migrate to more expensive servers with high-performance capabilities or advanced administration utilities only after your project meets initial success. This is in contrast to many of the other CGI alternatives, which require a significant initial investment for the purchase of a proprietary package. Price and portability are somewhat connected.

Secure

One of the main sources of vulnerabilities in traditional CGI stems from the fact that the programs are often executed by general-purpose operating system shells. So, the CGI programmer must be careful to filter out characters such as backquotes and semicolons that are treated specially by the shell. Implementing this precaution is harder than one might think, and weaknesses stemming from this problem are constantly being uncovered in widely used CGI libraries. A second source of problems is the fact that some CGI programs are processed by languages that do not automatically check array or string bounds. For example, in C and C++ it is perfectly legal to allocate a 100-element array and then write into the 999th “element,” which is really some random part of program memory. So, programmers who forget to perform this check open up their system to deliberate or accidental buffer overflow attacks. Servlets suffer from neither of these problems. Even if a servlet executes a system call (e.g., with Runtime.exec or JNI) to invoke a program on the local operating system, it does not use a shell to do so. And, of course, array bounds checking and other memory protection features are a central part of the Java programming language.

9.2 THE SERVLET LIFE CYCLE

When the servlet is first created, its init method is invoked, so init is where you put one-time setup code. After this, each user request results in a thread that calls the service method of the previously created instance. Multiple concurrent requests normally result in multiple threads calling service simultaneously, although your servlet can implement a special interface (SingleThreadModel) that stipulates that only a single thread is permitted to run at any one time. The service method then calls doGet, doPost, or another doXxx method, depending on the type of HTTP request it received. Finally, if the server decides to unload a servlet, it first calls the servlet’s destroy method.

The service Method

Each time the server receives a request for a servlet, the server spawns a new thread and calls service. The service method checks the HTTP request type (GET, POST, PUT, DELETE, etc.) and calls doGet, doPost, doPut, doDelete, etc., as appropriate. A GET request results from a normal request for a URL or from an HTML form that has no METHOD specified. A POST request results from an HTML form that specifically lists POST as the METHOD. Other HTTP requests are generated only by custom clients. Now, if you have a servlet that needs to handle both POST and GET requests identically, you may be tempted to override service directly rather than implementing both doGet and doPost. This is not a good idea. Instead, just have doPost call doGet (or vice versa).

The doGet, doPost, and doXxx Methods

These methods contain the real meat of your servlet. Ninety-nine percent of the time, you only care about GET or POST requests, so you override doGet and/or doPost. However, if you want to, you can also override doDelete for DELETE requests, doPut for PUT, doOptions for OPTIONS, and doTrace for TRACE. Recall, however, that you have automatic support for OPTIONS and TRACE. Normally, you do not need to implement doHead in order to handle HEAD requests (HEAD requests stipulate that the server should return the normal HTTP headers, but no associated document). You don't normally need to implement doHead because the system automatically calls doGet and uses the resultant status line and header settings to answer HEAD requests. However, it is occasionally useful to implement doHead so that you can generate responses to HEAD requests (i.e., requests from custom clients that want just the HTTP headers, not the actual document) more quickly—without building the actual document output.

The init Method

Most of the time, your servlets deal only with per-request data, and doGet or doPost are the only life-cycle methods you need. Occasionally, however, you want to perform complex setup tasks when the servlet is first loaded, but not repeat those tasks for each request. The init method is designed for this case; it is called when the servlet is first created, and not called again for each user request. So, it is used for one-time initializations, just as with the init method of applets. The servlet is normally created when a user first invokes a URL corresponding to the servlet, but you can also specify that the servlet be loaded when the server is first started. The init method performs two varieties of initializations: general initializations and initializations controlled by initialization parameters.

The destroy Method

The server may decide to remove a previously loaded servlet instance, perhaps because it is explicitly asked to do so by the server administrator or perhaps because the servlet is idle for a long time. Before it does, however, it calls the servlet's destroy method. This method gives your servlet a chance to close database connections, halt background threads, write cookie lists or hit counts to disk, and perform other such cleanup activities. Be aware, however, that it is possible for the Web server to crash. So, don't count on destroy as the only mechanism for saving state to disk. If your servlet performs activities like counting hits or accumulating lists of cookie values that indicate special access, you should also proactively write the data to disk periodically.

Example: Write a Servlet program to display ‘Hello World’.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloServlet extends HttpServlet
{
        public void doGet(
                HttpServletRequest req,HttpServletResponse res)
                throws IOException, ServletException
        {
                PrintWriter out = res.getWriter();
                out.println("<html><head><title>First Servlet
                                                </title></head>");
                out.println("<b>HelloServlet</b></html>");
        }
}
```

Example: Write a Servlet program to display the current date.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Date;
public class DateServlet extends HttpServlet
{
        public void doGet(
                HttpServletRequest req,HttpServletResponse res)
                throws IOException,ServletException
        {
                res.setContentType("Text/Html");
                PrintWriter out=res.getWriter();
                Date d=new Date();
                out.println("<Html><Head><Title>Today
                                                        </Title></Head>");
                out.println("<Body><H1> Date: "+d+"</H1>");
                out.flush();
                out.println("</Body></Html>");
        }
}
```

9.3 READING FORM DATA FROM SERVLETS

Reading Single Values: getParameter

To read a request (form) parameter, you simply call the getParameter method of HttpServletRequest, supplying the case-sensitive parameter name as an argument. You supply the parameter name exactly as it appeared in the HTML source code, and you get the result exactly as the end user entered it; any necessary URL-decoding is done automatically. An empty String is returned if the parameter exists but has no value (i.e., the user left the corresponding textfield empty when submitting the form), and null is returned if there was no such parameter. Parameter names are case sensitive so, for example, request. Get Parameter ("Param1") and request. get Parameter ("param1") are not interchangeable.

Reading Multiple Values: getParameterValues

If the same parameter name might appear in the form data more than once, you should call getParameterValues (which returns an array of strings) instead of getParameter (which returns a single string corresponding to the first occurrence of the parameter). The return value of getParameterValues is null for nonexistent parameter names and is a one element array when the parameter has only a single value. Now, if you are the author of the HTML form, it is usually best to ensure that each textfield, checkbox, or other user interface element has a unique name. That way, you can just stick with the simpler getParameter method and avoid getParameterValues altogether. Besides, multiselectable list boxes repeat the parameter name for each selected element in the list. So, you cannot always avoid multiple values.

Looking Up Parameter Names: getParameterNames

Use getParameterNames to get this list in the form of an Enumeration, each entry of which can be cast to a String and used in a getParameter or getParameterValues call. If there are no parameters in the current request, getParameterNames returns an empty Enumeration (not null). Note that Enumeration is an interface that merely guarantees that the actual class will have hasMoreElements and nextElement methods: there is no guarantee that any particular underlying data structure will be used. And, since some common data structures (hash tables, in particular) scramble the order of the elements, you should not count on getParameterNames returning the parameters in the order in which they appeared in the HTML form.

Example: Write a Servlet that accepts name and age of student sent from an HTML document and displays them on screen.

//Student.html file

```
<html>
<head><title>Student Information</title>
</head>
<form name=frm method=get
action=http:\\localhost:8080\Servlet\Student.class>
</form>
<body>
<table>
<tr><td>Student Name</td><td><input type=text
name=txtName></td></tr>
<tr><td>Student Age</td><td><input type=text
name=txtAge></td></tr>
</table>
<input type=submit name=submit>
</body>
</html>
```

//Student.java file

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class Student extends HttpServlet
{
public void doGet(HttpServletRequest req,
                    HttpServletResponse res)
                    throws ServletException,IOException
{
res.setContentType("text/html");
String name=(String)req.getParameter("txtName");
String age=(String)req.getParameter("txtAge");
PrintWriter out=res.getWriter();
out.println("Name = "+name);
out.println("Age= "+age);
}
}//class
```

Example: Write a Servlet that accepts single-valued as well as multi-valued parameters like check boxes and multiple selection list boxes from an HTML document and outputs them to the screen.

```
//Part I – HTML file
<html>
<head><title>Multivalued Parameter</title>
</head>
<form method=post action=http:\\localhost:8080\servlet\
MultiValued.class>
<table border=1>
<tr><td>Name</td><td><input type=text name=txtName></td></tr>
<tr><td>Tel.No</td><td><input type=text name=txtTelno></td></tr>
<tr><td>Language</td>
        <td><input type=checkbox name=chk value=eng>Eng</td>
        <td><input type=checkbox name=chk
value=mar>Marathi</td>
        <td><input type=checkbox name=chk
value=hin>Hindi</td></tr>
<tr><td>Software</td>
        <td><select multiple size=5 name=software>
                <option>Excel
                <option>VB
                <option>Word
                <option>Java
                <option>C++
        </select></td></tr>
<tr>
        <td><input type=submit value=submit></td>
        <td><input type=reset></td>
</tr>
</table>
</form>
</html>

//Part II – JAVA file
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
import java.io.*;

public class MultiValued extends HttpServlet
{
        public void doPost(HttpServletResponse
res,HttpServletRequest req)
throws IOException,ServletException
        {
                try
                {
                res.setContentType("text/html");
                Enumeration e=req.getParameterNames();
                PrintWriter out=res.getWriter();
                while(e.hasMoreElements())
                {
                        String name=(String)e.nextElement();
                        out.println(name);
                        String[] value=req.getParameterValues(name);
                        for(int i=0;i<value.length;i++)
                        {
                                out.print(value[i]+"\t");
                        }
                }
                }//try
                catch(Exception e)
                {
                System.out.println("ERROR "+e.getMessage());
                }
        }
}
```

Example: Write two servlets in which one servlet will display a form in which data entry can be done for the field's dept-no, dept-name and location. In the same form place a button called as submit and on click of that button this record should be posted to the table called as DEPT in the database. This inserting of record should be done in another servlet. The second servlet should also display all the previous record entered in the database.

```
//Part I
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class DeptForm extends HttpServlet
{
        public void service(HttpServletRequest req,
                                HttpServletResponse res)
                                throws IOException,ServletException
        {
                try
                {
                res.setContentType("text/html");
                PrintWriter out=res.getWriter();
                out.println("<Html><Head><Title>Department Info
                                                        </Title></Head>");
                out.println("<Form name=frm method="+"POST"+"
                                                action=DeptEntry.class>");
                out.println("DepartmentNo:   <input type=text
                                                name=txtNo><br>");
                out.println("DepartmentName: <input type=text
                                                name=txtName><br>");
                out.println("Location: <input type=text
                                                name=txtLoc><br>");
                out.println("<input type=submit name=Submit>");
                out.println("<input type=reset name=Reset>");
                out.println("</Form></Html>");
                }
                catch (Exception e)
                {
                System.out.println(e.getMessage());
                }
        }
}

//Part II
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.*;
public class DeptEntry extends HttpServlet
{
        public void doPost(HttpServletRequest req,
                            HttpServletResponse res)
                            throws IOException,ServletException
        {
                String a,b,c,d,e,f;
                int i;
                Connection con;
```

```
try
{
res.setContentType("text/html");
Class.forName("oracle.jdbc.driver.OracleDriver");
con=DriverManager.getConnection("jdbc:odbc:First”);
String Query="insert into dept Values(?,?,?)";
Statement st=con.createStatement();
PreparedStatement ps;
ps=con.prepareStatement(Query);
a=(String)req.getParameter("txtNo");
b=(String)req.getParameter("txtName");
c=(String)req.getParameter("txtLoc");
ps.setString(1,a);
ps.setString(2,b);
ps.setString(3,c);
ps.executeUpdate();

PrintWriter out=res.getWriter();
ResultSet rs=st.executeQuery("select * from dept");
ResultSetMetaData md=rs.getMetaData();
int num=md.getColumnCount();
out.println("<html><body><table border=1><tr>");
for(i=1;i<=num;i++)
{
out.print("<th>"+md.getColumnName(i)+"</th>");
}
out.println("</tr>");
while(rs.next())
{       d=rs.getString(1);
        e=rs.getString(2);
        f=rs.getString(3);
        out.println("<tr><td>");       out.println(d);
        out.println("</td><td>");      out.println(e);
        out.println("</td><td>");      out.println(f);
        out.println("</td></tr>");
}
out.println("</table>");
con.commit();
out.println("<a href=DeptForm.class>BACK</a>");
out.println("</body></html>");
```

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

        }
}
```

9.4 RESPONSE HEADERS

Setting the HTTP response headers often goes hand in hand with setting the status codes in the status line. For example, all the “document moved” status codes (300 through 307) have an accompanying Location header, and a 401 (Unauthorized) code always includes an accompanying WWW-Authenticate header. However, specifying headers can also play a useful role even when no unusual status code is set. Response headers can be used to specify cookies, to supply the page modification date (for client-side caching), to instruct the browser to reload the page after a designated interval, to give the file size so that persistent HTTP connections can be used, to designate the type of document being generated, and to perform many other tasks.

Setting Response Headers from Servlets

The most general way to specify headers is to use the setHeader method of HttpServletResponse. This method takes two strings: the header name and the header value. As with setting status codes, you must specify headers before returning the actual document.

- setHeader(String headerName, String headerValue) - This method sets the response header with the designated name to the given value. In addition to the general-purpose setHeader method, HttpServletResponse also has two specialized methods to set headers that contain dates and integers.
- setDateHeader(String header, long milliseconds) - This method saves you the trouble of translating a Java date in milli seconds since 1970 (as returned by System. Current TimeMillis, Date.getTime, or Calendar.getTimeInMillis) into a GMT time string.
- setIntHeader(String header, int headerValue) - This method spares you the minor inconvenience of converting an int to a String before inserting it into a header.

Finally, HttpServletResponse also supplies a number of convenience methods for specifying common headers. These methods are summarized as follows.

- setContentType(String mimeType) - This method sets the Content-Type header and is used by the majority of servlets.
- setContentLength(int length) - This method sets the Content-Length header, which is useful if the browser supports persistent (keep-alive) HTTP connections.
- addCookie(Cookie c) - This method inserts a cookie into the Set-Cookie header. There is no corresponding setCookie method, since it is normal to have multiple Set-Cookie lines.
- sendRedirect(String address) - The sendRedirect method sets the Location header as well as setting the status code to 302.

Understanding HTTP 1.1 Response Headers

- Allow: The Allow header specifies the request methods (GET, POST, etc.) that the server supports.
- Connection: A value of close for this response header instructs the browser not to use persistent HTTP connections.
- Content-Encoding: This header indicates the way in which the page was encoded during transmission.
- Content-Language: The Content-Language header signifies the language in which the document is written.
- Content-Length: This header indicates the number of bytes in the response.
- Content-Type: The Content-Type header gives the MIME (Multipurpose Internet Mail Extension) type of the response document. Setting this header is so common that there is a special method in HttpServletResponse for it: setContentType.
- Expires: This header stipulates the time at which the content should be considered out-of-date and thus no longer be cached. A servlet might use this header for a document that changes relatively frequently, to prevent the browser from displaying a stale cached value.
- Last-Modified: This very useful header indicates when the document was last changed.
- Refresh: This header indicates how soon (in seconds) the browser should ask for an updated page. For example, to tell the browser to ask for a new copy in 30 seconds, you would specify a value of 30 with response.setIntHeader("Refresh", 30);
- Set-Cookie: The Set-Cookie header specifies a cookie associated with the page. Each cookie requires a separate Set-

Cookie header. Servlets should not use response.setHeader("Set-Cookie", ...) but instead should use the special-purpose addCookie method of HttpServletResponse.

Example: Write a Servlet that creates Excel spreadsheet comparing apples and oranges.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class ApplesAndOranges extends HttpServlet
{
        public void doGet(HttpServletRequest request,
                                HttpServletResponse response)
        throws ServletException, IOException
        {
                response.setContentType("application/vnd.ms-
                excel");
                PrintWriter out = response.getWriter();
                out.println("\tQ1\tQ2\tQ3\tQ4\tTotal");
                out.println("Apples\t78\t87\t92\t29\t=SUM(B2:E2)");
                out.println("Oranges\t77\t86\t93\t30\t=SUM(B3:E3)");
        }
}
```

9.5 REQUEST HEADERS

HTTP request headers are distinct from the form (query) data. Form data results directly from user input and is sent as part of the URL for GET requests and on a separate line for POST requests. Request headers, on the other hand, are indirectly set by the browser and are sent immediately following the initial GET or POST request line. For instance, the following example shows an HTTP request that might result from a user submitting a book-search request to a servlet at http://www.somebookstore. com/servlet/Search. The request includes the headers Accept, Accept-Encoding, Connection, Cookie, Host, Referer, and User-Agent, all of which might be important to the operation of the servlet, but none of which can be derived from the form data or deduced automatically: the servlet needs to explicitly read the request headers to make use of this information.

```
GET /servlet/Search?keywords=servlets+jsp HTTP/1.1
Accept: image/gif, image/jpg, */*
Accept-Encoding: gzip
```

```
Connection: Keep-Alive
Cookie: userID=id456578
Host: www.somebookstore.com
Referer: http://www.somebookstore.com/findbooks.html
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
```

Reading headers is straightforward; just call the getHeader method of HttpServletRequest with the name of the header. This call returns a String if the specified header was supplied in the current request, null otherwise. In HTTP 1.0, all request headers are optional; in HTTP 1.1, only Host is required. So, always check for null before using a request header. Header names are not case sensitive. So, for example, request.getHeader("Connection") is interchangeable with request.getHeader("connection"). Although getHeader is the general-purpose way to read incoming headers, a few headers are so commonly used that they have special access methods in HttpServletRequest.

Following is a summary.

- getCookies - The getCookies method returns the contents of the Cookie header, parsed and stored in an array of Cookie objects.
- getAuthType and getRemoteUser - The getAuthType and getRemoteUser methods break the Authorization header into its component pieces.
- getContentLength - The getContentLength method returns the value of the Content-Length header (as an int).
- getContentType - The getContentType method returns the value of the Content-Type header (as a String).
- getDateHeader and getIntHeader - The getDateHeader and getIntHeader methods read the specified headers and then convert them to Date and int values, respectively.
- getHeaderNames - Rather than looking up one particular header, you can use the getHeaderNames method to get an Enumeration of all header names received on this particular request.
- getHeaders - In most cases, each header name appears only once in the request. Occasionally, however, a header can appear multiple times, with each occurrence listing a separate value. Accept-Language is one such example. You can use getHeaders to obtain an Enumeration of the values of all occurrences of the header.

Finally, in addition to looking up the request headers, you can get information on the main request line itself (i.e., the first line in the example request just shown), also by means of methods in HttpServletRequest. Here is a summary of the four main methods.

- getMethod - The getMethod method returns the main request method (normally, GET or POST, but methods like HEAD, PUT, and DELETE are possible).
- getRequestURI - The getRequestURI method returns the part of the URL that comes after the host and port but before the form data. For example, for a URL of http://randomhost. com/servlet/search.BookSearch?subject=jsp, get Request URI would return "/servlet/search. Book Search".
- getQueryString - The getQueryString method returns the form data. For example, with http://randomhost.com/servlet/search. Book Search? subject=jsp, getQueryString would return "subject=jsp".
- getProtocol - The getProtocol method returns the third part of the request line, which is generally HTTP/1.0 or HTTP/1.1. Servlets should usually check getProtocol before specifying response headers that are specific to HTTP 1.1.

Example: Write a program which shows all the request headers sent on the current request

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

public class ShowRequestHeaders extends HttpServlet
{
public void doGet(HttpServletRequest request,
                                HttpServletResponse response)
                                throws ServletException, IOException
{
response.setContentType("text/html");
PrintWriter out = response.getWriter();
String title = "Servlet Example: Showing Request Headers";
out.println("<HTML>\n" );
out.println("<HEAD><TITLE>" + title + "</TITLE></HEAD>\n");
out.println("<BODY BGCOLOR=\"#FDF5E6\">\n");
out.println("<H1 ALIGN=\"CENTER\">" + title + "</H1>\n");
out.println("<B>Request Method: </B>"
                        +request.getMethod() + "<BR>\n");
out.println("<B>Request URI: </B>" +
                        request.getRequestURI() + "<BR>\n");
out.println("<B>Request Protocol: </B>" +
                        request.getProtocol() + "<BR>\n");
out.println("<TABLE BORDER=1 ALIGN=\"CENTER\">\n");
out.println("<TR BGCOLOR=\"#FFAD00\">\n");
out.println("<TH>Header Name<TH>Header Value");

Enumeration headerNames = request.getHeaderNames();
while(headerNames.hasMoreElements())
{
String headerName = (String)headerNames.nextElement();
out.println("<TR><TD>" + headerName);
out.println(" <TD>" + request.getHeader(headerName));
}
out.println("</TABLE>\n</BODY></HTML>");
}
public void doPost(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException
{
doGet(request, response);
}
}//class
```

9.6 SUMMARY

- Servlets are Java programs that run on Web acting as a middle layer between requests coming from Web browsers and databases or applications on the HTTP server.
- Java servlets are more efficient, easier to use, more powerful, more portable, safer, and cheaper than traditional CGI and many alternative CGI-like technologies.
- Some servlets may not read anything from the Request object, based on the Servlet that is invoked only the processing would be done and the result will be returned to the client. In this case only the service method would be called.
- In some case the servlet read the data using the Request object process the data and return the result to the client.
- The Request object is used to read single as well as multiple parameters from the HTML objects with the help of methods of HttpServletRequest.

- The Response object is used to write the Headers by the user on the client side, this can be achieved using various methods from the HttpServletResponse.

9.7 UNIT END EXERCISE

1) What is a Servlet? How do they perform their tasks?
2) State any three reasons why Servlets are used to build Web Pages Dynamically?
3) State the advantages of Servlets over "Traditional" CGI?
4) Write a short note on Servlet Life Cycle?
5) Explain the methods used for reading Form Data from Servlets.
6) State and explain any three methods of HttpServletRequest?
7) State and explain any three methods of HttpServletResponse?
8) Write a Servlet that accepts a string from the user and displayed the string as a marquee in response.
9) Write a Servlet to accept a table name and to display all the records in the table.
10) Write a servlet that accepts roll number from a student and obtains the result "Pass Class", "First Class" etc by checking the appropriate fields from the students table.

9.8 FURTHER READING

- Eric Jendrock, Jennifer Ball, D Carson and others, The Java EE 5 Tutorial, Pearson Education, Third Edition, 2003

- Bryan Basham, Kathy Sierra, Bert Bates, Head First Servlets and JSP, O'reilly (SPD), Second Edition, 2008

- The Java Tutorials of Sun Microsystems Inc.

❖❖❖❖

10

ADVANCE SERVLETS

Unit Structure:

10.0 Objectives
10.1 Status Codes
10.2 Filtering Requests and Responses
10.3 Cookies
10.4 HttpSession
10.5 Summary
10.6 Unit end exercise
10.7 Further Reading

10.0 OBJECTIVES

The objective of this chapter is to learn the advance features os servlets such as status codes, filtering, cookies and session.

10.1 STATUS CODES

The HTTP response status line consists of an HTTP version, a status code, and an associated message. Since the message is directly associated with the status code and the HTTP version is determined by the server, all a servlet needs to do is to set the status code. A code of 200 is set automatically, so servlets don't usually need to specify a status code at all. When they do want to, they use response.setStatus, response.sendRedirect, or response.sendError.

Setting Arbitrary Status Codes: setStatus

When you want to set an arbitrary status code, do so with the setStatus method of HttpServletResponse. If your response includes a special status code and a document, be sure to call setStatus before actually returning any of the content with the PrintWriter. The reason is that an HTTP response consists of the status line, one or more headers, a blank line, and the actual document, in that order. Servlets do not necessarily buffer the document, so you have to either set the status code before using the PrintWriter or carefully check that the buffer hasn't been flushed and content actually sent to the browser.

The setStatus method takes an int (the status code) as an argument, but instead of using explicit numbers, for readability and to avoid typos, use the constants defined in HttpServletResponse. The name of each constant is derived from the standard HTTP 1.1 message for each constant, all upper case with a prefix of SC (for Status Code) and spaces changed to underscores. Thus, since the message for 404 is Not Found, the equivalent constant in HttpServletResponse is SC_NOT_FOUND.

Setting 302 and 404 Status Codes: sendRedirect and send Error

Although the general method of setting status codes is simply to call response.setStatus(int), there are two common cases for which a shortcut method in HttpServletResponse is provided. Just be aware that both of these methods throw IOException, whereas setStatus does not. Since the doGet and doPost methods already throw IOException, this difference only matters if you pass the response object to another method.

public void sendRedirect(String url) - The 302 status code directs the browser to connect to a new location. The sendRedirect method generates a 302 response along with a Location header giving the URL of the new document. Either an absolute or a relative URL is permitted; the system automatically translates relative URLs into absolute ones before putting them in the Location header.

public void sendError(int code, String message) - The 404 status code is used when no document is found on the server. The sendError method sends a status code (usually 404) along with a short message that is automatically formatted inside an HTML document and sent to the client.

Setting a status code does not necessarily mean that you omit the document. For example, although most servers automatically generate a small File Not Found message for 404 responses, a servlet might want to customize this response. Again, remember that if you do send output, you have to call setStatus or sendError first.

These codes fall into five general categories:

100–199: Codes in the 100s are informational, indicating that the client should respond with some other action.

200–299: Values in the 200s signify that the request was successful.

300–399: Values in the 300s are used for files that have moved and usually include a Location header indicating the new address.

400–499: Values in the 400s indicate an error by the client.

500–599: Codes in the 500s signify an error by the server.

Example: Write a Servlet that sends IE users to the Netscape home page and Netscape (and all other) users to the Microsoft home page

```
import javax.servlet.*;
import javax.servlet.http.*;
public class WrongDestination extends HttpServlet
{
public void doGet(HttpServletRequest request,
                        HttpServletResponse response)
                        throws ServletException, IOException
{
        String userAgent = request.getHeader("User-Agent");
        if ((userAgent != null) &&(userAgent.indexOf("MSIE") != -1))
                response.sendRedirect("http://home.netscape.com");
        else
                response.sendRedirect("http://www.microsoft.com");
}
}
```

10.2 FILTERING REQUESTS AND RESPONSES

A filter is an object that can transform the header and content (or both) of a request or response. Filters differ from web components in that filters usually do not themselves create a response. Instead, a filter provides functionality that can be “attached” to any kind of web resource. Consequently, a filter should not have any dependencies on a web resource for which it is acting as a filter; this way it can be composed with more than one type of web resource.

The main tasks that a filter can perform are as follows:

- Query the request and act accordingly.
- Block the request-and-response pair from passing any further.
- Modify the request headers and data. You do this by providing a customized version of the request.

- Modify the response headers and data. You do this by providing a customized version of the response.
- Interact with external resources.

Applications of filters include authentication, logging, image conversion, data compression, encryption, tokenizing streams, XML transformations, and so on. You can configure a web resource to be filtered by a chain of zero, one, or more filters in a specific order. This chain is specified when the web application containing the component is deployed and is instantiated when a web container loads the component.

In summary, the tasks involved in using filters are

- Programming the filter
- Programming customized requests and responses
- Specifying the filter chain for each web resource

Programming Filters

The filtering API is defined by the Filter, FilterChain, and FilterConfig interfaces in the javax.servlet package. You define a filter by implementing the Filter interface. The most important method in this interface is **doFilter**, which is passed request, response, and filter chain objects. This method can perform the following actions:

- Examine the request headers.
- Customize the request object if the filter wishes to modify request headers or data.
- Customize the response object if the filter wishes to modify response headers or data.
- Invoke the next entity in the filter chain. If the current filter is the last filter in the chain that ends with the target web component or static resource, the next entity is the resource at the end of the chain; otherwise, it is the next filter that was configured in the WAR. The filter invokes the next entity by calling the doFilter method on the chain object (passing in the request and response it was called with, or the wrapped versions it may have created). Alternatively, it can choose to block the request by not making the call to invoke the next entity. In the latter case, the filter is responsible for filling out the response.
- Examine response headers after it has invoked the next filter in the chain.
- Throw an exception to indicate an error in processing.

In addition to doFilter, you must implement the init and destroy methods. The init method is called by the container when the filter is instantiated. If you wish to pass initialization parameters to the filter, you retrieve them from the FilterConfig object passed to init.

10.3 COOKIES

Cookies are small bits of textual information that a Web server sends to a browser and that the browser later returns unchanged when visiting the same Web site or domain. By letting the server read information it sent the client previously, the site can provide visitors with a number of conveniences such as presenting the site the way the visitor previously customized it or letting identifiable visitors in without their having to reenter a password.

Benefits of Cookies

There are four typical ways in which cookies can add value to your site. We summarize these benefits below:

- Identifying a user during an e-commerce session - This type of short-term tracking is so important that another API is layered on top of cookies for this purpose.
- Remembering usernames and passwords - Cookies let a user log in to a site automatically, providing a significant convenience for users of unshared computers.
- Customizing sites - Sites can use cookies to remember user preferences.
- Focusing advertising - Cookies let the site remember which topics interest certain users and show advertisements relevant to those interests.

Sending cookies to the client involves three steps:

- Creating a Cookie object - You call the Cookie constructor with a cookie name and a cookie value, both of which are strings.
- Setting the maximum age - If you want the browser to store the cookie on disk instead of just keeping it in memory, you use setMaxAge to specify how long (in seconds) the cookie should be valid.
- Placing the Cookie into the HTTP response headers - You use response.addCookie to accomplish this. If you forget this step, no cookie is sent to the browser!

To **read the cookies** that come back from the client, you should perform the following two tasks, which are summarized below:

- Call request.getCookies. This yields an array of Cookie objects.

- Loop down the array, calling getName on each one until you find the cookie of interest. You then typically call getValue and use the value in some application-specific way.

Here are the methods that set the cookie attributes:
public void setMaxAge(int lifetime)
public int getMaxAge()

These methods tell how much time (in seconds) should elapse before the cookie expires. A negative value, which is the default, indicates that the cookie will last only for the current browsing session (i.e., until the user quits the browser) and will not be stored on disk. Specifying a value of 0 instructs the browser to delete the cookie.

public String getName()

The getName method retrieves the name of the cookie. The name and the value are the two pieces you virtually always care about. However, since the name is supplied to the Cookie constructor, there is no setName method; you cannot change the name once the cookie is created. On the other hand, getName is used on almost every cookie received by the server. Since the getCookies method of HttpServletRequest returns an array of Cookie objects, a common practice is to loop down the array, calling getName until you have a particular name, then to check the value with getValue.

public void setValue(String cookieValue)
public String getValue()

The setValue method specifies the value associated with the cookie; getValue looks it up. Again, the name and the value are the two parts of a cookie that you almost always care about, although in a few cases, a name is used as a boolean flag and its value is ignored (i.e., the existence of a cookie with the designated name is all that matters). However, since the cookie value is supplied to the Cookie constructor, setValue is typically reserved for cases when you change the values of incoming cookies and then send them back out.

Example: Write a program which stores a Cookie and the read the cookie to display the information.

```
//Part I
<html><body><center>
<form name=form1 method=get action=”AddCookieServlet”>
<B>Enter a Value</B>
<input type=text name=data>
<input type=submit>
</form></center></body></html>
```

```
//Part II
import javax.servlet.*;
import javax.servlet.http.*;
public class AddCookieServlet extends HttpServlet
{
        public void doGet(
                HttpServletRequest req, HttpServletResponse res)
                throws IOException, ServletException
        {
                String data = req.getParametar();
                Cookie c = new Cookie("My Cookie",data);
                res.addCookie(c);
                res.setCountentType("text/html");
                PrintWriter out = res.getWriter();
                out.println("My Cookie has been set to");
                out.println(data);
        }
}

//Part III
import javax.servlet.*;
import javax.servlet.http.*;
public class GetCookie extends HttpServlet
{
        public void doGet(
                HttpServletRequest req,HttpServletResponse res)
                throws IOException, ServletException
        {
                Cookie c[] = req.getCookies();
                res.setContentType("text/html");
                PrintWriter out = res.getWriter();
                for(int i = 0; i < c.length; i++)
                {
                        String name = c[i].getName();
                        String value = c[i].getValue();
                        out.println(name +"\t"+ value);
                }
        }
}
```

10.4 HTTP SESSION

It provides a way to identify a user across more than one page request or visit to a Web site. The servlet engine uses this interface to create a session between an HTTP client and an HTTP server. The session persists for a specified time period, across more than one connection or page request from the user. A session usually corresponds to one user, who may visit a site many times. The server can maintain a session either by using cookies or by rewriting URLs.

This interface allows servlets to

- View and manipulate information about a session, such as the session identifier, creation time, or context
- Bind objects to sessions, allowing you to use an online shopping cart to hold data that persists across multiple user connections

HttpSession defines methods that store these types of data:

- Standard session properties, such as a session identifier or session context
- Data that the application provides, accessed using this interface and stored using a dictionary-like interface

An HTTP session represents the server's view of the session. The server considers a session new under any of these conditions:

- The client does not yet know about the session
- The session has not yet begun
- The client chooses not to join the session, for example, if the server supports only cookies and the client rejects the cookies the server sends

When the session is new, the isNew() method returns true.

Method Summary	
String getId()	Returns a string containing the unique identifier assigned to this session.
Object getValue(String name)	Returns the object bound with the specified name in this session or null if no object of that name exists.
String[]getValueNames()	Returns an array containing the names of all the objects bound to this session.
boolean isNew()	Returns true if the Web server has created a session but the client has not yet joined.

void setAttribute(String name, Object value)	This method binds an object to this session, using the name specified.
Object getAttribute(String name)	This method returns the object bound with the specified name in this session, or null if no object is bound under the name.

Example: Create a form, which accepts user information such as name and background color. Session allows storing the client information about name and background. If the user is new then display the page asking for name and background else set the background and find number of visits to the page.

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class SessionServlet extends HttpServlet
{
public void service(
		HttpServletResponse res,HttpServletRequest req)
		throws IOException,ServletException
{
	try {
	res.setContentType("Text/Html");
	Integer hitCount;
	PrintWriter out=res.getWriter();
	HttpSession s=req.getSession(true);
	if(s.isNew()){
		out.println("<Html>");
		out.println("<Form method="+"GET"+"  action
		=http://localhost:8080/servlet/SessionServlet>");
		out.println("<b>Please select bgcolor</b>");
		out.println("<input type=radio name=optColor
				value=red>Red");
		out.println("<input type=radio name=optColor
				value=green>Green");
		out.println("<input type=radio name=optColor
				value=blue>Blue");
		out.println("<input type=text name=txtName>");
		out.println("<br><br>");
		out.println("<input type=submit value=Submit>");
```

```
                out.println("</form></Html>");
        }//if
        else{
                String name=(String)req.getParameter("txtName");
                String color=(String)req.getParameter("optColor");
                if(name!=null && color!=null){
                        out.println("Name: "+name);
                        hitCount=new Integer(1);
                        out.println("<a
                        href=SessionServlet>SessionServlet");
                        s.setAttribute("txtName",name);
                        s.setAttribute("optColor",color);
                        s.setAttribute("Hit",hitCount);
                }else{
                        hitCount=(Integer)s.getValue("Hit");
                        hitCount=new Integer(hitCount.intValue()+1);
                        s.putValue("Hit",hitCount);
                        out.println("<Html><body text=cyan bgcolor="
                                +s.getAttribute("optColor")+">");
                        out.println("You Have Been Selected"
                                +s.getAttribute("optColor")+"Color");
                        out.println("<br><br>Your Name Is"
                                +s.getAttribute("txtName"));
                        out.println("<br><br>Number Of Visits==>"
                                        +hitCount);
                        out.println("<br><br>");
                        out.println("<a
                        href=SessionServlet>SessionServlet</a>");
                        out.println("</body></html>");
                }
        }
}//try
catch(Exception e){}
} }//class
```

10.5 SUMMARY

- When you want to set a status code, we use the setStatus method of HttpServletResponse.
- A filter is an object that can transform the header and content (or both) of a request or response.
- Cookies are small bits of textual information that a Web server sends to a browser and that the browser later returns unchanged when visiting the same Web site or domain.
- The session persists for a specified time period, across more than one connection or page request from the user.

10.6 UNIT END EXERCISE

1) Explain the use of the following methods
    a. setStatus
    b. sendRedirect
    c. sendError
2) What are Cookies? State the benefits of using Cookies?
3) Explain with an example how Cookie class is used.
4) Write a short note on HttpSession?
5) Write a servlet that accepts a number and name from an HTML file, compares the number with predefined number and returns a message " You win" or "You lose" if the user's number matches the predefined number similar to lottery.
6) Write a Servlet that accepts user's information using three pages, first page accepts personal information, second accepts academic information and the third accepts extra-curricular information. The Servlet stores the data in sessions and in the end displays all the information.

10.7 FURTHER READING

- Eric Jendrock, Jennifer Ball, D Carson and others, The Java EE 5 Tutorial, Pearson Education, Third Edition, 2003
- Bryan Basham, Kathy Sierra, Bert Bates, Head First Servlets and JSP, O'reilly (SPD), Second Edition, 2008
- The Java Tutorials of Sun Microsystems Inc.

❖❖❖❖

11

INTRODUCTION TO JSP

Unit Structure:

11.0 Objectives
11.1 Introduction to JSP
11.2 The Life Cycle of a JSP Page
11.3 JSP Syntax Basics
11.4 Unified Expression Language
11.5 Summary
11.6 Unit end exercise
11.7 Further Reading

11.0 OBJECTIVES

The objectives of this chapter are to learn what JSP is and how to create useful JSP pages. In this chapter we will cover the basic of JSP, lifecycle of JSP and the expression language.

11.1 INTRODUCTION TO JSP

JSP enjoys cross-platform and cross-Web-server support, but effectively melds the power of server-side Java technology with the WYSIWYG features of static HTML pages. JSP pages typically comprise of:

- Static HTML/XML components.
- Special JSP tags
- Optionally, snippets of code written in the Java programming language called "scriptlets."

JSP Advantages

- **Separation of static from dynamic content:** With servlets, the logic for generation of the dynamic content is an intrinsic part of the servlet itself, and is closely tied to the static presentation templates responsible for the user interface. Thus, even minor changes made to the UI typically result in the recompilation of the servlet. This tight coupling of presentation and content results in brittle, inflexible applications. However, with JSP, the logic to generate the dynamic content is kept separate from the static presentation templates by encapsulating it within external JavaBeans components. These are then created and used by the JSP page using special tags and scriptlets. When a page designer makes any changes to the presentation template, the JSP page is automatically recompiled and reloaded into the web server by the JSP engine.

- **Write Once Run Anywhere:** JSP technology brings the "Write Once, Run Anywhere" paradigm to interactive Web pages. JSP pages can be moved easily across platforms, and across web servers, without any changes.
- **Dynamic content can be served in a variety of formats:** There is nothing that mandates the static template data within a JSP page to be of a certain format. Consequently, JSP can service a diverse clientele ranging from conventional browsers using HTML/DHTML, to handheld wireless devices like mobile phones and PDAs using WML, to other B2B applications using XML.
- **Completely leverages the Servlet API:** If you are a servlet developer, there is very little that you have to "unlearn" to move over to JSP. In fact, servlet developers are at a distinct advantage because JSP is nothing but a high-level abstraction of servlets. You can do almost anything that can be done with servlets using JSP--but more easily!

Comparing JSP with ASP

Although the features offered by JSP may seem similar to that offered by Microsoft's Active Server Pages (ASP), they are fundamentally different technologies, as shown by the following table:

	Java Server Pages	**Active Server Pages**
Web Server Support	Most popular web servers including Apache, Netscape, and Microsoft IIS can be easily enabled with JSP.	Native support only within Microsoft IIS or Personal Web Server. Support for select servers using third-party products.
Platform Support	Platform independent. Runs on all Java-enabled platforms.	Is fully supported under Windows. Deployment on other platforms is cumbersome due to reliance on the Win32-based component model.

Component Model	Relies on reusable, cross-platform components like JavaBeans, Enterprise JavaBeans, and custom tag libraries.	Uses the Win32-based COM component model.
Scripting	Can use the Java programming language or JavaScript.	Supports VBScript and JScript for scripting.
Security	Works with the Java security model.	Can work with the Windows NT security architecture.
Database Access	Uses JDBC for data access.	Uses Active Data Objects for data access.
Customizable Tags	JSP is extensible with custom tag libraries.	Cannot use custom tag libraries and is not extensible.

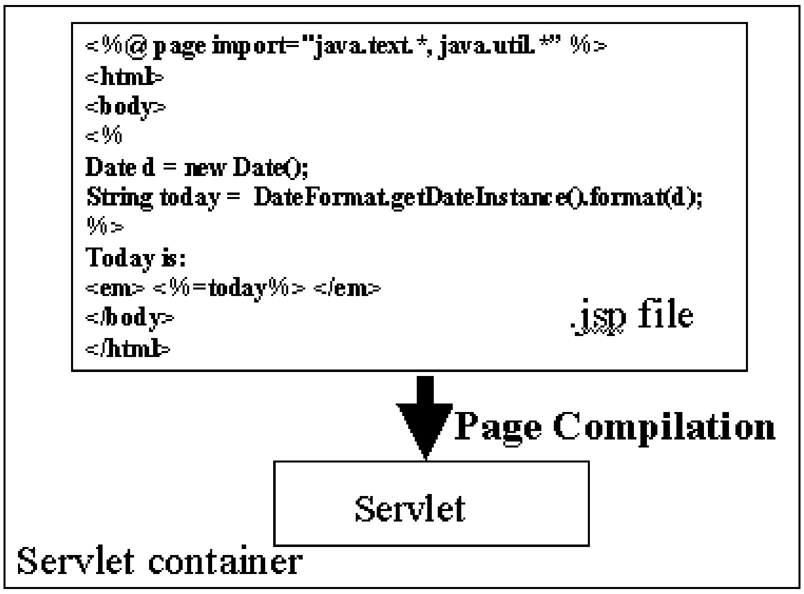
Example: Write a JSP page to display the current date and time.

```
<Html>
<Head>
<Title>JSP Expressions</Title>
</Head>
<Body>
<H2>JSP Expressions</H2>
<ul>
        <li>Current time:      <%= new java.util.Date() %>
        <li>Server:            <%= application.getServerInfo() %>
        <li>Session Id:        <%= session.getId() %>
        <li>The <code>test param</code> form parameter:
<%= request.getParameter("testParam")%>
</ul>
</Body>
</Html>
```

11.2 THE LIFE CYCLE OF A JSP PAGE

The purpose of JSP is to provide a declarative, presentation-centric method of developing servlets. As noted before, the JSP specification itself is defined as a standard extension on top the Servlet API. Consequently, it should not be too surprisingly that under the covers, servlets and JSP pages have a lot in common.

Typically, JSP pages are subject to a translation phase and a request processing phase. The translation phase is carried out only once, unless the JSP page changes, in which case it is repeated. Assuming there were no syntax errors within the page, the result is a JSP page implementation class file that implements the Servlet interface, as shown below.

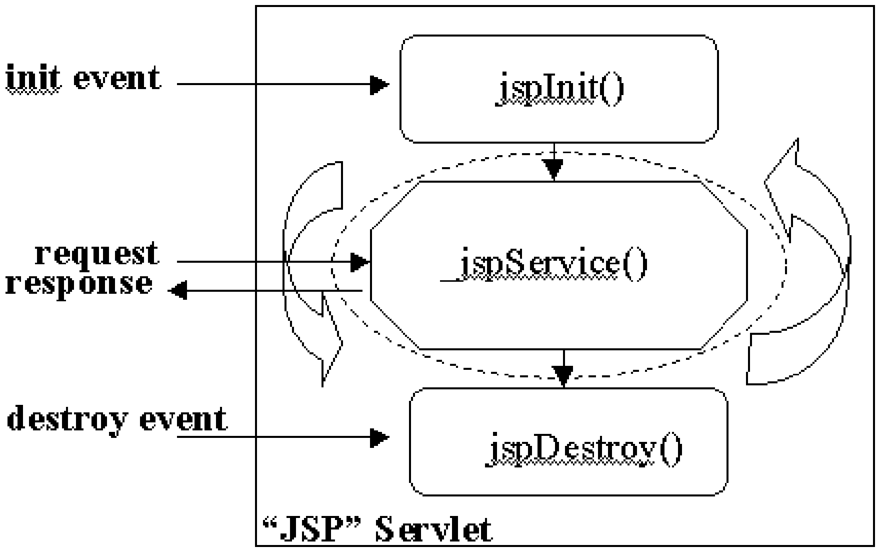


```
<%@ page import="java.text.*, java.util.*" %>
<html>
<body>
<%
Date d = new Date();
String today =  DateFormat.getDateInstance().format(d);
%>
Today is:
<em> <%=today%> </em>
</body>
</html>
```

The translation phase is typically carried out by the JSP engine itself, when it receives an incoming request for the JSP page for the first time. Many details of the translation phase, like the location where the source and class files are stored are implementation dependent.

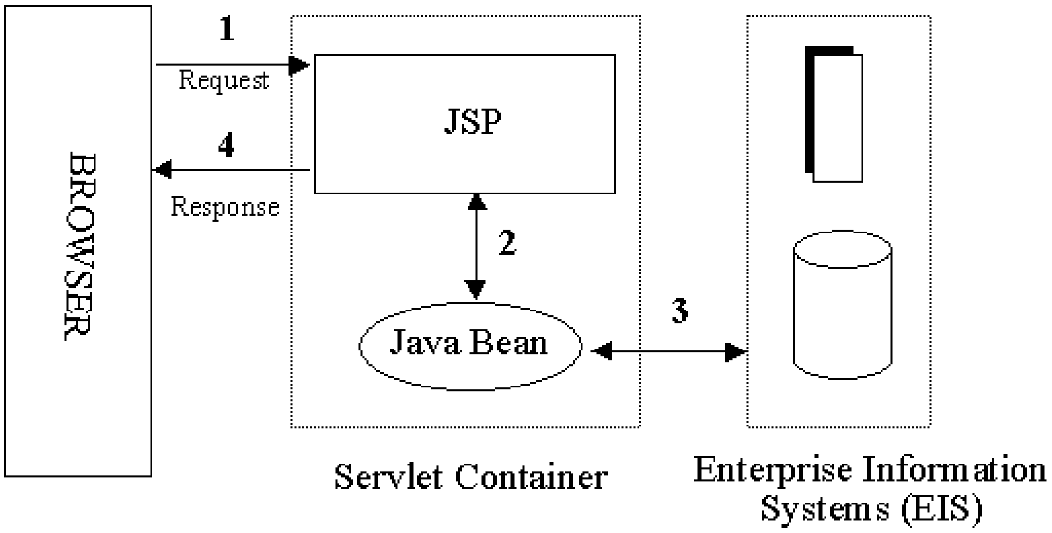
The JSP page implementation class file extends HttpJspBase, which in turn implements the Servlet interface. Observe how the service method of this class, _jspService(), essentially inlines the contents of the JSP page. Although _jspService() cannot be overridden, the developer can describe initialization and destroy events by providing implementations for the jspInit() and jspDestroy() methods within their JSP pages.

Once this class file is loaded within the servlet container, the _jspService() method is responsible for replying to a client's request. By default, the _jspService() method is dispatched on a separate thread by the servlet container in processing concurrent client requests, as shown below:



JSP Access Models

The early JSP specifications advocated two philosophical approaches, popularly known as Model 1 and Model 2 architectures, for applying JSP technology. Consider the Model 1 architecture, shown below:

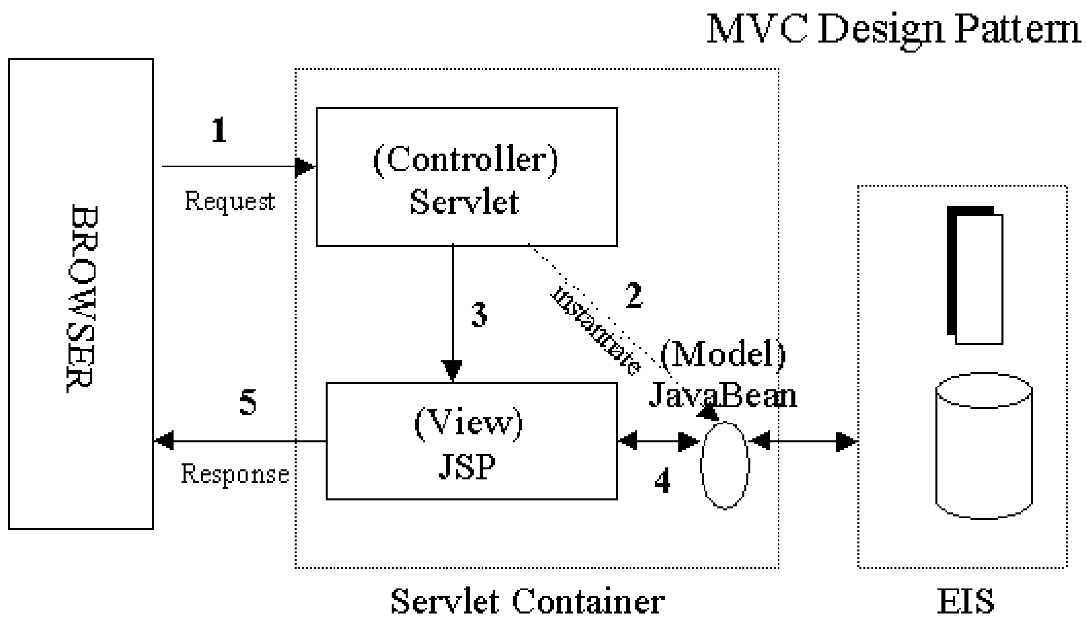


In the Model 1 architecture, the incoming request from a web browser is sent directly to the JSP page, which is responsible for processing it and replying back to the client. There is still separation of presentation from content, because all data access is performed using beans.

Although the Model 1 architecture is suitable for simple applications, it may not be desirable for complex implementations. Indiscriminate usage of this architecture usually leads to a significant amount of scriptlets or Java code embedded within the

JSP page, especially if there is a significant amount of request processing to be performed. While this may not seem to be much of a problem for Java developers, it is certainly an issue if your JSP pages are created and maintained by designers--which is usually the norm on large projects. Another downside of this architecture is that each of the JSP pages must be individually responsible for managing application state and verifying authentication and security.

The Model 2 architecture, shown below, is a server-side implementation of the popular Model/View/Controller design pattern. Here, the processing is divided between presentation and front components. Presentation components are JSP pages that generate the HTML/XML response that determines the user interface when rendered by the browser. Front components (also known as controllers) do not handle any presentation issues, but rather, process all the HTTP requests. Here, they are responsible for creating any beans or objects used by the presentation components, as well as deciding, depending on the user's actions, which presentation component to forward the request to. Front components can be implemented as either a servlet or JSP page.



The advantage of this architecture is that there is no processing logic within the presentation component itself; it is simply responsible for retrieving any objects or beans that may have been previously created by the controller, and extracting the dynamic content within for insertion within its static templates. Consequently, this clean separation of presentation from content leads to a clear delineation of the roles and responsibilities of the developers and page designers on the programming team. Another benefit of this approach is that the front components present a single point of entry into the application, thus making the management of application state, security, and presentation uniform and easier to maintain.

Example: Write a JSP file, which displays the parameters passed to the file.

Register.jsp

```
<Html>
<Head>
<Title>Register</Title>
</Head>
<form method=get action="http://localhost:8080/StudentInfo.jsp">
<table border=1>
<tr><td>Name:</td><td> <input type=text name=txtName></td>
<tr><td>Age: </td><td><input type=text name=txtAge></td>
<tr><td>Tel Nos: </td><td><input type=text name=txtTelNo></td>
<tr><td><input type=submit></td><td> <input type=reset></td>
</table>
</form>
</html>
```

StudentInfo.jsp

```
<html>
<head>
<Title>Student Info</Title>
</Head>
<Body>
<table border=1>
<tr><td>Name</td><td><%=request.getParameter("txtName")
%></td></tr>
<tr><td>Age</td><td><%=request.getParameter("txtAge")
%></td></tr>
<tr><td>Tel No</td><td><%=request.getParameter("txtTelNo")
%></td></tr>
</table>
</body>
</html>
```

Example: Write a JSP page, which displays three text boxes for Department Number, Department Name and Location. On click of the submit button call another JSP page which will

enter the values in the database with the help of PreparedStatement class. Also use jspInit() and jspDestroy() to open and close the connection. (Register.jsp).

DeptForm.jsp

```
<html>
<Head><title>Department Form</title>
</head>
<body>
<form method=GET action="http://localhost:8080/Register1.jsp">
<table>
<tr><td>DepartmentNo:  </td><td> <input type=text
name=txtNo></td></tr>
<tr><td>DepartmentName: </td><td><input type=text
name=txtName></td></tr>
<tr><td>Location:</td><td> <input type=text
name=txtLoc></td></tr>
</table>
<input type=submit name=Submit>
<input type=reset name=Reset>
</Form>
</body>
</Html>
```

Register1.jsp

```
<%@ page import="java.sql.*" %>
<%! String a,b,c,d,e,f,Query; %>
<%! Connection con;
Statement st;
PreparedStatement ps; %>
<%! int i,num; %>

<%!
public void jspInit()
{
try
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
con=DriverManager.getConnection("jdbc:odbc:ty289");
st=con.createStatement();
Query="insert into Dept values(?,?,?)";
```

```
ps=con.prepareStatement(Query);
}
catch(Exception e){System.out.println("Error: "+e.getMessage());}
}
%>
<%
a=(String)request.getParameter("txtNo");
b=(String)request.getParameter("txtName");
c=(String)request.getParameter("txtLoc");
ps.setInt(1,Integer.parseInt(a));
ps.setString(2,b);
ps.setString(3,c);
ps.executeUpdate();
con.commit();
ResultSet rs=st.executeQuery("select * from Dept");
%>
<html><body>
<table border=1>
<tr><th>Dept No </th><th>Dept Name</th><th>Location</th></tr>
<%
        while(rs.next())
        {
%>
        <tr>
<%      for(int j=0;j<=2;j++)
        {
                Object obj=rs.getObject(j+1);
%>
        <td><%=obj.toString()%></td>
<%
        }
        }
%>
</tr>
</table>
<%!
        public void jspDestroy()
{   try
        {
                ps.close();
        st.close();
        }
        catch(Exception e){System.out.println("Error:
"+e.getMessage());}
}%>
</body>
</html>
```

11.3 JSP SYNTAX BASICS

JSP syntax is fairly straightforward, and can be classified into directives, scripting elements, and standard actions.

Directives

JSP directives are messages for the JSP engine. They do not directly produce any visible output, but tell the engine what to do with the rest of the JSP page. JSP directives are always enclosed within the <%@ ... %> tag. The two primary directives are page and include.

Page Directive

Typically, the page directive is found at the top of almost all of your JSP pages. There can be any number of page directives within a JSP page, although the attribute/value pair must be unique. Unrecognized attributes or values result in a translation error. For example,

```
<%@ page import="java.util.*, com.foo.*" buffer="16k" %>
```

makes available the types declared within the included packages for scripting and sets the page buffering to 16K.

Purpose of the page Directive

• Give high-level information about the Servlet that will result from the JSP page

• Can control

– Which classes are imported

– What class the servlet extends

– If the servlet participates in sessions

– The size and behavior of the output buffer

– What page handles unexpected errors

The import Attribute

• Format

```
<%@ page import="package.class" %>
```

<%@ page import="package.class1,...,package.classN" %>

• Purpose

Generate import statements at top of servlet definition

Although JSP pages can be almost anywhere on server, classes used by JSP pages must be in normal servlet dirs

E.g.:

…/classes or

…/classes/directoryMatchingPackage

• Always use packages for utilities that will be used by JSP!

The session Attribute

• Format

<%@ page session="true" %> <%-- Default --%>

<%@ page session="false" %>

• Purpose

To designate that page not be part of a session

By default, it is part of a session

Saves memory on server if you have a high-traffic site

All related pages have to do this for it to be useful

The buffer Attribute

• Format

<%@ page buffer="sizekb" %>

<%@ page buffer="none" %>

• Purpose

To give the size of the buffer used by the out variable

Buffering lets you set HTTP headers even after some page content has been generated (as long as buffer has not filled up or been explicitly flushed)

Servers are allowed to use a larger size than you ask for, but not a smaller size

Default is system-specific, but must be at least 8kb

The errorPage Attribute

• Format

<%@ page errorPage="Relative URL" %>

• Purpose

Specifies a JSP page that should process any exceptions thrown but not caught in the current page

The exception thrown will be automatically available to the designated error page by means of the "exception" variable

The web.xml file lets you specify application-wide error pages that apply whenever certain exceptions or certain HTTP status codes result.
• The errorPage attribute is for page-specific error pages

The isErrorPage Attribute

• Format

<%@ page isErrorPage="true" %>

<%@ page isErrorPage="false" %> <%-- Default --%>

• Purpose

Indicates whether or not the current page can act as the error page for another JSP page

A new predefined variable called exception is created and accessible from error pages

Use this for emergency backup only; explicitly handle as many exceptions as possible

• Don't forget to always check query data for missing or malformed values

The extends Attribute

• Format

<%@ page extends="package.class" %>

• Purpose

To specify parent class of servlet that will result from JSP page

Use with extreme caution

Can prevent system from using high-performance custom superclasses

Typical purpose is to let you extend classes that come from the server vendor (e.g., to support personalization features), not to extend your own classes.

Declarations

JSP declarations let you define page-level variables to save information or define supporting methods that the rest of a JSP page may need. While it is easy to get led away and have a lot of code within your JSP page, this move will eventually turn out to be a maintenance nightmare. For that reason, and to improve reusability, it is best that logic-intensive processing is encapsulated as JavaBean components.

Declarations are found within the <%! ... %> tag. Always end variable declarations with a semicolon, as any content must be valid Java statements:
<%! int i=0; %>

You can also declare methods. For example, you can override the initialization event in the JSP life cycle by declaring:

```
<%! public void jspInit() {
        //some initialization code
   }
%>
```

Expressions

With expressions in JSP, the results of evaluating the expression are converted to a string and directly included within the output page. Typically expressions are used to display simple values of variables or return values by invoking a bean's getter methods. JSP expressions begin within <%= ... %> tags and do not include semicolons:

```
 <%= fooVariable %>
 <%= fooBean.getName() %>
```

Scriptlets

JSP code fragments or scriptlets are embedded within <% ... %> tags. This Java code is run when the request is serviced by the JSP page. You can have just about any valid Java code within a scriptlet, and is not limited to one line of source code. For example, the following displays the string "Hello" within H1, H2, H3, and H4 tags, combining the use of expressions and scriptlets:

```
<% for (int i=1; i<=4; i++) { %>
   <H<%=i%>>Hello</H<%=i%>>
<% } %>
```

Comments

Although you can always include HTML comments in JSP pages, users can view these if they view the page's source. If you don't want users to be able to see your comments, embed them within the <%-- ... --%> tag:

```
<%-- comment for server side only --%>
```

A most useful feature of JSP comments is that they can be used to selectively block out scriptlets or tags from compilation. Thus, they can play a significant role during the debugging and testing process.

Object Scopes

It is important to understand the scope or visibility of Java objects within JSP pages that are processing a request. Objects may be created implicitly using JSP directives, explicitly through actions, or, in rare cases, directly using scripting code. The instantiated objects can be associated with a scope attribute defining where there is a reference to the object and when that reference is removed. The following diagram indicates the various scopes that can be associated with a newly created object:

JSP Implicit Objects

As a convenience feature, the JSP container makes available implicit objects that can be used within scriptlets and expressions, without the page author first having to create them. These objects act as wrappers around underlying Java classes or interfaces typically defined within the Servlet API. The nine implicit objects:

- request: represents the HttpServletRequest triggering the service invocation. Request scope.
- response: represents HttpServletResponse to the request. Not used often by page authors. Page scope.
- pageContext: encapsulates implementation-dependent features in PageContext. Page scope.
- application: represents the ServletContext obtained from servlet configuration object. Application scope.
- out: a JspWriter object that writes into the output stream. Page scope.
- config: represents the ServletConfig for the JSP. Page scope.
- page: synonym for the "this" operator, as an HttpJspPage. Not used often by page authors. Page scope.
- session: An HttpSession. Session scope. More on sessions shortly.
- exception: the uncaught Throwable object that resulted in the error page being invoked. Page scope.

Note that these implicit objects are only visible within the system generated _jspService() method. They are not visible within methods you define yourself in declarations.

11.4 UNIFIED EXPRESSION LANGUAGE

The primary new feature of JSP 2.1 is the unified expression language (unified EL), which represents a union of the expression language offered by JSP 2.0 and the expression language created for JavaServer Faces technology

The expression language introduced in JSP 2.0 allows page authors to use simple expressions to dynamically read data from JavaBeans components. For example, the test attribute of the following conditional tag is supplied with an EL expression that compares the number of items in the session-scoped bean named cart with 0.

```
<c:if test="${sessionScope.cart.numberOfItems > 0}">
...
</c:if>
```

JSP supports a simple request/response life cycle, during which a page is executed and the HTML markup is rendered immediately. Therefore, the simple, read-only expression language offered by JSP 2.0 was well suited to the needs of JSP applications.

To summarize, the new, unified expression language allows page authors to use simple expressions to perform the following tasks:

- Dynamically read application data stored in JavaBeans components, various data structures, and implicit objects
- Dynamically write data, such as user input into forms, to JavaBeans components
- Invoke arbitrary static and public methods
- Dynamically perform arithmetic operations

The unified EL also allows custom tag developers to specify which of the following kinds of expressions that a custom tag attribute will accept:

- Immediate evaluation expressions or deferred evaluation expressions. An immediate evaluation expression is evaluated immediately by the JSP engine. A deferred evaluation expression can be evaluated later by the underlying technology using the expression language.
- Value expression or method expression. A value expression references data, whereas a method expression invokes a method.
- Rvalue expression or Lvalue expression. An rvalue expression can only read a value, whereas an lvalue expression can both read and write that value to an external object.
- Finally, the unified EL also provides a pluggable API for resolving expressions so that application developers can implement their own resolvers that can handle expressions not already supported by the unified EL.

11.5 SUMMARY

- JSP pages have cross-platform and cross-Web-server support, but effectively melds the power of server-side Java technology with the WYSIWYG features of static HTML pages.
- JSP pages are subject to a translation phase and a request processing phase.

- The incoming request from a web browser is sent directly to the JSP page, which is responsible for processing it and replying back to the client.
- JSP directives are messages for the JSP engine.
- The page directive is found at the top of JSP pages and gives high-level information about the Servlet that will result from the JSP page.
- The expression language introduced in JSP 2.0 allows page authors to use simple expressions to dynamically read data from JavaBeans components.

11.6 UNIT END EXERCISE

1) State and explain the advantages of JSP?
2) What are the major differences between JSP and ASP?
3) What are the advantages of using JSP over Servlets?
4) Describe various Implicit objects of the JSP? What is the scope of those objects?
5) Write a short note on JSP Access Model?
6) Explain PAGE directive with all its attribute.
7) What is meant by declaration in JSP? How it is different from scriplet?
8) Write a JSP page to display the current date and time.
9) Write a JSP page to connect to a database and display the contents of a database table using the HTML TABLE tag. The column names should also be fetched from the database.
10) Write a JSP page, which displays three text boxes for user name, password, and email. On click of the submit button call another JSP page which will enter the values in the database with the help of PreparedStatement class. Also use jspInit() and jspDestroy() to open and close the connection.

11.7 FURTHER READING

- Eric Jendrock, Jennifer Ball, D Carson and others, The Java EE 5 Tutorial, Pearson Education, Third Edition, 2003
- Bryan Basham, Kathy Sierra, Bert Bates, Head First Servlets and JSP, O’reilly (SPD), Second Edition, 2008
- The Java Tutorials of Sun Microsystems Inc.

❖❖❖❖

12

ADVANCE JSP

Unit Structure:

12.0 Objectives
12.1 Reusing Content in JSP Pages
12.2 Using JavaBeans Components
12.3 Using Custom tags
12.4 Transferring Control to another Web Component
12.5 Summary
12.6 Unit end exercise
12.7 Further Reading

12.0 OBJECTIVES

The objective of this chapter is to learn the advance concepts of JSP such as reusing content, custom tags and Java Beans Components. After this chapter you will be able to create more advance JSP pages.

12.1 REUSING CONTENT IN JSP PAGES

There are many mechanisms for reusing JSP content in a JSP page. Three mechanisms that can be categorized as direct reuse are discussed here:

- The include directive
- Preludes and codas
- The jsp:include element

The include directive is processed when the JSP page is translated into a servlet class. The effect of the directive is to insert the text contained in another file (either static content or another JSP page) into the including JSP page. You would probably use the include directive to include banner content, copyright information, or any chunk of content that you might want to reuse in another page. The syntax for the include directive is as follows: <%@ include file="filename" %>. For example, all the Duke's Bookstore application pages could include the file banner.jspf, which contains the banner content, by using the following directive: <%@ include file="banner.jspf" %>

Another way to do a static include is to use the prelude and coda mechanisms Because you must put an include directive in each file that reuses the resource referenced by the directive, this approach has its limitations. Preludes and codas can be applied only to the beginnings and ends of pages.

The jsp:include element is processed when a JSP page is executed. The include action allows you to include either a static or a dynamic resource in a JSP file. The results of including static and dynamic resources are quite different. If the resource is static, its content is inserted into the calling JSP file. If the resource is dynamic, the request is sent to the included resource, the included page is executed, and then the result is included in the response from the calling JSP page. The syntax for the jsp:include element is:
<jsp:include page="includedPage" />

Example: Write a JSP page that will include in it a simple static file with the help of <%@ include file="?" %> and a simple JSP page with the help of <jsp:include page="?"/> tag.

<u>Include.jsp</u>

```
<html>
<body bgcolor="white">
<br>
<H1 align="center">JavaServer Pages 1.0</H1>
<H2 align="center">Include Example</H2>
<P>&nbsp;</P>
<P>&nbsp;</P>
<font color="red">
<%@ page buffer="5" autoFlush="false" %>
<p>In place evaluation of another JSP which gives you the current
time:

<%@ include file="foo.jsp" %>

<p> <jsp:include page="/examples/jsp/samples/include/foo.html"
flush="true"/> by including the output of another JSP:

<jsp:include page="foo.jsp" flush="true"/>

</html>
```

<u>foo.jsp</u>

```
<body bgcolor="white">
<font color="red">
<%= System.currentTimeMillis() %>
```

12.2 USING JAVABEAN COMPONENTS

The component model for JSP technology is based on JavaBeans component architecture. JavaBeans components are nothing but Java objects, which follow a well-defined design/naming pattern: the bean encapsulates its properties by declaring them private and provides public accessor (getter/setter) methods for reading and modifying their values.

Before you can access a bean within a JSP page, it is necessary to identify the bean and obtain a reference to it. The <jsp:useBean> tag tries to obtain a reference to an existing instance using the specified id and scope, as the bean may have been previously created and placed into the session or application scope from within a different JSP page. The bean is newly instantiated using the Java class name specified through the class attribute only if a reference was not obtained from the specified scope. Consider the tag:

```
        <jsp:useBean id="user" class="beans.Person"
scope="session" />
```

In this example, the Person instance is created just once and placed into the session. If this useBean tag is later encountered within a different JSP page, a reference to the original instance that was created before is retrieved from the session.

The <jsp:useBean> tag can also optionally include a body, such as

```
<jsp:useBean id="user" class="beans.Person" scope="session">
<%
      user.setDate(DateFormat.getDateInstance().format(new
Date()));
      / /etc..
%>
</jsp:useBean>
```

Any scriptlet (or <jsp:setProperty> tags, which are explained shortly) present within the body of a <jsp:useBean> tag are executed only when the bean is instantiated, and are used to initialize the bean's properties.

Once you have declared a JavaBean component, you have access to its properties to customize it. The value of a bean's property is accessed using the <jsp:getProperty> tag. With the <jsp:getProperty> tag, you specify the name of the bean to use (from the id field of useBean), as well as the name of the property whose value you are interested in. The actual value is then directly printed to the output:

```
<jsp:getProperty name="user" property="name" />
```

Changing the property of a JavaBean component requires you to use the <jsp:setProperty> tag. For this tag, you identify the bean and property to modify and provide the new value:

```
<jsp:setProperty name="user" property="name" value="jGuru" />
```

or

```
<jsp:setProperty name="user" property="name"
value="<%=expression %>" />
```

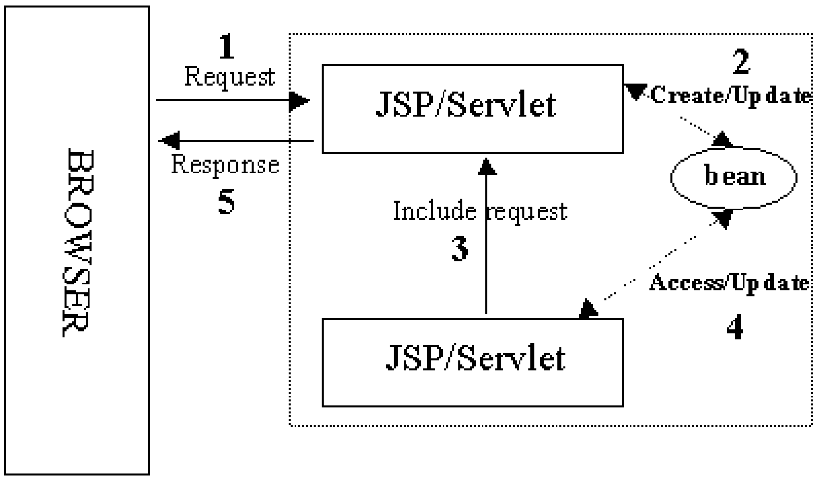
When developing beans for processing form data, you can follow a common design pattern by matching the names of the bean properties with the names of the form input elements. You also need to define the corresponding getter/setter methods for each property within the bean. The advantage in this is that you can now direct the JSP engine to parse all the incoming values from the HTML form elements that are part of the request object, then assign them to their corresponding bean properties with a single statement, like this:

```
<jsp:setProperty name="user" property="*"/>
```

This runtime magic is possible through a process called introspection, which lets a class expose its properties on request. The introspection is managed by the JSP engine, and implemented through the Java reflection mechanism. This feature alone can be a lifesaver when processing complex forms containing a significant number of input elements.

If the names of your bean properties do not match those of the form's input elements, they can still be mapped explicitly to your property by naming the parameter as:

```
<jsp:setProperty name="user" property="address"
param="parameterName" />
```



Example: Create a java bean that gives information about the current time. The bean has getter properties for time, hour, minute, and second. Write a JSP page that uses the bean and display all the information.

```
package myclass;
import java.util.Calendar;
import java.util.Date;
public class CalendarBean
{
        private Calendar calendar;
        public CalendarBean()     {
                calendar=Calendar.getInstance();
        }
        public Date getTime()              {
                return calendar.getTime();
        }
        public int getHour()          {
                return calendar.get(Calendar.HOUR_OF_DAY);
        }
        public int getMinute()             {
                return calendar.get(Calendar.MINUTE);
        }
        public int getSecond()             {
                return calendar.get(Calendar.SECOND);
        }
}
```

```
BeanTime.jsp
<html>
<body>
<jsp:useBean class="myclass.CalendarBean" id="cal" />
<pre>
        Time:  <jsp:getProperty name="cal" property="Time" /><br>
        Hour:  <jsp:getProperty name="cal" property="Hour" /><br>
        Minute:<jsp:getProperty name="cal" property="Minute"
        /><br>
        Seconds:<jsp:getProperty name="cal" property="Second"
        /><br>
</pre>
</body>
</html>
```

12.3 USING CUSTOM TAGS

Custom tags are user-defined JSP language elements that encapsulate recurring tasks. Custom tags are distributed in a tag library, which defines a set of related custom tags and contains the objects that implement the tags.

Custom tags have the syntax

```
        <prefix:tag attr1="value" ... attrN="value" />
```

or

```
        <prefix:tag attr1="value" ... attrN="value" >
        body
        </prefix:tag>
```

where prefix distinguishes tags for a library, tag is the tag identifier, and attr1 ... attrN are attributes that modify the behavior of the tag.

To use a custom tag in a JSP page, you must

- Declare the tag library containing the tag
- Make the tag library implementation available to the web application

12.4 TRANSFERRING CONTROL TO ANOTHERWEB COMPONENT

The mechanism for transferring control to another web component from a JSP page uses the functionality provided by the Java Servlet API. You access this functionality from a JSP page by using the jsp:forward element:

```
        <jsp:forward page="/main.jsp" />
```

Note that if any data has already been returned to a client, the jsp:forward element will fail with an IllegalStateException.

jsp:param Element

When an include or forward element is invoked, the original request object is provided to the target page. If you wish to provide additional data to that page, you can append parameters to the request object by using the jsp:param element:

```
<jsp:include page="..." >
        <jsp:param name="param1" value="value1"/>
</jsp:include>
```

When jsp:include or jsp:forward is executed, the included page or forwarded page will see the original request object, with the original parameters augmented with the new parameters and new values taking precedence over existing values when applicable. For example, if the request has a parameter A=foo and a parameter A=bar is specified for forward, the forwarded request will have A=bar,foo.Note that the new parameter has precedence.

The scope of the new parameters is the jsp:include or jsp:forward call; that is, in the case of an jsp:include the new parameters (and values) will not apply after the include.

12.5 SUMMARY

- Three mechanisms for reusing JSP content in a JSP page are include directive, Preludes & codas and jsp:include element.

- The <jsp:useBean> tag tries to obtain a reference to an existing instance using the specified id and scope, as the bean may have been previously created and placed into the session or application scope from within a different JSP page.

- Custom tags are distributed in a tag library, which defines a set of related custom tags and contains the objects that implement the tags.

- Using the jsp:forward element we can transfer control to another web component from a JSP page.

12.6 UNIT END EXERCISE

1) Explain INCLUDE directive with all its attribute.
2) Describe the jsp:useBean tag with an example?
3) Expalin how control can be transferred to another Web Component.
4) Write a JSP page that will include in it a simple static file with the help of <%@ include file="?" %> and a simple JSP page with the help of <jsp:include page="?"/> tag.
5) Create a java bean that gives information about the current time. The bean has getter properties for time, hour, minute, and second. Write a JSP page that uses the bean and display all the information.
6) Create a multi-page registration form in which the user input is spread across 3 pages. The data is stored in the session with the help of Java Beans. After all the information is entered, read the contents of the java bean and display the contents on a new page.

12.7 FURTHER READING

- Eric Jendrock, Jennifer Ball, D Carson and others, The Java EE 5 Tutorial, Pearson Education, Third Edition, 2003
- Bryan Basham, Kathy Sierra, Bert Bates, Head First Servlets and JSP, O'reilly (SPD), Second Edition, 2008
- The Java Tutorials of Sun Microsystems Inc.

❖❖❖❖

13

INTRODUCTION TO EJB

Unit Structure:

13.0 Objectives
13.1 Introduction to EJB
13.2 Benefits of EJB
13.3 Difference between JavaBeans and Enterprise JavaBeans
13.4 JEE Architecture overview
13.5 JEE Application components
13.6 Java EE Clients
13.7 Summary
13.8 Unit end exercise
13.9 Further Reading

13.0 OBJECTIVES

The objectives of this chapter are to learn what EJB is and it works. Here we will also understand the difference between EJB and beans, architecture and components.

13.1 INTRODUCTION TO EJB

- EJB is defined as an architecture for the development and deployment of component-based, robust, highly scalable business applications. By using EJB, you can write scalable, reliable, and secure applications without writing your own complex distributed component framework.
- EJB is about rapid application development for the server side. You can quickly and easily construct server-side components in Java. This can be done by leveraging a prewritten distributed infrastructure provided by the industry.
- EJB is designed to support application portability and reusability across Enterprise middleware services of any vendor.

13.2 BENEFITS OF EJB

- Component portability - The EJB architecture provides a simple, elegant component container model. Java server components can be developed once and deployed in any EJB-compliant server.
- Architecture independence - The EJB architecture is independent of any specific platform, proprietary protocol, or middleware infrastructure. Applications developed for one platform can be redeployed on other platforms.
- Developer productivity - The EJB architecture improves the productivity of application developers by standardizing and automating the use of complex infrastructure services such as transaction management and security checking. Developers can create complex applications by focusing on business logic rather than environmental and transactional issues.
- Customization - Enterprise bean applications can be customized without access to the source code. Application behavior and runtime settings are defined through attributes that can be changed when the enterprise bean is deployed.
- Multitier technology - The EJB architecture overlays existing infrastructure services.
- Versatility and scalability - The EJB architecture can be used for small-scale or large-scale business transactions. As processing requirements grow, the enterprise beans can be migrated to more powerful operating environments.

13.3 DIFFERENCE BETWEEN JAVABEANS AND ENTERPRISE JAVABEANS

Enterprise JavaBeans	**JavaBeans**
1. They are non-visible remote objects.	1. They can be either visible or non-visible.
2. They are remotely executable components deployed on the server.	2. They are intended to be local to a single process on the client side.
3. They use the Deployment Descriptor to describe themselves.	3. They use BeanInfo classes, and Property Editors. And they customize to describe themselves.
4. They cannot be deployed as ActiveX control, since OCXs run on desktop.	4. They can also be deployed as ActiveX controls.

Enterprise JavaBeans can be used while:

- developing the reusable business logic component in enterprise application
- developing a fast growing distributed application, which is scalable
- application supports transaction management to ensure the integrity of the database
- application deals with variety of clients and session management for thousands of clients

13.4 JEE ARCHITECTURE OVERVIEW

The aim of the Java EE 5 platform is to provide developers a powerful set of APIs. This is in order to:

- reduce development time
- reduce application complexity
- improve application performance

The Java EE platform uses a distributed Multi-Tiered application model for Enterprise applications. The application logic is divided to form components according to the function. The various application components that make up a Java EE application are installed on different machines. This installation depends on the tier in the multi-tiered Java EE environment to which the application component belongs.

Java EE applications are divided in the tiers described in the following list:

- Client-Tier components run on the Client machine
- Web-Tier components run on the Java EE server
- Business-Tier components run on the Java EE server
- Enterprise Information System (EIS)-Tier software run on the EIS server

A Java EE application can consist of three or four tiers. However, Java EE Multi-Tiered applications are generally considered to be Three-Tiered applications because they are distributed over three locations:
1. the Client machines
2. the Java EE server machine
3. the database or legacy machines at the back end

The Three-Tiered applications that run in this manner extend the standard Two-Tiered “Client and Server” model by placing a “Multi-threaded application server” between the “Client application” and the “back-end storage”.+

13.5 JEE APPLICATION COMPONENTS:

Java EE applications are made up of components. A Java EE component is a self-contained functional software unit.

- The Java EE component is assembled in a Java EE application with its related classes and files.
- The Java EE component communicates with other components, as well.

The Java EE specification defines the following Java EE components:

1. Application clients and applets: They are components that run on the client.
2. Java Servlet, JavaServer Faces, and JavaServer Pages technology components: They are web components that run on the server.
3. Enterprise JavaBeans (EJB) components (Enterprise beans): They are business components that run on the server.

Java EE components are written in the Java programming language and are compiled in the same way as any program in the language. However, the difference between Java EE components and standard Java classes is that Java EE components are assembled in a Java EE application. Here they are verified to be well formed and in compliance with the Java EE specification. Then they are deployed to production, where they are run and managed by the Java EE server.

13.6 JAVA EE CLIENTS:

13.6.1. Web Clients:

A web client consists of two parts:

i. dynamic web pages that contain various types of markup languages (HTML, XML, and so on), which are generated by web components running in the web tier, and

ii. a web browser, which renders the pages received from the server.

A web client is sometimes called a “thin client”. Thin clients usually do not query databases, execute complex business rules, or connect to legacy applications. When you use a “thin client”, such heavyweight operations are off-loaded to Enterprise beans executing on the Java EE server. Therein they can leverage the security, speed, services, and reliability of Java EE server-side technologies.

13.6.2. Applets:

A web page received from the web tier can include an “embedded applet”. An applet is a small client application written in the Java programming language that executes in the Java Virtual Machine installed in the web browser. However, client systems will likely need the Java Plug-in, and possibly a security policy file, for the applet to successfully execute in the web browser.

Web components are the preferred API for creating a web client program because no plug-ins or security policy files are required on the client systems. The web components enable a cleaner and more modular application design, as well. This is because the web components provide a method to separate “applications programming” from “web page design”. Thus the personnel involved in web page design do not need to understand Java programming language syntax to do their jobs.

13.6.3. Application Clients:

An application client runs on a client machine. The application client provides a better method for users to handle tasks, which require a richer user interface than can be provided by a markup language. The application client typically has a graphical user interface (GUI) created from the Swing or the Abstract Window Toolkit (AWT) API. However, a command-line interface is certainly possible.

Application clients directly access Enterprise beans that run in the “business tier”. However, if application requirements warrant it, an application client can open an HTTP connection to establish communication with a servlet running in the “web tier”. Application clients written in languages other than Java can interact with Java EE 5 servers. This provision enables the Java EE 5 platform to interoperate with legacy systems, clients, and non-Java languages.

13.7 SUMMARY

- EJB is defined as an architecture for the development and deployment of component-based, robust, highly scalable business applications.

- EJB can be used while developing the reusable business logic component in enterprise application
- The Java EE platform uses a distributed Multi-Tiered application model for Enterprise applications.
- A Java EE application can consist of three or four tiers.
- Java EE applications are made up of components. A Java EE component is a self-contained functional software unit.
- Java EE components are written in the Java programming language and are compiled in the same way as any program in the language.

13.8 UNIT END EXERCISE

1) What is EJB? State the benefits of EJB?
2) State the difference between JavaBeans and Enterprise JavaBeans?
3) Explain the JEE Architecture?
4) Describe the JEE Application Components?
5) State and explain Java EE clients?

13.9 FURTHER READING

- Eric Jendrock, Jennifer Ball, D Carson and others, The Java EE 5 Tutorial, Pearson Education, Third Edition, 2003
- Joe Wigglesworth and Paula McMillan, Java Programming: Advanced Topics, Thomson Course Technology (SPD), Third Edition, 2004
- The Java Tutorials of Sun Microsystems Inc

❖❖❖❖

14

TYPES OF EJB'S

Unit Structure:

14.0 Objectives
14.1 Types of Enterprise JavaBeans
14.2 Session Beans
14.3 Message-driven Bean
14.4 Deciding on Remote or Local Access
14.5 Method Parameters and Access
14.6 Summary
14.7 Unit end exercise
14.8 Further Reading

14.0 OBJECTIVES

The objective of this chapter is to learn and understand the types of EJB's. Here we will learn about the session beans, message beans and two types of access – Remote and Local.

14.1 TYPES OF ENTERPRISE JAVABEANS:

There are two kinds of Enterprise Beans:
• Session Beans
•Message-driven Beans

• Session Beans:

A Session Bean represents a transient conversation with a client. When the Client completes its execution, the Session Bean and it's data are gone.

• Message-driven Beans:

A Message-driven Bean combines features of a session bean and a message listener, allowing a business component to asynchronously receive messages. Commonly, these are known as Java Message Service (JMS) messages. In Java EE 5, the Entity Beans have been replaced by Java Persistence API entities. An Entity represents persistent data stored in one row of a database table. If the Client terminates, or if the Server shuts down, the Persistence Manager ensures that the entity data is saved.

14.2 SESSION BEANS:

A Session Bean represents a single Client inside the Application Server.

- Session Beans are reusable components that contain logic for business processes.
- For example: A Session Bean can perform price quoting, order entry, video compression, banking transactions, stock trades, database operations, complex calculations and more.
- To access an application that is deployed on the Server, the Client invokes the methods of the Session Bean.
- The Session Bean performs work for its Client, thus shielding the Client from complexity by executing business tasks inside the Server.
- As it's name suggests, a Session Bean is similar to an interactive session. However, a Session Bean is not shared.
- A Session Bean can have only one client, in the same manner as an interactive session can have only one user.
- Like an interactive session, a Session bean is not persistent. When the client terminates, it's Session Bean terminates and is no longer associated with the client.

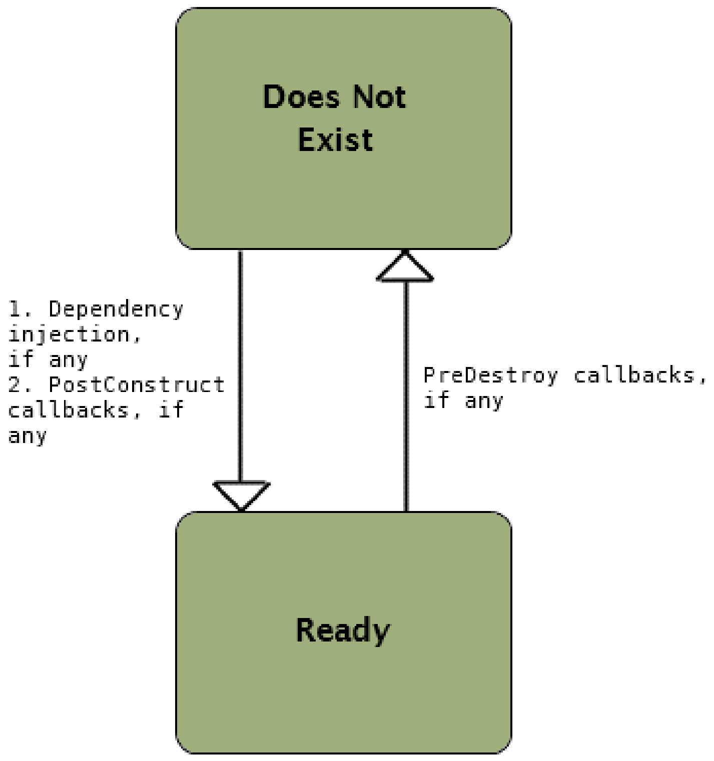
Session Beans Types:

Based on the "span of conversation" between the Client and the Bean, there are two types of Session Beans:

1. Stateless Session Bean
2. Stateful Session Bean

All Enterprise Beans hold conversations at some level. A "conversation" is an interaction between a Bean and a Client. It comprises of a number of "method calls" between the Client and the Bean.

Stateless Session Bean - Life Cycle:


Does Not Exist
1. Dependency injection, if any
2. PostConstruct callbacks, if any
PreDestroy callbacks, if any
Ready

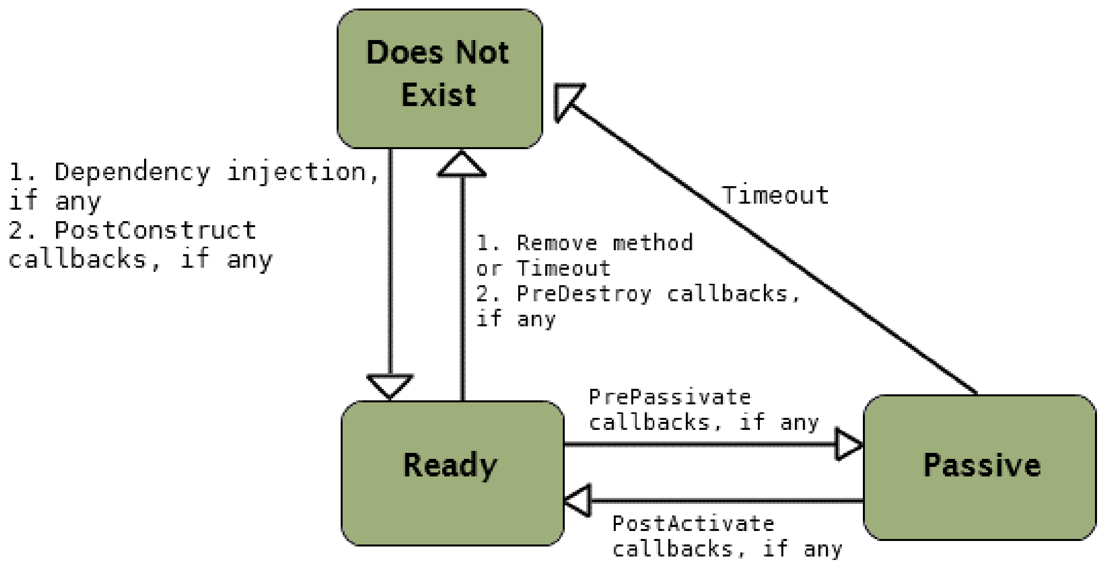
1. Does Not Exist to Ready: The client initiates the life cycle by obtaining a reference to a Stateless Session Bean. The container performs any dependency injection, and then invokes the method annotated @PostConstruct, if any. The client can now invoke the business methods of the bean.

2. Ready to Does Not Exist: At the end of the session bean life cycle, the EJB container calls the method annotated @PreDestroy, if any. The Bean instance is then ready for garbage collection.

Callback Methods:

- @PostConstruct: The container invokes this method on newly constructed Bean instances after all dependency injections are completed, and before the first business method is invoked on the Enterprise Bean.
- @PreDestroy: These methods are invoked after any method annotated @Remove is completed, and before the container removes the Enterprise Bean instance.

Stateful Session Bean - Life Cycle:


Does Not Exist
1. Dependency injection, if any
2. PostConstruct callbacks, if any
1. Remove method or Timeout
2. PreDestroy callbacks, if any
Timeout
PrePassivate callbacks, if any
Ready
Passive
PostActivate callbacks, if any

1. Does Not Exist to Ready: The Client initiates the life cycle by obtaining a reference to a Stateful Session Bean. Dependency injection is performed by container, and then invokes the method annotated @PostConstruct, if any. The client can now invoke the business methods of the bean.
2. Ready to Passive: In the Ready state, the EJB container may decide to passivate the Bean by moving it from the memory to the secondary storage.
3. Passive to Ready: If there is any @PrePassivate annotated method, container invokes it immediately before passivating the Bean. If a Client invokes a business method on the Bean while it is in the Passive state, the EJB container activates the Bean. The container then calls the method annotated with @PostActivate and moves it to the Ready state.
4. Ready to Does Not Exist: At the end, the client calls a method annotated with @Remove, and the EJB container calls the method annotated with @PreDestroy. The Bean's instance is then ready for garbage collection. Only the method annotated with @Remove can be controlled with your code.

Callback methods:
Given below are the annotations with the help of which you can declare Bean Class methods as Life Cycle Callback methods:
• javax.annotation.PostConstruct
• javax.annotation.PreDestroy
• javax.ejb.PostActivate
• javax.ejb.PrePassivate

1. @PostConstruct: The container calls these methods on newly constructed Bean instances after all dependency injections are completed and before the first business method is invoked on the Enterprise Bean.
2. @PreDestroy: These methods are called after any method annotated with @Remove has completed its execution, and before the container removes the Enterprise Bean instance.
3. @PostActivate: The container calls these methods after it moves the Bean from secondary storage to the memory, i.e. Active state.
4. @PrePassivate: The container calls these methods before it passivates the Enterprise Bean. This means before the container shifts the bean from memory to secondary storage.

Stateless Session Beans	**Stateful Session Beans**
1. They do not possess Internal state.	1. They possess Internal state.
2. They cannot be passivated.	2. They can undergo Passivation and Activation.
3. They can serve for multiple client.	3. They are specific to a single client.
4. They create Network Traffic.	4. Stateful Beans hurt scalability.

When to use Session Beans?

1. Generally Session Beans are used in the following circumstances:

- When there is only one client accessing the bean instance at a given time.
- When the bean is not persistent, that is when the bean is going to exist no longer.
- The bean is implementing the web services.

2. Stateful Session Beans are useful in the following circumstances:

- What information the bean wants to hold about the client across method invocation.
- When the bean works as the mediator between the client and the other component of the application.
- When the bean has to manage the work flow of several other enterprise beans.

3. Stateless Session Beans are appropriate in the circumstances illustrated below:

- If the bean does not contain the data for a specific client.
- If there is only one method invocation among all the clients to perform the generic task.

14.3 MESSAGE-DRIVEN BEAN:

• A Message-driven Bean is an Enterprise Bean that allows Java EE applications to asynchronously process messages. It normally acts as a “JMS Message Listener”, which is similar to an “event listener” except that it receives “JMS messages” instead of “events”.

• The messages can be sent by any Java EE component (an Application Client, another Enterprise Bean, or a web component), by a JMS application, or by a system that does not use Java EE technology.

• Message-driven Beans can process JMS messages or other kinds of messages.

• A Message-driven Bean resembles a Stateless Session Bean:

- A Message-driven Bean instances do not retain data or conversational state for a specific Client.
- All instances of a Message-driven Bean are equivalent.
- A single Message-driven Bean can process messages from multiple clients

Following are the characteristics of a Message-driven Bean (MDB):

- MDBs execute upon receipt of a single Client message.
- MDBs are asynchronously invoked.
- MDBs are relatively short-lived.
- MDBs do not represent the directly shared data in the database. However, they can access and update this data.
- MDBs can be transaction-aware.
- MDBs are stateless.

When to use Message Driven Bean:

- Session Beans allow sending JMS messages. However, they allow synchronous receiving of JMS messages, and not asynchronous.
- To avoid tying up server resources, do not use blocking synchronous receives in a server-side component. In general, do not send or receive JMS messages in a “synchronous” manner.
- To “asynchronously” receive messages, use a Message-driven Bean.

JMS Concept:

• What is Message?
Message is a unit of information or data which can be sent from one processing computer/application to other/same computer/applications.

• What is Messaging?
Messaging is a method of communication between software components or applications.

• How Messaging works?
A messaging system is a peer-to-peer facility. A messaging client can send messages to, and receive messages from, any other client. Each client connects to a messaging agent that provides facilities for creating, sending, receiving, and reading messages.

• What is JMS?
The Java Message service is a client-side API for accessing messaging systems.

JMS Messaging models:

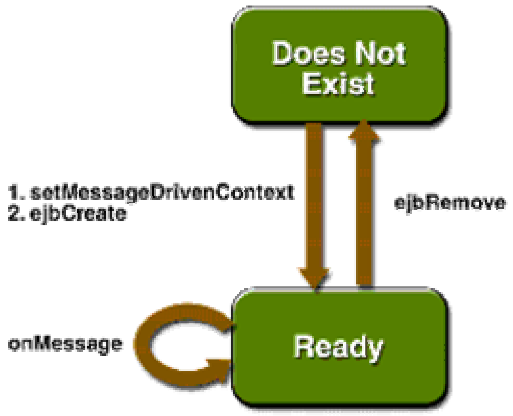
JMS communicates in synchronous or in asynchronous mode by using “point-to-point” and the “publish-subscribe” models respectively. Point-to-Point and Publish/Subscribe are the two most commonly used models. These two models conclude the following concepts:

- Producer: The client, responsible for sending the message to the destination is known as the “producer”.
- Consumer: The client, responsible for receiving the message is known as the “consumer”.
- Destination: Destination is the object used by the client to specify the target that uses it to send the message to or to receive the message from.

Working of Message-driven bean:

- In Message-driven beans (MDB), the client components don't locate Message-driven beans, and directly invoke methods. Instead, the JMS clients send messages to message queues managed by the JMS server (for example: an email inbox can be a message queue) for which the javax.jms.MessageListener interface is implemented.
- The message queue is monitored by a special kind of EJB(s) – Message-driven Beans (MDBs) – that processes the incoming messages and perform the services requested by the message.
- The MDBs are the end-point for JMS service request messages. You assign a Message-driven Bean's destination during deployment by using Application Server resources.

Life cycle of Message-driven Bean:



- The EJB container usually creates a pool of Message-driven Bean instances. For each instance, the EJB container performs these tasks:
    - o If the Message-driven Bean uses dependency injection, the Container injects these references before instantiating the instance.
    - o The Container calls the method annotated @PostConstruct, if any.
    - o Like a Stateless Session Bean, a Message-driven Bean is never passivated. It has only two states:
        - ▪ Not Exist
        - ▪ Ready to receive messages

  - o At the end of the life cycle, the Container calls the method annotated @PreDestroy, if any. The Bean instance is then ready for garbage collection.

- To create a new instance of a Message-driven Bean, the Container does the following: instantiates the Bean, performs any required resource injection and calls the @PostConstruct callback method, if it exists
- To remove an instance of a Message-driven Bean, the Container calls the @PreDestroy callback method.
- On message arrival, the Container calls the “onMessage method” of the Message-driven Bean to process the message.
- The onMessage method normally casts the message to one of the five JMS Message Types, and handles it in accordance with the business logic of the Application.
- The onMessage method can call helper methods, or it can invoke a Session Bean to process the information in the message, or to store it in a database.
- A message can be delivered to a Message-driven Bean within a transaction context, such that all operations within the “onMessage method” are part of a single transaction. If message processing is rolled back, the message will be redelivered.

14.4 DECIDING ON REMOTE OR LOCAL ACCESS

When you design a Java EE application, one of the first decisions you make is the type of client access allowed by the enterprise beans: remote, local, or web service. Whether to allow local or remote access depends on the following factors.

- **Tight or loose coupling of related beans**: Tightly coupled beans depend on one another. For example, if a session bean that processes sales orders calls a session bean that emails a confirmation message to the customer, these beans are tightly coupled. Tightly coupled beans are good candidates for local access. Because they fit together as a logical unit, they typically call each other often and would benefit from the increased performance that is possible with local access.
- **Type of client**: If an enterprise bean is accessed by application clients, it should allow remote access. In a production environment, these clients almost always run on machines other

than those on which the GlassFish Server is running. If an enterprise bean’s clients are web components or other enterprise beans, the type of access depends on how you want to distribute your components.

- **Component distribution**: Java EE applications are scalable because their server-side components can be distributed across multiple machines. In a distributed application, for example, the server that the web components run on may not be the one on which the enterprise beans they access are deployed. In this distributed scenario, the enterprise beans should allow remote access.
- **Performance**: Owing to such factors as network latency, remote calls may be slower than local calls. On the other hand, if you distribute components among different servers, you may improve the application’s overall performance. Both of these statements are generalizations; performance can vary in different operational environments.Nevertheless, you should keep in mind how your application design might affect performance.

Although it is uncommon, it is possible for an enterprise bean to allow both remote and local access. If this is the case, either the business interface of the bean must be explicitly designated as a business interface by being decorated with the @Remote or @Local annotations, or the bean class must explicitly designate the business interfaces by using the @Remote and @Local annotations. The same business interface cannot be both a local and a remote business interface.

14.5 METHOD PARAMETERS AND ACCESS

The type of access affects the parameters of the bean methods that are called by clients. The following sections apply not only to method parameters but also to method return values.

Isolation

The parameters of remote calls are more isolated than those of local calls. With remote calls, the client and the bean operate on different copies of a parameter object. If the client changes the value of the object, the value of the copy in the bean does not change. This layer of isolation can help protect the bean if the client accidentally modifies the data.

In a local call, both the client and the bean can modify the same parameter object. In general, you should not rely on this side effect of local calls. Perhaps someday you will want to distribute your components, replacing the local calls with remote ones.

As with remote clients, web service clients operate on different copies of parameters than does the bean that implements the web service.

Granularity of Accessed Data

Because remote calls are likely to be slower than local calls, the parameters in remote methods should be relatively coarse-grained. A coarse-grained object contains more data than a fine-grained one, so fewer access calls are required. For the same reason, the parameters of the methods called by web service clients should also be coarse-grained.

Example: Develop "Converter" Stateless Session Bean. Write Enterprise application for converting Japanese yen currency to Eurodollars currency. converter consists of an enterprise bean, which performs the calculations. Use following formula: 1 Euro = 115.3100 Yens. Develop a web client to test the converter.

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-
8">
<title>JSP Page</title>
</head>
<body>
<form method="get"
action="http://localhost:8080/StateLessEJB/ConverterServlet">
<b><h2>Converter Bean</h2></b>
<table border=2>
<tr>
<td>Enter Amount</td>
<td><input type="Text" name=txtnum></td>
</tr>
<tr>
<td><input type=Submit name=cmdsubmit></td>
<td><input type=Reset name=cmdreset></td>
</tr>
</table>
</form>
</body>
</html>
```

ConverterBeanRemote.java

```
package server;

import java.math.BigDecimal;
import javax.ejb.Remote;

@Remote
public interface ConverterBeanRemote
{
   public BigDecimal dollarToYen(BigDecimal dollars);
   public BigDecimal yenToEuro(BigDecimal yen);
}
```

ConverterBean.java

```
package server;
import javax.ejb.Stateless;
import java.math.BigDecimal;
@Stateless
public class ConverterBean
{
   private BigDecimal euroRate = new BigDecimal("0.0070");
   private BigDecimal yenRate = new BigDecimal("112.58");
   public BigDecimal dollarToYen(BigDecimal dollars)
   {
      BigDecimal result = dollars.multiply(yenRate);

      return result.setScale(2, BigDecimal.ROUND_UP);
   }
   public BigDecimal yenToEuro(BigDecimal yen)
   {
      BigDecimal result = yen.multiply(euroRate);

      return result.setScale(2, BigDecimal.ROUND_UP);
   }
}
```

ConverterServlet.java

```
package server;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(name="ConverterServlet",
urlPatterns={"/ConverterServlet"})
public class ConverterServlet extends HttpServlet {

   @EJB ConverterBeanRemote conv;
   protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
   throws ServletException, IOException {
      response.setContentType("text/html;charset=UTF-8");
      PrintWriter out = response.getWriter();
      try {
         out.println("<html>");
         out.println("<head>");
         out.println("<title>Servlet ConverterServlet</title>");
         out.println("</head>");
         out.println("<body>");

         String str=request.getParameter("txtnum");
         int number=Integer.parseInt(str);
         BigDecimal num=new BigDecimal(number);

         out.println("<h2>Dollor to yen  "
                           + conv.dollarToYen(num) +"</h2>");
         out.println("<h2>Yen to euro  "
                           + conv.yenToEuro(num) +"</h2>");
                  out.println("</body>");
         out.println("</html>");
      } finally {
         out.close();
      }
   }
```

Example: Develop a Stateful session bean to add items to a cart. Develop a web client to test the converter.

CartBeanRemote.java

```
package server;
import java.util.Collection;
import javax.ejb.Remote;

@Remote
public interface CartBeanRemote{
   public void addItem(String item);
   public void removeItem(String item);
   public Collection getItems();
}
```

CartBean.java

```
package server;
import java.util.ArrayList;
import java.util.Collection;
import javax.annotation.PostConstruct;
import javax.ejb.Stateful;

@Stateful
public class CartBean implements CartBeanRemote
{
   private ArrayList items;
   @PostConstruct
   public void initialize() {
      items = new ArrayList();
   }
   @Override
   public void addItem(String item) {
      items.add(item);
   }
   @Override
   public void removeItem(String item) {
      items.remove(item);
   }
   @Override
   public Collection getItems() {
      return items;
```

```
    }
}
protected void processRequest(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException
    {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try
        {
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet CartServlet</title>");
        out.println("</head>");
        out.println("<body>");
        final Context context= new InitialContext();
        CartBeanRemote cart = (CartBeanRemote)context.lookup
                        ("java:global/CartStatefulEJB/CartBean");
        out.println("<br>Adding items to cart<br>");
        cart.addItem("Pizza");
        cart.addItem("Pasta");
        cart.addItem("Noodles");
        cart.addItem("Bread");
        cart.addItem("Butter");
        out.println("<br>Listing cart contents<br>");
        Collection items = cart.getItems();
        for (Iterator i = items.iterator(); i.hasNext();)
        {
        String item = (String) i.next();
        out.println("<br>" + item);
        }
        }catch (Exception ex){
            out.println("ERROR -->" + ex.getMessage());
        }
        out.println("</body>");
        out.println("</html>");
        out.close();
    }
```

14.6 SUMMARY

- A Session Bean represents a transient conversation with a client.
- A Message-driven Bean combines features of a session bean and a message listener, allowing a business component to asynchronously receive messages.
- Based on the “span of conversation” between the Client and the Bean, there are two types of Session Beans:
    - o Stateless Session Bean
    - o Stateful Session Bean
- In Message-driven beans (MDB) the JMS clients send messages to message queues managed by the JMS server for which the javax.jms.MessageListener interface is implemented.

14.7 UNIT END EXERCISE

1. Explain the Lifecycle of Stateless Session Bean.
2. List and explain Callback methods of Stateless Session Bean.
3. Explain the Lifecycle of Stateful Session Bean.
4. List and explain Callback methods of Stateful Session Bean.
5. What factors are considered for Remote or Local access?
6. Explain the Lifecycle of Message Driven Bean.
7. What is MessageListener? Also explain onMessage().
8. What are the various ways of passing parameters in EJB?

14.8 FURTHER READING

- Eric Jendrock, Jennifer Ball, D Carson and others, The Java EE 5 Tutorial, Pearson Education, Third Edition, 2003
- Joe Wigglesworth and Paula McMillan, Java Programming: Advanced Topics, Thomson Course Technology (SPD), Third Edition, 2004
- The Java Tutorials of Sun Microsystems Inc

❖❖❖❖

15

WEB SERVICES

Unit Structure:

15.0 Objectives
15.1 Introduction to Web Services
15.2 Building Web Services with JAX-WS
15.3 Creating a Simple Web Service and Client with JAX-WS.
15.4 Summary
15.5 Unit end exercise
15.6 Further Reading

15.0 OBJECTIVES

The objective of this chapter is to learn what Web Service is and how they are created. Here we will also understand the need why it is used and where it is used.

15.1 INTRODUCTION TO WEB SERVICES

Web services are the mechanism to develop a Service-Oriented-Architecture (SOA). SOA is an architectural approach for designing large scale distributed systems to integrate heterogeneous application on the service interfaces. Web services technologies support to the Service-Oriented-Architecture in various ways. Some of them are illustrated below:

- A service requestor uses the selection criteria to the query registry for finding the services description.
- A service requestor can bind and use the service if it finds a suitable descriptor.

Web services are used in various fields such converting a temperature value from Fahrenheit to Celsius. More realistic examples built using the web services are heterogeneous applications such as billing application and report generator, interconnected in-house architectures. A service interface is just like an object interface with a slight difference that the contract between the interface and the client is more flexible and the implementation of client and the service is not much tightly coupled as compared to EJB or other distributed platform. Looser coupling allows the client and service implementation to run on various platforms, independently such as Microsoft .NET is capable of using a Java EE application server to access a service running on it. From the client's point of view, web services's life cycle is more static as compared to average objects because web services stay around rather than pop-up and go away, even if the services are implemented using the object technology.

15.2 BUILDING WEB SERVICES WITH JAX-WS

JAX-WS stands for Java API for XML Web Services. JAX-WS is a technology for building web services and clients that communicate using XML. JAX-WS allows developers to write message-oriented as well as RPC-oriented web services.

In JAX-WS, a web service operation invocation is represented by an XML-based protocol such as SOAP. The SOAP specification defines the envelope structure, encoding rules, and conventions for representing web service invocations and responses. These calls and responses are transmitted as SOAP messages (XML files) over HTTP.

Although SOAP messages are complex, the JAX-WS API hides this complexity from the application developer. On the server side, the developer specifies the web service operations by defining methods in an interface written in the Java programming language. The developer also codes one or more classes that implement those methods. Client programs are also easy to code.

A client creates a proxy (a local object representing the service) and then simply invokes methods on the proxy. With JAX-WS, the developer does not generate or parse SOAP messages. It is the JAX-WS runtime system that converts the API calls and responses to and from SOAP messages.

With JAX-WS, clients and web services have a big advantage: the platform independence of the Java programming language. In addition, JAX-WS is not restrictive: a JAX-WS client can access a web service that is not running on the Java platform, and vice versa. This flexibility is possible because JAX-WS uses technologies defined by the World Wide Web Consortium (W3C): HTTP, SOAP, and the Web ServiceDescription Language (WSDL).WSDL specifies an XML format for describing a service as a set of endpoints operating on messages.

15.3 USING JAX-WS 2.0 TO CREATE A SIMPLE WEB SERVICE

JAX-WS 2.0 is extremely easy to use. Below you will see how to create a simple web service using JAX-WS 2.0 with Java SE 6 technology. The first thing you need is a class with one or more methods that you wish to export as a web service:

```
package hello;
public class CircleFunctions {
   public double getArea(double radius) {
       return java.lang.Math.PI * (r * r);
    }
   public double getCircumference(double radius) {
       return 2 * java.lang.Math.PI * r;
   }
}
```

To export these methods, you must add two things: an import statement for the javax.jws.WebService package and a @WebService annotation at the beginning that tells the Java interpreter that you intend to publish the methods of this class as a web service. The following code example shows the additions in bold.

```
package hello;

import javax.jws.WebService;
@WebService
public class CircleFunctions {
   public double getArea(double r) {
       return java.lang.Math.PI * (r * r);
    }
   public double getCircumference(double r) {
        return 2 * java.lang.Math.PI * r;
    }
}
```

You can use the static publish() method of the javax.xml.ws.Endpoint class to publish the class as a web service in the specified context root:

```
import javax.xml.ws.Endpoint;
public static void main(String[] args) {
      Endpoint.publish(
         "http://localhost:8080/WebServiceExample/circlefunctions",
        new CircleFunctions());
}
```

Now, compile the source code normally using javac. However, you must perform one more step: Call the Wsgen tool, as follows.

```
> wsgen –cp . hello.CircleFunctions
```

The Wsgen tool will generate a number of source files in a subdirectory called wsgen, which it then compiles. Although you should never have to edit these files, you can browse these source code files to get an idea of how JAX-WS 2.0 creates the appropriate stub files for use while publishing the web service. Note that the original source files must be located in a package when you call the Wsgen tool. Otherwise, you may get an error that dictates that classes annotated with @WebService, such as Circle Functions, must declare a separate javax. jws. Web service.target Namespace element because the source files are not part of a package.

That's it. When you run the application, the Java SE 6 platform has a small web application server that will publish the web service at the address http://localhost:8080/WebService Example/circle functions while the JVM is running.* You can verify that the web service is running by displaying the Web Services Definition Language (WSDL) file of the circlefunctions web service.While the JVM is still running, open a browser and go to the following location:

http://localhost:8080/WebServiceExample/circlefunctions?WSDL

If you see a large amount of XML that describes the functionality behind the web service, then the deployment has been successful.

15.4 SUMMARY

- Web services are the mechanism to develop a Service-Oriented-Architecture (SOA). SOA is an architectural approach for designing large scale distributed systems to integrate heterogeneous application on the service interfaces.
- JAX-WS stands for Java API for XML Web Services. JAX-WS is a technology for building web services and clients that communicate using XML.
- JAX-WS is not restrictive: a JAX-WS client can access a web service that is not running on the Java platform, and vice versa.

- JAX-WS uses technologies defined by the World Wide Web Consortium (W3C): HTTP, SOAP, and the Web Service Description Language (WSDL).

15.5 UNIT END EXERCISE

1. Write a short note on JAX-WS.
2. Write a Web Service which will return factorial of a number passed to it.

15.6 FURTHER READING

- Eric Jendrock, Jennifer Ball, D Carson and others, The Java EE 5 Tutorial, Pearson Education, Third Edition, 2003
- Joe Wigglesworth and Paula McMillan, Java Programming: Advanced Topics, Thomson Course Technology (SPD), Third Edition, 2004
- The Java Tutorials of Sun Microsystems Inc

❖❖❖❖

BIBLIOGRAPY

- The Java Tutorials of Sun Microsystems Inc.
- The Java EE 5 Tutorial, Pearson Education, Third Edition, 2003, Eric Jendrock, Jennifer Ball, D Carson
- Java2: The Complete Reference, Herbert Schildt
- http://java.sun.com
- http://www.roseindia.net
- http://docs.oracle.com
- www.tutorialspoint.com
- www.easywayserver.com
- www.download.oracle.com
- www.java2s.com
- www.coreservlets.com

❖❖❖❖