

MAR GREGORIOS COLLEGE OF ARTS & SCIENCE

Block No.8, College Road, Mogappair West, Chennai – 37

Affiliated to the University of Madras
Approved by the Government of Tamil Nadu
An ISO 9001:2015 Certified Institution



PG DEPARTMENT OF COMPUTER SCIENCE

SUBJECT NAME: ADVANCED JAVA PROGRAMMING

SUBJECT CODE: PED1B

SEMESTER: I

PREPARED BY: PROF.S.JAMES BENEDICT FELIX

ADVANCED JAVA PROGRAMMING

UNIT – I

SERVLET OVERVIEW

1. Servlet Overview

Java Servlets are programs that run on a Web or Application server and act as a middle layer between a requests coming from a Web browser or other HTTP client and databases or applications on the HTTP server.

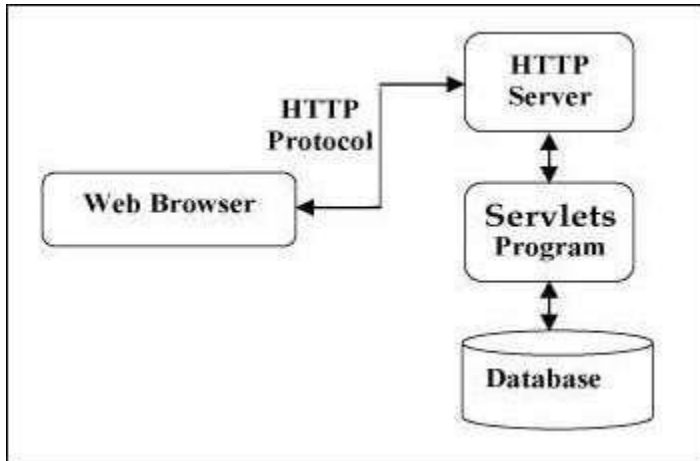
Using Servlets, you can collect input from users through web page forms, present records from a database or another source, and create web pages dynamically.

Java Servlets often serve the same purpose as programs implemented using the Common Gateway Interface (CGI). But Servlets offer several advantages in comparison with the CGI.

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

Servlets Architecture

The following diagram shows the position of Servlets in a Web Application.



Servlets Tasks

Servlets perform the following major tasks –

- Read the explicit data sent by the clients (browsers). This includes an HTML form on a Web page or it could also come from an applet or a custom HTTP client program.
- Read the implicit HTTP request data sent by the clients (browsers). This includes cookies, media types and compression schemes the browser understands, and so forth.
- Process the data and generate the results. This process may require talking to a database, executing an RMI or CORBA call, invoking a Web service, or computing the response directly.
- Send the explicit data (i.e., the document) to the clients (browsers). This document can be sent in a variety of formats, including text (HTML or XML), binary (GIF images), Excel, etc.
- Send the implicit HTTP response to the clients (browsers). This includes telling the browsers or other clients what type of document is being returned (e.g., HTML), setting cookies and caching parameters, and other such tasks.

Servlets Packages

Java Servlets are Java classes run by a web server that has an interpreter that supports the Java Servlet specification.

Servlets can be created using the **javax.servlet** and **javax.servlet.http** packages, which are a standard part of the Java's enterprise edition, an expanded version of the Java class library that supports large-scale development projects.

These classes implement the Java Servlet and JSP specifications. At the time of writing this tutorial, the versions are Java Servlet 2.5 and JSP 2.1.

Java servlets have been created and compiled just like any other Java class. After you install the servlet packages and add them to your computer's Classpath, you can compile servlets with the JDK's Java compiler or any other current compiler.

2. Servlet Life Cycle

A servlet life cycle can be defined as the entire process from its creation till the destruction. The following are the paths followed by a servlet.

- The servlet is initialized by calling the **init()** method.
- The servlet calls **service()** method to process a client's request.
- The servlet is terminated by calling the **destroy()** method.
- Finally, servlet is garbage collected by the garbage collector of the JVM.

Now let us discuss the life cycle methods in detail.

a) The `init()` Method

The `init` method is called only once. It is called only when the servlet is created, and not called for any user requests afterwards. 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.

When a user invokes a servlet, a single instance of each servlet gets created, with each user request resulting in a new thread that is handed off to `doGet` or `doPost` as appropriate. The `init()` method simply creates or loads some data that will be used throughout the life of the servlet.

The init method definition looks like this –

```
public void init() throws ServletException {
    // Initialization code...
}
```

b) The service() Method

The service() method is the main method to perform the actual task. The servlet container (i.e. web server) calls the service() method to handle requests coming from the client (browsers) and to write the formatted response back to the client.

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. methods as appropriate.

Here is the signature of this method –

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

The service () method is called by the container and service method invokes doGet, doPost, doPut, doDelete, etc. methods as appropriate. So you have nothing to do with service() method but you override either doGet() or doPost() depending on what type of request you receive from the client.

The doGet() and doPost() are most frequently used methods with in each service request. Here is the signature of these two methods.

c) The doGet() Method

A GET request results from a normal request for a URL or from an HTML form that has no METHOD specified and it should be handled by doGet() method.

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
```

```
throws ServletException, IOException {
// Servlet code
}
```

d) The doPost() Method

A POST request results from an HTML form that specifically lists POST as the METHOD and it should be handled by doPost() method.

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
// Servlet code
}
```

e) The destroy() Method

The destroy() method is called only once at the end of the life cycle of a servlet. 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.

After the destroy() method is called, the servlet object is marked for garbage collection. The destroy method definition looks like this –

```
public void destroy() {
// Finalization code...
}
```

3. Web Server

Server is a device or a computer program that accepts and responds to the request made by other program, known as client. It is used to manage the network resources and for running the program or software that provides services.

There are two types of servers:

- a) Web Server
- b) Application Server

Web Server

Web server contains only web or servlet container. It can be used for servlet, jsp, struts, jsf etc. It can't be used for EJB.

It is a computer where the web content can be stored. In general web server can be used to host the web sites but there also used some other web servers also such as FTP, email, storage, gaming etc.

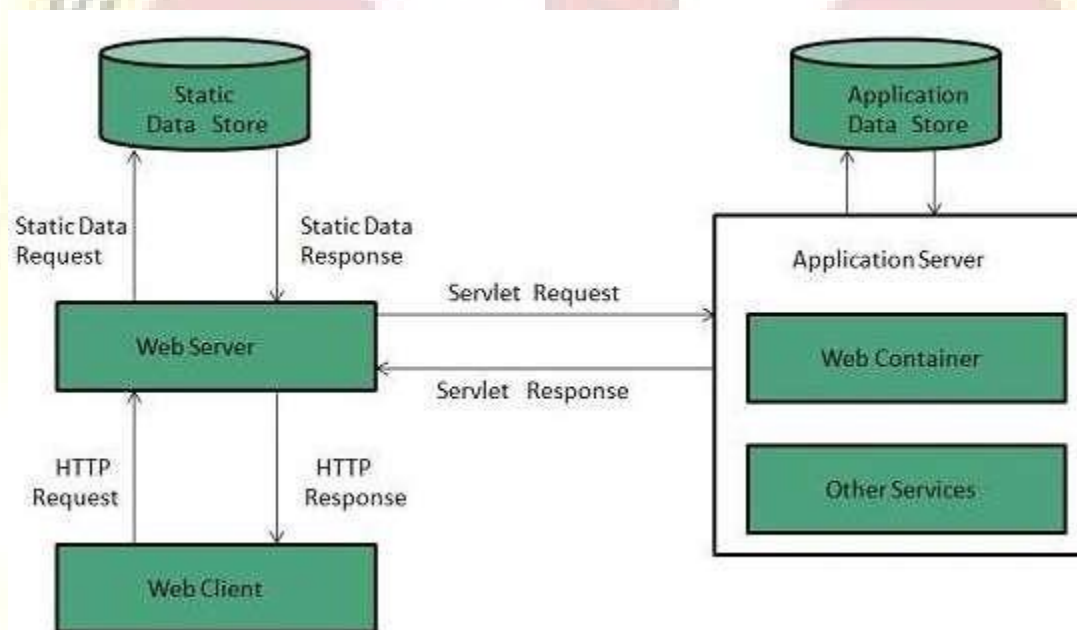
Examples of Web Servers are: **Apache Tomcat** and **Resin**.

Web Server Working

It can respond to the client request in either of the following two possible ways:

- Generating response by using the script and communicating with database.
- Sending file to the client associated with the requested URL.

The block diagram representation of Web Server is shown below:



Important points

- If the requested web page at the client side is not found, then web server will send the HTTP response: Error 404 Not found.
- When the web server searches for the requested page, if the requested page is found, then it will send to the client with an HTTP response.
- If the client requests some other resources, then the web server will contact the application server and data is stored for constructing the HTTP response.

Application Server

Application server contains Web and EJB containers. It can be used for servlet, jsp, struts, jsf, ejb etc. It is a component based product that lies in the middle-tier of a server centric architecture.

It provides the middleware services for state maintenance and security, along with persistence and data access. It is a type of server designed to install, operate and host associated services and applications for the IT services, end users and organizations.

The block diagram representation of Application Server is shown below:

4. Simple Servlet

Servlets are Java classes which service HTTP requests and implement the `javax.servlet.Servlet` interface. Web application developers typically write servlets that extend `javax.servlet.http.HttpServlet`, an abstract class that implements the Servlet interface and is specially designed to handle HTTP requests.

Sample Code

Following is the sample source code structure of a servlet example to show Hello World

–


```
// Import required java libraries
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Extend HttpServlet class
public class HelloWorld extends HttpServlet {

    private String message;

    public void init() throws ServletException {
        // Do required initialization
        message = "Hello World";
    }

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        // Set response content type
        response.setContentType("text/html");
```

```
// Actual logic goes here.  
PrintWriter out = response.getWriter();  
out.println("<h1>" + message + "</h1>");  
}  
  
public void destroy() {  
    // do nothing.  
}  
}
```

Compiling a Servlet

Let us create a file with name `HelloWorld.java` with the code shown above. Place this file at `C:\ServletDevel` (in Windows) or at `/usr/ServletDevel` (in Unix). This path location must be added to `CLASSPATH` before proceeding further.

Assuming your environment is setup properly, go in **ServletDevel** directory and compile `HelloWorld.java` as follows –

```
$ javac HelloWorld.java
```

If the servlet depends on any other libraries, you have to include those JAR files on your `CLASSPATH` as well. I have included only `javax.servlet-api.jar` JAR file because I'm not using any other library in Hello World program.

This command line uses the built-in `javac` compiler that comes with the Sun Microsystems Java Software Development Kit (JDK). For this command to work properly, you have to include the location of the Java SDK that you are using in the `PATH` environment variable.

If everything goes fine, above compilation would produce **HelloWorld.class** file in the same directory. Next section would explain how a compiled servlet would be deployed in production.

Servlet Deployment

By default, a servlet application is located at the path `<Tomcat-installationdirectory>/webapps/ROOT` and the class file would reside in `<Tomcat-installationdirectory>/webapps/ROOT/WEB-INF/classes`.

If you have a fully qualified class name of **com.myorg.MyServlet**, then this servlet class must be located in `WEB-INF/classes/com/myorg/MyServlet.class`.

For now, let us copy `HelloWorld.class` into `<Tomcat-installationdirectory>/webapps/ROOT/WEB-INF/classes` and create following entries in **web.xml** file located in `<Tomcat-installation-directory>/webapps/ROOT/WEB-INF/`

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

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

Above entries to be created inside `<web-app>...</web-app>` tags available in `web.xml` file. There could be various entries in this table already available, but never mind.

You are almost done, now let us start tomcat server using `<Tomcat-installationdirectory>\bin\startup.bat` (on Windows) or `<Tomcat-installationdirectory>/bin/startup.sh` (on Linux/Solaris etc.) and finally

type **http://localhost:8080/HelloWorld** in the browser's address box. If everything goes fine, you would get the following result



5. Servlet Packages

There are two packages in Java Servlet that provide various features to servlet. These two packages are `javax.servlet` and `javax.servlet.http`.

javax.servlet package: This package contains various servlet interfaces and classes which are capable of handling any type of protocol.

javax.servlet.http package: This package contains various interfaces and classes which are capable of handling a specific http type of protocol.

Overview of some important interfaces and classes

javax.servlet package interface

Some of the important interfaces are listed below.

Interface	Overview
Servlet	This interface is used to create a servlet class. Each servlet class must require to implement this interface either directly or indirectly.
ServletRequest	The object of this interface is used to retrieve the information from the user.
ServletResponse	The object of this interface is used to provide response to the user.
ServletConfig	ServletConfig object is used to provide the information to the servlet class explicitly.
ServletContext	The object of ServletContext is used to provide the information to the web application explicitly.

javax.servlet package classes

Some of the important classes are listed below.

Classes	Overview
GenericServlet	This is used to create servlet class. Internally, it implements the Servlet interface.
ServletInputStream	This class is used to read the binary data from user requests.
ServletOutputStream	This class is used to send binary data to the user side.
ServletException	This class is used to handle the exceptions occur in servlets.
ServletContextEvent	If any changes are made in servlet context of web application, this class notifies.

javax.servlet.http package interface

Some of the important interface of this package are listed below:

Interface	Overview
HttpServletRequest	The object of this interface is used to get the information from the user under http protocol.
HttpServletResponse	The object of this interface is used to provide the response of the request under http protocol.
HttpSession	This interface is used to track the information of users.
HttpSessionAttributeListener	This interface notifies if any change occurs in HttpSession attribute.
HttpSessionListener	This interface notifies if any changes occur in HttpSession lifecycle.

javax.servlet.http package classes

Some of the important interface of this package are listed below.

Class	Overview
HttpServlet	This class is used to create servlet class.
Cookie	This class is used to maintain the session of the state.
HttpSessionEvent	This class notifies if any changes occur in the session of web application.
HttpSessionBindingEvent	This class notifies when any attribute is bound, unbound or replaced in a session.

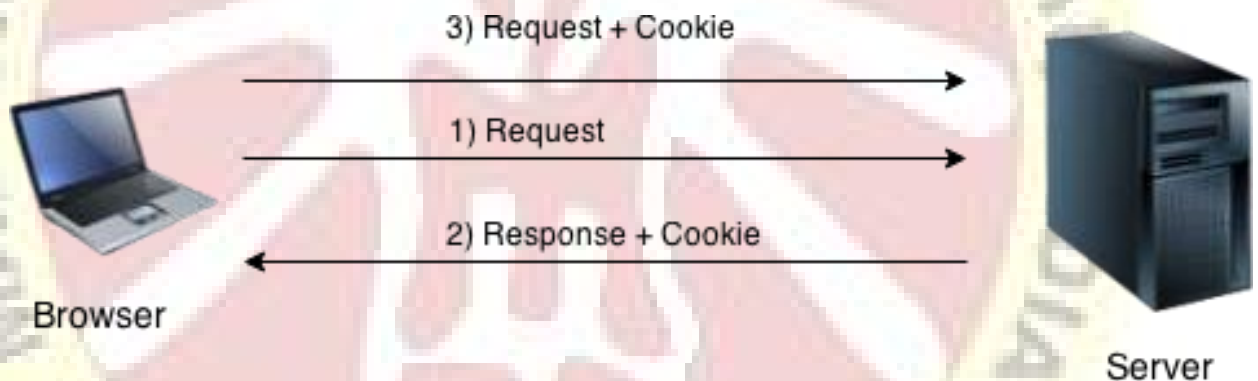
6. Using Cookies

A **cookie** is a small piece of information that is persisted between the multiple client requests.

A cookie has a name, a single value, and optional attributes such as a comment, path and domain qualifiers, a maximum age, and a version number.

How Cookie works

By default, each request is considered as a new request. In cookies technique, we add cookie with response from the servlet. So cookie is stored in the cache of the browser. After that if request is sent by the user, cookie is added with request by default. Thus, we recognize the user as the old user.



Types of Cookie

There are 2 types of cookies in servlets.

1. Non-persistent cookie
2. Persistent cookie

Non-persistent cookie

It is **valid for single session** only. It is removed each time when user closes the browser.

Persistent cookie

It is **valid for multiple session** . It is not removed each time when user closes the browser. It is removed only if user logout or signout.

Advantage of Cookies

1. Simplest technique of maintaining the state.
2. Cookies are maintained at client side.

Disadvantage of Cookies

1. It will not work if cookie is disabled from the browser.
2. Only textual information can be set in Cookie object.

Cookie class

javax.servlet.http.Cookie class provides the functionality of using cookies. It provides a lot of useful methods for cookies.

Constructor of Cookie class

Constructor	Description
Cookie()	constructs a cookie.
Cookie(String name, String value)	constructs a cookie with a specified name and value.

Useful Methods of Cookie class

There are given some commonly used methods of the Cookie class.

Method	Description
<code>public void setMaxAge(int expiry)</code>	Sets the maximum age of the cookie in seconds.
<code>public String getName()</code>	Returns the name of the cookie. The name cannot be changed after creation.
<code>public String getValue()</code>	Returns the value of the cookie.
<code>public void setName(String name)</code>	changes the name of the cookie.
<code>public void setValue(String value)</code>	changes the value of the cookie.

Other methods required for using Cookies

For adding cookie or getting the value from the cookie, we need some methods provided by other interfaces.

They are:

1. **public void addCookie(Cookie ck):**method of HttpServletResponse interface is used to add cookie in response object.
2. **public Cookie[] getCookies():**method of HttpServletRequest interface is used to return all the cookies from the browser.

How to create Cookie?

Let's see the simple code to create cookie.

1. `Cookie ck=new Cookie("user","sonoo jaiswal");//creating cookie object`
2. `response.addCookie(ck);//adding cookie in the response`

How to delete Cookie?

Let's see the simple code to delete cookie. It is mainly used to logout or signout the user.

1. `Cookie ck=new Cookie("user","");//deleting value of cookie`
2. `ck.setMaxAge(0);//changing the maximum age to 0 seconds`
3. `response.addCookie(ck);//adding cookie in the response`

How to get Cookies?

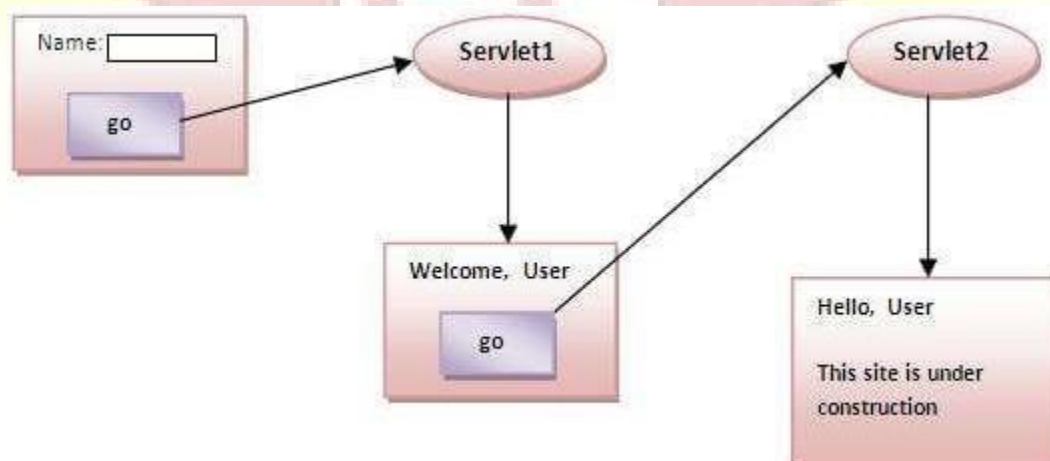
Let's see the simple code to get all the cookies.

```
Cookie ck[]=request.getCookies();
for(int i=0;i<ck.length;i++){
    out.print("<br>" +ck[i].getName()+" "+ck[i].getValue());//printing name and value of cookie }

```

Simple example of Servlet Cookies

In this example, we are storing the name of the user in the cookie object and accessing it in another servlet. As we know well that session corresponds to the particular user. So if you access it from too many browsers with different values, you will get the different value.



index.html

```
<form action="servlet1" method="post">
Name:<input type="text" name="userName"/><br/>
<input type="submit" value="go"/>
</form>
```

FirstServlet.java

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

public class FirstServlet extends HttpServlet {

public void doPost(HttpServletRequest request, HttpServletResponse response){
try{
response.setContentType("text/html");
PrintWriter out = response.getWriter();

String n=request.getParameter("userName");

out.print("Welcome "+n);

Cookie ck=new Cookie("uname",n);//creating cookie object
response.addCookie(ck);//adding cookie in the response

//creating submit button
out.print("<form action='servlet2'>");
out.print("<input type='submit' value='go'>");
out.print("</form>");

out.close();

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

SecondServlet.java

```

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

public class SecondServlet extends HttpServlet {

public void doPost(HttpServletRequest request, HttpServletResponse response){
    try{

response.setContentType("text/html");
PrintWriter out = response.getWriter();

Cookie ck[]=request.getCookies();
out.print("Hello "+ck[0].getValue());
out.close();

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

```

web.xml

```

<web-app>
<servlet>
<servlet-name>s1</servlet-name>
<servlet-class>FirstServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>s1</servlet-name>
<url-pattern>/servlet1</url-pattern>
</servlet-mapping>

```



```
<servlet>
<servlet-name>s2</servlet-name>
<servlet-class>SecondServlet</servlet-class>
</servlet>
```

```
<servlet-mapping>
<servlet-name>s2</servlet-name>
<url-pattern>/servlet2</url-pattern>
</servlet-mapping>
</web-app>
```

OUTPUT



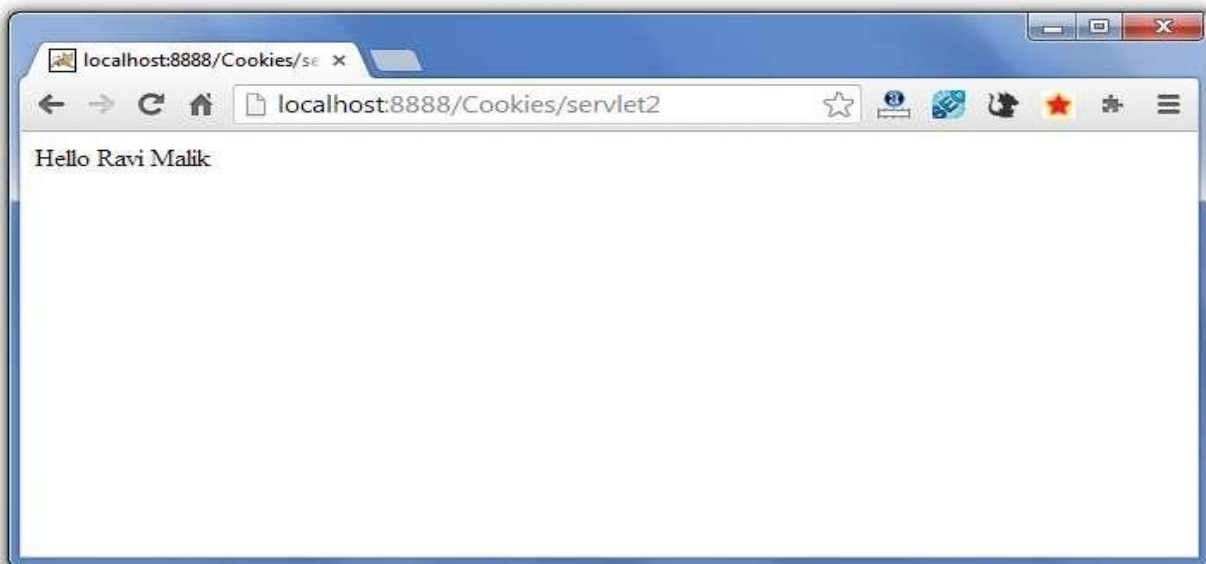
localhost:8888/Cookies/ fi

localhost.atas/cookies/

Ravi Malik

localhost:8888/Cookies/servlet

Welcome Ravi Malik



7. Session Tracking in Servlets

Session simply means a particular interval of time.

Session Tracking is a way to maintain state (data) of an user. It is also known as **session management** in servlet.

Http protocol is a stateless so we need to maintain state using session tracking techniques. Each time user requests to the server, server treats the request as the new request. So we need to maintain the state of an user to recognize to particular user.

HTTP is stateless that means each request is considered as the new request. It is shown in the figure given below:

Why use Session Tracking?

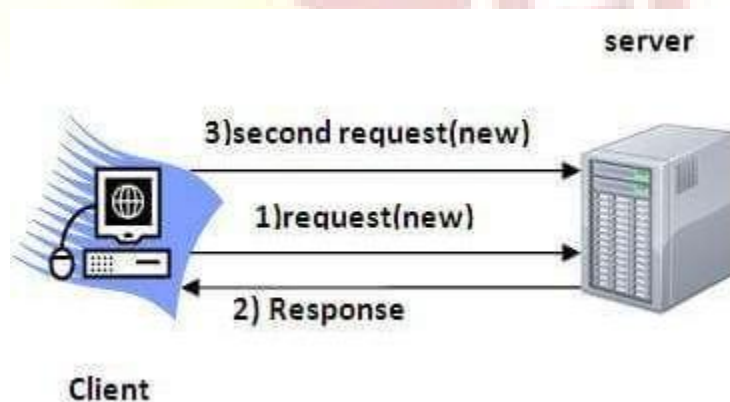
To recognize the user It is used to recognize the particular user.

Why use Session Tracking?

To recognize the user It is used to recognize the particular user.

Why use Session Tracking?

To recognize the user It is used to recognize the particular user.



Session Tracking Techniques

There are four techniques used in Session tracking:

1. Cookies
2. Hidden Form Field
3. URL Rewriting
4. HttpSession

Session Tracking Example

This example describes how to use the HttpSession object to find out the creation time and the last-accessed time for a session. We would associate a new session with the request if one does not already exist.

```
// Import required java libraries
```

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

// Extend HttpServlet class
public class SessionTrack extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        // Create a session object if it is already not created.
        HttpSession session = request.getSession(true);

        // Get session creation time.
        Date createTime = new Date(session.getCreationTime());

        // Get last access time of this web page.
        Date lastAccessTime = new Date(session.getLastAccessedTime());

        String title = "Welcome Back to my website";
        Integer visitCount = new Integer(0);
        String visitCountKey = new String("visitCount");
        String userIDKey = new String("userID");
        String userID = new String("ABCD");

        // Check if this is new comer on your web page.
        if (session.isNew()) {
            title = "Welcome to my website";
            session.setAttribute(userIDKey, userID);
        } else {
```

```

visitCount = (Integer)session.getAttribute(visitCountKey);
visitCount = visitCount + 1;
userID = (String)session.getAttribute(userIDKey);
}
session.setAttribute(visitCountKey, visitCount);

```

// Set response content type

```

response.setContentType("text/html");
PrintWriter out = response.getWriter();

```

```
String docType =
```

```

    "<!doctype html public "-//w3c//dtd html 4.0 " +
    "transitional//en">\n";

```

```
out.println(docType +
```

```

    "<html>\n" +
    "<head><title>" + title + "</title></head>\n" +

```

```

    "<body bgcolor = \"#f0f0f0\">\n" +

```

```

    "<h1 align = \"center\">" + title + "</h1>\n" +

```

```

    "<h2 align = \"center\">Session Infomation</h2>\n" +

```

```

    "<table border = \"1\" align = \"center\">\n" +

```

```

        "<tr bgcolor = \"#949494\">\n" +

```

```

            " <th>Session info</th><th>value</th>

```

```

        </tr>\n" +

```

```

        "<tr>\n" +

```

```

            " <td>id</td>\n" +

```

```

            " <td>" + session.getId() + "</td>

```

```

        </tr>\n" +

```

```

"<tr>\n" +
  " <td>Creation Time</td>\n" +
  " <td>" + createTime + " </td>
</tr>\n" +

"<tr>\n" +
  " <td>Time of Last Access</td>\n" +
  " <td>" + lastAccessTime + " </td>
</tr>\n" +

"<tr>\n" +
  " <td>User ID</td>\n" +
  " <td>" + userID + " </td>
</tr>\n" +

"<tr>\n" +
  " <td>Number of visits</td>\n" +
  " <td>" + visitCount + " </td>
</tr>\n" +
"</table>\n" +
"</body>
</html>"
);
}
}

```

Compile the above servlet **SessionTrack** and create appropriate entry in web.xml file. Now running *http://localhost:8080/SessionTrack* would display the following result when you would run for the first time –

Welcome to my website

Session Information

Session info	value
Id	0AE3EC93FF44E3C525B4351B77ABB2D5
Creation Time	Tue Jun 08 17:26:40 GMT+04:00 2010
Time of Last Access	Tue Jun 08 17:26:40 GMT+04:00 2010
User ID	ABCD
Number of visits	0

Now try to run the same servlet for second time, it would display following result.

Welcome Back to my website

Session Information

info type	value
Id	0AE3EC93FF44E3C525B4351B77ABB2D5
Creation Time	Tue Jun 08 17:26:40 GMT+04:00 2010
Time of Last Access	Tue Jun 08 17:26:40 GMT+04:00 2010
User ID	ABCD
Number of visits	1

8. Security Issues

The `servlet-security` quickstart demonstrates the use of Java EE declarative security to control access to Servlets and Security in JBoss Enterprise Application Platform Server.

When you deploy this example, two users are automatically created for you: user `quickstartUser` with password `quickstartPwd1!` and user `guest` with password `guestPwd1!`. This data is located in the `src/main/resources/import.sql` file.

This quickstart takes the following steps to implement Servlet security:

1. Web Application:

- Adds a security constraint to the Servlet using the `@ServletSecurity` `@HttpConstraint` annotations.
- Adds a security domain reference to `WEB-INF/jboss-web.xml`.
- Adds a `login-config` that sets the `auth-method` to `BASIC` in the `WEB-INF/web.xml`.

2. Application Server (`standalone.xml`):

- Defines a security domain in the `elytron` subsystem that uses the JDBC security realm to obtain the security data used to authenticate and authorize users.
 - Defines an `http-authentication-factory` in the `elytron` subsystem that uses the security domain created in step 1 for BASIC authentication.
 - Adds an `application-security-domain` mapping in the `undertow` subsystem to map the Servlet security domain to the HTTP authentication factory defined in step 2.
3. Database Configuration:
- Adds an application user with access rights to the application.

```
User Name: quickstartUser Password: quickstartPwd1! Role: quickstarts
```

- Adds another user with no access rights to the application.

```
User Name: guest Password: guestPwd1! Role: notauthorized
```

9. using JDBC in Servlets

To start with interfacing Java Servlet Program with JDBC Connection:

1. Proper JDBC Environment should set-up along with database creation.
2. To do so, download the `mysql-connector.jar` file from the internet,
3. As it is downloaded, move the jar file to the `apache-tomcat` server folder,
4. Place the file in `lib` folder present in the `apache-tomcat` directory.

5. To start with the basic concept of interfacing:

- **Step 1: Creation of Database and Table in MySQL**

As soon as jar file is placed in the folder, create a database and table in MySQL,

```
mysql> create database demoprj;
```

```
Query OK, 1 row affected (4.10 sec)
```

```
mysql> use demoprj
```

Database changed

```
mysql> create table demo(id int(10), string varchar(20));
```

Query OK, 0 rows affected (1.93 sec)

```
mysql> desc demo;
```

Field	Type	Null	Key	Default	Extra
id	int(10)	YES		NULL	
string	varchar(20)	YES		NULL	

2 rows in set (0.40 sec)

Step 2: Implementation of required Web-pages

Create a form in HTML file, where take all the inputs required to insert data into the database. Specify the servlet name in it, with the POST method as security is important aspects in database connectivity.

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
<title>Insert Data</title>
```

```
</head>
```

```
<body>
```

<!-- Give Servlet reference to the form as an instances

GET and POST services can be according to the problem statement-->

```
<form action="/InsertData" method="post">
```

```
<p>ID:</p>
```

<!-- Create an element with mandatory name attribute,

so that data can be transfer to the servlet using getParameter() -->

```
<input type="text" name="id"/>
```

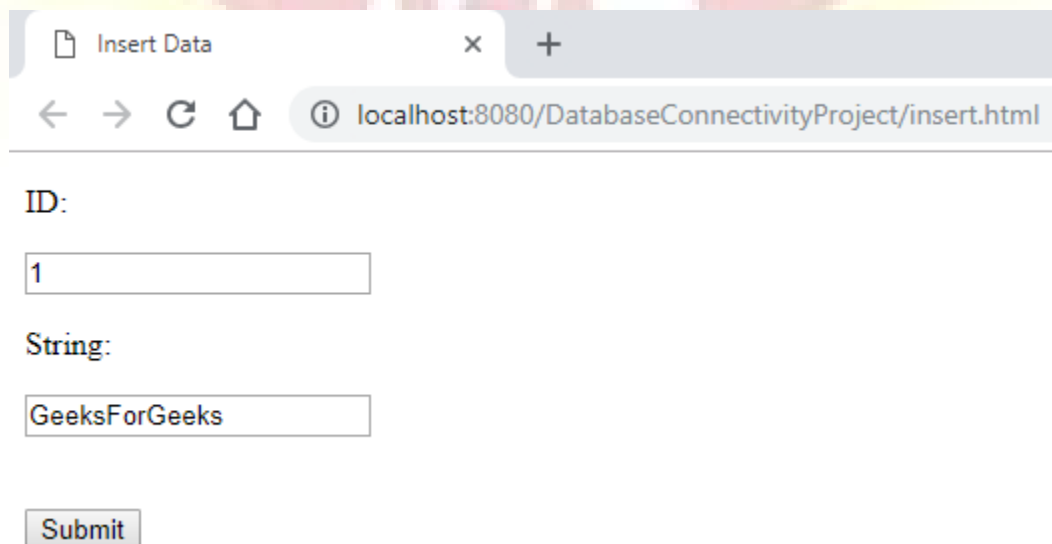
```
<br/>
```

```
<p>String:</p>
```

```
<input type="text" name="string"/>
```

```
<br/><br/><br/>
```

```
<input type="submit"/>
```



The screenshot shows a web browser window with the title 'Insert Data'. The address bar displays 'localhost:8080/DatabaseConnectivityProject/insert.html'. The form contains two text input fields: the first is labeled 'ID:' and contains the value '1'; the second is labeled 'String:' and contains the value 'GeeksForGeeks'. Below the input fields is a 'Submit' button.

Step 3: Creation of Java Servlet program with JDBC Connection

To create a JDBC Connection steps are

1. Import all the packages
2. Register the JDBC Driver
3. Open a connection
4. Execute the query, and retrieve the result
5. Clean up the JDBC Environment

Create a separate class to create a connection of database, as it is a lame process to writing the same code snippet in all the program. Create a .java file which returns a Connection object.

```
import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

// This class can be used to initialize the database connection

public class DatabaseConnection {

    protected static Connection initializeDatabase()

        throws SQLException, ClassNotFoundException

    {

        // Initialize all the information regarding

        // Database Connection

        String dbDriver = "com.mysql.jdbc.Driver";

        String dbURL = "jdbc:mysql://localhost:3306/";
```

```
// Database name to access

String dbName = "demoprj";

String dbUsername = "root";

String dbPassword = "root";

Class.forName(dbDriver);

Connection con = DriverManager.getConnection(dbURL + dbName,

                                         dbUsername,

                                         dbPassword);

return con;

}
```

- **Step 4: To use this class method, create an object in Java Servlet program**
Below program shows Servlet Class which create a connection and insert the data in the **demo** table,

```
import java.io.IOException;

import java.io.PrintWriter;

import java.sql.Connection;

import java.sql.PreparedStatement;
```

```
import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

// Import Database Connection Class file
import code.DatabaseConnection;

// Servlet Name
@WebServlet("/InsertData")

public class InsertData extends HttpServlet {

    private static final long serialVersionUID = 1L;

    protected void doPost(HttpServletRequest request,

HttpServletResponse response)

        throws ServletException, IOException

    {

        try {
```



```
// Initialize the database

Connection con = DatabaseConnection.initializeDatabase();

// Create a SQL query to insert data into demo table

// demo table consists of two columns, so two '?' is used

PreparedStatement st = con

    .prepareStatement("insert into demo values(?, ?)");

// For the first parameter,

// get the data using request object

// sets the data to st pointer

st.setInt(1, Integer.valueOf(request.getParameter("id")));

// Same for second parameter

st.setString(2, request.getParameter("string"));

// Execute the insert command using executeUpdate()

// to make changes in database

st.executeUpdate();

// Close all the connections

st.close();

con.close();
```


Result in MySQL Interface

```
mysql> select * from demo;
```

```
+-----+-----+
| id | string |
+-----+-----+
| 1 | GeeksForGeeks |
+-----+-----+
1 row in set (0.06 sec)
```

This article shows the basic connection of JDBC with Java Servlet Program, to insert data in large volume then proper validation should be done as if any data which is not in proper format will encounter an error. All the data inserting in Database should be encrypted.

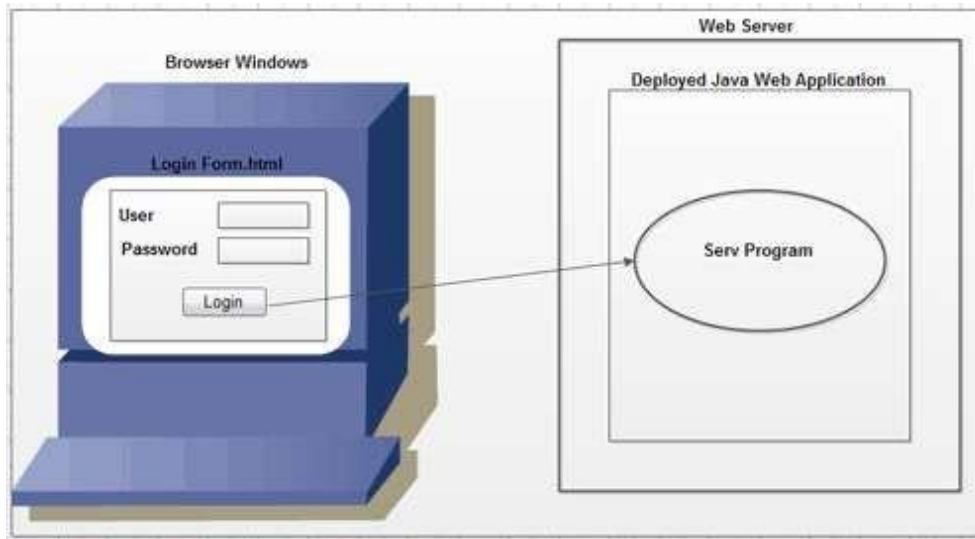
10. HTML to Servlet Communication

A web resource application is a combination of static as well as dynamic web resource programs, images, etc. A static web resource resides in server and is executed in client side web browser, e.g., HTML. A dynamic web resource program resides in server, is executed in context of server and gives response back to the client, e.g., Servlet, JSP. In web application, static web resource takes data from client side and takes it to the dynamic web resource as per request. The dynamic web resource processes the data and sends the response back to client in the form of response.

In HTML to servlet communication, when we open the browser window, the browser page is empty. First, we enter the HTML (static web resource) that resides in the server. After the HTML page is downloaded from the server to the client side browser, the client will be able to enter the data in the HTML form. After submission of the data, request is generated to dynamic web resource. Hence, we generate request for the server twice in order to get HTML to servlet communication.

So far we have sent request to servlet program from browser window by typing request URL in the browser window in this process to send data along with request we need to add query string to the request URL explicitly. But this work can be done only by technical people. Non technical end users like civil engineer, chemical engineer, kids can't do this work so they need a graphical user interface to generous request with or without data .For that

purpose we can use either Html form page or hyperlink to generous with or without data.



The form page generated request carries form data as request parameters along with request.

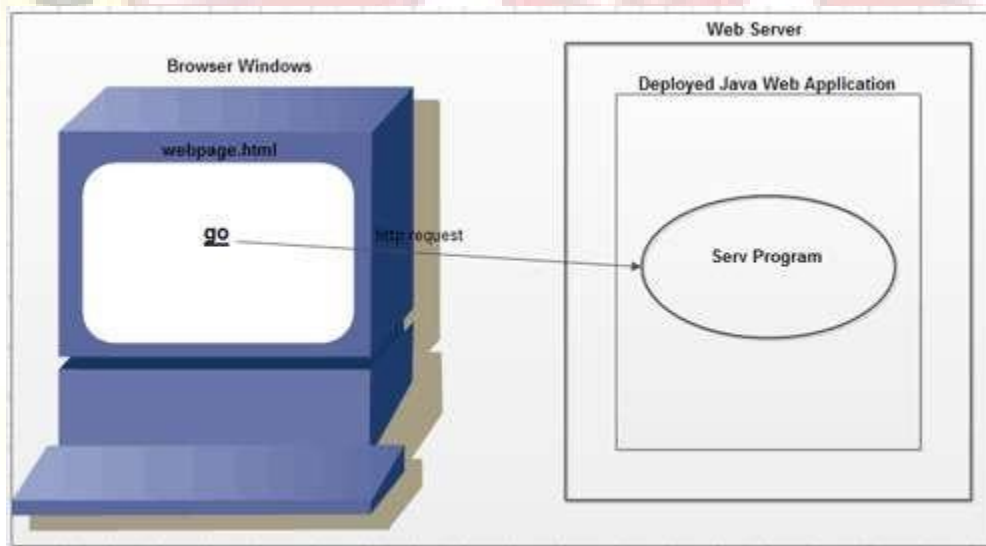
Hyperlink based web page to servlet communication

In this section, we will learn about HTML to servlet communication by using *Hyperlink*. For establishing this type of interaction, we have to pass the URL pattern of the servlet as a value of “href attribute” of the “action tag”. It is to be noted that when we are using hyperlink, the request is submitted to the servlet using “http get method”. Hyperlink are favorable for either of inter or intra linking between the pages, but not for communication between active resources. In other words, only static data can be passed from HTML to servlet and not the dynamic data

Example 1: `Click Here to go to servlet program`

Example 2: ` Login </ a>`

In second example, the static data UName=King and Pass=Queen will be send to servlet.



Generally the hyperlink generated request is blank request that means it can not carry any data along with the request.

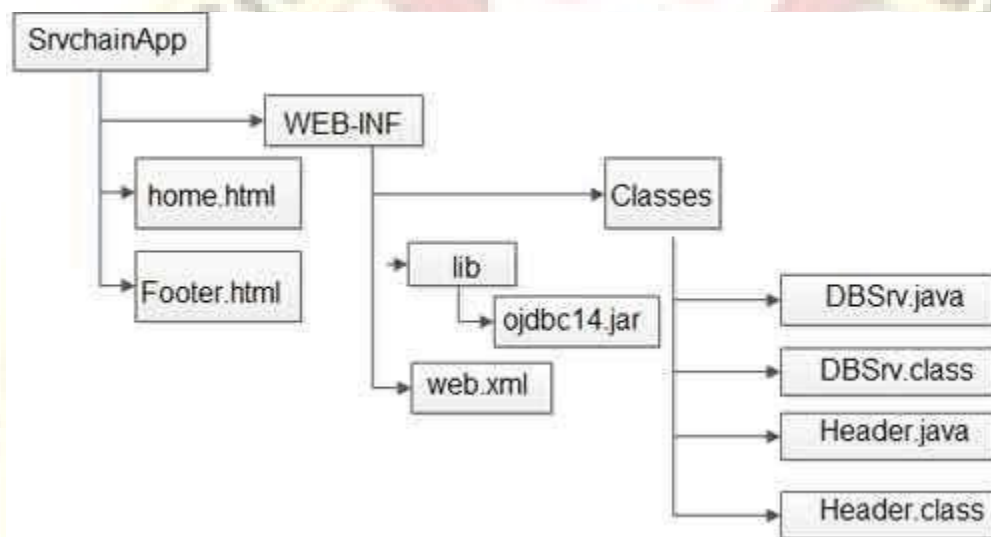
html files of web application must be placed parallel to WEB-INF folder in deployment directory structure of web application there is no need of configure then in web.xml file.

Example Application (hyperlink-based HTML to servlet program communication)

- WishSrv servlet program generates the wish message based on the server machine current time.
- .html-based web page are always static pages, whereas servlet and JSP program based web pages can be static or dynamic pages.

Step 1: Prepare the deployment directory structure of web application.

Deployment Directory Structure



Step 2: Develop the source code of above servlet program or webApplication

- Place servlet program request URL with URL pattern as the value of href attribute.

ABC.HTML

<!-- Web page having hyper links -->

GET WISHING

WishSrv.java

```
import javax.servlet.*;
```

```
import javax.servlet.http.*;
```

```
import java.io.*;

import Java.util.*;

public class wishsrv extends HttpServlet

{

public void service(HttpServletRequest req,HttpServletResponse res) throws ServletException ,
IOException

{

//get the printwriter object

printWriter pw=res.getWriter();

res.setContentType("text/html");

//set the mime type response

Calendar cl=Calendar.getInstance();

//give curent date and time

//get cuurent hour of day

int h=cl.get(Calendar.HOUR_OF_DAY);

if(h<=12)

pw.println("<center><font color='red' size=6> GOOD MORNING </FONT>

</CENTER>");

else if (h>17)

pw.println("<center><font color='red' size=6> GOOD AFTERNOON </FONT>

</CENTER>");

else

pw.println(., "<center><font color='red' size=6> GOOD NIGHT </FONT></CENTER>");
```

```
pw.println (“<center><font color='red' size=6> <a href=' https://ecomputernotes.com :2020/
Wishapp/home.html'> HOME</a></FONT></CENTER>”);
```

```
pw,close() ;
```

```
}//end of service
```

```
}// end of class
```

web.xml

```
<web-app>
    <servlet>
        <servlet-name>abc</servlet-name>
        <servlet-class>wishSrv</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>abc</servlet-name>
        <url-pattern>/wurl</url-pattern>
    </servlet-mapping>
</web-app>
```

Explanation of the above Program

In this program our aim is to click on a link so that it points to a servlet program and we have to include that link within the HTML page by using the *href* attribute of HTML. Within this *href* tag we pass the URL pattern of the servlet program.

ABC.html:

First we design the HTML page that is displayed on the browser window. In the HTML program we use the *href* attribute, and within this we pass the URL pattern of the servlet program as follows:

```
<a href=http://localhost :2020!WishApp/WurbGETWISHING
```


When we click on the GETWISHING link at that time the request goes to the servlet program.

WishSrv.java:

- In the above program first one needs to import all the packages like javax.

servlet. * I j avax.servlet.http. * | java.io.* **and** java.util. *. **Our** class WishServ extends from the HttpServlet class within this class the life cycle method, i.e, service(.,_) is overridden.

- **printWriter pw;res.getWriter();**

With the help of the PrintWriter class the response print on the browser window.

- **res.setContentType(“text/html”);**

With the help of the setContentType(“ _ “)of the ServletResponse interface set the format of the browser window.

- **Calendar cl=Calendar.getInstance();**

Here we use the Calendar class which is present within the java.util package that is why java.util package is imported. getInstance() is a static method of the Calendar class. Hence this method is invoked through the class name, i.e., “Calendar class”. The return type of the getInstance() is the object of the Calendar class. So by mentioning” *Calendar cl=Calendar.getInstance();*” we can create the object of the Calendar class. This getInstance() gives the current date and time.

- **int h=cl.get(Calendar.HOUR_OF_DAY);**

In order to get the current hour of the day we use the constant of the Calendar class, i.e., HOUR_OF_DAY which is invoked through the class name. Then by using the get() of the Calendar class we retrieve the current hour of the day and assign it to the variable “h” of int type.

- *if(h<=12)*

By using the if conditional statement, i.e., if we compare that if(h<=12) if this condition is satisfied then control enter into the if block and print the message “GOOD MORNING” on the browser window otherwise the control goes to the elseif block. If the condition of the elseif block is not satisfied then at last the control moves to the else block and print the “GOOD

NIGHT” message and also print the message “HOME”. As in this case within the href attribute we pass the URL pattern of the home.htm!. When we click on HOME, the home page is opened.

- *pw.close()*;

Then by calling the close() of the PrintWriter class close the PrintWriter stream class.

Step 3: Compile the source files of all servlet programs.

Step 4: Configure all the four servlet programs in web.xml file having four different URL patterns.

Step 5: Start the server (Tomcat)

Step 6: Deploy the web application.

Copy WishApp folder to Tomcat_home \ webapps folder.

11. Applet to servlet communication

HTML exhibits high performance by taking less time to load in the browser. However, when we use HTML page for important user details, by default, all the parameters that are passed appended in the URL. This compromises with the security. On the other hand, applet takes more time to load but there is no problem with Java security. This is an advantage of this technique.

Applet is a compiled Java class that can be sent over the network. Applets are an alternative to HTML form page for developing websites. HTML form gives good performance, takes less time to load but has poor security. Whereas, Applets give poor performance, take time to load but have good security.

There are two types of Applets:

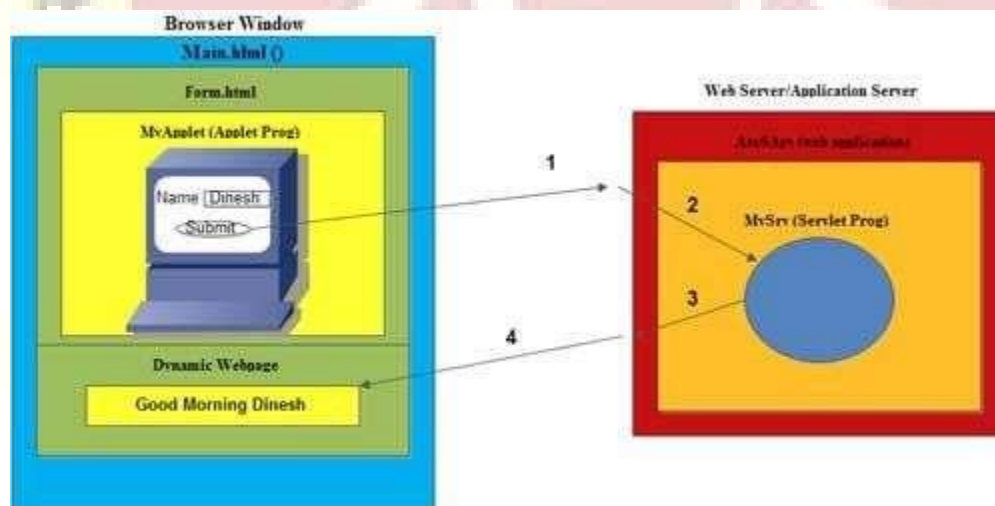
1. Untrusted Applets: It cannot interact with files and file system, so writing malicious codes is not possible. Applets are untrusted.
2. Trusted Applets: It can interact with files and file system so can write malicious codes.

Difference between *applet* and *servlet*

APPLET	SERVLET
Used to develop client side web-resource program to generate static web page.	Used to develop server side web-resource program to generate dynamic web-page.
Needs browser window or appletviewer for execution.	Needs servlet container for execution.
Applet program comes to browser window from server for execution.	Servlet program reside and execute in web resource.
The life cycle methods are init(), start(), stop() and destroy().	The life cycle methods are init(-), Service(-,-) and destroy().

Similar to HTML-to-servlet communication we need to perform applet-to servlet communication. In HTML-to-servlet communication the browser window automatically forms request URL and automatically forms query string having form data, when submit button is clicked. In applet-to-servlet communication all these activities should be taken care of by the programmers manually or explicitly as event handling operation.

Example on Application of Applet to Servlet Communication



Frame is a logical partition of the web page. Frame with name is called Named ,
Frame.

Step 1: Prepare the deployment directory structure of web application.

Request url <http://localhost:2020/AtoSApp/Main.html>

Step 2: Develop the source code of above servlet program or web Application.

Source

Code

MyApplet.java

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

public class wishsrv extends HttpServlet {
    public void service(HttpServletRequest req,HttpServletResponse res)
        throws ServletException , IOException {
        //general settings
        PrintWriter pw=res.getWriter();
        setContentType("text/html") ;
        //read form data
        String name=req.getParameter("uname") ;
        //generate wish message
        Calendar cl=Calendar.getInstance();
        int h=cl.get(Calendar.HOUR_OF_DAY);
        if (h<=12)
            pw.println ("Good Morning :"+name) ;
        elseif(h<=16)
            pw.println("Good Afternoon: "+name);
        elseif(h<=20)
            pw.println("Good Evening :"+name);
        else
```

```

    pw.println("Good Night :"+name);
    //close stream obj
    pw.close();
} //doGet
} //class
<> javac MyServlet.java

```

web.xml

Configure MyServlet program with /testurl url pattern and also configure Main.html as welcome file.

```

<web-app>
  <servlet>
    <servlet-name>abc</servlet-name>
    <servlet-class>MyServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>abc</servlet-name>
    <url-pattern>/testurl</url-pattren>
  </servlet-mapping>
  <welcome-file-list>
    <welcome-file>Main.html</welcome-file>
  </welcome-file-list>
</web-app>

```

Main.html

```

<frameset rows = "40% , *">
  <frame name = "f1" SRC = "Form.html">
  <frame = &nbsp;"f2" />
</frameset>

```

MyApplet.java

```

// MyApplet. Java
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
import java.net.*;
public class MyApplet extends Applet implements ActionListener {
    Button b;
    TextField tfl;
    Label l1;
    public void init () {
        l1 = new Label ("User name :");
        add(l1) ;
        tfl = new TextField(10);
        add(tfl) ;
        b=new Button("Send");
        b.addActionListener(this) ;
        add(b) ;
    }
    public void actionPerformed(ActionEvent ae) {
        try{
            //read text value
            String name=tfl.getText().replace();
            //frame query String
            String qrystr="(?"uname="+name) ;
            //frame request url having query String
            String url="(https://ecomputernotes.com:2020/AtoSApp/testurl"+qrystr);
            //create URL class object
            URL requrl = new URL (ur1);
            //getAppletContext obj
            AppletContext apc=getAppletContext();
        } catch(Exception ee)

```

```
{}
{}

```

Form.html

```
<applet code= "MyApplet.class" width= "500" height= "500">
</applet>
```

Step 3: Compile the source files of all servlet programs.

Step 4: Configure all the four servlet programs in web.xml file having four different url patterns.

Step 5: Start the server (Tomcat).

Step 6: Deploy the web application.

Copy AutoSApp folder Tomcat_home \ webapps folder.

Step 7: Test the web application.

Open browser window type this url-[https://ecomputernotes.com:2020/ AtoSApp/ Main.html](https://ecomputernotes.com:2020/AtoSApp/Main.html)

ADVANCED JAVA PROGRAMMING

UNIT-II

JAVA BEANS

The Software Component Assembly Model

Components and Containers

JavaBeans are Java software components that are designed for maximum reuse. They are often visible GUI components, but can also be invisible algorithmic components. They support the software component model pioneered by Microsoft's Visual Basic and Borland's Delphi. This model focuses on the use of components and containers.

Components are specialized, self-contained software entities that can be replicated, customized, and inserted into applications and applets. Containers are simply components that contain other components. A container is used as a framework for visually organizing components. Visual development tools allow components to be dragged and dropped into a container, resized, and positioned.

You are familiar with the concept of components and containers from your study of the AWT. The components and containers of the JavaBeans component model are similar in many ways to the Component and Container classes of the AWT.

- Components come in a variety of different implementations and support a wide range of functions.
- Numerous individual components can be created and tailored for different applications.
- Components are contained within containers.
- Components can also be containers and contain other components.
- Interaction with components occurs through event handling and method invocation.

In other ways, the components and containers of JavaBeans extend beyond the Component and Container classes of the AWT.

- JavaBeans components and containers are not restricted to the AWT. Almost any kind of Java object can be implemented as a JavaBean.
- Components written in other programming languages can be reused in Java software development via special Java interface code. You'll learn how to use non-Java components, such as Component Object Model (COM) objects in Chapter 54, "Dirty Java."
- Components written in Java can be used in other component implementations, such as ActiveX, via special interfaces referred to as bridges. You'll also study bridges in Chapter 54, "Dirty Java."

The important point to remember about JavaBeans components and containers is that they support a hierarchical software development approach where simple components can be assembled within containers to produce more complex components. This capability allows software developers to make maximum reuse of existing software components when creating new software or improving existing software.

Introspection and Discovery

Component interfaces are well-defined and may be discovered during a component's execution. This feature, referred to as introspection, allows visual programming tools to drag and drop a component onto an application or applet design and dynamically determine what component interface methods and properties are available. Interface methods are public methods of a bean

that are available for use by other components. Properties are attributes of a bean that are implemented by the bean class's field variables and accessed via *accessor* methods.

JavaBeans support introspection at multiple levels. At a low level, introspection can be accomplished using the reflection capabilities of the `java.lang.reflect` package. These capabilities allow Java objects to discover information about the public methods, fields, and constructors of loaded classes during program execution. Reflection allows introspection to be accomplished for all beans. All you have to do is declare a method or variable as public and it can be discovered using reflection.

An intermediate level introspection capability provided by JavaBeans utilizes design patterns. Design patterns are method naming conventions that are used by the introspection classes of `java.beans` to infer information about reflected methods based on their names. For example, design patterns can be used by visual design tools to identify a bean's event generation and processing capabilities by looking for methods that follow the naming conventions for event generation and event listening. Design tools can use design patterns to obtain a great deal of information about a bean in the absence of explicitly provided information.

Design patterns are a low overhead approach to supporting introspection in component development. All you have to do is adhere to the naming convention of design patterns and visual design tools will be able to make helpful inferences about how your components are used.

At the highest level, JavaBeans supports introspection through the use of classes and interfaces that provide explicit information about a bean's methods, properties, and events. By explicitly providing this information to visual design tools, you can add help information and extra levels of design documentation that will be automatically recognized and presented in the visual design environment.

Persistence

The property sheets of visual design tools are used to tailor the properties of components for specific applications. The modified properties are stored in such a manner that they remain with the component from design to execution. The capability to store changes to a component's properties is known as persistence. Persistence allows components to be customized for later use. For example, during design, you can create two button beans••one with a blue background color and a yellow foreground color and another with a red background color and a white foreground color. The color modifications are stored along with instances of each bean object.

When the beans are displayed during a program's execution, they are displayed using the modified colors.

JavaBeans supports persistence through object serialization. *Object serialization* is the capability to write a Java object to a stream in such a way that the definition and current state of the object are preserved. When a serialized object is read from a stream, the object is initialized and in exactly the same state it was in when it was written to the stream. Figure 24.5 summarizes how object serialization supports persistence. Chapter 40, "Using Object Serialization and JavaSpaces," covers object serialization.

Events

Visual development tools allow components to be dragged and dropped into a container, resized, and positioned. The visual nature of these tools greatly simplifies the development of user interfaces. However, component-based visual development tools go beyond simple screen layout. They also allow component event handling to be described in a visual manner.

You should be familiar with events, having worked with event handling code in most of the examples in this book. In general, events are generated in response to certain actions, such as the user clicking or moving the mouse or pressing a keyboard key. The event is handled by an event handler. Beans can handle events that occur local to them. For example, a button-based bean is required to handle the clicking of a button. Beans can also call upon other beans to complete the handling of an event. For example, a button bean can handle the button-clicked event by causing a text string to be displayed in a status-display bean. Visual development tools support the connection of event sources (for example, the button bean) with event listeners (for example, the status-display bean) using graphical design tools. In many cases, event handling can be performed without having to write event-handling code. You'll see a concrete example of this when you use the BDK in the next chapter. This code is automatically generated by the visual design tools. Figure 24.6 graphically depicts the relationship between event sources and event listeners.

Visual Design

One of the ultimate benefits of using a component-based approach to software development is that you can use visual design tools to support your software development efforts. These tools greatly simplify the process of complex software development. They also allow you to develop higher-quality software, more quickly, and at a lower cost. Some of the features typically found in component-based visual design tools are as follows:

- Components and containers can be dragged onto a visual design worksheet.
- Components can be dragged into containers and assembled into more complex, higher-level components.
- Visual layout tools can be used to organize components within containers.
- Property sheets can be used to tailor component properties for different applications.
- Component interaction editors can be used to connect the events generated by one component with the interface methods of other components.
- Code can be automatically generated to implement visual interface designs.
- Traditional software development tools, such as source code editors, compilers, debuggers, and version control managers can be integrated within the visual design environment.

The JavaBeans Development Kit

Inside the BDK

The BDK provides several examples of JavaBeans, a tutorial, and supporting documentation. But most important, it provides a tool, referred to as the BeanBox, that can be used to display, customize, and test the beans that you'll develop. The BeanBox also serves as a primitive visual development tool. You'll use the BeanBox to see the important aspects of visual component-based software development as it applies to JavaBeans. Download and install the BDK before continuing on to the next section. Once you've installed the BDK, restart your system to make sure that all installation changes take effect.

Using the BeanBox

The BeanBox of the BDK is an example of a simple visual development tool for JavaBeans. It is located in the `c:\bdk\beanbox` directory. Change to this directory and start the BeanBox as follows: `c:\bdk\beanbox>run`

The BeanBox application loads and displays three windows labeled ToolBox, BeanBox, and PropertySheet, as shown in Figures 25.1, 25.2, and 25.3.

The ToolBox window contains a list of available Java beans. These beans are components that can be used to build more complex beans, Java applications, or applets.

Visual software development tools, such as the BeanBox, allow beans to be visually organized by placing them at the location where you want them to be displayed.

Understanding the Example Beans

You should be impressed by how easy it was to develop an interesting (or at least entertaining) application using the BeanBox and JavaBeans. In fact, you didn't have to write a single line of code to create the application. That's the power of component-based software development.

Given a good stock of beans, you can quickly and easily assemble a large variety of useful applications.

In the example of the previous section, you learned how to use the OurButton and Juggler beans. The ToolBox that comes with the BeanBox lists 16 beans. I recommend that you play around with these beans to familiarize yourself with how they work. You studied the theory behind component-based software in the previous chapter. Now is the time to get some practical experience to back up your theoretical understanding. Try to see how the BeanBox and the example beans support the component-based model described in Chapter 24.

Just to whet your appetite, what follows is a short description of the beans that are in the ToolBox.

- BlueButton••A simple blue button with background, foreground, label, and font properties
- OrangeButton••A simple orange button with background, foreground, label, and font properties
- OurButton••A gray button with additional font properties
- ExplicitButton••A simple gray button with background, foreground, label, and font properties
- EventMonitor••A text area that is used to view events as they happen
- JellyBean••A jelly bean that is associated with a cost
- Juggler••A juggler animation
- TickTock••An interval timer
- Voter••A component that maintains a yes or no state

- `ChangeReporter`••A text field
- `Molecule`••A graphical field for displaying 3D pictures of molecules
- `QuoteMonitor`••A component that displays stock quotes received from a quote server
- `JDBC SELECT`••An SQL interface to a database
- `SorterBean`••An animation of a bubble sort
- `BridgeTester`••A bean used to test bean bridges (refer to Chapter 28, "Using Bridges")
- `TransitionalBean`••A button that changes colors

Now fire up the BeanBox and try out some of these beans. You'll learn how the `java.beans` packages support the implementation of the capabilities that you observe with the BeanBox.

Developing Beans

Customizable and Persistent Properties

Properties are attributes of a bean that can be modified to change the appearance or behavior of a bean. They are accessed through special methods, referred to as *accessor* methods. Visual development tools allow properties to be changed through the use of *property sheets*, lists of properties that can be specified for a bean. Visual building tools, like the BeanBox, display a property sheet in response to a bean's selection. You used property sheets to change the `animationRate` property of the `Juggler` bean and the `label` property of the `OurButton` bean.

In addition to the simple property editing capabilities exhibited by the BeanBox example, individual beans can define custom property editors that allow properties to be edited using specialized dialog boxes. These custom property editors are implemented as special classes that are associated with the bean's class. The custom property editors are available to visual development tools, but because they are not part of the bean's class, they do not need to be compiled into applications or applets. This lets you provide extra *design* capabilities for a bean without having to develop bloated applications. Suppose that you are using a bean that provides extensive customization support. You change the bean's background color to red and its foreground color to white, change a label associated with the bean, and alter a few other properties. You may wonder what happens to the property changes. How are the changes packaged along with the bean's class?

Beans store any property changes so that new property values come into effect and are displayed when the modified bean is used in an application. The capability to permanently store property changes is known as *persistence*. JavaBeans implement persistence by serializing bean objects that are instances of a bean class. *Serialization* is the process of writing the current state of an object to a stream. Because beans are serialized, they must implement the `java.io.Serializable` or `java.io.Externalizable` interfaces. Beans that implement `java.io.Serializable` are automatically saved. Beans that implement `java.io.Externalizable` are responsible for saving themselves.

When a bean object is saved through serialization, all of the values of the variables of the object are saved. In this way, any property changes are carried along with the object. The only exceptions to this are variables that are identified as transient. The values of transient variables are not serialized.

Bean Properties

Beans support a few different types of properties. In the BeanBox tutorial, you saw examples of *simple* properties. The `animationRate` property of the Juggler bean used a simple numeric value, and the `label` property of the OurButton bean used a text value.

An *indexed property* is a property that can take on an array of values. Indexed properties are used to keep track of a group of related values of the same type. For example, an indexed property could be used to maintain the values of a scrollable list.

A *bound* property is a property that alerts other objects when its value changes. For example, you could use a bound property to implement a temperature control dial. Whenever the user changes the control, notification of the change is propagated to objects that regulate temperature.

A *constrained* property differs from a bound property in that it notifies other objects of an *impending* change. Constrained properties give the notified objects the power to veto a property change. You could use a constrained property to implement a bean that fires a missile under twoperson control. When one person initiates a missile launch, a notification is sent to a second user, who could either confirm or deny the launch.

Accessor Methods

All properties are accessed through accessor methods. There are two types of accessor methods: *getter* methods and *setter* methods. Getter methods retrieve the values of properties, and setter methods set property values. The names of getter methods begin with `get` and are followed by the

name of the property to which they apply. The names of setter methods begin with set and are followed by the property name.

Methods Used with Simple Properties

If a bean has a property named fooz of type foozType that can be read and written, it should have the following accessor methods:

```
public foozType getFooz() public void
setFooz(foozType foozValue)
```

A property is read-only or write-only if one of the preceding accessor methods are missing.

Methods Used with Indexed Properties

A bean that has an indexed property will have methods that support the reading and writing of individual array elements as well as the entire array. For example, if a bean has an indexed widget property in which each element of the array is of type widgetType, it will have the following accessor methods:

```
public widgetType getWidget(int index) public
widgetType[] getWidget()
public void setWidget(int index, widgetType widgetValue) public void
setWidget(widgetType[] widgetValues)
```

Methods Used with Bound Properties

Beans with bound properties have getter and setter methods, as previously identified, depending upon whether the property values are simple or indexed. Bound properties require certain objects to be notified when they change. The change notification is accomplished through the generation of a PropertyChangeEvent. Objects that want to be notified of a property change to a bound property must register as listeners. Accordingly, the bean that's implementing the bound property supplies methods of the form:

```
public void addPropertyChangeListener(PropertyChangeListener l) public void
removePropertyChangeListener(PropertyChangeListener l)
```

The preceding listener registration methods do not identify specific bound properties. To register listeners for the PropertyChangeEvent of a specific property, the following

```

methods      must      be      provided:      public      void
addPropertyNameListener(PropertyChangeListener l)
public void removePropertyNameListener(PropertyChangeListener l)

```

In the preceding methods, `PropertyName` is replaced by the name of the bound property.

Objects that implement the `PropertyChangeListener` interface must implement the `propertyChange()` method. This method is invoked by the bean for all of its registered listeners to inform them of a property change.

Methods Used with Constrained Properties

The previously discussed methods used with simple and indexed properties also apply to constrained properties. In addition, the following event registration methods are provided:

```

public void addVetoableChangeListener(VetoableChangeListener l) public void
removeVetoableChangeListener(VetoableChangeListener l) public void
addPropertyNameListener(VetoableChangeListener l) public void
removePropertyNameListener(VetoableChangeListener l)

```

Objects that implement the `VetoableChangeListener` interface must implement the `vetoableChange()` method. This method is invoked by the bean for all of its registered listeners to inform them of a property change. Any object that does not approve of a property change can throw a

`PropertyVetoException` within its `vetoableChange()` method to inform the bean whose constrained property was changed that the change was not approved.

Introspection

In order for beans to be used by visual development tools, the beans must be able to dynamically inform the tools of their interface methods and properties and also what kind of events they may generate or respond to. This capability is referred to as *introspection*. The `Introspector` class of `java.beans` provides a set of static methods for tools to obtain information about the properties, methods, and events of a bean.

The Introspector supports introspection in the following ways:

- Reflection and design patterns••The `java.lang.reflect` package provides the capability to identify the fields and methods of a class. The Introspector uses this capability to review the names of the methods of a bean's class. It identifies a bean's properties by looking at the method names for the getter and setter naming patterns, identified in previous sections of this chapter. It identifies a bean's event generation and processing capabilities by looking for methods that follow the naming conventions for event generation and event listening. The Introspector automatically applies reflection and design (naming) patterns to a bean class to obtain information for design tools in the absence of explicitly provided information.
- Explicit specification••Information about a bean may be optionally provided by a special bean information class that implements the `BeanInfo` interface. The `BeanInfo` interface provides methods for explicitly conveying information about a bean's methods, properties, and events. The Introspector recognizes `BeanInfo` classes by their name. The name of a `BeanInfo` class is the name of the bean class followed by `BeanInfo`. For example, if a bean was implemented via the `MyGizmo` class, the related `BeanInfo` class would be named `MyGizmoBeanInfo`.

Connecting Events to Interface Methods

Beans, being primarily GUI components, generate and respond to events. Visual development tools provide the capability to link events generated by one bean with event• handling methods implemented by other beans. For example, a button component may generate an event as the result of the user clicking on that button. A visual development tool would enable you to connect the handling of this event to the interface methods of other beans. The bean generating the event is referred to as the *event source*. The bean listening for (and handling) the event is referred to as the *event listener*.

Inside `java.beans`

Now that you have a feel for what beans are, how they are used, and some of the mechanisms they employ, let's take a look at the classes and interfaces of the `java.beans` packages. These classes and interfaces are organized into the categories of design support, introspection support, and change event•handling support.

Design Support

The classes in this category help visual development tools to use beans in a design environment.

The Beans class provides seven static methods that are used by application builders:

- `instantiate()`••Creates an instance of a bean from a serialized object.
- `isInstanceOf()`••Determines if a bean is of a specified class or interface.
- `getInstanceof()`••Returns an object that represents a particular view of a bean.
- `isDesignTime()`••Determines whether beans are running in an application builder environment.
- `setDesignTime()`••Identifies the fact that beans are running in an application builder environment.
- `isGuiAvailable()`••Determines whether a GUI is available for beans.
- `setGuiAvailable()`••Identifies the fact that a GUI is available for beans.

The Visibility interface is implemented by classes that support the capability to answer questions about the availability of a GUI for a bean. It provides the `avoidingGui()`, `dontUseGui()`, `needsGui()`, and `okToUseGui()` methods. The VisibilityState interface provides the `isOkToUseGui()` method. The methods of the PropertyEditor interface are implemented by classes that support custom property editing. These methods support a range of property editors, from simple to complex. The `setValue()` method is used to identify the object that is to be edited. The `getValue()` method returns the edited value. The `isPaintable()` and `paintValue()` methods support the painting of property values on a Graphics object. The `getJavaInitializationString()` method returns a string of Java code that is used to initialize a property value. The `setAsText()` and `getAsText()` methods are used to set and retrieve a property value as a String object. The `getTags()` method returns an array of String objects that are acceptable values for a property. The `supportsCustomEditor()` method returns a boolean value indicating whether a custom editor is provided by a PropertyEditor. The `getCustomEditor()` method returns an object that is of a subclass of Component and is used as a custom editor for a bean's property. The `addPropertyChangeListener()` and `removePropertyChangeListener()` methods are used to register event handlers for the PropertyChangeEvent associated with a property.

The PropertyEditorManager class provides static methods that help application builders find property editors for specific properties. The `registerEditor()` method is used to register an editor

class for a particular property class. The `getEditorSearchPath()` and `setEditorSearchPath()` methods support package name lists for finding property editors. The `findEditor()` method finds a property editor for a specified class. Unregistered property editors are identified by the name of the property followed by Editor.

The `PropertyEditorSupport` class is a utility class that implements the `PropertyEditor` interface. It is subclassed to simplify the development of property editors.

The methods of the `Customizer` interface are implemented by classes that provide a graphical interface for customizing a bean. These classes are required to be subclasses of `java.awt.Component` so that they can be displayed in a panel. The `addPropertyChangeListener()` method is used to enable an object that implements the `PropertyChangeListener` interface as an event handler for the `PropertyChangeEvent` of the object being customized. The `removePropertyChangeListener()` method is used to remove a `PropertyChangeListener`. The `setObject()` method is used to identify the object that is to be customized.

Introspection Support

The classes and interfaces in this category provide information to application builders about the interface methods, properties, and events of a bean.

The Introspector Class

The `Introspector` class provides static methods that are used by application builders to obtain information about a bean's class. The `Introspector` gathers this information using information explicitly provided by the bean designer whenever possible and uses reflection and design patterns when explicit information is not available. The `getBeanInfo()` method returns information about a class as a `BeanInfo` object. The `getBeanInfoSearchPath()` method returns a `String` array to be used as a search path for finding `BeanInfo` classes. The `setBeanInfoSearchPath()` method updates the list of package names used to find `BeanInfo` classes. The `decapitalize()` method is used to convert a `String` object to a standard variable name in terms of capitalization.

The BeanInfo Interface

The methods of the `BeanInfo` interface are implemented by classes that want to provide additional information about a bean. The `getBeanDescriptor()` method returns a `BeanDescriptor`

object that provides information about a bean. The `getIcon()` method returns an `Image` object that is used as an icon to represent a bean. It uses the icon constants defined in `BeanInfo` to determine which type of icon should be returned. The `getEventSetDescriptors()` method returns an array of `EventSetDescriptor` objects that describe the events generated (fired) by a bean. The `getDefaultEventIndex()` method returns the index of the most commonly used event of a bean. The `getPropertyDescriptors()` method returns an array of `PropertyDescriptor` objects that support the editing of a bean's properties. The `getDefaultPropertyIndex()` method returns the most commonly updated property of a bean. The `getMethodDescriptors()` method returns an array of `MethodDescriptor` objects that describe a bean's externally accessible methods. The `getAdditionalBeanInfo()` method returns an array of objects that implement the `BeanInfo` interface.

The SimpleBeanInfo Class

The `SimpleBeanInfo` class provides a default implementation of the `BeanInfo` interface. It is subclassed to implement `BeanInfo` classes.

The FeatureDescriptor Class and Its Subclasses

The `FeatureDescriptor` class is the top-level class of a class hierarchy that is used by `BeanInfo` objects to report information to application builders. It provides methods that are used by its subclasses for information gathering and reporting.

The `BeanDescriptor` class provides global information about a bean, such as the bean's class and its `Customizer` class, if any. The `EventSetDescriptor` class provides information on the events generated by a bean. The `PropertyDescriptor` class provides information on a property's accessor methods and property editor. It is extended by the `IndexedPropertyDescriptor` class, which provides access to the type of the array implemented as an indexed property and information about the property's accessor methods.

The `MethodDescriptor` and `ParameterDescriptor` classes provide information about a bean's methods and parameters.

Change Event•Handling Support

The `PropertyChangeEvent` is generated by beans that implement bound and constrained properties as the result of a change in the values of these properties. The `PropertyChangeListener` interface is implemented by those classes that listen for the `PropertyChangeEvent`. It consists of a single method, `propertyChange()`, that is used to handle the event.

The `VetoableChangeListener` interface is implemented by classes that handle the `PropertyChangeEvent` and throw a `VetoableChangeEvent` in response to certain property changes. The `vetoableChange()` method is used to handle the `PropertyChangeEvent`.

The `PropertyChangeSupport` class is a utility class that can be subclassed by beans that implement bound properties. It provides a default implementation of the `addPropertyChangeListener()`, `removePropertyChangeListener()`, and `firePropertyChange()` methods.

The `VetoableChangeSupport` class, like the `PropertyChangeSupport` class, is a utility class that can be subclassed by beans that implement constrained properties. It provides a default implementation of the `addVetoableChangeListener()`, `removeVetoableChangeListener()`, and `fireVetoableChange()` methods.

Aggregation

The `Aggregate` interface has been added in JDK 1.2 as a means of aggregating several objects into a single bean. It is extended by the `Delegate` interface, which provides methods for accessing `Aggregate` objects. The `AggregateObject` class is an abstract class that implements the `Delegate` interface and provides a foundation for creating other aggregate classes. Note that aggregation has nothing to do with inheritance. It is just a way of combining multiple objects into a single bean.

The `java.beans.beancontext` Package

JDK 1.2 introduces the `java.beans.beancontext` package, which provides classes and interfaces for enabling beans to access their execution environment, referred to as their *bean context*. The `BeanContextChild` interface provides methods for getting and setting this context and for managing context•related event listeners. `BeanContextChild` is extended by the `BeanContext` interface, which provides methods by which beans can access resources and services that are available within their context. Objects that implement `BeanContext` function as containers for other beans.

The `BeanContextMemberShipListener` interface provides an event listener interface for events that occur as the result of changes to the beans that are members of a bean context. The `DesignMode` interface of `java.beans` provides the capability for a `BeanContext` object to determine whether it is being executed in a design or execution mode.

In addition to the interfaces described in the previous paragraph, the `java.beans.beancontext` package provides the following six classes:

- `BeanContextEvent`••Events of this class are fired when the state of a bean context changes.
- `BeanContextMembershipEvent`••Extends `BeanContextEvent` to support changes in the membership of a bean context.
- `BeanContextAddedEvent`••Events of this class are fired when a bean is added to a bean context. This class extends `BeanContextMembershipEvent`.
- `BeanContextRemovedEvent`••Events of this class are fired when a bean is removed from a bean context. This class extends `BeanContextMembershipEvent`.
- `BeanContextSupport`••Provides an implementation of the `BeanContext` interface.
- `BeanContextSupport.BCChildInfo`••Used to maintain information on the beans that are contained within a bean context.

The easiest way to implement a bean context is to extend the `BeanContextSupport` class. `BeanContextSupport` provides numerous methods for managing beans that are contained within a particular context.

Developing Beans

A Gauge Bean

When you studied all of the classes and interfaces of `java.beans` in previous sections, you might have been left with the impression that beans are complicated and hard to develop. In fact, the opposite is true. You can easily convert existing classes to beans with minimal programming overhead.

It contains the code for a bean that displays a simple gauge. The gauge is displayed as a 3D•style box that is filled somewhere between its minimum and maximum values. The color of the

gauge's border and its fill color are both configurable. So are its dimensions and horizontal/vertical orientation.

THE Gauge.java BEAN.

```
import java.io.Serializable;
import java.beans.*; import
java.awt.*;      import
java.awt.event.*;
public class Gauge extends Canvas implements Serializable {
    // Set constants and default values public static
    final int HORIZONTAL = 1; public static final int
    VERTICAL = 2; public static final int WIDTH =
    100; public static final int HEIGHT = 20; public
    int orientation = HORIZONTAL; public int width
    = WIDTH; public int height = HEIGHT; public
    double minValue = 0.0; public double maxValue
    = 1.0; public double currentValue = 0.0; public
    Color gaugeColor = Color.lightGray; public Color
    valueColor = Color.blue; public Gauge() {
    super(); }
    public Dimension getPreferredSize() {
    return new Dimension(width,height); }
    // Draw bean
    public synchronized void paint(Graphics g) {
    g.setColor(gaugeColor);
    g.fill3DRect(0,0,width-1,height-1,false); int
    border=3;
    int innerHeight=height-2*border; int
    innerWidth=width-2*border;
    double scale=(double)(currentValue-minValue)/
```

```

    (double)(maxValue-minValue); int
    gaugeValue;
    g.setColor(valueColor); if(orientation==HORIZONTAL){
        gaugeValue=(int)((double)innerWidth*scale);
        g.fillRect(border,border,gaugeValue,innerHeight); }else{

gaugeValue=(int)((double)innerHeight*scale);
    g.fillRect(border,border+(innerHeight-ÂgaugeValue),innerWidth,gaugeValue);
    }
    }
    // Methods for accessing bean properties
    public double getCurrentValue(){ return
    currentValue;
    }
    public void setCurrentValue(double newCurrentValue){
    if(newCurrentValue>=minValue && newCurrentValue<=maxValue)
    currentValue=newCurrentValue;
    }
    public double getMinValue(){ return
    minValue;
    }
    public void setMinValue(double newMinValue){
    if(newMinValue<=currentValue)  minValue=newMinValue;
    }
    public double getMaxValue(){ return
    maxValue;
    }
    public void setMaxValue(double newMaxValue){
    if(newMaxValue >= currentValue)  maxValue=newMaxValue;
    }

```



```
public int getWidth(){ return
width;
}
public void setWidth(int newWidth){
if(newWidth > 0){ width=newWidth;
updateSize();

}
}
public int getHeight(){ return
height;
}
public void setHeight(int newHeight){
if(newHeight > 0){ height=newHeight;
updateSize();
}
}
public Color getGaugeColor(){
return gaugeColor;
}
public void setGaugeColor(Color newGaugeColor){
gaugeColor=newGaugeColor;
} public Color getValueColor(){

return valueColor;
}
public void setValueColor(Color newValueColor){
valueColor=newValueColor;
}
```

```

public boolean isHorizontal(){
if(orientation==HORIZONTAL) return true; else
return false;
}
public void setHorizontal(boolean newOrientation){ if(newOrientation){
if(orientation==VERTICAL) switchDimensions();
}else{
if(orientation==HORIZONTAL) switchDimensions();
orientation=VERTICAL;
}
updateSize();
}
void switchDimensions(){
int temp=width;
width=height; height=temp;
}
void updateSize(){ setSize(width,height);
Container container=getParent();
if(container!=null){ container.invalidate();
container.doLayout();
}
}
}
}

```

To see how the bean works, copy the Gauge.jar file from your ch26 directory to the \bdk\jars directory and then start up the BeanBox using the following commands: copy Gauge.jar \bdk\jars cd \bdk\beanbox run

The BeanBox opens and displays the ToolBox, BeanBox, and PropertySheet windows. You will notice a new bean at the bottom of the ToolBox.

Click the Gauge bean's ToolBox icon and then click in the BeanBox. The bean is displayed as a horizontal 3D box.

The bean's property sheet displays a number of properties that may be changed to alter the bean's appearance. The foreground, background, and font properties are the default properties of visible beans. These properties reflect the `getForeground()`, `setForeground()`, `getBackground()`, `setBackground()`, `getFont()`, and `setFont()` methods of the `Component` class.

The `minValue` and `maxValue` properties identify the minimum and maximum values associated with the gauge. The `currentValue` property identifies the current value of the gauge.

The `width` and `height` properties control the gauge's dimensions. The `horizontal` property takes on boolean values. When set to `True`, the gauge is displayed horizontally. When set to `False`, the gauge is displayed vertically. When the orientation of a gauge is switched, so are its width and height.

The `gaugeColor` and `valueColor` properties identify the color of the gauge's border and the color to be displayed to identify the gauge's current value.

To see how the gauge's properties work, make the following changes:

- Change the `currentValue` property to `.7`.
- Change the `horizontal` property to `False`.
- Change the `height` property to `200`.
- Change the `gaugeColor` property to `green`.
- Change the `valueColor` property to `orange`.

How the Gauge Bean Works

The first thing that you'll notice about the Gauge bean's source code is that it imports `java.io.Serializable`. All bean classes implement `Serializable` or `Externalizable`. These interfaces support bean persistence, allowing beans to be read from and written to permanent storage (for

example, hard disk). When a bean implements `Serializable`, serialization of the bean's data is performed automatically by Java, meaning you don't have to figure out how to write objects to streams or read them back in. When a bean implements `Externalizable`, the bean is responsible for performing all the serialization overhead. `Serializable` is obviously the easiest to implement of the two interfaces. All you have to do is add `Serializable` to your class's implements clause and poof!•serialization is automatically supported. Chapter 40, "Using Object Serialization and JavaSpaces," covers object serialization.

Besides serialization, you won't notice anything bean-specific about the rest of the `Gauge` class. In fact, it looks just like any other custom AWT class. `Gauge` extends `Canvas` so that it can draw to a `Graphics` object. It defines a few constants for use in initializing its field variables. You should note that these field variables correspond to the `Gauge` bean's properties.

The `getPreferredSize()` method is an important method for visible beans to implement. It tells application builder tools how much room is needed to display a bean. All of your visible beans should implement `getPreferredSize()`.

The `paint()` method draws the bean on a `Graphics` object. Visible beans need to implement `paint()` in order to display themselves. The `paint()` method of `Gauge` works by drawing a 3D rectangle using the `gaugeColor` and then drawing an inner rectangle using the `valueColor`. The dimensions of the inner rectangle are calculated based on the value of `currentValue` and the orientation variables.

`Gauge` provides getter and setter methods for each of its properties. These methods adhere to the naming conventions used for bean properties. The `Introspector` class of `java.beans` automatically reports the properties corresponding to these methods to application builder tools, such as the `BeanBox`.

The `switchDimensions()` method is used to switch the values of width and height when the bean's orientation is switched.

The `updateSize()` method is invoked when the bean changes its size. It invokes `setSize()` to inform a layout manager of its new size. It invokes the `invalidate()` method of its container to invalidate the container's layout and `doLayout()` to cause the component to be redisplayed.

The GaugeBeanInfo Class

You may be wondering, "What about all of those other classes and interfaces of java.beans?" For simple beans, you don't really need them. However, we created a GaugeBeanInfo class so that the bean's icon can be displayed.

THE GaugeBeanInfo CLASS.

```
import java.beans.*; import
java.awt.*;
public class GaugeBeanInfo extends SimpleBeanInfo {
// Return icon to be used with bean
public Image getIcon(int size) {
switch(size){ case
ICON_COLOR_16x16:
return loadImage("gauge16c.gif"); case
ICON_COLOR_32x32:
return loadImage("gauge32c.gif"); case
ICON_MONO_16x16:
return loadImage("gauge16m.gif");
case ICON_MONO_32x32: return
loadImage("gauge32c.gif");
}
return null;
}
}
```

The GaugeBeanInfo class extends the SimpleBeanInfo class and implements one method••getIcon(). The getIcon() method is invoked by application builders to obtain an icon for a bean. It uses the constants defined in the BeanInfo interface to select a color or monochrome icon of size 16¥16 or 32¥32 bits.

The Gauge.mf Manifest File

The Gauge.mf manifest file is used to build the Gauge.jar file. It identifies the Gauge.class file as a bean. To create the Gauge.jar file, use the following command: `jar cfm Gauge.jar Gauge.mf Gauge*.class gauge*.gif`

All the files that you need for this example are installed in your ch26 directory. Remember to copy your beans' .jar files from the ch26 directory to the \jdk\jars directory to have them loaded by the BeanBox. The contents of the Gauge.mf file are as follows:

Manifest-Version: 1.0

Name: Gauge.class

Java-Bean: True

Notable Beans

The HotJava HTML Component

One of the most powerful bean sets on the market is the HotJava HTML Component from JavaSoft. This product consists of several beans that can be used to add Web•browsing support to window applications. It parses and renders HTML files that are loaded from the Web and includes the following features:

- HTML 3.2 support
- HTTP 1.1 compatibility
- Frames and tables support
- The ability to use cookies
- Multimedia support
- JAR file support
- Implementation of the FTP, Gopher, SMTP, and SOCKS protocols

A trial version can be downloaded from <http://java.sun.com/products/hotjava/bean/index.html>. Go ahead and download it now so you can work along with the example in this section.

Installing the HotJava HTML Component

The HotJava HTML Component is easy to install. The Microsoft Windows version comes as a .zip file. UnZip the file to a temporary directory and copy the HotJavaBean.jar and TextBean.jar files to your \jdk\jars directory.

Running the HotJava HTML Component in the BeanBox

After installing HotJavaBean.jar and TextBean.jar, run your BeanBox. You will notice the following five beans have been added to the ToolBox:

- TextBean••A text field for entering the URL of a document to be browsed.
- HotJavaDocumentStack••A bean that keeps track of the URLs that have been browsed.
- AuthenticatorBean••An invisible bean that supports user authentication.
- HotJavaBrowserBean••An HTML•rendering bean.
- HotJavaSystemState••An invisible bean that maintains configuration information about the HotJava HTML Component.

Using InfoBus

How InfoBus Works

Normally, all beans that are loaded from the same classloader are visible to each other. Beans can find each other by searching the container•component hierarchy or their bean context. They can then use reflection and design patterns to determine which services are provided by other beans. However, this approach is often cumbersome and prone to error. The software engineers at Lotus Development Corporation and JavaSoft recognized that a standard approach to data

exchange between beans was needed and collaborated to simplify inter-bean communication. The InfoBus is the result of this effort.

The InfoBus is analogous to a PC system bus. Data producers and consumers connect to an InfoBus in the same way that PC cards connect to a PC's system bus. Data producers use the bus to send data items to data consumers. The InfoBus is asynchronous and symmetric. This means that the producer and consumer do not have to synchronize to exchange data, and any member of the bus can send data to any other member of the bus.

The InfoBus operates as follows:

- Beans, components, and other objects join the InfoBus by implementing the InfoBusMember interface, obtaining an InfoBus instance, and using an appropriate method to join the instance.
- Data producers implement the InfoBusDataProducer interface, and data consumers implement the InfoBusDataConsumer interface. These interfaces define methods for handling events required for data exchange.
- Data producers signal that named data items are available on an InfoBus object by invoking the object's fireItemAvailable() method.
- Data consumers get named data items from an InfoBus object by invoking the requestDataItem() method of the InfoBusItemAvailableEvent event received via the InfoBusDataConsumer interface.

This list summarizes the typical usage of the InfoBus. However, the InfoBus is flexible and provides additional usage options, which you'll learn about in the next section. The advantage of InfoBus is that it eliminates the need for inference and discovery on the part of beans. Instead, it provides a standard, structured mechanism for named data items to be exchanged.

The InfoBus API

The InfoBus is a standard extension API consisting of the javax.infobus package, which defines 14 classes and 17 interfaces that support all aspects of InfoBus operation. The InfoBus class is the primary class of the package, supporting bus membership and communication between bus members. The InfoBusMember interface is the interface required of all bus members. The InfoBusMemberSupport class provides a default implementation of this interface.

Data producers implement `InfoBusDataProducer`, and consumers implement `InfoBusDataConsumer`. The `InfoBusDataController` interface is implemented by members that control the operation of the InfoBus. By default, no bus controllers are required.

The `DataItem` interface is used to provide descriptive information about a data item. The data provided by a data item can be accessed through the following InfoBus access interfaces:

- `ImmediateAccess`••Used to access String or other objects.
- `ArrayAccess`••Used to access arrays.
- `DbAccess`••Used to provide access to a database.
- `RowsetAccess`••Used to access the rows of a database.
- `ScrollableRowsetAccess`••Used to access a set of database rows.

The `DataItemView` interface provides a two-dimensional database view. The `RowsetValidate` is used to validate the contents of a row of a database.

The `DefaultPolicy` class provides a default InfoBus security policy implementation. It implements the `InfoBusPolicyHelper` interface, which is required of InfoBus security policies.

The InfoBus supports two event hierarchies. The `InfoBusEvent` class is the base event class used with InfoBus communication. It is extended by `InfoBusItemAvailableEvent`, `InfoBusItemRequestedEvent`, and `InfoBusItemRevokedEvent`. The `InfoBusEventListener` interface is used to handle these events. The `DataItemChangeEvent` class is used to inform bus members about the availability and changes to a data item. It is extended by `DataItemAddedEvent`, `DataItemDeletedEvent`, `DataItemRevokedEvent`, `DataItemValueChangedEvent`, and `RowsetCursorMovedEvent`. The `DataItemChangeListener` interface handles these events.

The `InfoBusPropertyMap` interface is used with InfoBus 1.1 to support the `DataItemChangeEvent`. The `DataItemChangeManager` interface is used to manage multiple `DataItemChangeListeners`.

Glasgow Developments

The JavaBeans improvements resulting from Glasgow can be grouped into the following three functional areas:

- The Extensible Runtime Containment and Services Protocol••A protocol that lets beans find out information about their container and the services that it provides.

- The JavaBeans Activation Framework (JAF)••A framework for mapping data to beans based on the data's types.
- The Drag and Drop Subsystem••The Drag and Drop API

Each of these three areas greatly enhances the capabilities of JavaBeans. The Extensible Runtime Containment and Services Protocol allows beans to be constructed in a hierarchical fashion, with bean containers providing direct services to their bean components. The JAF allows beans to be activated during execution time to process data of a variety of types. The Drag and Drop API allows JavaBeans to provide the same GUI capabilities as other component frameworks.

The Extensible Runtime Containment and Services Protocol

As described in "The Software Component Assembly Model," the overall objective of componentbased software development is to develop software components that can be used to assemble software on a component•by•component basis. In this model, existing components are used to assemble new components, which may be used to develop even more complex components. These components are then assembled into applets or applications via visual programming tools.

The JavaBeans implementation provided with JDK 1.1 allowed beans to be assembled into component hierarchies, with parent beans containing one or more child beans. However, the JDK

JavaBeans implementation did not provide any facilities for child beans to learn about their parent containers or the services they provide. As a result, child beans were not able to interact with their parents (or siblings) or make use of their (family) environment. For example, suppose that you want to use a multimedia bean as a container for part of an application, and you want to add custom bean controls to the multimedia bean. The bean controls have no way of obtaining information about their container or the multimedia services it provides.

The Extensible Runtime Containment and Services Protocol solves the lack of communication between child beans and their parent containers. This protocol adds the following capabilities to JDK 1.1 JavaBeans:

- Specifies the environment, or *context*, in which a bean executes.
- Allows services to be dynamically added to a bean's environment.
- Provides a mechanism for beans to interrogate their environment, discover which services are provided, and make use of those services.

These capabilities are provided through the `java.beans.beancontext` package, which is introduced in "Developing Beans." This package provides classes and interfaces for enabling beans to access their execution environment, referred to as their *bean context*. The `BeanContextChild` and `BeanContext` interfaces implement this concept.

The `BeanContextChild` interface provides methods for getting and setting this context, and for managing context-related event listeners. All context-aware beans must implement this interface.

`BeanContextChild` is extended by the `BeanContext` interface, which provides methods by which beans can access resources and services that are available within their context. Objects that implement `BeanContext` function as containers for other beans (which implement `BeanContextChild`). When a child bean that implements `BeanContextChild` is added to a parent bean container that implements `BeanContext`, the parent invokes the `setBeanContext()` method of its child. When the child wants to access its environment (`BeanContext`), it invokes its `getBeanContext()` method.

A bean can access its environment through its `BeanContext`. A `BeanContext` may or may not expose its services to the beans that it contains. The `BeanContextServicesSupport` interface extends the `BeanContext` interface to provide child beans with access to the services of their `BeanContext`. A bean that provides services to its children must implement this interface. The `getCurrentServiceClasses()`, `hasServices()`, and `getService()` methods are invoked by the child beans to access these services. The `BeanContextServiceProvider` interface is implemented by objects that provide instances of a particular service. The `BeanContextContainer` interface is implemented by `BeanContext` objects that are associated with AWT containers.

The `BeanContextChildSupport` class provides a default implementation of the `BeanContextChild` interface.

The `BeanContextSupport` class extends `BeanContextChildSupport` to provide an implementation of the `BeanContext` interface. This class provides variables and methods for managing the beans that are contained in the context. It defines two inner classes: `BeanContextSupport.BCChild` and `BeanContextSupport.BCIterator`. These classes are used to maintain information on the beans that are contained within a bean context.

The `BeanContextServicesSupport` class extends `BeanContextSupport` and implements the `BeanContextServices` interface. It defines two inner classes:

BeanContextServicesSupport.BCSSChild and BeanContextServicesSupport.

BCSSServiceProvider.

BeanContextServicesSupport.BCSSChild extends BeanContextSupport.BCSChild, and

BeanContextServicesSupport. BCSSServiceProvider is used to access

BeanContextServiceProvider objects.

BeanContext Event Handling

The Extensible Runtime Containment and Services Protocol allows a beans's BeanContext to change during a program's execution. It provides events and event handling interfaces to deal with changes in a bean's BeanContext. The BeanContextEvent class is the superclass of all BeanContextrelated events. It is extended by BeanContextMembershipEvent, BeanContextServiceAvailableEvent, and BeanContextServiceRevokedEvent. The BeanContextMembershipEvent class defines events that occur as the result of changes in a bean's membership in a BeanContext. It is extended by BeanContextAddedEvent and BeanContextRemovedEvent. The BeanContextServiceAvailableEvent reports changes in the availability of services, and BeanContextServiceRevokedEvent reports the revocation of services.

The BeanContextMemberShipListener interface is used to handle the BeanContextMembershipEvent event. The BeanContextServicesListener interface is used to handle the BeanContextServiceAvailableEvent event. The BeanContextServiceRevokedListener interface is used to handle the BeanContextServiceRevokedEvent event.

The JavaBeans Activation Framework

In many applications, software is called upon to process data of arbitrary types. The application is required to determine the type of data that it is to process, determine which operations can be performed on the data, and instantiate software components for performing those operations. For example, consider a Web browser that displays data loaded from a URL. Some URLs reference HTML files, some reference image files, and others may reference files containing scripting data. The Web browser determines the type of data contained in the file using MIME type information provided by Web servers. It then selects external or internal components that display the file's data, launches instances of those components, and feeds the file data to those components for display.

The JavaBeans Activation Framework is used to support this type of data processing by associating beans with the types of data that they support. It provides the following capabilities:

- A mechanism for associating data types with different types of data.
- The capability to determine the operations supported by data of a particular type.
- A mapping of data operations to the beans that support those operations.
- The capability to instantiate beans to support specific data operations.

The JAF API is implemented by the `javax.activation` package, which is a standard API extension.

This package consists of the following classes and interfaces:

- `DataHandler`••This class provides a standard interface to data of different types. It is the entry point to the JAF.
- `DataSource`••An interface that provides encapsulated access to data of different types. It reports the data's type and provides access to the data via input and output streams.
- `FileDataSource`••Extends `DataSource` to provide access to file data.
- `URLDataSource`••Extends `DataSource` to provide access to data that is accessible via a URL.
- `DataContentHandler`••Used by `DataHandler` to convert `DataSource` objects to the objects they represent, and to convert objects to byte streams of a particular MIME type.
- `DataContentHandlerFactory`••A factory for creating `DataContentHandler` objects for specific MIME types.
- `CommandMap`••Provides access to the commands (operations) supported for a particular MIME type, and maps these commands to objects that support those commands.
- `MailcapCommandMap`••Extends `CommandMap` to support mailcap (RFC 1524) files.
- `CommandObject`••An interface implemented by JavaBeans to make them JAF•aware. It allows the beans to respond to commands and access the `DataHandler` associated with the data they are commanded to process.
- `CommandInfo`••Used by `CommandMap` to return the results of commands that have been made of JAF•aware beans.
- `MimeType`••Encapsulates a MIME type.
- `FileTypeMap`••Associated files with MIME types.

- `MimetypesFileTypeMap`••Extends `FileTypeMap` to identify a file's MIME type based on its extension.
- `MimeTypeParameterList`••Encapsulates the parameter list of a MIME type.
- `ActivationDataFlavor`••Extends `java.awt.datatransfer.DataFlavor` to provide better MIME type processing.

Before the JAF can be used, a `CommandMap` must be created that maps MIME types, and operations on those types, to bean classes. This is typically accomplished using the `MailcapCommandMap` class. External mailcap files can be used to set up the mapping, or it can be set up during program initialization. Once the `CommandMap` has been created, the JAF is ready for use.

The JAF is used by creating a `DataSource` for data that is to be processed. This data is typically stored in a file or referenced by a URL. A `DataHandler` is then constructed from the `DataSource`. The `getContentType()` method of `DataHandler` is used to retrieve the MIME type associated with the `DataSource`. This MIME type is used to retrieve a `CommandInfo` array from the `CommandMap`. The `CommandInfo` array presents the list of operations that are supported by the `DataSource`. A selected `CommandInfo` object is passed to the `getBean()` method of `DataHandler` to create an instance of a bean that supports a specific operation on a MIME type. The bean is added to the applet or application's GUI, and the bean's `setCommandContext()` method is invoked to cause the bean to perform the desired operation on the data contained in the `DataSource`.

JAF•compliant beans are required to implement the `CommandObject` interface. This interface consists of the single `setCommandContext()` method.

The `JAFApp` program in the next section illustrates the use of the interfaces and classes described in the previous paragraphs.

ADVANCED JAVA PROGRAMMING

UNIT-III

EJB

EJB stands for **Enterprise Java Beans**. EJB is an essential part of a J2EE platform. J2EE platform has component based architecture to provide multi-tiered, distributed and highly transactional features to enterprise level applications.

EJB provides an architecture to develop and deploy component based enterprise applications considering robustness, high scalability, and high performance. An EJB application can be deployed on any of the application server compliant with the J2EE 1.3 standard specification.

1. EJB architecture

The **EJB** stands for Enterprise Java beans that is a server-based **architecture** that follows the specifications and requirements of the enterprise environment. **EJB** is conceptually based on the Java RMI(Remote Method Invocation) specification. In **EJB**, the beans are run in a container having four-tier **architecture**.

The **EJB** architecture has two main layers, i.e., **Application Server** and **EJB Container**, based on which the EJB architecture exist.

The EJB architecture consists of three main components: enterprise beans (EJBs), the EJB container, and the Java **application** server. EJBs run inside an EJB container, and the EJB container runs inside a Java **application** server

2. EJB requirements

3. DESIGNS AND IMPLEMENTATION:

Designing Enterprise JavaBeans

These sections discuss design options for WebLogic Server Enterprise JavaBeans (EJBs), bean behaviors to consider during the design process, and recommended design patterns.

- Choosing the Right Bean Type
- Persistence Management Alternatives
- Transaction Design and Management Options
- Satisfying Application Requirements with WebLogic Server EJBs

Choosing the Right Bean Type

When you choose the bean type for a processing task, consider the different natures and capabilities of the various bean types.

Bean types vary in terms of the bean's relationship with its client. Some bean types stick with a client throughout a series of processes, serving as a sort of general contractor—acquiring and orchestrating services for the client. There are other bean types that act like subcontractors, they deliver the same single function to multiple client-oriented general contractor beans. A client-oriented bean keeps track of client interactions and other information associated with the client process, throughout a client session—a capability referred to as *maintaining state*. Beans that deliver commodity services to multiple client-oriented beans do not maintain state.

Persistence Management Alternatives

- Persistence management strategy determines how an entity bean's database access is performed.
- Configure the persistence management strategy—either container-managed or bean-managed—for an entity bean in the **persistence-type** element in **ejb-jar.xml**.

Transaction Design and Management Options

A transaction is a unit of work that changes application state—whether on disk, in memory or in a database—that, once started, is completed entirely, or not at all.

Understanding Transaction Demarcation Strategies and Performance

Transactions can be demarcated—started, and ended with a commit or rollback—by the EJB container, by bean code, or by client code.

Demarcating Transactions at the Server Level is Most Efficient

Transactions are costly application resources, especially database transactions, because they reserve a network connection for the duration of the transaction. In a multi-tiered architecture—with database, application server, and Web layers—you optimize performance by reducing the network traffic "round trip." The best approach is to start and stop transactions at the application server level, in the EJB container.

Container-Managed Transactions Are Simpler to Develop and Perform Well

Container-managed transactions (CMTs) are supported by all bean types: session, entity, and message-driven. They provide good performance, and simplify development because the enterprise bean code does not include statements that begin and end the transaction.

Each method in a CMT bean can be associated with a single transaction, but does not have to be. In a container-managed transaction, the EJB container manages the transaction, including start, stop, commit, and rollback. Usually, the container starts a transaction just before a bean method starts, and commits it just before the method exits.

For information about the elements related to transaction management in `ejb-jar.xml` and `weblogic-ejb-jar.xml`, see [Container-Managed Transactions Elements](#).

Satisfying Application Requirements with WebLogic Server EJBs

WebLogic Server offers a variety of value-added features for enterprise beans that you can configure to meet the requirements of your application. They are described in WebLogic Server Value-Added EJB Features.

FEATURES AND DESIGN PATTERN

Availability and reliability

Failover for Clustered EJBs Increases Reliability

Load Balancing Among Clustered EJBs Increases Scalability

Scalability	Stateless Beans Offer Performance and Scalability Advantages
Data Consistency	<p>Use Container-Managed Persistence (CMP) for Productivity and Portability</p> <p>Use Read-Write Beans for Higher Data Consistency.</p> <p>Transaction Isolation: A Performance vs. Data Consistency Choice</p> <p>Keep Bean-Managed Transactions Short</p>
Developer and Administrator Productivity	<p>Use Container-Managed Persistence (CMP) for Productivity and Portability</p> <p>Container-Managed Transactions Are Simpler to Develop and Perform Well</p>
Performance	<p>Choosing bean types and design patterns:</p> <p>Combine Read-Only and Read-Write Beans to Optimize Performance</p> <p>Use Read-Only Beans to Improve Performance If Stale Data Is Tolerable</p> <p>Use Session Facades to Optimize Performance for Remote Entity Beans</p> <p>Avoid the Use of Transfer Objects</p> <p>Stateless Beans Offer Performance and Scalability Advantages</p> <p>Clustering features:</p> <p>Load Balancing Among Clustered EJBs Increases Scalability</p> <p>Pooling and caching:</p> <p>Performance-Enhancing Features for WebLogic Server EJBs</p> <p>Transaction management:</p> <p>Container-Managed Transactions Are Simpler to Develop and Perform Well</p> <p>Demarcating Transactions at the Server Level is Most Efficient</p> <p>Transaction Isolation: A Performance vs. Data Consistency Choice</p> <p>Costly Option: Distributing Transactions Across Databases</p> <p><u>Keep Bean-Managed Transactions Short</u></p>

Implementing Enterprise JavaBeans

The sections that follow describe the EJB implementation process, and provide guidance for how to get an EJB up and running in WebLogic Server.

It is assumed that you understand WebLogic Server's value-added EJB features, have selected a design pattern for your application, and have made key design decisions.

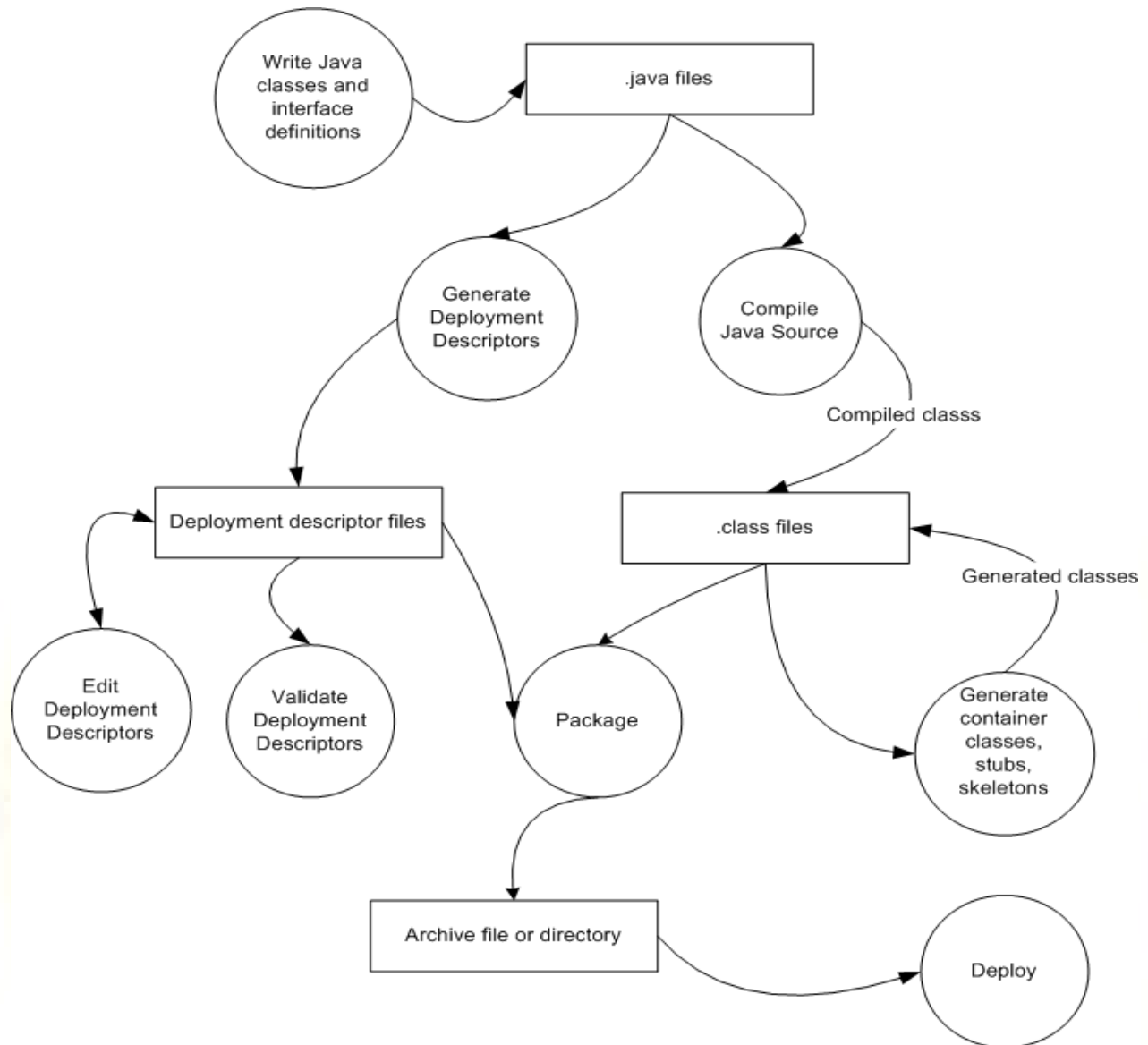
For a review of WebLogic Server EJB features, see [WebLogic Server Value-Added EJB Features](#).

For discussion of design options for EJBs, factors to consider during the design process, and recommended design patterns see [Designing Enterprise JavaBeans](#).

- [Overview of the EJB Development Process](#)
- [Create a Source Directory](#)
- [Create EJB Classes and Interfaces](#)
- [Programming the EJB Timer Service](#)
- [Declare Web Service References](#)
- [Compile Java Source](#)
- [Generate Deployment Descriptors](#)
- [Edit Deployment Descriptors](#)
- [Generate EJB Wrapper Classes, and Stub and Skeleton Files](#)
- [Package](#)
- [Deploy](#)
- [Solving Problems During Development](#)
- [WebLogic Server Tools for Developing EJBs](#)

Overview of the EJB Development Process

This section is a brief overview of the EJB development process. It describes the key implementation tasks and associated results.



Create EJB Classes and Interfaces

The classes required depend on the type of EJB you are developing, as described in EJB Components..

Oracle offers productivity tools for developing class and interface files. The EJBGen command line utility automates the process of creating class and interface files, and also generates

deployment descriptor files for the EJB. For more information and instructions for using these tools see EJBGen Reference.

The sections that follow provide tips and guidelines for using WebLogic Server-specific EJB features.

Programming the EJB Timer Service

WebLogic Server supports the EJB timer service defined in the EJB 2.1 Specification and EJB

Specification. The EJB timer service is an EJB-container provided service that allows you to create timers that schedule callbacks to occur when a timer object expires. Timer objects can be created for entity beans, message-driven beans, and stateless session beans. Timer objects expire at a specified time, after an elapsed period of time, or at specified intervals. For instance, you can use the timer service to send out notification when an EJB remains in a certain state for an elapsed period of time.

The WebLogic EJB timer service is intended to be used as a coarse-grained timer service. Rather than having a large number of timer objects performing the same task on a unique set of data, Oracle recommends using a small number of timers that perform bulk tasks on the data. For example, assume you have an EJB that represents an employee's expense report. Each expense report must be approved by a manager before it can be processed. You could use one EJB timer to periodically inspect all pending expense reports and send an email to the corresponding manager to remind them to either approve or reject the reports that are waiting for their approval

Declare Web Service References

This release of WebLogic Server complies with the EJB 2.1 requirements related to declaring and accessing external Web Services. Web Service references, declared in an EJB's deployment descriptor, maps a logical name for a Web Service to an actual Web Service interface, which

allows you to refer to the Web Service using a logical name. The bean code then performs a JNDI lookup using the Web Service reference name.

For more information, see *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

Compile Java Source

For a list of tools that support the compilation process, see [Table 4-1](#).

For information on the compilation process, see *Developing Applications for Oracle WebLogic Server*.

Generate Deployment Descriptors

If you annotate your Bean class file with JDK 1.5 annotations, you can use EJBGen to generate the Remote and Home classes and the deployment descriptor files for an EJB application.

Oracle recommends that you use EJBGen to generate deployment descriptors. For more information, see [Appendix E, "EJBGen Reference."](#)

Edit Deployment Descriptors

Elements in **`ejb-jar.xml`**, **`weblogic-ejb-jar.xml`**, and for container-managed persistence entity beans, **`weblogic-cmp-jar.xml`**, control the run-time characteristics of your application.

If you need to modify a descriptor element, you can edit the descriptor file with any plain text editor. However, to avoid introducing errors, use a tool designed for XML editing. Descriptor elements that you can edit with the WebLogic Server Administration Console are listed in [Table 4-1](#).

The following sections are a quick reference to WebLogic Server-specific deployment elements. Each section contains the elements related to a type of feature or behavior. The table in each section defines relevant elements in terms of the behavior it controls, the bean type it relates to (if bean type-specific), the parent element in **`weblogic-ejb-jar.xml`** that contains the element,

and the behavior you can expect if you do not explicitly specify the element in **weblogic-ejb-jar.xml**.

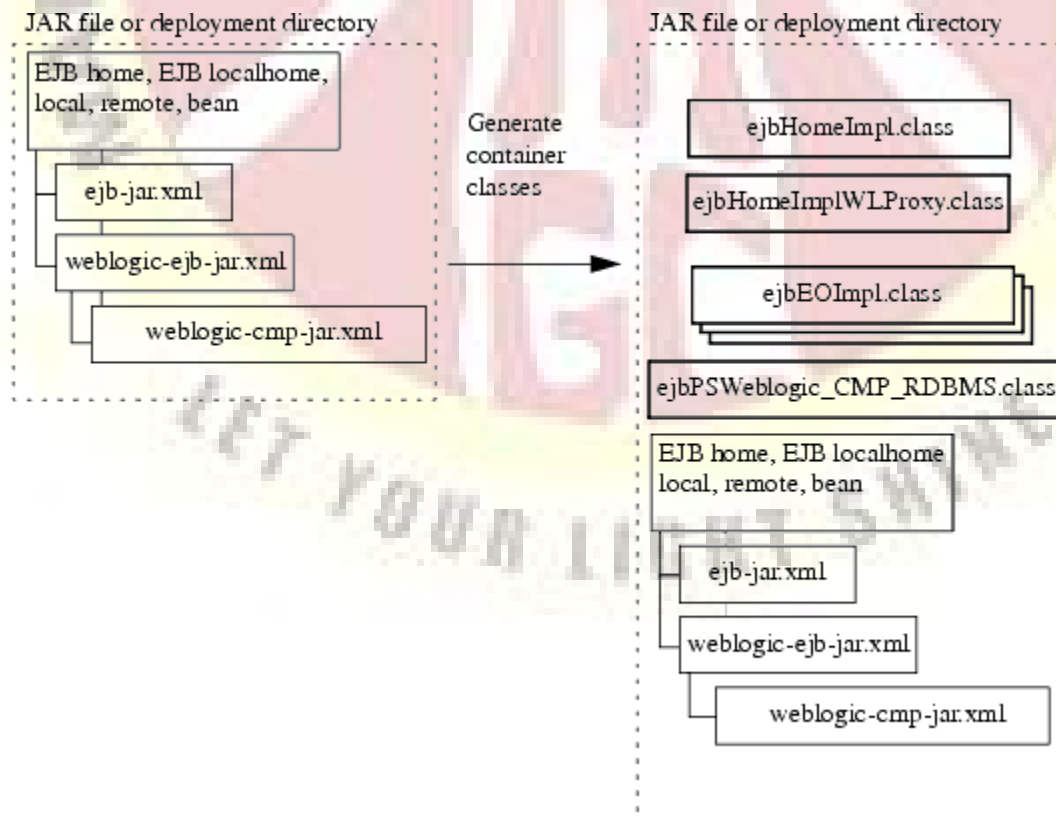
Generate EJB Wrapper Classes, and Stub and Skeleton Files

Container classes include the internal representation of the EJB that WebLogic Server uses and the implementation of the external interfaces (home, local, and/or remote) that clients use. You can use Oracle Workshop for WebLogic Platform or **appc** to generate container classes.

Container classes are generated in according to the descriptor elements in **weblogic-ejb-jar.xml**. For example, if you specify clustering elements, **appc** creates cluster-aware classes that will be used for deployment. You can use **appc** directly from the command line by supplying the required options and arguments. See [appc](#) for more information.

The following figure shows the container classes added to the deployment unit when the EAR or JAR file is generated.

Figure 4-2 Generating EJB Container Classes



appc and Generated Class Name Collisions

Although infrequent, when you generate classes with **appc**, you may encounter a generated class name collision which could result in a **ClassCastException** and other undesirable behavior. This is because the names of the generated classes are based on three keys: the bean class name, the bean class package, and the **ejb-name** for the bean. This problem occurs when you use an EAR file that contains multiple JAR files and at least two of the JAR files contain an EJB with both the same bean class, package, or classname, and both of those EJBs have the same **ejb-name** in their respective JAR files. If you experience this problem, change the **ejb-name** of one of the beans to make it unique.

Because the **ejb-name** is one of the keys on which the file name is based and the **ejb-name** must be unique within a JAR file, this problem never occurs with two EJBs in the same JAR file.

Also, because each EAR file has its own classloader, this problem never occurs with two EJBs in different EAR files.

Package

Oracle recommends that you package EJBs as part of an enterprise application. For more information, see "[Deploying and Packaging from a Split Development Directory](#)" in *Developing Applications for Oracle WebLogic Server*.

Packaging Considerations for EJBs with Clients in Other Applications

WebLogic Server supports the use of **ejb-client.jar** files for packaging the EJB classes that a programmatic client in a different application requires to access the EJB.

Specify the name of the client JAR in the **ejb-client-jar** element of the bean's **ejb-jar.xml** file.

When you run the **appc** compiler, a JAR file with the classes required to access the EJB is generated.

Make the client JAR available to the remote client. For Web applications, put the **ejb-client.jar** in the **/lib** directory. For non-Web clients, include **ejb-client.jar** in the client's classpath.

Deploy

Deploying an EJB enables WebLogic Server to serve the components of an EJB to clients. You can deploy an EJB using one of several procedures, depending on your environment and whether or not your EJB is in production.

For general instructions on deploying WebLogic Server applications and modules, including EJBs, see *Deploying Applications to Oracle WebLogic Server*. For EJB-specific deployment issues and procedures, see [Chapter 8, "Deployment Guidelines for Enterprise JavaBeans"](#) in this book — *Programming WebLogic Enterprise JavaBeans for Oracle WebLogic Server*.

Solving Problems During Development

The following sections describe WebLogic Server features that are useful for checking out and debugging deployed EJBs.

Adding Line Numbers to Class Files

If you compile your EJBs with **appc**, you can use the **appc -lineNumbers** command option to add line numbers to generated class files to aid in debugging. For information, see [Appendix D, "appc Reference."](#)

Monitoring Data

WebLogic Server collects a variety of data about the run-time operation of a deployed EJB. This data, which you can view in the Deployments node of the Administration Console, can be useful in determining if an EJB has completed desired processing steps. To access EJB run-time statistics, expand the Deployment node in the Administration Console, navigate to the JAR EAR that contains the bean, and select the Monitoring tab.

For information about the data available, see these pages in *Oracle WebLogic Server Administration Console Help*:

- ["Deployments-->EJB --> Monitoring --> Stateful Session EJBs"](#)
- ["Deployments-->EJB-->Monitoring-->Stateless EJBs"](#)
- ["Deployments-->EJB --> Monitoring--> Message-Driven EJBs"](#)
- ["Deployments-->EJB --> Monitoring --> Entity EJBs"](#)

Creating Debug Messages

For instructions on how to create messages in your application to help you troubleshoot and solve bugs and problems, see *Configuring Log Files and Filtering Log Messages for Oracle WebLogic Server*.

WebLogic Server Tools for Developing EJBs

This section describes Oracle tools that support the EJB development process. For a comparison of the features available in each tool

Oracle JDeveloper

Oracle JDeveloper is a full-featured Java IDE that can be used for end-to-end development of EJBs. For more information, see the Oracle JDeveloper online help. For information about installing JDeveloper, see *Oracle Fusion Middleware Installation Guide for Oracle JDeveloper*.

Oracle Enterprise Pack for Eclipse

Oracle Enterprise Eclipse (OEPE) provides a collection of plug-ins to the Eclipse IDE platform that facilitate development of WebLogic Web services. For more information, see the Eclipse IDE platform online help.

Administration Console

In the Administration Console, you can view, modify, and persist to the descriptor file within the EJB a number of deployment descriptor elements. Descriptors are modified in the Administration Server copy of the EJB as well as in any deployed copies of the EJB (after deployment). When you modify descriptors, changes are made to your (the user's) original copy of the EJB (prior to deployment).

4 . EJB SESSION BEAN

A session bean is an EJB 3.0 or EJB 2.1 enterprise bean component created by a client for the duration of a single client/server session. A session bean performs operations for the client.

Although a session bean can be transactional, it is not recoverable should a system failure occur.

Session bean objects are either stateless (see ["What is a Stateless Session Bean?"](#)) or stateful: maintaining conversational state across method calls and transactions (see ["What is a Stateful](#)

Session Bean?"). If a session bean maintains state, then OC4J manages this state if the object must be removed from memory ("When Does Stateful Session Bean Passivation Occur?"). However, the session bean object itself must manage its own persistent data.

From a client's perspective, a session bean is a nonpersistent object that implements some business logic running on the application server. For example, in an on-line store application, you can use a session bean to implement a ShoppingCartBean that provides a Cart interface that the client uses to invoke such methods as purchaseItem and checkout.

Each client is allocated its own session object. A client does not directly access instances of the session bean's class: a client accesses a session object through the session bean's home ("Implementing the Home Interfaces") and component ("Implementing the Component Interfaces") interfaces. The client of a session bean may be a local client, a remote client, or a Web service client (stateless session bean only), depending on the interface provided by the bean and used by the client.

5. EJB ENTITY BEANS

An entity bean is an EJB 2.1 enterprise bean component that manages persistent data, performs complex business logic, potentially uses several dependent Java objects, and can be uniquely identified by a primary key.

Entity beans persist business data using one of the two following methods:

Automatically by the container using an entity bean with container-managed persistence

Programmatically through methods implemented in an entity bean with bean-managed persistence

For information on choosing between container-managed persistence and container-managed persistence architectures

Entity beans are persistent because their data is stored persistently in some form of data storage, such as a database: entity beans survive a server failure, failover, or a network failure. When an entity bean is reinstantiated, the state of the previous instance is automatically restored. OC4J manages this state if the entity bean must be removed from memory

A entity bean models a business entity or multiple actions within a single business process. Entity beans are often used to facilitate business services that involve data and computations on that data. For example, you might implement an entity bean to retrieve and perform computation on items within a purchase order. Your entity bean can manage multiple, dependent, persistent objects in performing its tasks.

A common design pattern pairs entity beans with a session bean that acts as the client interface. The entity bean functions as a coarse-grained object that encapsulates functionality and represents persistent data and relationships to dependent (typically, fine-grained) objects. Thus, you decouple the client from the data so that if the data changes, the client is not affected. For efficiency, the session bean can be collocated with entity beans and can coordinate

between multiple entity beans through their local interfaces. This is known as a session facade design.

6. EJB clients

- clients interacts with Enterprise Beans instances through proxy objects (generated by tools provided by container provider). Commonly, the communication is based on IIOP protocol and proxy objects are regular CORBA stubs
- the presence of the container wrapping the bean is completely hidden to the client
- the passivation and activation of bean instances is completely transparent to the client
- the reference to the bean is acquired from the factory (called "home interface") which is specific for each bean type. The factory is capable for creating new bean instances and finding the existing ones (based on some searching criteria). The reference to the factory itself is obtained using the Java Naming and Directory (JNDI) interface
- due to selection IIOP as the communication protocol, the bean may also integrate with RMI/IIOP application or even the client written in any programming language supported by CORBA

7. DEPLOYMENT TIPS, TRICKS, AND TRAPS, FOR BUILDING DISTRIBUTED AND OTHER SYSTEM

8. Implementation and future directions of EJB

9. variables in perl

Variables are the reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or strings in these variables.

We have learnt that Perl has the following three basic data types –

- Scalars
- Arrays
- Hashes

Accordingly, we are going to use three types of variables in Perl. A **scalar** variable will precede by a dollar sign (\$) and it can store either a number, a string, or a reference. An **array** variable will precede by sign @ and it will store ordered lists of scalars. Finally, the **Hash** variable will precede by sign % and will be used to store sets of key/value pairs.

Perl maintains every variable type in a separate namespace. So you can, without fear of conflict, use the same name for a scalar variable, an array, or a hash. This means that \$foo and @foo are two different variables.

10. Perl control structures and operators

Control Structures

Perl is an *iterative language* in which control flows from the first statement in the program to the last statement unless something interrupts. Some of the things that can interrupt this linear flow are conditional branches and loop structures. Perl offers approximately a dozen such constructs, which are described below. The basic form will be shown for each followed by a partial example.

statement block

Statement blocks provide a mechanism for grouping statements that are to be executed as a result some expression being evaluated. They are used in all of the control structures

discussed below. Statement blocks are designated by pairs of curly braces.

Form: BLOCK

Example:

```
{  
  stmt_1;  
  stmt_2;  
  stmt_3;  
}
```

if statement

Form: if (EXPR) BLOCK

Example:

```
if (expression) {  
  true_stmt_1;  
  true_stmt_2;  
  true_stmt_3;  
}
```

if/else statement

Form: if (EXPR) BLOCK else BLOCK

Example:

```
if (expression) {  
  true_stmt_1;  
  true_stmt_2;  
  true_stmt_3;  
}
```

```

} else {
    false_stmt_1;
    false_stmt_2;
    false_stmt_3;
}

```

if/elseif/else statement

Form: if (EXPR) BLOCK elseif (EXPR) BLOCK . . . else BLOCK

Example:

```

if (expression_A) {
    A_true_stmt_1;
    A_true_stmt_2;
    A_true_stmt_3;
} elseif (expression_B) {
    B_true_stmt_1;
    B_true_stmt_2;
    B_true_stmt_3;
} else {
    false_stmt_1;
    false_stmt_2;
    false_stmt_3;
}

```

while statement

Form: LABEL: while (EXPR) BLOCK

The LABEL in this and the following control structures is optional. In addition to description, it also provides function in the quasi-goto statements: last, next, and redo.

Perl conventional practice calls for labels to be expressed in uppercase to avoid confusion with variables or key words.

Example:

```
ALABEL: while (expression) {
    stmt_1;
    stmt_2;
    stmt_3;
}
```

until statement

Form: LABEL: until (EXPR) BLOCK

Example:

```
ALABEL: until (expression) { # while not
    stmt_1;
    stmt_2;
    stmt_3;
}
```

for statement

Form: LABEL: for (EXPR; EXPR; EXPR) BLOCK

Example:

```
ALABEL: for (initial exp; test exp; increment exp) { # e.g., ($i=1; $i<5; $i++)
    stmt_1;
    stmt_2;
    stmt_3;
}
```

foreach statement

Form: LABEL: foreach VAR (EXPR) BLOCK

Example:

```
ALABEL: foreach $i (@aList) {
    stmt_1;
    stmt_2;
    stmt_3;
}
```

last operator

The last operator, as well as the next and redo operators that follow, apply only to loop control structures. They cause execution to jump from where they occur to some other position, defined with respect to the block structure of the encompassing control structure. Thus, they function as limited forms of *goto* statements.

Last causes control to jump from where it occurs to the first statement following the enclosing block.

Example:

```
ALABEL: while (expression) {
    stmt_1;
    stmt_2;
    last;
    stmt_3;
}
```

last jumps to here

If last occurs within nested control structures, the jump can be made to the end of an outer loop by adding a label to that loop and specifying the label in the last statement.

Example:

```

ALABEL: while (expression) {
    stmt_1;
    stmt_2;
    BLABEL: while (expression) {
        stmt_a;
        stmt_b;
        last ALABEL;
        stmt_c;
    }
    stmt_3;
}
# last jumps to here

```

next operator

The next operator is similar to last except that execution jumps to the end of the block, but remains *inside* the block, rather than exiting the block. Thus, iteration continues normally.

Example:

```

ALABEL: while (expression) {
    stmt_1;
    stmt_2;
    next;
    stmt_3;
# next jumps to here
}

```

As with last, next can be used with a label to jump to an outer designated loop.

redo operator

The redo operator is similar to next except that execution jumps to the top of the block

without re-evaluating the control expression.

Example:

```
ALABEL: while (expression) {
# redo jumps to here
    stmt_1;
    stmt_2;
    redo;
    stmt_3;
}
```

As with last, next can be used with a label to jump to an outer designated loop.

Table of Contents

- 1 Numeric operators
 - 1.1 Arithmetic operators
 - 1.2 Bitwise Operators
 - 1.3 Comparison operators for numbers
- 2 String operators
 - 2.1 String comparison operators
 - 2.2 String concatenation operators
 - 2.3 The chomp() operator
- 3 Logical operators

Numeric operators

Perl provides numeric operators to help you operate on [numbers](#) including arithmetic, Boolean and bitwise operations. Let's examine the different kinds of operators in more detail.

Arithmetic operators

Perl arithmetic operators deal with basic math such as adding, subtracting, multiplying, dividing, etc. To add (+) or subtract (-) numbers, you would do something as follows:

```
#!/usr/bin/perl
use warnings;
use strict;

print 10 + 20, "\n"; # 20
print 20 - 10, "\n"; # 10
```

To multiply or divide numbers, you use divide (/) and multiply (*) operators as follows:

```
#!/usr/bin/perl
use warnings;
use strict;

print 10 * 20, "\n"; # 200

print 20 / 10, "\n"; # 2
```

When you combine adding, subtracting, multiplying, and dividing operators together, Perl will perform the calculation in an order, which is known as operator precedence.

The multiply and divide operators have higher precedence than add and subtract operators, therefore, Perl performs multiplying and dividing before adding and subtracting. See the following example:

```
print 10 + 20/2 - 5 * 2, "\n"; # 10
```

Perl performs $20/2$ and $5*2$ first, therefore you will get $10 + 10 - 10 = 10$.

You can use brackets () to force Perl to perform calculation based on precedence you want as shown in the following example:

```
print (((10 + 20)/2 - 5) * 2); # 20;
```

To raise one number to the power of another number, you use exponentiation operator (**) e.g., $2**3 = 2 * 2 * 2$. The following example demonstrates the exponentiation operators:

```
#!/usr/bin/perl
use warnings;
```

```
use strict;
```

```
print 2**3, "\n"; # = 2 * 2 * 2 = 8.
```

```
print 3**4, "\n"; # = 3 * 3 * 3 * 3 = 81.
```

To get the remainder of the division of one number by another, you use modulo operator (%).

It is handy to use the modulo operator (%) to check if a number is odd or even by dividing it by 2 to get the remainder. If the remainder is zero, the number is even, otherwise, the number is odd.

See the following example:

```
#!/usr/bin/perl
```

```
use warnings;
```

```
use strict;
```

```
print 4 % 2, "\n"; # 0 even
```

```
print 5 % 2, "\n"; # 1 odd
```

Bitwise Operators

Bitwise operators allow you to operate on numbers one bit at a time. Think a number as a series of bits e.g., 125 can be represented in binary form as 1111101. Perl provides all basic bitwise operators including and (&), or (|), exclusive or (^), not (~) operators, shift right (>>) and shift left (<<) operators.

The bitwise operators perform from right to left. In other words, bitwise operators perform from rightmost bit to the left most bit.

The following example demonstrates all bitwise operators:

```
#!/usr/bin/perl
```

```
use warnings;
```

```
use strict;
```

```
my $a = 0b0101; # 5
```

```
my $b = 0b0011; # 3
```


Equality	Operators
Equal	==
Not Equal	!=
Comparison	<=>
Less than	<
Greater than	>
Less than or equal	<=
Greater than or equal	>=

All the operators in the table above are obvious except the number comparison operator `<=>` which is also known as spaceship operator.

The number comparison operator is often used in sorting numbers. See the code below:

```
$a <=> $b
```

The number operator returns:

- 1 if \$a is greater than \$b
- 0 if \$a and \$b are equal
- -1 if \$a is lower than \$b

Take a look at the following example:

```
#!/usr/bin/perl
```

```
use warnings;
```

```
use strict;
```

```
my $a = 10;
```

```
my $b = 20;
```



```
print $a <=> $b, "\n";
```

```
$b = 10;
```

```
print $a <=> $b, "\n";
```

```
$b = 5;
```

```
print $a <=> $b, "\n";
```

String operators

String comparison operators

Perl provides the corresponding comparison operators for strings. Let's take a look at the table below:

Equality	Operators
Equal	eq
Not Equal	ne
Comparison	cmp
Less than	lt
Greater than	gt
Less than or equal	le
Greater than or equal	ge

String concatenation operators

Perl provides the concatenation (.) and repetition (x) operators that allow you to manipulate strings. Let's take a look at the concatenation operator (.) first:

```
print "This is" . " concatenation operator" . "\n";
```

The concatenation operator (.) combines two strings together.

A string can be repeated with the repetition (`x`) operator:

```
print "a message " x 4, "\n";
```

The `chomp()` operator

The `chomp()` operator (or [function](#)) removes the last character in a string and returns a number of characters that was removed. The `chomp()` operator is very useful when dealing with user's input, because it helps you remove the new line character `\n` from the string that user entered.

```
#!/usr/bin/perl
```

```
use warnings;
```

```
use strict;
```

```
my $s;
```

```
chomp($s = <STDIN>);
```

```
print $s;
```

The `<STDIN>` is used to get input from users.

Logical operators

Logical operators are often used in control statements such as [if](#), [while](#), [given](#), etc., to control the flow of the program. The following are logical operators in Perl:

- `$a && $b` performs logical AND of two variables or expressions. The logical `&&` operator checks if both variables or expressions are true.
- `$a || $b` performs logical OR of two variables or expressions. The logical `||` operator checks either a variable or expression is true.
- `!$a` performs logical NOT of the variable or expression. The logical `!` operator inverts the value of the followed variable or expression. In the other words, it converts true to false or false to true.

You will learn how to use logical operators in the conditional statements such as [if](#), [while](#) and [given](#).

In this tutorial, you've learned some basic Perl operators. These operators are very important so make sure that you get familiar with them.

11. Functions and scope

FUNCTIONS

SCOPE/ BENEFITS

The important benefits of EJB –

- Simplified development of large-scale enterprise level application.
- Application Server/EJB container provides most of the system level services like transaction handling, logging, load balancing, persistence mechanism, exception handling, and so on. Developer has to focus only on business logic of the application.
- EJB container manages life cycle of EJB instances, thus developer needs not to worry about when to create/delete EJB objects.

ADVANCED JAVA PROGRAMMING

UNIT-IV

RMI

RMI Overview

RMI stands for **Remote Method Invocation**. It is a mechanism that allows an object residing in one system (JVM) to access/invoke an object running on another JVM.

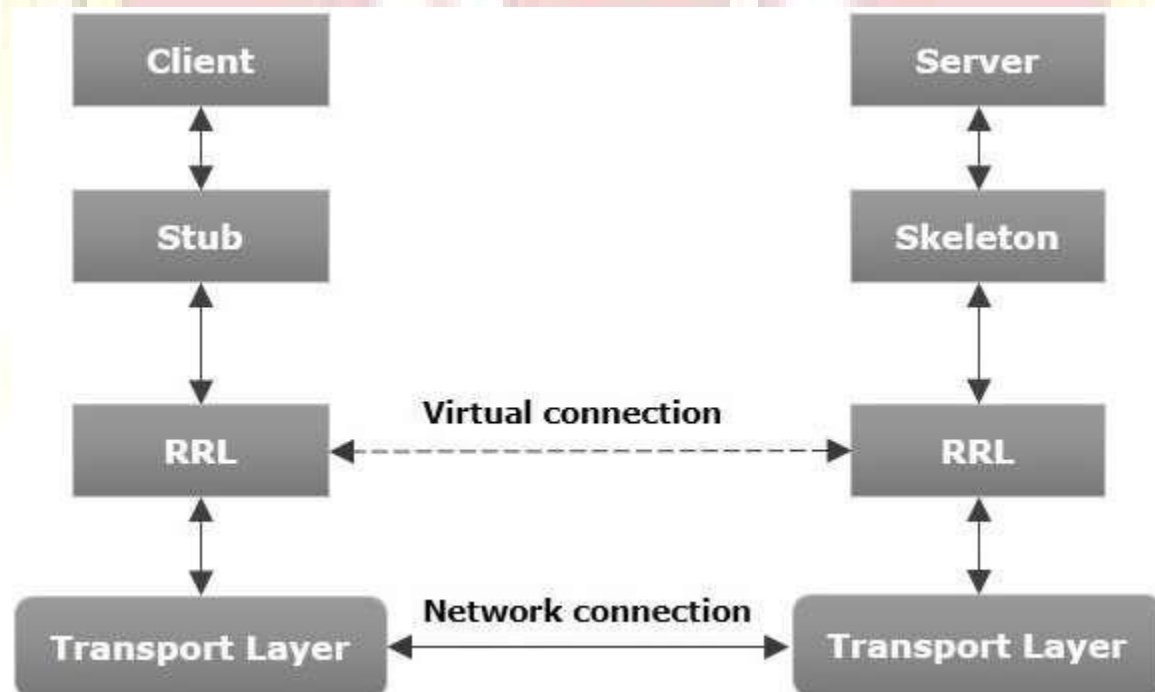
RMI is used to build distributed applications; it provides remote communication between Java programs. It is provided in the package **java.rmi**.

Architecture of an RMI Application

In an RMI application, we write two programs, a **server program** (resides on the server) and a **client program** (resides on the client).

- Inside the server program, a remote object is created and reference of that object is made available for the client (using the registry).
- The client program requests the remote objects on the server and tries to invoke its methods.

The following diagram shows the architecture of an RMI application.



Let us now discuss the components of this architecture.

- **Transport Layer** – This layer connects the client and the server. It manages the existing connection and also sets up new connections.

- **Stub** – A stub is a representation (proxy) of the remote object at client. It resides in the client system; it acts as a gateway for the client program.
- **Skeleton** – This is the object which resides on the server side. **stub** communicates with this skeleton to pass request to the remote object.
- **RRL(Remote Reference Layer)** – It is the layer which manages the references made by the client to the remote object.

Working of an RMI Application

The following points summarize how an RMI application works –

- When the client makes a call to the remote object, it is received by the stub which eventually passes this request to the RRL.
- When the client-side RRL receives the request, it invokes a method called **invoke()** of the object **remoteRef**. It passes the request to the RRL on the server side.
- The RRL on the server side passes the request to the Skeleton (proxy on the server) which finally invokes the required object on the server.
- The result is passed all the way back to the client.

Marshalling and Unmarshalling

Whenever a client invokes a method that accepts parameters on a remote object, the parameters are bundled into a message before being sent over the network. These parameters may be of primitive type or objects. In case of primitive type, the parameters are put together and a header is attached to it. In case the parameters are objects, then they are serialized. This process is known as **marshalling**.

At the server side, the packed parameters are unbundled and then the required method is invoked. This process is known as **unmarshalling**.

RMI Registry

RMI registry is a namespace on which all server objects are placed. Each time the server creates an object, it registers this object with the RMI registry (using **bind()** or **reBind()** methods). These are registered using a unique name known as **bind name**.

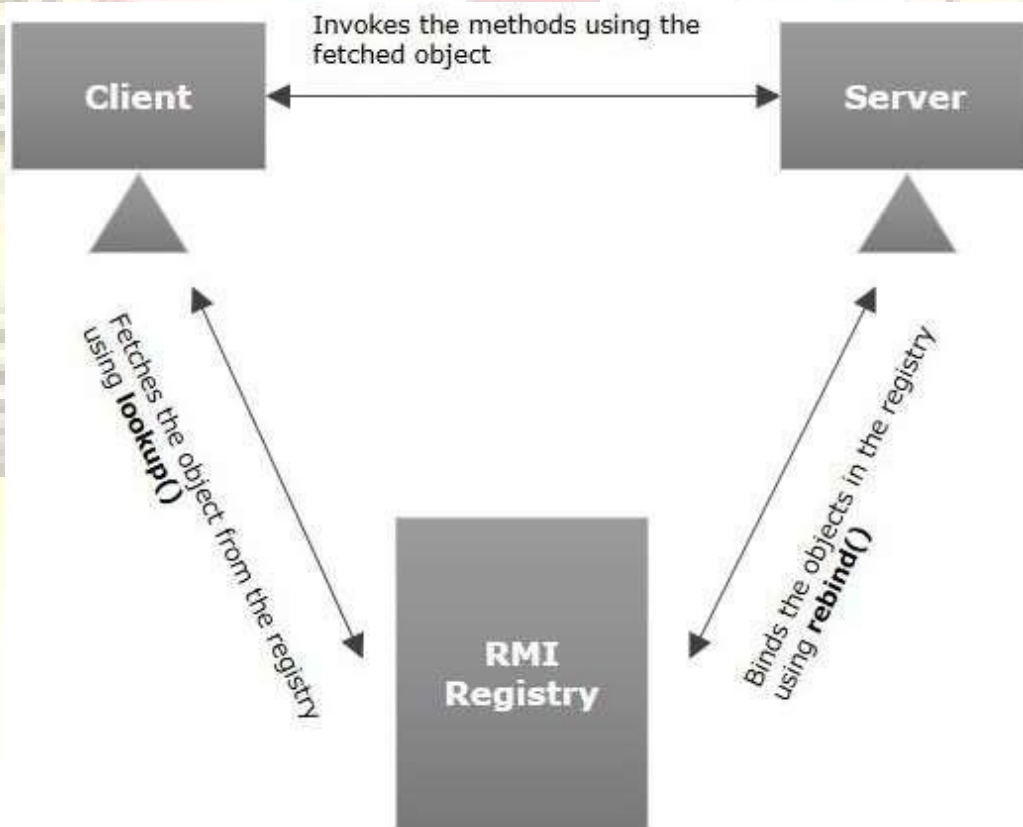
To invoke a remote object, the client needs a reference of that object. At that time, the client fetches the object from the registry using its bind name (using **lookup()** method).

The following illustration explains the entire process –

Goals of RMI

Following are the goals of RMI –

- To minimize the complexity of the application.
- To preserve type safety.
- Distributed garbage collection.
- Minimize the difference between working with local and remote objects. Pr



Developing applications by RMI

To write an RMI Java application, you would have to follow the steps given below –

- Define the remote interface
- Develop the implementation class (remote object)
- Develop the server program
- Develop the client program
- Compile the application
- Execute the application

Defining the Remote Interface

A remote interface provides the description of all the methods of a particular remote object. The client communicates with this remote interface.

To create a remote interface –

- Create an interface that extends the predefined interface **Remote** which belongs to the package.
- Declare all the business methods that can be invoked by the client in this interface.
- Since there is a chance of network issues during remote calls, an exception named **RemoteException** may occur; throw it.

Following is an example of a remote interface. Here we have defined an interface with the name **Hello** and it has a method called **printMsg()**.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

// Creating Remote interface for our application
public interface Hello extends Remote {
    void printMsg() throws RemoteException;
}
```

Developing the Implementation Class (Remote Object)

We need to implement the remote interface created in the earlier step. (We can write an implementation class separately or we can directly make the server program implement this interface.)

To develop an implementation class –

- Implement the interface created in the previous step.
- Provide implementation to all the abstract methods of the remote interface.

Following is an implementation class. Here, we have created a class named **ImplExample** and implemented the interface **Hello** created in the previous step and provided **body** for this method which prints a message.

```
// Implementing the remote interface
public class ImplExample implements Hello {

    // Implementing the interface method
    public void printMsg() {
        System.out.println("This is an example RMI program");
    }
}
```

```

}
}

```

Developing the Server Program

An RMI server program should implement the remote interface or extend the implementation class. Here, we should create a remote object and bind it to the **RMIregistry**.

To develop a server program –

- Create a client class from where you want invoke the remote object.
- **Create a remote object** by instantiating the implementation class as shown below.
- Export the remote object using the method **exportObject()** of the class named **UnicastRemoteObject** which belongs to the package **java.rmi.server**.
- Get the RMI registry using the **getRegistry()** method of the **LocateRegistry** class which belongs to the package **java.rmi.registry**.
- Bind the remote object created to the registry using the **bind()** method of the class named **Registry**. To this method, pass a string representing the bind name and the object exported, as parameters.

Following is an example of an RMI server program.

```

import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class Server extends ImplExample {
    public Server() {}
    public static void main(String args[]) {
        try {
            // Instantiating the implementation class
            ImplExample obj = new ImplExample();

```

```

// Exporting the object of implementation class
// (here we are exporting the remote object to the stub)
Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);

// Binding the remote object (stub) in the registry
Registry registry = LocateRegistry.getRegistry();

registry.bind("Hello", stub);
System.err.println("Server ready");
} catch (Exception e) {
    System.err.println("Server exception: " + e.toString());
    e.printStackTrace();
}
}

```

Developing the Client Program

Write a client program in it, fetch the remote object and invoke the required method using this object.

To develop a client program –

- Create a client class from where your intended to invoke the remote object.
- Get the RMI registry using the **getRegistry()** method of the **LocateRegistry** class which belongs to the package **java.rmi.registry**.
- Fetch the object from the registry using the method **lookup()** of the class **Registry** which belongs to the package **java.rmi.registry**.

To this method, you need to pass a string value representing the bind name as a parameter. This will return you the remote object.

- The lookup() returns an object of type remote, down cast it to the type Hello.
- Finally invoke the required method using the obtained remote object.

Following is an example of an RMI client program.

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {
    private Client() {}
    public static void main(String[] args) {
        try {
            // Getting the registry
            Registry registry = LocateRegistry.getRegistry(null);

            // Looking up the registry for the remote object
            Hello stub = (Hello) registry.lookup("Hello");

            // Calling the remote method using the obtained object
            stub.printMsg();

            // System.out.println("Remote method invoked");
        } catch (Exception e) {
            System.err.println("Client exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

Compiling the Application

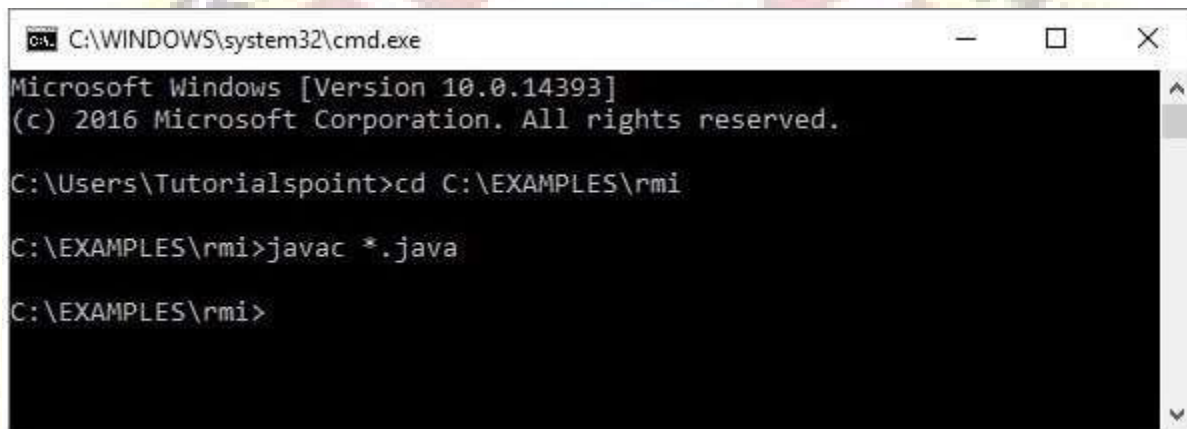
To compile the application –

- Compile the Remote interface.
- Compile the implementation class.
- Compile the server program.
- Compile the client program.

Or,

Open the folder where you have stored all the programs and compile all the Java files as shown below.

Javac *.java



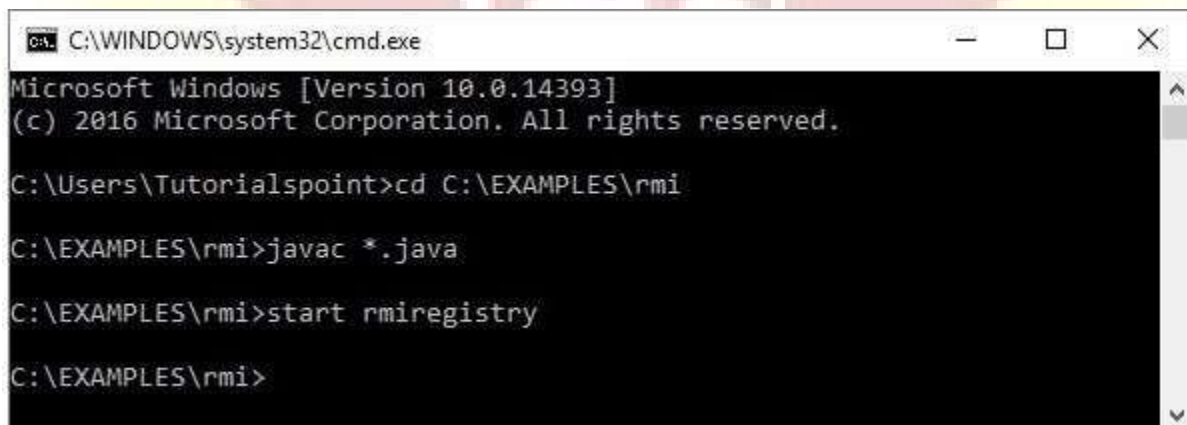
```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Tutorialspoint>cd C:\EXAMPLES\rmi
C:\EXAMPLES\rmi>javac *.java
C:\EXAMPLES\rmi>
```

Executing the Application

Step 1 – Start the **rmi** registry using the following command.

start rmiregistry



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Tutorialspoint>cd C:\EXAMPLES\rmi
C:\EXAMPLES\rmi>javac *.java
C:\EXAMPLES\rmi>start rmiregistry
C:\EXAMPLES\rmi>
```

This will start an **rmi** registry on a separate window as shown below.



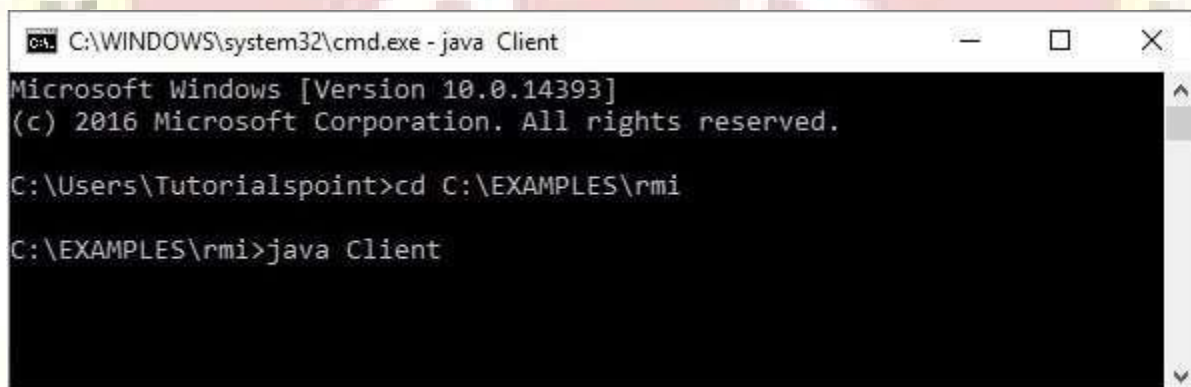
Step 2 – Run the server class file as shown below.

Java Server



Step 3 – Run the client class file as shown below.

java Client



Verification – As soon you start the client, you would see the following output in the server.

```

C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Tutorialspoint>cd C:\EXAMPLES\rmi

C:\EXAMPLES\rmi>javac *.java

C:\EXAMPLES\rmi>start rmiregistry

C:\EXAMPLES\rmi>java Server
Server ready
This is an example RMI program

```

Declaring and Implementation

Implementing a Remote Interface

This section discusses the task of implementing a class for the compute engine. In general, a class that implements a remote interface should at least do the following:

- Declare the remote interfaces being implemented
- Define the constructor for each remote object
- Provide an implementation for each remote method in the remote interfaces

An RMI server program needs to create the initial remote objects and *export* them to the RMI runtime, which makes them available to receive incoming remote invocations. This setup procedure can be either encapsulated in a method of the remote object implementation class itself or included in another class entirely. The setup procedure should do the following:

- Create and install a security manager
- Create and export one or more remote objects
- Register at least one remote object with the RMI registry (or with another naming service, such as a service accessible through the Java Naming and Directory Interface) for bootstrapping purposes

The complete implementation of the compute engine follows. The `engine.ComputeEngine` class implements the remote interface `Compute` and also includes the main method for setting up the compute engine. Here is the source code for the `ComputeEngine` class:


```
package engine;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import compute.Compute;
import compute.Task;

public class ComputeEngine implements Compute {

    public ComputeEngine() {
        super();
    }

    public <T> T executeTask(Task<T> t) {
        return t.execute();
    }

    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            String name = "Compute";
            Compute engine = new ComputeEngine();
            Compute stub =
                (Compute) UnicastRemoteObject.exportObject(engine, 0);
            Registry registry = LocateRegistry.getRegistry();
            registry.rebind(name, stub);
            System.out.println("ComputeEngine bound");
        } catch (Exception e) {
            System.err.println("ComputeEngine exception:");
            e.printStackTrace();
        }
    }
}
```

```

    }
}
}

```

The following sections discuss each component of the compute engine implementation.

Declaring the Remote Interfaces Being Implemented

The implementation class for the compute engine is declared as follows:

```
public class ComputeEngine implements Compute
```

This declaration states that the class implements the Compute remote interface and therefore can be used for a remote object.

The ComputeEngine class defines a remote object implementation class that implements a single remote interface and no other interfaces. The ComputeEngine class also contains two executable program elements that can only be invoked locally. The first of these elements is a constructor for ComputeEngine instances. The second of these elements is a main method that is used to create a ComputeEngine instance and make it available to clients.

Defining the Constructor for the Remote Object

The ComputeEngine class has a single constructor that takes no arguments. The code for the constructor is as follows:

```
public ComputeEngine() {
    super();
}

```

This constructor just invokes the superclass constructor, which is the no-argument constructor of the Object class. Although the superclass constructor gets invoked even if omitted from the ComputeEngine constructor, it is included for clarity.

Providing Implementations for Each Remote Method

The class for a remote object provides implementations for each remote method specified in the remote interfaces. The Compute interface contains a single remote method, executeTask, which is implemented as follows:

```
public <T> T executeTask(Task<T> t) {  
    return t.execute();  
}
```

This method implements the protocol between the ComputeEngine remote object and its clients. Each client provides the ComputeEngine with a Task object that has a particular implementation of the Task interface's execute method. The ComputeEngine executes each client's task and returns the result of the task's execute method directly to the client.

Passing Objects in RMI

Arguments to or return values from remote methods can be of almost any type, including local objects, remote objects, and primitive data types. More precisely, any entity of any type can be passed to or from a remote method as long as the entity is an instance of a type that is a primitive data type, a remote object, or a *serializable* object, which means that it implements the interface `java.io.Serializable`.

Some object types do not meet any of these criteria and thus cannot be passed to or returned from a remote method. Most of these objects, such as threads or file descriptors, encapsulate information that makes sense only within a single address space. Many of the core classes, including the classes in the packages `java.lang` and `java.util`, implement the `Serializable` interface.

The rules governing how arguments and return values are passed are as follows:

- Remote objects are essentially passed by reference. A *remote object reference* is a stub, which is a client-side proxy that implements the complete set of remote interfaces that the remote object implements.
- Local objects are passed by copy, using object serialization. By default, all fields are copied except fields that are marked `static` or `transient`. Default serialization behavior can be overridden on a class-by-class basis.

Passing a remote object by reference means that any changes made to the state of the object by remote method invocations are reflected in the original remote object. When a remote object is passed, only those interfaces that are remote interfaces are available to the receiver. Any methods defined in the implementation class or defined in non-remote interfaces implemented by the class are not available to that receiver.

For example, if you were to pass a reference to an instance of the `ComputeEngine` class, the receiver would have access only to the compute engine's `executeTask` method. That receiver would not see the `ComputeEngine` constructor, its main method, or its implementation of any methods of `java.lang.Object`.

In the parameters and return values of remote method invocations, objects that are not remote objects are passed by value. Thus, a copy of the object is created in the receiving Java virtual machine. Any changes to the object's state by the receiver are reflected only in the receiver's copy, not in the sender's original instance. Any changes to the object's state by the sender are reflected only in the sender's original instance, not in the receiver's copy.

Implementing the Server's main Method

The most complex method of the `ComputeEngine` implementation is the main method. The main method is used to start the `ComputeEngine` and therefore needs to do the necessary initialization and housekeeping to prepare the server to accept calls from clients. This method is not a remote method, which means that it cannot be invoked from a different Java virtual machine. Because the main method is declared static, the method is not associated with an object at all but rather with the class `ComputeEngine`.

Creating and Installing a Security Manager

The main method's first task is to create and install a security manager, which protects access to system resources from untrusted downloaded code running within the Java virtual machine. A security manager determines whether downloaded code has access to the local file system or can perform any other privileged operations.

If an RMI program does not install a security manager, RMI will not download classes (other than from the local class path) for objects received as arguments or return values of remote method invocations. This restriction ensures that the operations performed by downloaded code are subject to a security policy.

Here's the code that creates and installs a security manager:

```
if (System.getSecurityManager() == null) {  
    System.setSecurityManager(new SecurityManager());  
}
```

Making the Remote Object Available to Clients

Next, the main method creates an instance of `ComputeEngine` and exports it to the RMI runtime with the following statements:

```
Compute engine = new ComputeEngine();  
Compute stub =  
    (Compute) UnicastRemoteObject.exportObject(engine, 0);
```

The static `UnicastRemoteObject.exportObject` method exports the supplied remote object so that it can receive invocations of its remote methods from remote clients. The second argument, an `int`, specifies which TCP port to use to listen for incoming remote invocation requests for the object. It is common to use the value zero, which specifies the use of an anonymous port. The actual port will then be chosen at runtime by RMI or the underlying operating system. However, a non-zero value can also be used to specify a specific port to use for listening. Once the `exportObject` invocation has returned successfully, the `ComputeEngine` remote object is ready to process incoming remote invocations.

The `exportObject` method returns a stub for the exported remote object. Note that the type of the variable `stub` must be `Compute`, not `ComputeEngine`, because the stub for a remote object only implements the remote interfaces that the exported remote object implements.

The `exportObject` method declares that it can throw a `RemoteException`, which is a checked exception type. The main method handles this exception with its `try/catch` block. If the exception were not handled in this way, `RemoteException` would have to be declared in the `throws` clause of the main method. An attempt to export a remote object can throw a `RemoteException` if the necessary communication resources are not available, such as if the requested port is bound for some other purpose.

Before a client can invoke a method on a remote object, it must first obtain a reference to the remote object. Obtaining a reference can be done in the same way that any other object reference is obtained in a program, such as by getting the reference as part of the return value of a method or as part of a data structure that contains such a reference.

The system provides a particular type of remote object, the RMI registry, for finding references to other remote objects. The RMI registry is a simple remote object naming service that enables clients to obtain a reference to a remote object by name. The registry is typically only used to locate the first remote object that an RMI client needs to use. That first remote object might then provide support for finding other objects.

The `java.rmi.registry.Registry` remote interface is the API for binding (or registering) and looking up remote objects in the registry. The `java.rmi.registry.LocateRegistry` class provides static methods for synthesizing a remote reference to a registry at a particular network address (host and port). These methods create the remote reference object containing the specified network address without performing any remote communication. `LocateRegistry` also provides static methods for creating a new registry in the current Java virtual machine, although this example does not use those methods. Once a remote object is registered with an RMI registry on the local host, clients on any host can look up the remote object by name, obtain its reference, and then invoke remote methods on the object. The registry can be shared by all servers running on a host, or an individual server process can create and use its own registry.

The `ComputeEngine` class creates a name for the object with the following statement:

```
String name = "Compute";
```

The code then adds the name to the RMI registry running on the server. This step is done later with the following statements:

```
Registry registry = LocateRegistry.getRegistry();  
registry.rebind(name, stub);
```

This `rebind` invocation makes a remote call to the RMI registry on the local host. Like any remote call, this call can result in a `RemoteException` being thrown, which is handled by the catch block at the end of the main method.

Note the following about the `Registry.rebind` invocation:

- The no-argument overload of `LocateRegistry.getRegistry` synthesizes a reference to a registry on the local host and on the default registry port, 1099. You must use an overload that has an `int` parameter if the registry is created on a port other than 1099.
- When a remote invocation on the registry is made, a stub for the remote object is passed instead of a copy of the remote object itself. Remote implementation objects, such as instances of `ComputeEngine`, never leave the Java virtual machine in which they were created. Thus, when a client performs a lookup in a server's remote object registry, a copy of the stub is returned. Remote objects in such cases are thus effectively passed by (remote) reference rather than by value.
- For security reasons, an application can only bind, unbind, or rebind remote object references with a registry running on the same host. This restriction prevents a remote client from removing or overwriting any of the entries in a server's registry. A lookup, however, can be requested from any host, local or remote.

Once the server has registered with the local RMI registry, it prints a message indicating that it is ready to start handling calls. Then, the `main` method completes. It is not necessary to have a thread wait to keep the server alive. As long as there is a reference to the `ComputeEngine` object in another Java virtual machine, local or remote, the `ComputeEngine` object will not be shut down or garbage collected. Because the program binds a reference to the `ComputeEngine` in the registry, it is reachable from a remote client, the registry itself. The RMI system keeps the `ComputeEngine`'s process running. The `ComputeEngine` is available to accept calls and won't be reclaimed until its binding is removed from the registry *and* no remote clients hold a remote reference to the `ComputeEngine` object.

The final piece of code in the `ComputeEngine.main` method handles any exception that might arise. The only checked exception type that could be thrown in the code is `RemoteException`, either by the `UnicastRemoteObject.exportObject` invocation or by the registry `rebind` invocation. In either case, the program cannot do much more than exit after printing an error message. In some distributed applications, recovering from the failure to make a remote invocation is possible. For example, the application could attempt to retry the operation or choose another server to continue the operation.

Stubs and Skeleton

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

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

Understanding stub and skeleton

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

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

stub

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

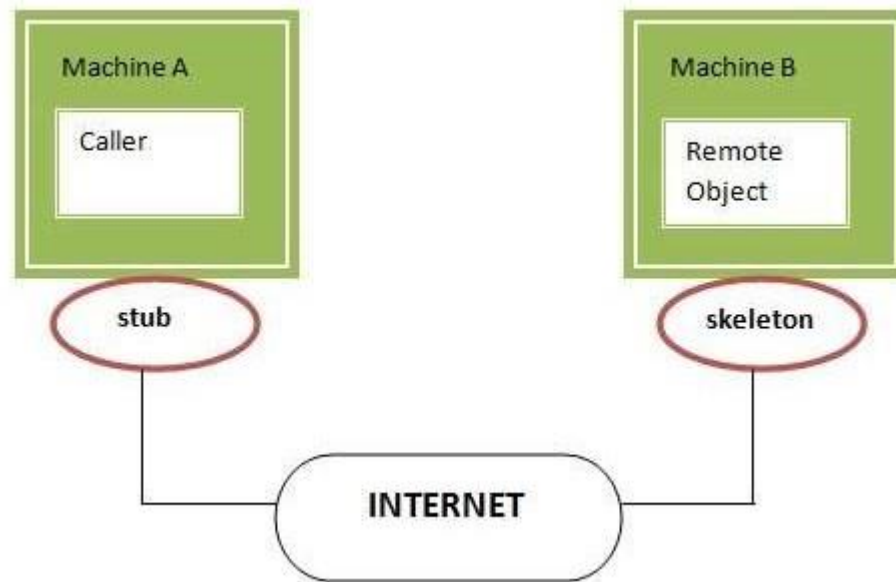
1. It initiates a connection with remote Virtual Machine (JVM),
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads (unmarshals) the return value or exception, and
5. It finally, returns the value to the caller.

skeleton

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

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

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



skeletons.

Understanding requirements for the distributed applications

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

1. The application need to locate the remote method
2. It need to provide the communication with the remote objects, and
3. The application need to load the class definitions for the objects.

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

Java RMI Example

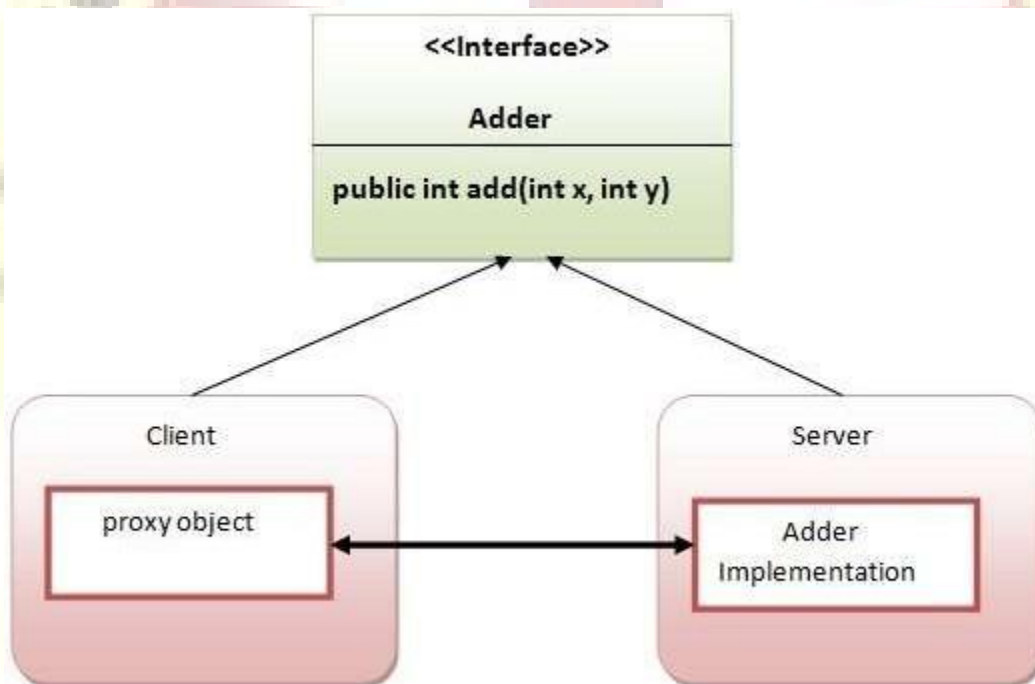
The is given the 6 steps to write the RMI program.

1. Create the remote interface
2. Provide the implementation of the remote interface
3. Compile the implementation class and create the stub and skeleton objects using the rmic tool

4. Start the registry service by rmiregistry tool
5. Create and start the remote application
6. Create and start the client application

RMI Example

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



1) create the remote interface

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

1. **import** java.rmi.*;
 2. **public interface** Adder **extends** Remote{
 3. **public int** add(**int** x,**int** y)**throws** RemoteException;
 4. }
-

2) Provide the implementation of the remote interface

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

- Either extend the UnicastRemoteObject class,
- or use the exportObject() method of the UnicastRemoteObject class

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

1. **import** java.rmi.*;
 2. **import** java.rmi.server.*;
 3. **public class** AdderRemote **extends** UnicastRemoteObject **implements** Adder{
 4. AdderRemote()**throws** RemoteException{
 5. **super**();
 6. }
 7. **public int** add(**int** x,**int** y){**return** x+y;}
 8. }
-

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

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

1. rmic AdderRemote
-

4) Start the registry service by the rmiregistry tool

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

1. rmiregistry 5000

5) Create and run the server application

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

public static java.rmi.Remote lookup(java.lang.String) throws java.rmi.NotBoundException, java.net.MalformedURLException, java.rmi.RemoteException;	It returns the object.
public static void bind(java.lang.String, java.rmi.Remote) throws java.rmi.AlreadyBoundException, java.net.MalformedURLException, java.rmi.RemoteException;	It binds the given name.
public static void unbind(java.lang.String) throws java.rmi.RemoteException, java.rmi.NotBoundException, java.net.MalformedURLException;	It destroys the bound with
public static void rebind(java.lang.String, java.rmi.Remote) throws java.rmi.RemoteException, java.net.MalformedURLException;	It binds the name.
public static java.lang.String[] list(java.lang.String) throws java.rmi.RemoteException, java.net.MalformedURLException;	It returns an remote object

In this example, we are binding the remote object by the name sonoo.

1. **import** java.rmi.*;
2. **import** java.rmi.registry.*;
3. **public class** MyServer{
4. **public static void** main(String args[]){
5. **try**{
6. Adder stub=**new** AdderRemote();
7. Naming.rebind("rmi://localhost:5000/sonoo",stub);

```

8. }catch(Exception e){System.out.println(e);}
9. }
10. }

```

6) Create and run the client application

At the client we are getting the stub object by the lookup() method of the Naming class and invoking the method on this object. In this example, we are running the server and client applications, in the same machine so we are using localhost. If you want to access the remote object from another machine, change the localhost to the host name (or IP address) where the remote object is located.

```

1. import java.rmi.*;
2. public class MyClient{
3. public static void main(String args[]){
4. try{
5. Adder stub=(Adder)Naming.lookup("rmi://localhost:5000/sonoo");
6. System.out.println(stub.add(34,4));
7. }catch(Exception e){}
8. }
9. }

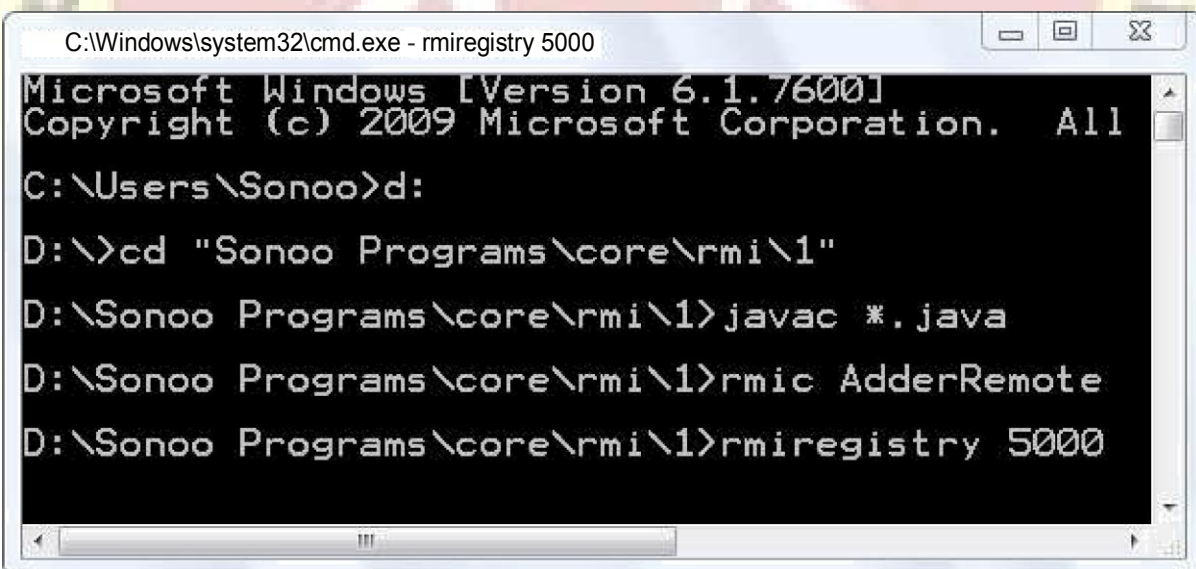
```

[download this example of rmi](#)

1. For running **this** rmi example,
- 2.
3. 1) compile all the java files
- 4.
5. javac *.java
- 6.

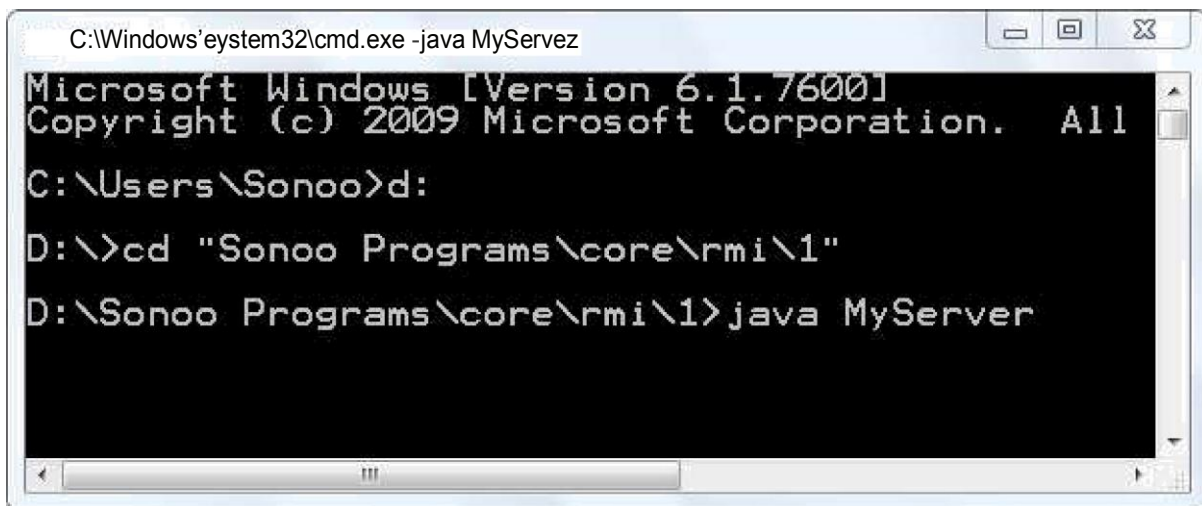
7. 2) create stub and skeleton object by rmic tool
- 8.
9. rmic AdderRemote
- 10.
11. 3) start rmi registry in one command prompt
- 12.
13. rmiregistry 5000
- 14.
15. 4) start the server in another command prompt
- 16.
17. java MyServer
- 18.
19. 5) start the client application in another command prompt
- 20.
21. java MyClient

Output of this RMI example

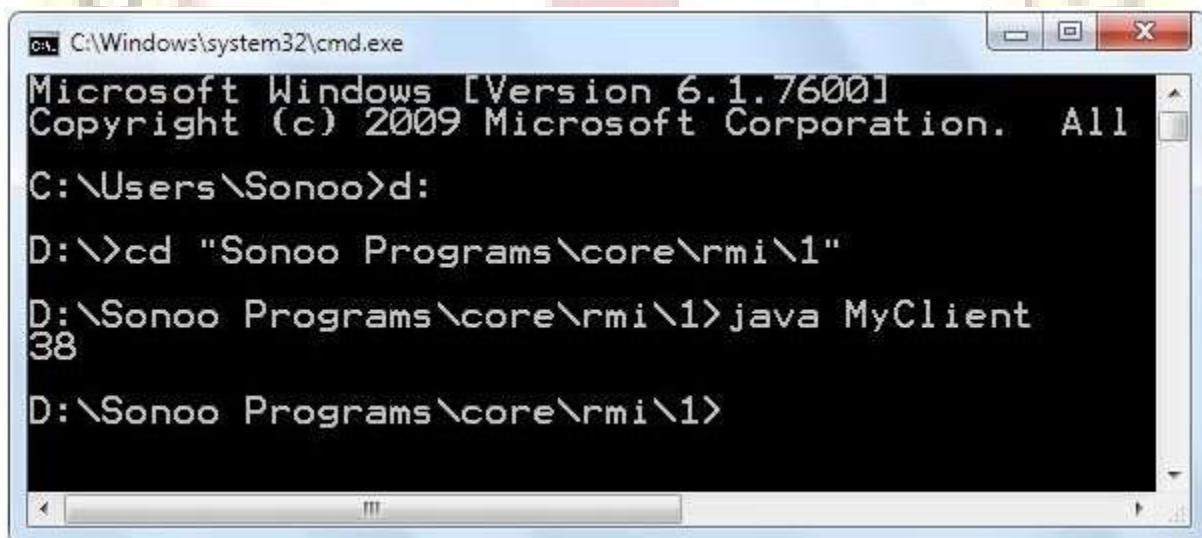


```
C:\Windows\system32\cmd.exe - rmiregistry 5000
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Sonoo>d:
D:\>cd "Sonoo Programs\core\rmi\1"
D:\Sonoo Programs\core\rmi\1>javac *.java
D:\Sonoo Programs\core\rmi\1>rmic AdderRemote
D:\Sonoo Programs\core\rmi\1>rmiregistry 5000
```



```
C:\Windows\system32\cmd.exe -java MyServerz
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All
C:\Users\Sonoo>d:
D:\>cd "Sonoo Programs\core\rmi\1"
D:\Sonoo Programs\core\rmi\1>java MyServer
```



```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All
C:\Users\Sonoo>d:
D:\>cd "Sonoo Programs\core\rmi\1"
D:\Sonoo Programs\core\rmi\1>java MyClient
38
D:\Sonoo Programs\core\rmi\1>
```

Meaningful example of RMI application with database

Consider a scenario, there are two applications running in different machines. Let's say MachineA and MachineB, machineA is located in United States and MachineB in India. MachineB want to get list of all the customers of MachineA application.

Let's develop the RMI application by following the steps.

1) Create the table

First of all, we need to create the table in the database. Here, we are using Oracle10 database.

CUSTOMER400											
Table	Data	Indexes	Model	Constraints	Grants	Statistics	UI Defaults	Triggers	Dependencies	SQL	
Query	Count Rows	Insert Row									
EDIT	ACC_NO	FIRSTNAME	LASTNAME	EMAIL	AMOUNT						
	67539876	James	Franklin	franklin1james@gmail.com	500000						
	67534876	Ravi	Kumar	ravimalik@gmail.com	98000						
	67579872	Vimal	Jaiswal	jaiswalvimal32@gmail.com	9380000						
					row(s) 1 - 3 of 3						
Download											

2) Create Customer class and Remote interface

File: Customer.java

1. **package** com.javatpoint;
2. **public class** Customer **implements** java.io.Serializable{
3. **private int** acc_no;
4. **private** String firstname,lastname,email;
5. **private float** amount;
6. //getters and setters
7. }

Note: Customer class must be Serializable.

File: Bank.java

1. **package** com.javatpoint;
2. **import** java.rmi.*;
3. **import** java.util.*;
4. **interface** Bank **extends** Remote{
5. **public** List<Customer> getCustomers()**throws** RemoteException;
6. }

3) Create the class that provides the implementation of Remote interface

File: BankImpl.java

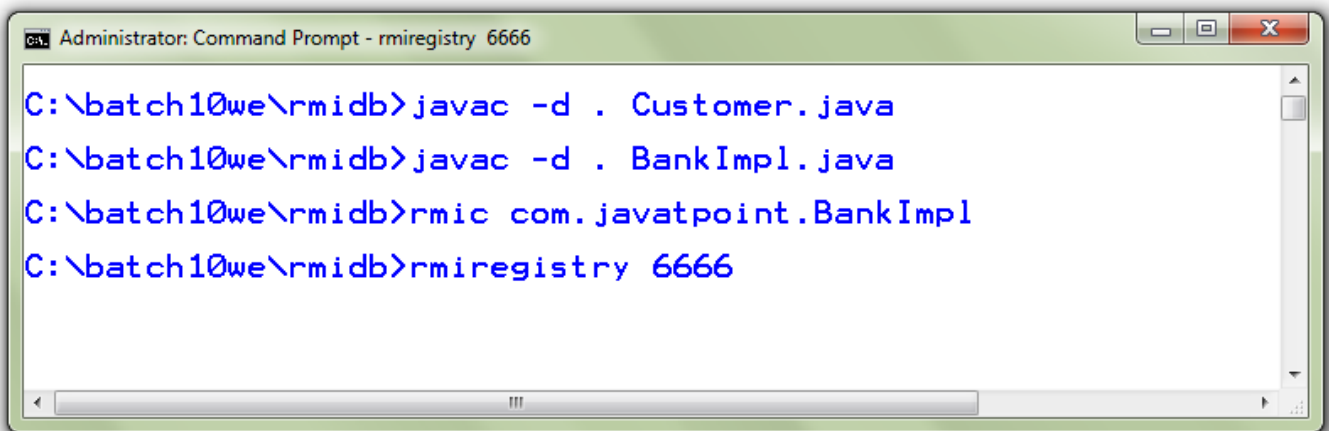
1. **package** com.javatpoint;
2. **import** java.rmi.*;
3. **import** java.rmi.server.*;
4. **import** java.sql.*;
5. **import** java.util.*;
6. **class** BankImpl **extends** UnicastRemoteObject **implements** Bank{
7. BankImpl()**throws** RemoteException{}
- 8.
9. **public** List<Customer> getCustomers(){
10. List<Customer> list=**new** ArrayList<Customer>();
11. **try**{
12. Class.forName("oracle.jdbc.driver.OracleDriver");
13. Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system",
"oracle");
14. PreparedStatement ps=con.prepareStatement("select * from customer400");
15. ResultSet rs=ps.executeQuery();
- 16.
17. **while**(rs.next()){
18. Customer c=**new** Customer();
19. c.setAcc_no(rs.getInt(1));
20. c.setFirstname(rs.getString(2));
21. c.setLastname(rs.getString(3));
22. c.setEmail(rs.getString(4));
23. c.setAmount(rs.getFloat(5));
24. list.add(c);

```

25. }
26.
27. con.close();
28. }catch(Exception e){System.out.println(e);}
29. return list;
30. }//end of getCustomers()
31. }

```

4) Compile the class `rmic` tool and start the registry service by `rmiregistry` tool



```

Administrator: Command Prompt - rmiregistry 6666
C:\batch10we\rmidb>javac -d . Customer.java
C:\batch10we\rmidb>javac -d . BankImpl.java
C:\batch10we\rmidb>rmic com.javatpoint.BankImpl
C:\batch10we\rmidb>rmiregistry 6666

```

5) Create and run the Server

File: `MyServer.java`

```

1. package com.javatpoint;
2. import java.rmi.*;
3. public class MyServer{
4. public static void main(String args[])throws Exception{
5. Remote r=new BankImpl();
6. Naming.rebind("rmi://localhost:6666/javatpoint",r);
7. }}

```



```
Administrator: C:\Windows\system32\cmd.exe - java com.javatpoint.MyServer
C:\batch10we\rmidb>javac -d . MyServer.java
C:\batch10we\rmidb>java com.javatpoint.MyServer
```

6) Create and run the Client

File: *MyClient.java*

1. **package** com.javatpoint;
2. **import** java.util.*;
3. **import** java.rmi.*;
4. **public class** MyClient {
5. **public static void** main(String args[])**throws** Exception{
6. Bank b=(Bank)Naming.lookup("rmi://localhost:6666/javatpoint");
- 7.
8. List<Customer> list=b.getCustomers();
9. **for**(Customer c:list){
10. System.out.println(c.getAcc_no()+" "+c.getFirstname()+" "+c.getLastname()
11. +" "+c.getEmail()+" "+c.getAmount());
12. }
- 13.
14. }}

```

Administrator: C:\Windows\system32\cmd.exe

C:\batch10we\rmidb>javac -d . MyClient.java

C:\batch10we\rmidb>java com.javatpoint.MyClient
67539876 James Franklin franklin1james@gmail.com 500000.0
67534876 Ravi Kumar ravimalik@gmail.com 98000.0
67579872 Vimal Jaiswal jaiswalvimal32@gmail.com 9380000.0

C:\batch10we\rmidb>

```

WRITING RMI CLIENTS

To develop a client program –

- Create a client class from where your intended to invoke the remote object.
- Get the RMI registry using the **getRegistry()** method of the **LocateRegistry** class which belongs to the package **java.rmi.registry**.
- Fetch the object from the registry using the method **lookup()** of the class **Registry** which belongs to the package **java.rmi.registry**.

To this method, you need to pass a string value representing the bind name as a parameter. This will return you the remote object.

- The lookup() returns an object of type remote, down cast it to the type Hello.
- Finally invoke the required method using the obtained remote object.

Following is an example of an RMI client program.

```

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {
    private Client() {}
    public static void main(String[] args) {
        try {
            // Getting the registry
            Registry registry = LocateRegistry.getRegistry(null);

            // Looking up the registry for the remote object
            Hello stub = (Hello) registry.lookup("Hello");

            // Calling the remote method using the obtained object
            stub.printMsg();

            // System.out.println("Remote method invoked");
        } catch (Exception e) {
            System.err.println("Client exception: " + e.toString());
            e.printStackTrace();
        }
    }
}

```

Compiling the Application

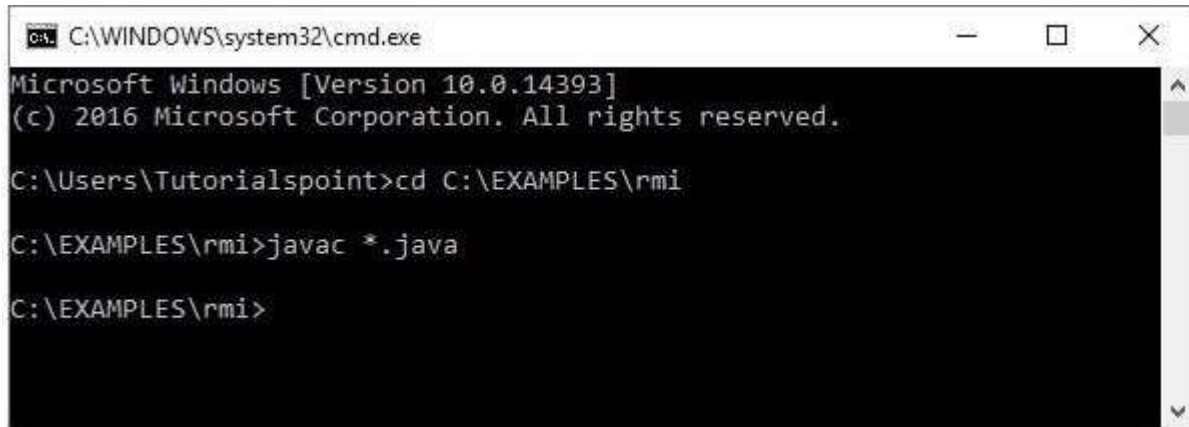
To compile the application –

- Compile the Remote interface.
- Compile the implementation class.
- Compile the server program.
- Compile the client program.

Or,

Open the folder where you have stored all the programs and compile all the Java files as shown below.

Javac *.java



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Tutorialspoint>cd C:\EXAMPLES\rmi

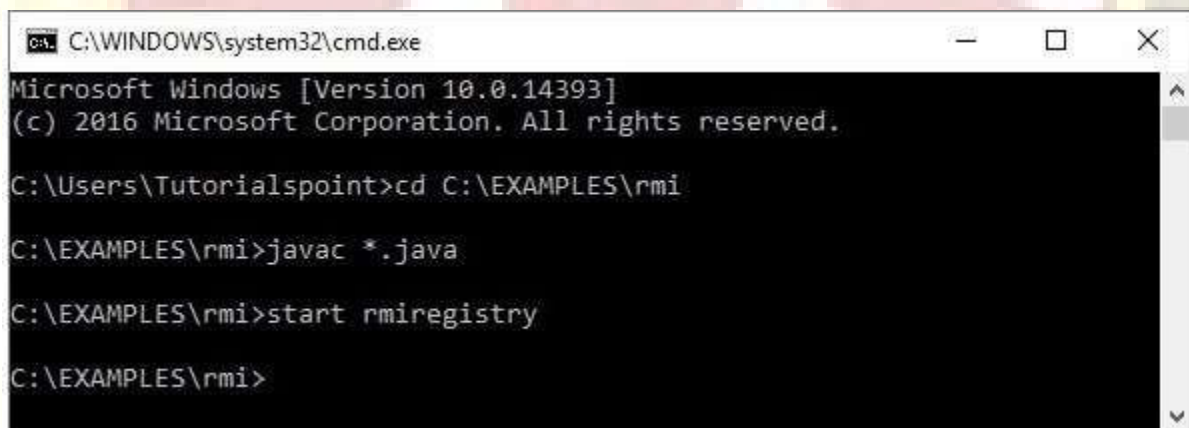
C:\EXAMPLES\rmi>javac *.java

C:\EXAMPLES\rmi>
```

Executing the Application

Step 1 – Start the **rmi** registry using the following command.

start rmiregistry



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Tutorialspoint>cd C:\EXAMPLES\rmi

C:\EXAMPLES\rmi>javac *.java

C:\EXAMPLES\rmi>start rmiregistry

C:\EXAMPLES\rmi>
```

This will start an **rmi** registry on a separate window as shown below.



```
C:\Program Files\Java\jdk1.8.0_101\bin\rmiregistry.exe
```

Step 2 – Run the server class file as shown below.

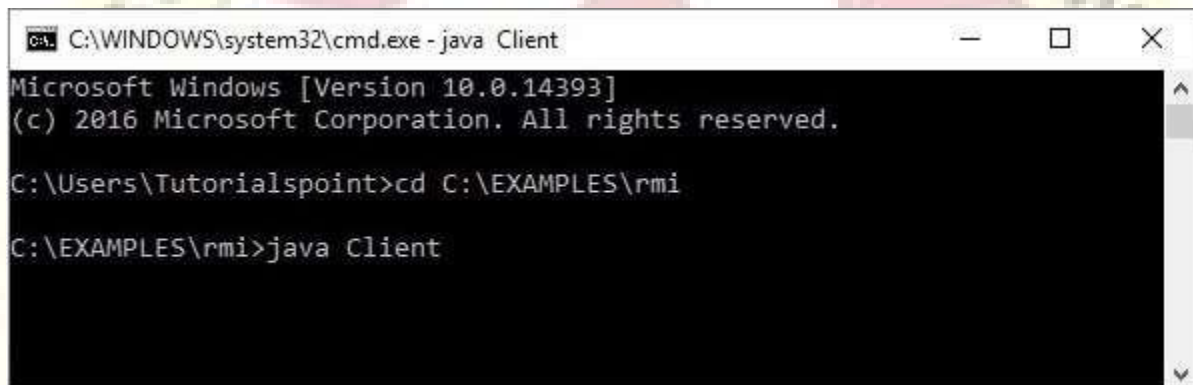
Java Server



```
C:\WINDOWS\system32\cmd.exe - java Server
C:\EXAMPLES\rmi>java Server
Server ready
```

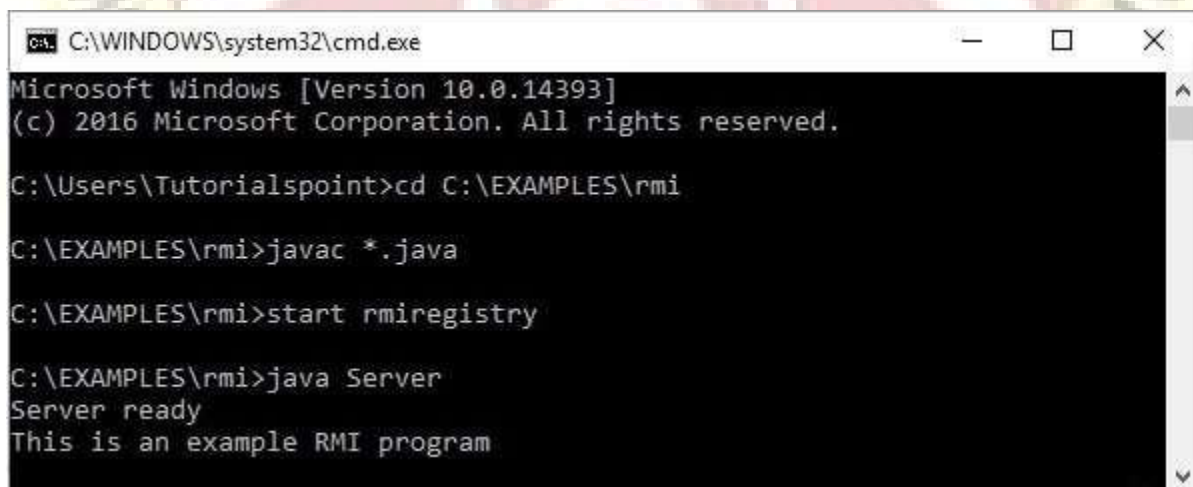
Step 3 – Run the client class file as shown below.

java Client



```
C:\WINDOWS\system32\cmd.exe - java Client
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.
C:\Users\Tutorialspoint>cd C:\EXAMPLES\rmi
C:\EXAMPLES\rmi>java Client
```

Verification – As soon you start the client, you would see the following output in the server.



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.
C:\Users\Tutorialspoint>cd C:\EXAMPLES\rmi
C:\EXAMPLES\rmi>javac *.java
C:\EXAMPLES\rmi>start rmiregistry
C:\EXAMPLES\rmi>java Server
Server ready
This is an example RMI program
```

ORB PROTOCOL

The ORB

This description of the Object Request Broker (ORB) provides background information to help you understand how the ORB works.

The IBM® ORB that is provided with this release is used by WebSphere® Application Server. It is one of the enterprise features of the Java™ Standard Edition. The ORB is both a tool and a runtime component. It provides distributed computing through the CORBA Internet Inter-Orb Protocol (IIOP) communication protocol. The protocol is defined by the Object Management Group (OMG). The ORB runtime environment consists of a Java implementation of a CORBA ORB. The ORB toolkit provides APIs and tools for both the Remote Method Invocation (RMI) programming model and the Interface Definition Language (IDL) programming model.

- [CORBA](#)

The Common Object Request Broker Architecture (CORBA) is an open, vendor-independent specification for distributed computing. It is published by the Object Management Group (OMG).

- [RMI and RMI-IIOP](#)

This description compares the two types of remote communication in Java; Remote Method Invocation (RMI) and RMI-IIOP.

- [Java IDL or RMI-IIOP?](#)

There are circumstances in which you might choose to use RMI-IIOP and others in which you might choose to use Java IDL.

- [RMI-IIOP limitations](#)

You must understand the limitations of RMI-IIOP when you develop an RMI-IIOP application, and when you deploy an existing CORBA application in a Java-IIOP environment.

- [Examples of client-server applications](#)

CORBA, RMI (JRMP), and RMI-IIOP approaches are used to present three client-server example applications. All the applications use the RMI-IIOP IBM ORB.

- **How the ORB works**

This description tells you how the ORB works, by explaining what the ORB does transparently for the client. An important part of the work is performed by the server side of the ORB.

- **Additional features of the ORB**

Portable object adapter, fragmentation, portable interceptors, and Interoperable Naming Service are described.

- **Using the ORB**

To use the Object Request Broker (ORB) effectively, you must understand the properties that control the behavior of the ORB.

- **ORB problem determination**

One of your first tasks when debugging an ORB problem is to determine whether the problem is in the client-side or in the server-side of the distributed application. Think of a typical RMI-IIOP session as a simple, synchronous communication between a client that is requesting access to an object, and a server that is providing it.

- **CORBA support**

The Java Platform, Standard Edition (JSE) supports, at a minimum, the specifications that are defined in the compliance document from Oracle. In some cases, the IBM JSE ORB supports more recent versions of the specifications.

ADVANCED JAVA PROGRAMMING

UNIT-V

JSP

INTRODUCTION JSP

JSP Tutorial



JSP technology is used to create web application just like Servlet technology. It can be thought of as an extension to Servlet because it provides more functionality than servlet such as expression language, JSTL, etc.

A JSP page consists of HTML tags and JSP tags. The JSP pages are easier to maintain than Servlet because we can separate designing and development. It provides some additional features such as Expression Language, Custom Tags, etc.

Advantages of JSP over Servlet

There are many advantages of JSP over the Servlet. They are as follows:

1) Extension to Servlet

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

2) Easy to maintain

JSP can be easily managed because we can easily separate our business logic with presentation logic. In Servlet technology, we mix our business logic with the presentation logic.

3) Fast Development: No need to recompile and redeploy

If JSP page is modified, we don't need to recompile and redeploy the project. The Servlet code needs to be updated and recompiled if we have to change the look and feel of the application.

4) Less code than Servlet

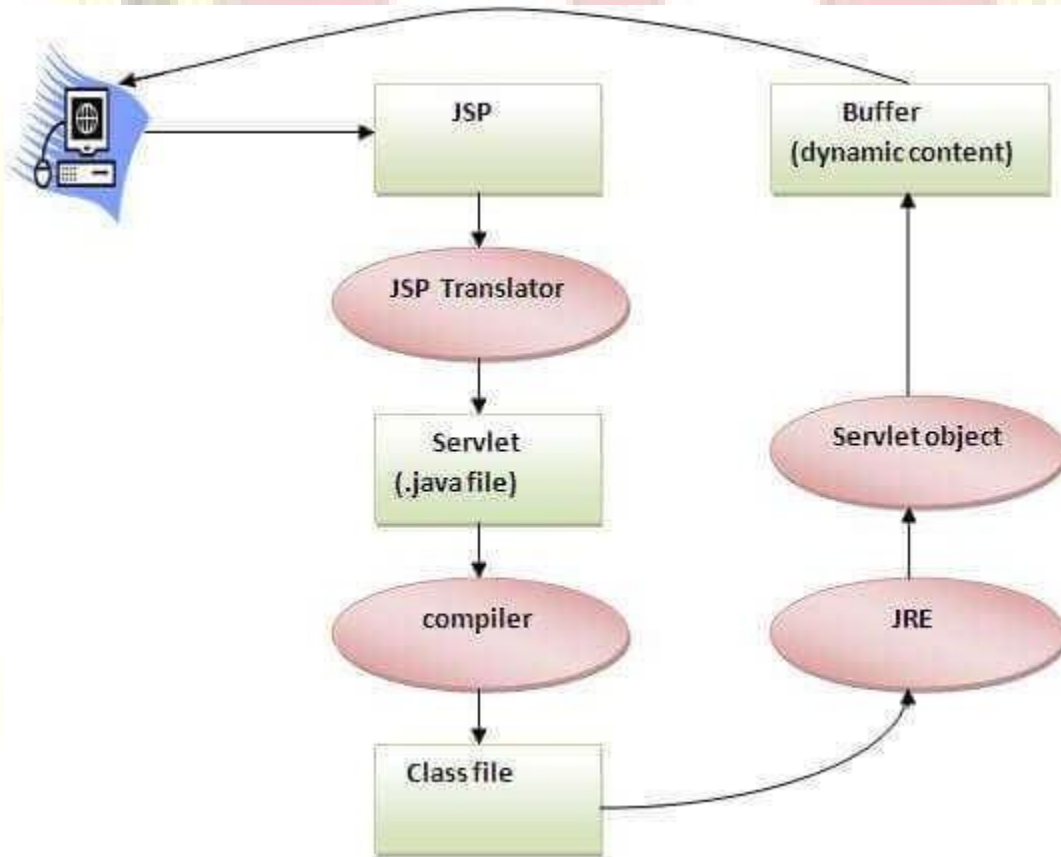
In JSP, we can use many tags such as action tags, JSTL, custom tags, etc. that reduces the code. Moreover, we can use EL, implicit objects, etc.

The Lifecycle of a JSP Page

The JSP pages follow these phases:

- Translation of JSP Page
- Compilation of JSP Page
- Classloading (the classloader loads class file)
- Instantiation (Object of the Generated Servlet is created).
- Initialization (the container invokes `jspInit()` method).
- Request processing (the container invokes `_jspService()` method).
- Destroy (the container invokes `jspDestroy()` method).

Note: *`jspInit()`, `_jspService()` and `jspDestroy()` are the life cycle methods of JSP.*



As depicted in the above diagram, JSP page is translated into Servlet by the help of JSP translator. The JSP translator is a part of the web server which is responsible for translating the JSP page into Servlet. After that, Servlet page is compiled by the compiler and gets converted into the class file. Moreover, all the processes that happen in Servlet are performed on JSP later like initialization, committing response to the browser and destroy.

Creating a simple JSP Page

To create the first JSP page, write some HTML code as given below, and save it by .jsp extension. We have saved this file as index.jsp. Put it in a folder and paste the folder in the web-apps directory in apache tomcat to run the JSP page.

index.jsp

Let's see the simple example of JSP where we are using the scriptlet tag to put Java code in the JSP page. We will learn scriptlet tag later.

1. `<html>`
2. `<body>`
3. `<% out.print(2*5); %>`
4. `</body>`
5. `</html>`

It will print **10** on the browser.

How to run a simple JSP Page?

Follow the following steps to execute this JSP page:

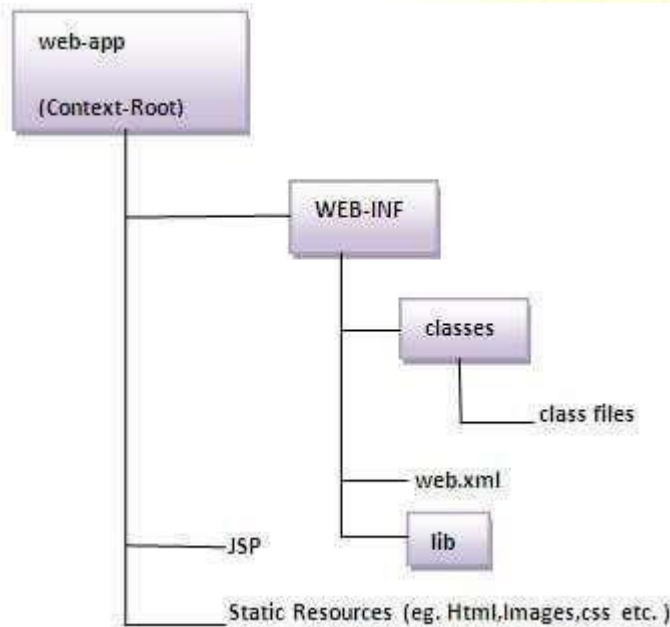
- Start the server
- Put the JSP file in a folder and deploy on the server
- Visit the browser by the URL `http://localhost:portno/contextRoot/jspfile`, for example, `http://localhost:8888/myapplication/index.jsp`

Do I need to follow the directory structure to run a simple JSP?

No, there is no need of directory structure if you don't have class files or TLD files. For example, put JSP files in a folder directly and deploy that folder. It will be running fine. However, if you are using Bean class, Servlet or TLD file, the directory structure is required.

The Directory structure of JSP

The directory structure of JSP page is same as Servlet. We contain the JSP page outside the WEB-INF folder or in any directory.



EXAMINING MVC AND JSP

MVC in JSP

1. [MVC in JSP](#)
2. [Example of following MVC in JSP](#)

MVC stands for Model View and Controller. It is a **design pattern** that separates the business logic, presentation logic and data.

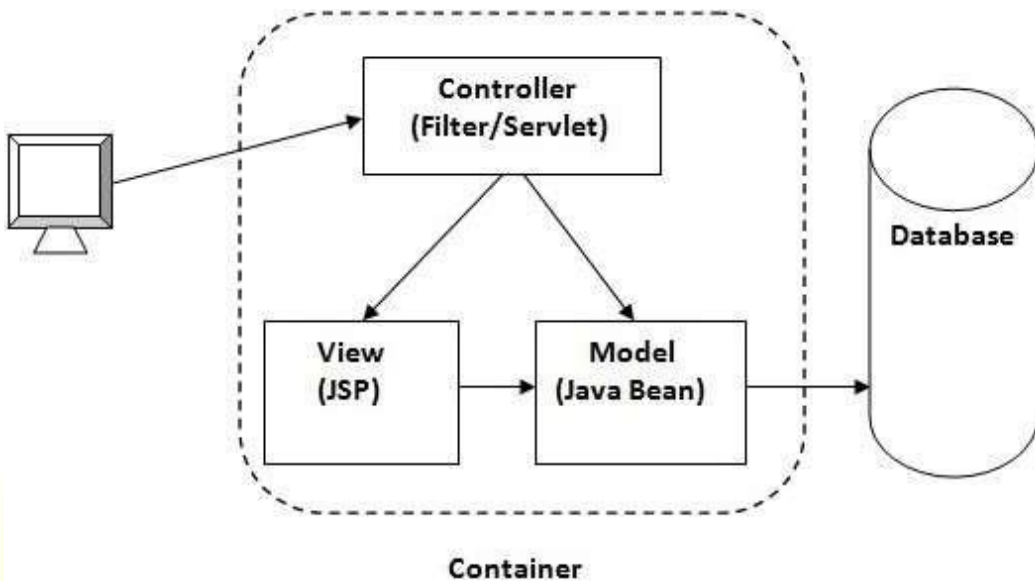
Controller acts as an interface between View and Model. Controller intercepts all the incoming requests.

Model represents the state of the application i.e. data. It can also have business logic.

View represents the presentaion i.e. UI(User Interface).

Advantage of MVC (Model 2) Architecture

1. Navigation Control is centralized
2. Easy to maintain the large application



If you new to MVC, please visit [Model1 vs Model2 first](#).

MVC Example in JSP

In this example, we are using servlet as a controller, jsp as a view component, Java Bean class as a model.

In this example, we have created 5 pages:

- **index.jsp** a page that gets input from the user.
- **ControllerServlet.java** a servlet that acts as a controller.
- **login-success.jsp** and **login-error.jsp** files acts as view components.
- **web.xml** file for mapping the servlet.

File: index.jsp

1. `<form action="ControllerServlet" method="post">`
2. `Name:<input type="text" name="name">
`
3. `Password:<input type="password" name="password">
`
4. `<input type="submit" value="login">`
5. `</form>`

File: ControllerServlet

```
1. package com.javatpoint;
2. import java.io.IOException;
3. import java.io.PrintWriter;
4. import javax.servlet.RequestDispatcher;
5. import javax.servlet.ServletException;
6. import javax.servlet.http.HttpServlet;
7. import javax.servlet.http.HttpServletRequest;
8. import javax.servlet.http.HttpServletResponse;
9. public class ControllerServlet extends HttpServlet {
10.     protected void doPost(HttpServletRequest request, HttpServletResponse response)
11.         throws ServletException, IOException {
12.         response.setContentType("text/html");
13.         PrintWriter out=response.getWriter();
14.
15.         String name=request.getParameter("name");
16.         String password=request.getParameter("password");
17.
18.         LoginBean bean=new LoginBean();
19.         bean.setName(name);
20.         bean.setPassword(password);
21.         request.setAttribute("bean",bean);
22.
23.         boolean status=bean.validate();
24.
25.         if(status){
26.             RequestDispatcher rd=request.getRequestDispatcher("login-success.jsp");
27.             rd.forward(request, response);
28.         }
29.         else{
30.             RequestDispatcher rd=request.getRequestDispatcher("login-error.jsp");
31.             rd.forward(request, response);
32.         }
```

```

33.
34. }
35.
36. @Override
37. protected void doGet(HttpServletRequest req, HttpServletResponse resp)
38.     throws ServletException, IOException {
39.     doPost(req, resp);
40. }
41. }

```

File: LoginBean.java

```

1. package com.javatpoint;
2. public class LoginBean {
3.     private String name,password;
4.
5.     public String getName() {
6.         return name;
7.     }
8.     public void setName(String name) {
9.         this.name = name;
10.    }
11.    public String getPassword() {
12.        return password;
13.    }
14.    public void setPassword(String password) {
15.        this.password = password;
16.    }
17.    public boolean validate(){
18.        if(password.equals("admin")){
19.            return true;
20.        }
21.        else{
22.            return false;
23.        }

```


24. }

25. }

File: login-success.jsp

1. `<%@page import="com.javatpoint.LoginBean"%>`
- 2.
3. `<p>You are successfully logged in!</p>`
4. `<%`
5. `LoginBean bean=(LoginBean)request.getAttribute("bean");`
6. `out.print("Welcome, "+bean.getName());`
7. `%>`

File: login-error.jsp

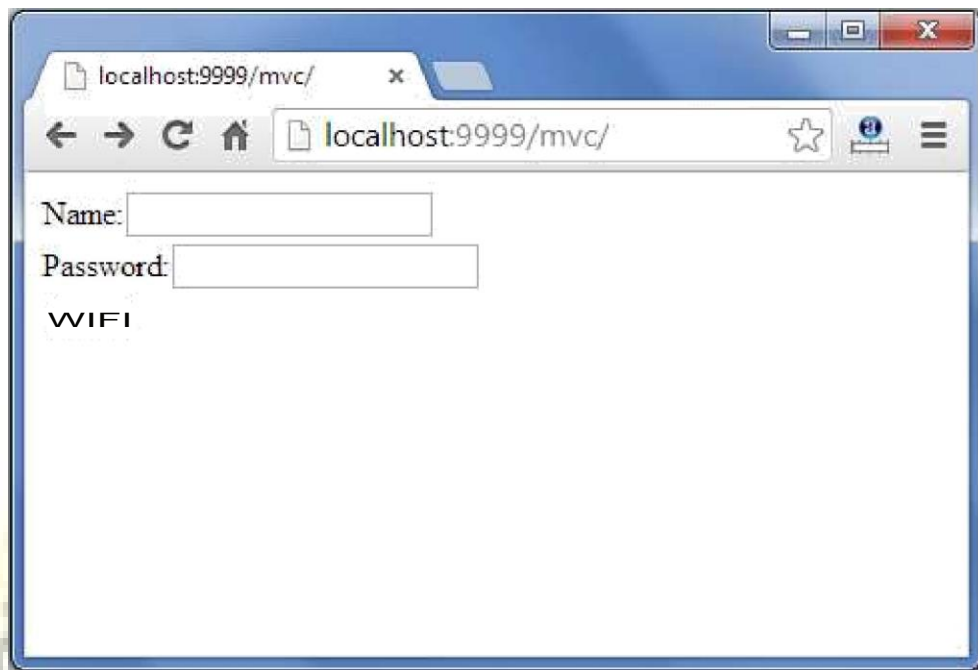
1. `<p>Sorry! username or password error</p>`
2. `<%@ include file="index.jsp" %>`

File: web.xml

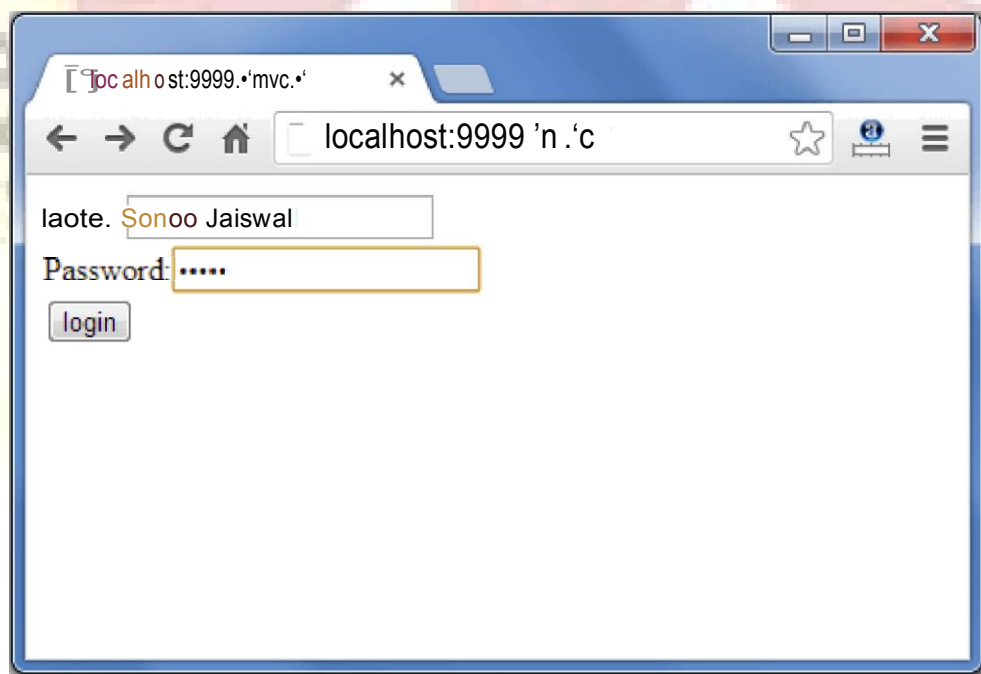
1. `<?xml version="1.0" encoding="UTF-8"?>`
2. `<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"`
3. `xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-`
4. `app_2_5.xsd"`
5. `xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-`
6. `app_3_0.xsd"`
7. `id="WebApp_ID" version="3.0">`
8. `<servlet>`
9. `<servlet-name>s1</servlet-name>`
10. `<servlet-class>com.javatpoint.ControllerServlet</servlet-class>`
11. `</servlet>`
12. `<servlet-mapping>`
13. `<servlet-name>s1</servlet-name>`
14. `<url-pattern>/ControllerServlet</url-pattern>`
15. `</servlet-mapping>`
16. `</web-app>`

[download this example \(developed using eclipse IDE\)](#)

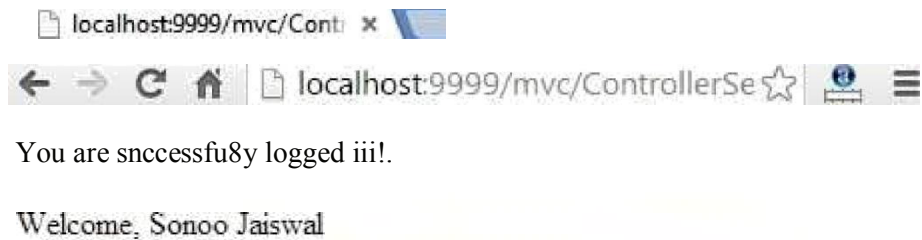
Output



A screenshot of a web browser window. The address bar shows "localhost:9999/mvc/". The page content includes a "Name:" label followed by an empty text input field, a "Password:" label followed by an empty text input field, and the text "WIFI" below the password field.



A screenshot of a web browser window. The address bar shows "localhost:9999/n.c". The page content includes a label "laote." followed by a text input field containing "Sano Jaiswal", a "Password:" label followed by a text input field containing masked characters ".....", and a "login" button below the password field.



JSP directives

1. [JSP directives](#)
1. [page directive](#)
2. [Attributes of page directive](#)

The **jsp directives** are messages that tells the web container how to translate a JSP page into the corresponding servlet.

There are three types of directives:

- page directive
- include directive
- taglib directive

Syntax of JSP Directive

1. `<%@ directive attribute="value" %>`

JSP page directive

The page directive defines attributes that apply to an entire JSP page.

Syntax of JSP page directive

1. `<%@ page attribute="value" %>`

Attributes of JSP page directive

- import
- contentType
- extends
- info

- buffer
- language
- isELIgnored
- isThreadSafe
- autoFlush
- session
- pageEncoding
- errorPage
- isErrorPage

1)import

The import attribute is used to import class,interface or all the members of a package.It is similar to import ke interface.

Example of import attribute

1. <html>
2. <
3. bod
4. y>
5. <%@ page **import**="java.util.Date" %>
6. Today is: <%= **new** Date() %>

- 6.
 7. `</body>`
 8. `</html>`
-

2)contentType

The contentType attribute defines the MIME(Multipurpose Internet Mail Extension) type of the HTTP response.The default value is "text/html;charset=ISO-8859-1".

Example of contentType attribute

1. `<html>`
 2. `<body>`
 - 3.
 4. `<%@ page contentType=application/msword %>`
 5. Today is: `<%= new java.util.Date() %>`
 - 6.
 7. `</body>`
 8. `</html>`
-

3)extends

The extends attribute defines the parent class that will be inherited by the generated servlet.It is rarely used.

4)info

This attribute simply sets the information of the JSP page which is retrieved later by using `getServletInfo()` method of Servlet interface.

Example of info attribute

1. `<html>`

2. <body>
- 3.
4. <%@ page info="composed by Sonoo Jaiswal" %>
5. Today is: <%= new java.util.Date() %>
- 6.
7. </body>
8. </html>

The web container will create a method `getServletInfo()` in the resulting servlet. For example:

1. **public** String `getServletInfo()` {
2. **return** "composed by Sonoo Jaiswal";
3. }

5)buffer

The buffer attribute sets the buffer size in kilobytes to handle output generated by the JSP page. The default size of the buffer is 8Kb.

Example of buffer attribute

1. <html>
2. <body>
- 3.
4. <%@ page buffer="16kb" %>
5. Today is: <%= new java.util.Date() %>
- 6.
7. </body>
8. </html>

6) language

The language attribute specifies the scripting language used in the JSP page. The default value is "java".

7) isELIgnored

We can ignore the Expression Language (EL) in jsp by the isELIgnored attribute. By default its value is false Language is enabled by default. We see Expression Language later.

1. `<%@ page isELIgnored="true" %>`//Now EL will be ignored

8) isThreadSafe

Servlet and JSP both are multithreaded. If you want to control this behaviour of JSP page, you can use isThreadSafe page directive. The value of isThreadSafe value is true. If you make it false, the web container will serialize the requests to the JSP page. i.e. it will wait until the JSP finishes responding to a request before passing another request to it. If you make the isThreadSafe attribute like:

```
<%@ page isThreadSafe="false" %>
```

The web container in such a case, will generate the servlet as:

1. **public class** SimplePage_jsp **extends** HttpJspBase
2. **implements** SingleThreadModel{
3.
4. }

9) errorPage

The errorPage attribute is used to define the error page, if exception occurs in the current page, it will be redirected to the error page.

Example of errorPage attribute

1. `//index.jsp`
2. `<html>`
3. `<body>`
- 4.
5. `<%@ page errorPage="myerrorpage.jsp" %>`
- 6.
7. `<%= 100/0 %>`
- 8.
9. `</body>`
10. `</html>`

10) isErrorPage

The isErrorPage attribute is used to declare that the current page is the error page.

Note: The exception object can only be used in the error page.

Example of isErrorPage attribute

1. `//myerrorpage.jsp`
2. `<html>`
3. `<body>`
- 4.
5. `<%@ page isErrorPage="true" %>`
- 6.
7. `Sorry an exception occurred!
`
8. `The exception is: <%= exception %>`
- 9.
10. `</body>`
11. `</html>`

Exception Handling in JSP

1. [Exception Handling in JSP](#)
2. [Example of exception handling in jsp by the elements of page directive](#)
3. [Example of exception handling in jsp by specifying the error-page element in web.xml file](#)

The exception is normally an object that is thrown at runtime. Exception Handling is the process to handle the runtime errors. There may occur exception any time in your web application. So handling exceptions is a safer side for the web developer. In JSP, there are two ways to perform exception handling:

1. By **errorPage** and **isErrorPage** attributes of page directive
2. By **<error-page>** element in web.xml file

Example of exception handling in jsp by the elements of page directive

In this case, you must define and create a page to handle the exceptions, as in the error.jsp page. The pages where may occur exception, define the errorPage attribute of page directive, as in the process.jsp page.

There are 3 files:

- index.jsp for input values
- process.jsp for dividing the two numbers and displaying the result
- error.jsp for handling the exception

index.jsp

1. `<form action="process.jsp">`
2. No 1:`<input type="text" name="n1" />

`
3. No 1:`<input type="text" name="n2" />

`
4. `<input type="submit" value="divide"/>`
5. `</form>`

process.jsp

1. `<%@ page errorPage="error.jsp" %>`
2. `<%`
- 3.
4. `String num1=request.getParameter("n1");`
5. `String num2=request.getParameter("n2");`
- 6.
7. `int a=Integer.parseInt(num1);`
8. `int b=Integer.parseInt(num2);`
9. `int c=a/b;`
10. `out.print("division of numbers is: "+c);`
- 11.
12. `%>`

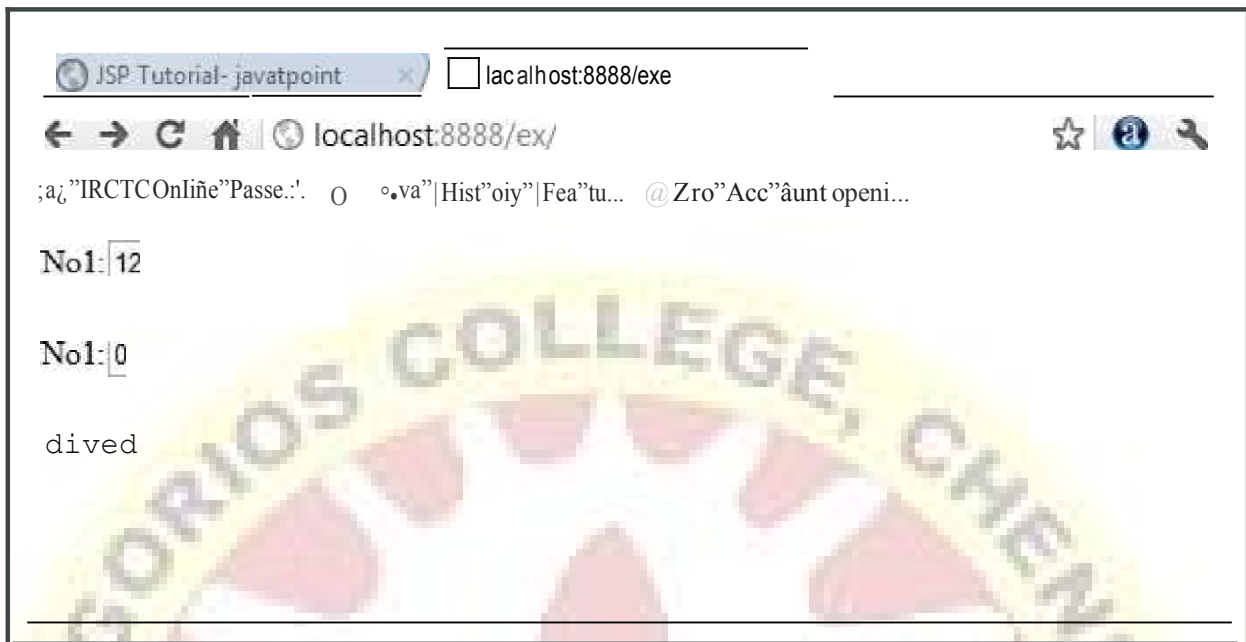
error.jsp

1. `<%@ page isErrorPage="true" %>`
- 2.
3. `<h3>Sorry an exception occured!</h3>`
- 4.
5. `Exception is: <%= exception %>`

[download this example](#)

Output of this example:





Example of exception handling in jsp by specifying the error-page element in web.xml file

This approach is better because you don't need to specify the `errorPage` attribute in each jsp page. Specifying the single entry in the `web.xml` file will handle the exception. In this case, either specify `exception-type` or `error-code` with the `location` element. If you want to handle all the exception, you will have to specify the `java.lang.Exception` in the `exception-type` element. Let's see the simple example:

There are 4 files:

- `web.xml` file for specifying the error-page element
- `index.jsp` for input values
- `process.jsp` for dividing the two numbers and displaying the result
- `error.jsp` for displaying the exception

1) *web.xml* file if you want to handle any exception

1. `<web-app>`
- 2.
3. `<error-page>`
4. `<exception-type>java.lang.Exception</exception-type>`
5. `<location>/error.jsp</location>`
6. `</error-page>`
- 7.
8. `</web-app>`

This approach is better if you want to handle any exception. If you know any specific error code and you want to handle that exception, specify the `error-code` element instead of `exception-type` as given below:

1) *web.xml* file if you want to handle the exception for a specific error code

1. `<web-app>`
- 2.
3. `<error-page>`

4. `<error-code>500</error-code>`
5. `<location>/error.jsp</location>`
6. `</error-page>`
- 7.
8. `</web-app>`

2) index.jsp file is same as in the above example

3) process.jsp

Now, you don't need to specify the `errorPage` attribute of `page` directive in the jsp page.

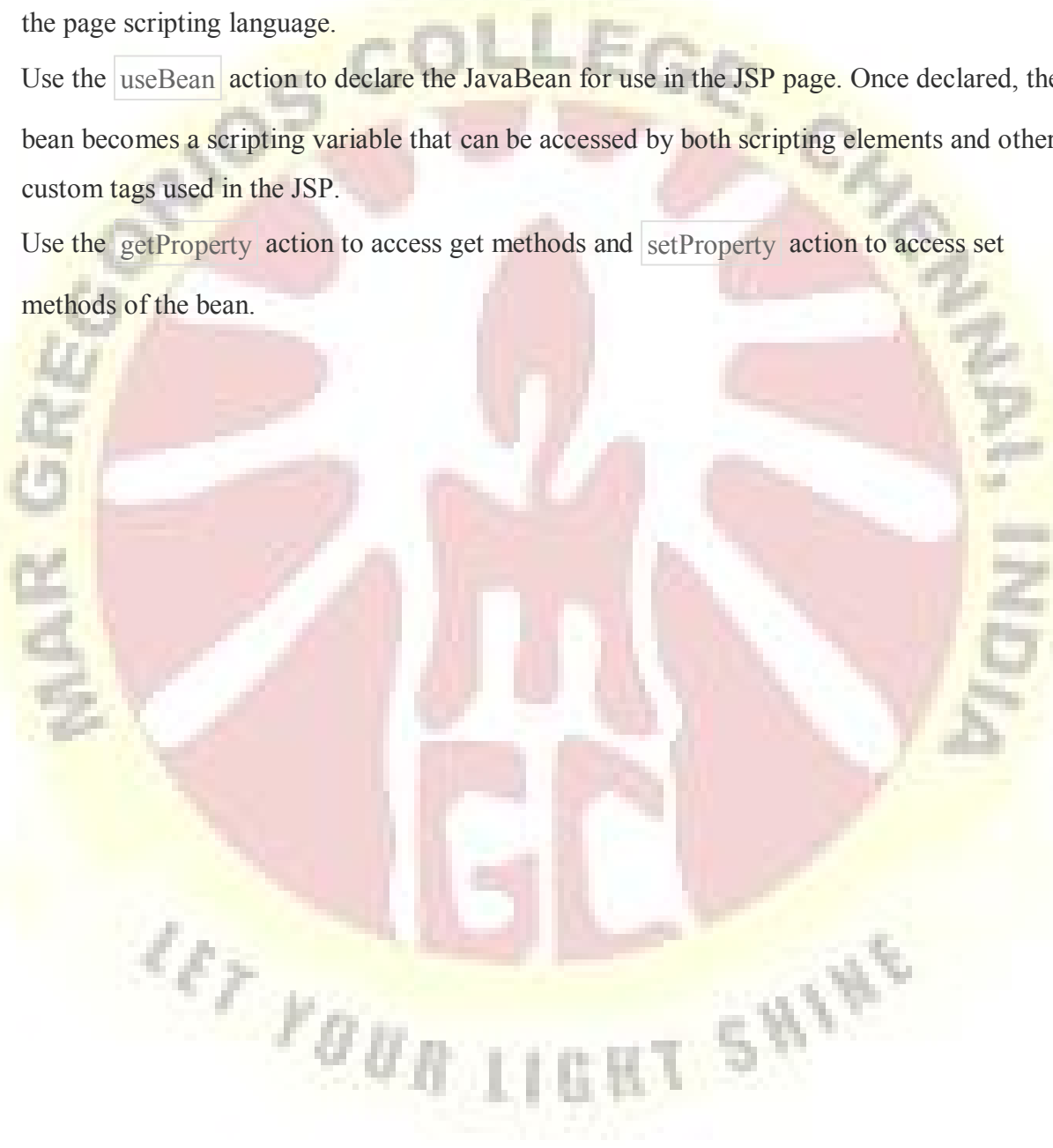
1. `<%@ page errorPage="error.jsp" %>`
2. `<%`
- 3.
4. `String num1=request.getParameter("n1");`
5. `String num2=request.getParameter("n2");`
- 6.
7. `int a=Integer.parseInt(num1);`
8. `int b=Integer.parseInt(num2);`
9. `int c=a/b;`
10. `out.print("division of numbers is: "+c);`
- 11.
12. `%>`

4) error.jsp file is same as in the above example

USING JAVA BEANS IN JSP

With this example we are going to demonstrate how to use a Bean in a JSP page. JavaServer Pages (JSP) is a server-side programming technology that enables the creation of dynamic, platform-independent method for building Web-based applications. JSP have access to the entire family of Java APIs, including the JDBC API to access enterprise databases. In short, to use a Bean in a JSP page you should:

- Create a Java Bean. The Java Bean is a specially constructed Java class that provides a default, no-argument constructor, implements the Serializable interface and it has getter and setter methods for its properties.
- Create a jsp page, using the `<%code fragment%>` scriptlet. It can contain any number of JAVA language statements, variable or method declarations, or expressions that are valid in the page scripting language.
- Use the `useBean` action to declare the JavaBean for use in the JSP page. Once declared, the bean becomes a scripting variable that can be accessed by both scripting elements and other custom tags used in the JSP.
- Use the `getProperty` action to access get methods and `setProperty` action to access set methods of the bean.



Let's take a look at the code snippets of a sample Bean and a JSP page that uses it, below:

SampleBean.java

```

01  package com.javacodegeeks.snippets.enterprise;
02
03  import java.util.Date;
04
05  public class
SampleBean { 06
07      private String param1;
08      private Date param2 =
new Date(); 09
10      public String getParam1() {
11          return
param1; 12 }
13      public void setParam1(String param1) {
14          this.param1 =
param1; 15 }
16
17      public Date getParam2() {
18          return
param2; 19 }
20      public void setParam2(Date param2) {
21          this.param2 =
param2; 22 }
23
24      @Override
25      public String toString() {
26          return "SampleBean [param1="+ param1 +", param2=" +
param2 + "]; 27      }
28
29  }

```

UseBean.jsp

```

01  <%@ page language="java" contentType="text/html; charset=UTF-8" %>

```



```
02 <%@ page import="com.javacodegeeks.snippets.enterprise.SampleBean"%>
03
04 <html>
05
06 <head>
07 <title>Java Code Geeks Snippets - Use a Bean in JSP
Page</title> 08 </head>
09
10 <body> 11
12 <jsp:useBean id="sampleBean" class="com.javacodegeeks.snippets.enterprise.SampleBean"
scope="se
13 <%-- intialize bean properties --%>
14 <jsp:setProperty name="sampleBean" property="param1" value="value1" />
15 </jsp:useBean>
16
17 Sample Bean: <%= sampleBean %>
18
19 param1: <jsp:getProperty name="sampleBean" property="param1" />
20 param2: <jsp:getProperty name="sampleBean" property="param2" />
21
22 </body>
```

URL:

```
http://myhost:8080/jcg snippets/UseBean.jsp
```

Output:

```
Sample Bean: SampleBean [param1=value1, param2=Thu Nov 17 21:28:03 EET 2011]
```

```
param1: value1 param2: Thu Nov 17 21:28:03 EET 2011
```

working with java mail

In this chapter, we will discuss how to send emails using JSP. To send an email using a JSP, you should have the **JavaMail API** and the **Java Activation Framework (JAF)** installed on your machine.

- You can download the latest version of [JavaMail \(Version 1.2\)](#) from the Java's standard website.
- You can download the latest version of JavaBeans Activation Framework [JAF \(Version 1.0.2\)](#) from the Java's standard website.

Download and unzip these files, in the newly-created top-level directories. You will find a number of jar files for both the applications. You need to add the **mail.jar** and the **activation.jar** files in your CLASSPATH.

Send a Simple Email

Here is an example to send a simple email from your machine. It is assumed that your **localhost** is connected to the Internet and that it is capable enough to send an email. Make sure all the jar files from the Java Email API package and the JAF package are available in CLASSPATH.

```
<%@ page import = "java.io.*,java.util.*,javax.mail.*"%>
<%@ page import = "javax.mail.internet.*,javax.activation.*"%>
<%@ page import = "javax.servlet.http.*,javax.servlet.*" %>
```

```
<%  
String result;  
  
// Recipient's email ID needs to be mentioned.  
String to = "abcd@gmail.com";  
  
// Sender's email ID needs to be mentioned  
String from = "mcmohd@gmail.com";  
  
// Assuming you are sending email from localhost  
String host = "localhost";  
  
// Get system properties object  
Properties properties = System.getProperties();  
  
// Setup mail server  
properties.setProperty("mail.smtp.host", host);  
  
// Get the default Session object.  
Session mailSession = Session.getDefaultInstance(properties);  
  
try {  
    // Create a default MimeMessage object.  
    MimeMessage message = new MimeMessage(mailSession);  
  
    // Set From: header field of the header.  
    message.setFrom(new InternetAddress(from));  
  
    // Set To: header field of the header.  
    message.addRecipient(Message.RecipientType.TO,  
        new InternetAddress(to));
```

```
// Set Subject: header field
message.setSubject("This is the Subject Line!");

// Now set the actual message
message.setText("This is actual message");

// Send message
Transport.send(message);
result = "Sent message successfully...";
} catch (MessagingException mex) {
    mex.printStackTrace();
    result = "Error: unable to send message...";
}
%>
<html>
<head>
<title>Send Email using JSP</title>
</head>
<body>
<center>
<h1>Send Email using JSP</h1>
</center>
<p align = "center">
<%
    out.println("Result: " + result + "\n");
%>
</p>
</body>
```

```
</html>
```

Let us now put the above code in **SendEmail.jsp** file and call this JSP using the URL **http://localhost:8080/SendEmail.jsp**. This will help send an email to the given email ID **abcd@gmail.com**. You will receive the following response –

Send Email using JSP

Result: Sent message successfully....

If you want to send an email to multiple recipients, then use the following methods to specify multiple email IDs –

```
void addRecipients(Message.RecipientType type, Address[] addresses)
throws MessagingException
```

Here is the description of the parameters –

- **type** – This would be set to TO, CC or BCC. Here CC represents Carbon Copy and BCC represents Black Carbon Copy. Example *Message.RecipientType.TO*
- **addresses** – This is the array of email ID. You would need to use the *InternetAddress()* method while specifying email IDs

Send an HTML Email

Here is an example to send an HTML email from your machine. It is assumed that your **localhost** is connected to the Internet and that it is capable enough to send an email. Make sure all the jar files from the **Java Email API package** and the **JAF package** are available in CLASSPATH.

This example is very similar to the previous one, except that here we are using the **setContent()** method to set content whose second argument is **"text/html"** to specify that the HTML content is included in the message.

Using this example, you can send as big an HTML content as you require.

```
<%@ page import = "java.io.*,java.util.*,javax.mail.*"%>
<%@ page import = "javax.mail.internet.*,javax.activation.*"%>
<%@ page import = "javax.servlet.http.*,javax.servlet.*" %>

<%
String result;

// Recipient's email ID needs to be mentioned.
String to = "abcd@gmail.com";

// Sender's email ID needs to be mentioned
String from = "mcmohd@gmail.com";

// Assuming you are sending email from localhost
String host = "localhost";

// Get system properties object
Properties properties = System.getProperties();

// Setup mail server
properties.setProperty("mail.smtp.host", host);

// Get the default Session object.
Session mailSession = Session.getDefaultInstance(properties);

try {
    // Create a default MimeMessage object.
    MimeMessage message = new MimeMessage(mailSession);

    // Set From: header field of the header.
    message.setFrom(new InternetAddress(from));
```

```
// Set To: header field of the header.
message.addRecipient(Message.RecipientType.TO, new InternetAddress(to));

// Set Subject: header field
message.setSubject("This is the Subject Line!");

// Send the actual HTML message, as big as you like
message.setContent("<h1>This is actual message</h1>", "text/html" );

// Send message
Transport.send(message);
result = "Sent message successfully... ";
} catch (MessagingException mex) {
mex.printStackTrace();
result = "Error: unable to send message... ";
}
%>

<html>
<head>
<title>Send HTML Email using JSP</title>
</head>

<body>
<center>
<h1>Send Email using JSP</h1>
</center>

<p align = "center">
<%
```

```

        out.println("Result: " + result + "\n");
    }
}
</p>
</body>
</html>

```

Let us now use the above JSP to send HTML message on a given email ID.

Send Attachment in Email

Following is an example to send an email with attachment from your machine –

```

<%@ page import = "java.io.*,java.util.*,javax.mail.*"%>
<%@ page import = "javax.mail.internet.*,javax.activation.*"%>
<%@ page import = "javax.servlet.http.*,javax.servlet.*" %>

<%
String result;

// Recipient's email ID needs to be mentioned.
String to = "abcd@gmail.com";

// Sender's email ID needs to be mentioned
String from = "mcmohd@gmail.com";

// Assuming you are sending email from localhost
String host = "localhost";

// Get system properties object
Properties properties = System.getProperties();

// Setup mail server
properties.setProperty("mail.smtp.host", host);

```



```
// Get the default Session object.
Session mailSession = Session.getDefaultInstance(properties);

try {
    // Create a default MimeMessage object.
    MimeMessage message = new MimeMessage(mailSession);

    // Set From: header field of the header.
    message.setFrom(new InternetAddress(from));

    // Set To: header field of the header.
    message.addRecipient(Message.RecipientType.TO, new InternetAddress(to));

    // Set Subject: header field
    message.setSubject("This is the Subject Line!");

    // Create the message part
    BodyPart messageBodyPart = new MimeBodyPart();

    // Fill the message
    messageBodyPart.setText("This is message body");

    // Create a multipart message
    Multipart multipart = new MimeMultipart();

    // Set text message part
    multipart.addBodyPart(messageBodyPart);

    // Part two is attachment
    messageBodyPart = new MimeBodyPart();
```

```
String filename = "file.txt";
DataSource source = new FileDataSource(filename);
messageBodyPart.setDataHandler(new DataHandler(source));
messageBodyPart.setFileName(filename);
multipart.addBodyPart(messageBodyPart);

// Send the complete message parts
message.setContent(multipart );

// Send message
Transport.send(message);
String title = "Send Email";
result = "Sent message successfully...";
} catch (MessagingException mex) {
mex.printStackTrace();
result = "Error: unable to send message...";
}
%>

<html>
<head>
<title>Send Attachment Email using JSP</title>
</head>

<body>
<center>
<h1>Send Attachment Email using JSP</h1>
</center>

<p align = "center">
```

```

    <%out.println("Result: " + result + "\n");%>
  </p>
</body>
</html>

```

Let us now run the above JSP to send a file as an attachment along with a message on a given email ID.

User Authentication Part

If it is required to provide user ID and Password to the email server for authentication purpose, then you can set these properties as follows –

```

props.setProperty("mail.user", "myuser");
props.setProperty("mail.password", "mypwd");

```

Rest of the email sending mechanism will remain as explained above.

Using Forms to Send Email

You can use HTML form to accept email parameters and then you can use the **request** object to get all the information as follows –

```

String to = request.getParameter("to");
String from = request.getParameter("from");
String subject = request.getParameter("subject");
String messageText = request.getParameter("body");

```

Once you have all the information, you can use the above mentioned programs to send email.

JavaMail Tutorial

1. [Java Mail API](#)
2. [Protocols used in JavaMail API](#)
3. [SMTP](#)
4. [POP](#)
5. [IMAP](#)
6. [MIME](#)

7. NNTP and Others

1. Java Mail Architecture

2. Java Mail API Core Classes

The **JavaMail** is an API that is used to compose, write and read electronic messages (emails).

The JavaMail API provides protocol-independent and platform-independent framework for sending and receiving mails.

The **javax.mail** and **javax.mail.activation** packages contains the core classes of JavaMail API.

The JavaMail facility can be applied to many events. It can be used at the time of registering the user (sending notification such as thanks for your interest to my site), forgot password (sending password to the users email id), sending notifications for important updates etc. So there can be various usage of java mail api.

Do You Know ?

- How to send and receive email using JavaMail API ?
- How to send email through gmail server ?
- How to send and receive email with attachment ?
- How to send email with html content including images?
- How to forward and delete the email ?

Protocols used in JavaMail API

There are some protocols that are used in JavaMail API.

- SMTP
- POP
- IMAP
- MIME
- NNTP and others

SMTP

SMTP is an acronym for Simple Mail Transfer Protocol. It provides a mechanism to deliver the email. We can use Apache James server, Postcast server, cmail server etc. as an SMTP server. But if we purchase the host space, an SMTP server is by default provided by the host provider. For example, my smtp server is mail.javatpoint.com. If we use the SMTP server provided by the host provider, authentication is required for sending and receiving emails.

POP

POP is an acronym for Post Office Protocol, also known as POP3. It provides a mechanism to receive the email. It provides support for single mail box for each user. We can use Apache James server, cmail server etc. as an POP server. But if we purchase the host space, an POP server is by default provided by the host provider. For example, the pop server provided by the host provider for my site is mail.javatpoint.com. This protocol is defined in RFC 1939.

IMAP

IMAP is an acronym for Internet Message Access Protocol. IMAP is an advanced protocol for receiving messages. It provides support for multiple mail box for each user, in addition to, mailbox can be shared by multiple users. It is defined in RFC 2060.

MIME

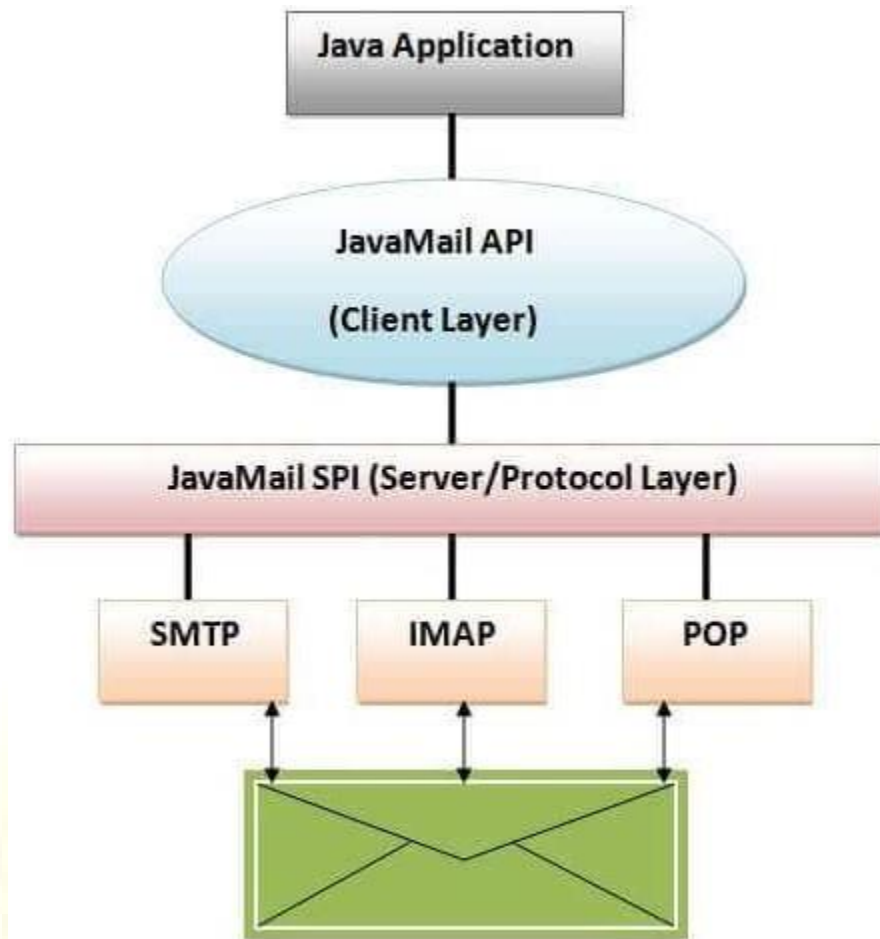
Multiple Internet Mail Extension (MIME) tells the browser what is being sent e.g. attachment, format of the known as mail transfer protocol but it is used by your mail program.

NNTP and Others

There are many protocols that are provided by third-party providers. Some of them are Network News Transfer Protocol (NNTP), Secure Multipurpose Internet Mail Extensions (S/MIME) etc.

JavaMail Architecture

The java application uses JavaMail API to compose, send and receive emails. The JavaMail API uses SPI (Service Provider Interfaces) that provides the intermediary services to the java application to deal with the different protocols. Let's understand it with the figure given below:



JavaMail API Core Classes

There are two packages that are used in Java Mail API: `javax.mail` and `javax.mail.internet` package. These packages contains many classes for Java Mail API. They are:

- `javax.mail.Session` class
- `javax.mail.Message` class
- `javax.mail.internet.MimeMessage` class
- `javax.mail.Address` class
- `javax.mail.internet.InternetAddress` class
- `javax.mail.Authenticator` class

- javax.mail.PasswordAuthentication class
- javax.mail.Transport class
- javax.mail.Store class
- javax.mail.Folder class etc.

We will know about these class one by one when it is getting used.

JMS Tutorial

JMS (Java Message Service) is an API that provides the facility to create, send and read messages. It provides loosely coupled, reliable and asynchronous communication.

JMS is also known as a messaging service.

Understanding Messaging

Messaging is a technique to communicate applications or software components.

JMS is mainly used to send and receive message from one application to another.

Requirement of JMS

Generally, user sends message to application. But, if we want to send message from one application to another, we need to use JMS API.

Consider a scenario, one application A is running in INDIA and another application B is running in USA. To send message from A application to B, we need to use JMS.

Advantage of JMS

1) **Asynchronous:** To receive the message, client is not required to send request. Message will arrive automatically to the client.

2) **Reliable:** It provides assurance that message is delivered.

Messaging Domains

There are two types of messaging domains in JMS.

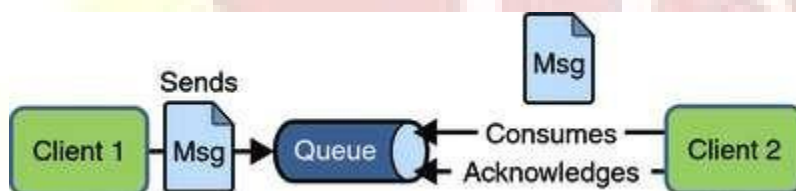
1. Point-to-Point Messaging Domain
2. Publisher/Subscriber Messaging Domain

1) Point-to-Point (PTP) Messaging Domain

In PTP model, one message is **delivered to one receiver** only. Here, **Queue** is used as a message oriented middleware (MOM).

The Queue is responsible to hold the message until receiver is ready.

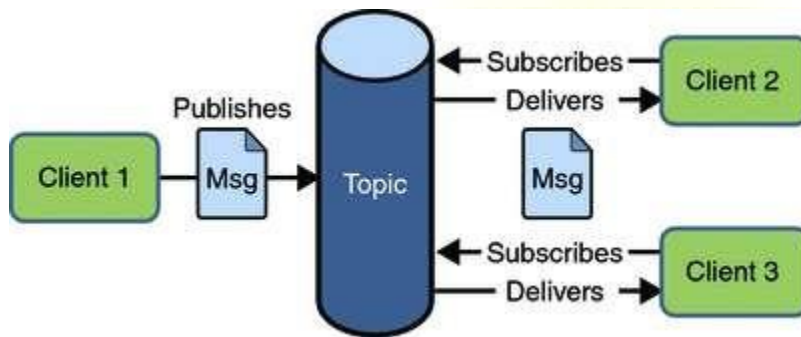
In PTP model, there is **no timing dependency** between sender and receiver.



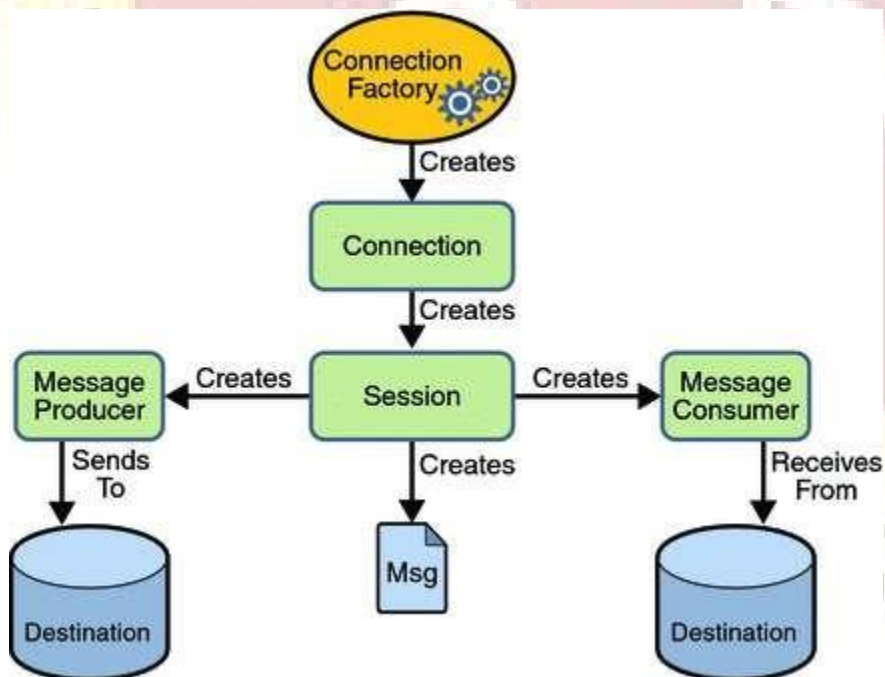
2) Publisher/Subscriber (Pub/Sub) Messaging Domain

In Pub/Sub model, one message is **delivered to all the subscribers**. It is like broadcasting. Here, **Topic** is used as a message oriented middleware that is responsible to hold and deliver messages.

In PTP model, there is **timing dependency** between publisher and subscriber.



JMS Programming Model



JMS Queue Example

To develop JMS queue example, you need to install any application server. Here, we are using **glassfish3** server where we are creating two JNDI.

1. Create connection factory named **myQueueConnectionFactory**
2. Create destination resource named **myQueue**

After creating JNDI, create server and receiver application. You need to run server and receiver in different console. Here, we are using eclipse IDE, it is opened in different console by default.

1) Create connection factory and destination resource

Open server admin console by the URL **http://localhost:4848**

Login with the username and password.

Click on the **JMS Resource -> Connection Factories -> New**, now write the pool name and select the Resource Type as QueueConnectionFactory then click on ok button.

The screenshot shows the GlassFish Server Admin Console interface. The browser address bar displays `localhost:4848/common/index.jsf`. The page title is "GlassFish™ Server Open Source Edition". The left sidebar shows a tree view of the server configuration, with "JMS Resources" expanded and "Connection Factories" selected. The main content area displays the "New JMS Connection Factory" configuration form.

New JMS Connection Factory (OK Cancel)

The creation of a new Java Message Service (JMS) connection factory also creates a connector connection pool for the factory and a connector resource.

General Settings

Pool Name: *

Resource Type: *

Description:

Status: Enabled

Pool Settings

Initial and Minimum Pool Size: Connections
Minimum and initial number of connections maintained in the pool

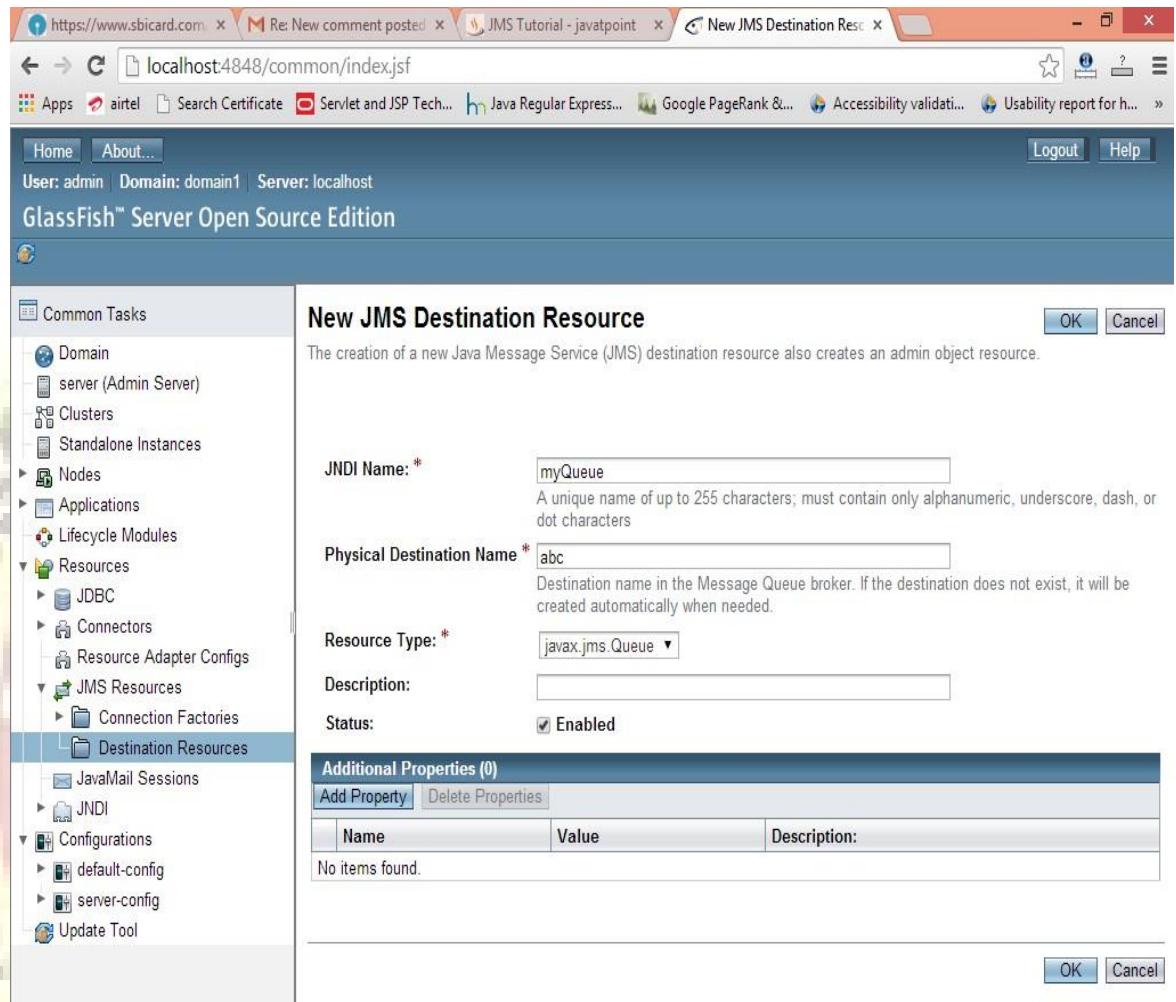
Maximum Pool Size: Connections
Maximum number of connections that can be created to satisfy client requests

Pool Resize Quantity: Connections
Number of connections to be removed when pool idle timeout expires

Idle Timeout: Seconds
Maximum time that connection can remain idle in the pool

Max Wait Time: Milliseconds
Amount of time caller waits before connection timeout is sent

Click on the **JMS Resource -> Destination Resources -> New**, now write the JNDI name and physical destination name then click on ok button.



2) Create sender and receiver application

Let's see the Sender and Receiver code. Note that Receiver is attached with listener which will be invoked when user sends message.

File: *MySender.java*

1. **import** java.io.BufferedReader;
2. **import** java.io.InputStreamReader;
3. **import** javax.naming.*;
4. **import** javax.jms.*;

```

6. public class MySender {
7.     public static void main(String[] args)
8.     try
9.         { //Create and start connection
10.            InitialContext ctx=new InitialContext();
11.            QueueConnectionFactory f=(QueueConnectionFactory)ctx.lookup("myQueueConnection
Factory");
12.            QueueConnection con=f.createQueueConnection();
13.            con.start();
14.            //2) create queue session
15.            QueueSession ses=con.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
16.            //3) get the Queue object
17.            Queue t=(Queue)ctx.lookup("myQueue");
18.            //4)create QueueSender object
19.            QueueSender sender=ses.createSender(t);
20.            //5) create TextMessage object
21.            TextMessage msg=ses.createTextMessage();
22.
23.            //6) write message
24.            BufferedReader b=new BufferedReader(new InputStreamReader(System.in));
25.            while(true)
26.            {
27.                System.out.println("Enter Msg, end to terminate:");
28.                String s=b.readLine();
29.                if (s.equals("end"))
30.                    break;
31.                msg.setText(s);
32.                //7) send message
33.                sender.send(msg);
34.                System.out.println("Message successfully sent.");
35.            }
36.            //8) connection close
37.            con.close();

```

```

38.     }catch(Exception e){System.out.println(e);}
39.   }
40. }

```

File: MyReceiver.java

```

1.  import javax.jms.*;
2.  import javax.naming.InitialContext;
3.
4.  public class MyReceiver {
5.      public static void main(String[] args) {
6.          try{
7.              //1) Create and start connection
8.              InitialContext ctx=new InitialContext();
9.              QueueConnectionFactory f=(QueueConnectionFactory)ctx.lookup("myQueueConnection
Factory");
10.             QueueConnection con=f.createQueueConnection();
11.             con.start();
12.             //2) create Queue session
13.             QueueSession ses=con.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
14.             //3) get the Queue object
15.             Queue t=(Queue)ctx.lookup("myQueue");
16.             //4)create QueueReceiver
17.             QueueReceiver receiver=ses.createReceiver(t);
18.
19.             //5) create listener object
20.             MyListener listener=new MyListener();
21.
22.             //6) register the listener object with receiver
23.             receiver.setMessageListener(listener);
24.
25.             System.out.println("Receiver1 is ready, waiting for messages...");
26.             System.out.println("press Ctrl+c to shutdown...");

```

```

27.     while(true){
28.         Thread.sleep(1000);
29.     }
30. }catch(Exception e){System.out.println(e);}
31. }
32.
33. }

```

File: MyListener.java

```

1. import javax.jms.*;
2. public class MyListener implements MessageListener {
3.
4.     public void onMessage(Message m) {
5.         try{
6.             TextMessage msg=(TextMessage)m;
7.
8.             System.out.println("following message is received:"+msg.getText());
9.         }catch(JMSEException e){System.out.println(e);}
10.    }
11. }

```

Run the Receiver class first then Sender class.

JMS Topic Example

It is same as JMS Queue, but you need to change Queue to Topic, Sender to Publisher and Receiver to Subscriber.

You need to create 2 JNDI named **myTopicConnectionFactory** and **myTopic**.

File: MySender.java


```
1. import java.io.BufferedReader;
2. import java.io.InputStreamReader;
3. import javax.naming.*;
4. import javax.jms.*;
5.
6. public class MySender {
7.     public static void main(String[] args) {
8.         try
9.             { //Create and start connection
10.                InitialContext ctx=new InitialContext();
11.                TopicConnectionFactory f=(TopicConnectionFactory)ctx.lookup("myTopicConnectionF
actory");
12.                TopicConnection con=f.createTopicConnection();
13.                con.start();
14.                //2) create queue session
15.                TopicSession ses=con.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
16.                //3) get the Topic object
17.                Topic t=(Topic)ctx.lookup("myTopic");
18.                //4)create TopicPublisher object
19.                TopicPublisher publisher=ses.createPublisher(t);
20.                //5) create TextMessage object
21.                TextMessage msg=ses.createTextMessage();
22.
23.                //6) write message
24.                BufferedReader b=new BufferedReader(new InputStreamReader(System.in));
25.                while(true)
26.                {
27.                    System.out.println("Enter Msg, end to terminate:");
28.                    String s=b.readLine();
29.                    if (s.equals("end"))
30.                        break;
```

```

31.     msg.setText(s);
32.     //7) send message
33.     publisher.publish(msg);
34.     System.out.println("Message successfully sent.");
35.     }
36.     //8) connection close
37.     con.close();
38.     }catch(Exception e){System.out.println(e);}
39. }
40. }

```

File: MyReceiver.java

```

1.  import javax.jms.*;
2.  import javax.naming.InitialContext;
3.
4.  public class MyReceiver {
5.      public static void main(String[] args) {
6.          try {
7.              //1) Create and start connection
8.              InitialContext ctx=new InitialContext();
9.              TopicConnectionFactory f=(TopicConnectionFactory)ctx.lookup("myTopicConnectionF
actory");
10.             TopicConnection con=f.createTopicConnection();
11.             con.start();
12.             //2) create topic session
13.             TopicSession ses=con.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
14.             //3) get the Topic object
15.             Topic t=(Topic)ctx.lookup("myTopic");
16.             //4)create TopicSubscriber
17.             TopicSubscriber receiver=ses.createSubscriber(t);
18.
19.             //5) create listener object

```



```

20. MyListener listener=new MyListener();
21.
22. //6) register the listener object with subscriber
23. receiver.setMessageListener(listener);
24.
25. System.out.println("Subscriber1 is ready, waiting for messages...");
26. System.out.println("press Ctrl+c to shutdown...");
27. while(true){
28.     Thread.sleep(1000);
29. }
30. }catch(Exception e){System.out.println(e);}
31. }
32.
33. }

```

File: MyListener.java

```

1. import javax.jms.*;
2. public class MyListener implements MessageListener {
3.
4.     public void onMessage(Message m) {
5.         try{
6.             TextMessage msg=(TextMessage)m;
7.
8.             System.out.println("following message is received:"+msg.getText());
9.         }catch(JMSEException e){System.out.println(e);}
10.    }
11. }

```
