SATHYABAMA UNIVERSITY

(Established under Section 3, UGC Act 1956)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

SCSX1022 J2EE

SCSX1022_J2EE

UNIT I

J2EE Platform

Java is one of the most mature and commonly used programming languages for building enterprise software. Over years java has evolved into three different platform editions namely:

The Java 2 Platform, Standard Edition (J2SE): Most commonly used platform, consisting of a run time environment and a set of APIs for building a wide range of applications that run various platforms and client applications for various enterprise applications.

The Java 2 Platform, Enterprise Edition (J2EE): J2EE is a platform for building server-side applications. It is an industry standard for developing and deploying the enterprise application. It was introduced in 1998 and released in 1990 dec.

The Java 2 Platform, Micro Edition (J2ME): J2ME, the latest edition enables building java application for micro devices such as mobile, set of boxes etc.

Programming for the Enterprise:

Java, which came in 1995, has an independent feature containing OOPS Concept which is useful in interacting with data application. It has replaced several proprietary and non-standard technologies as the preferred choice for building e-commerce and other web based enterprise applications. Today, J2EE is one of the two available alternatives for building e-commerce applications-the other being Microsoft's Window and .Net based technologies.

Enterprise Today:

An enterprise means a business organisation, and enterprise applications are those software applications that facilitate various activities in an enterprise.

Building applications for the enterprise has always been challenging. Some factors that contribute to this challenge and complexity are:

Diversity of information needs: In an enterprise, information is created and consumed by various users in number of different forms, depending on specific needs. It is very common to find that each business activity may process the same information in different forms.

Complexity of business processes: Most of the enterprise business processes involve complex information capture, processing and sharing. This leads to complex technical and architectural requirements for building enterprise applications.

Diversity of applications: Due to the complex nature of enterprise business processes, it is common to find that an enterprise consists of a large number of application each built at various times to fulfill different needs of various business process.

Over years, these challenges have taken monstrous shapes. In order to maintain a competitive edge, the adoptions of new technologies are gaining more importance. Some basic requirements to be met by any developers are:

- Programming productivity-Ability to develop and deploy applications
- Reliability-Ability to withstand/maintain the market at the time of release.
- Security-Make it impossible for hackers.
- Scalability-Ability to accommodate current changes in project.
- Integration-Ability to accommodate new features.

Enterprise Architecture Styles:

These architectural styles are quite common in today's enterprises. The Systems are generally data-driven with the server most of the times being a database server, and the client being a graphical user interface to operate on the data. There are various types of architecture styles.

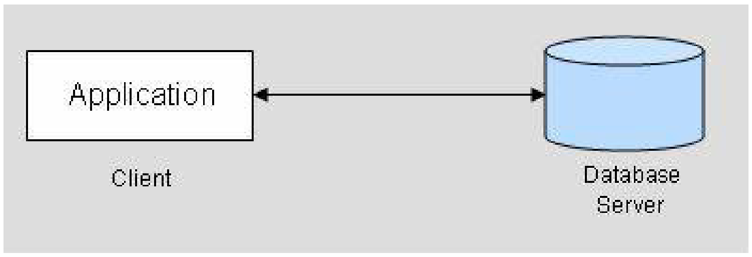
Tier is an application which is handled row by row or column by column.

(i)Two-Tier Architecture:

In traditional two tier application, the processing load is given to the client Pc, whereas the server acts only as a traffic controller between the application and data. When the entire application is processed on a PC, the application is forced to make multiple requests for data, thereby heavily tax the network.

Advantages:

The advantage of the two-tier design is its simplicity.

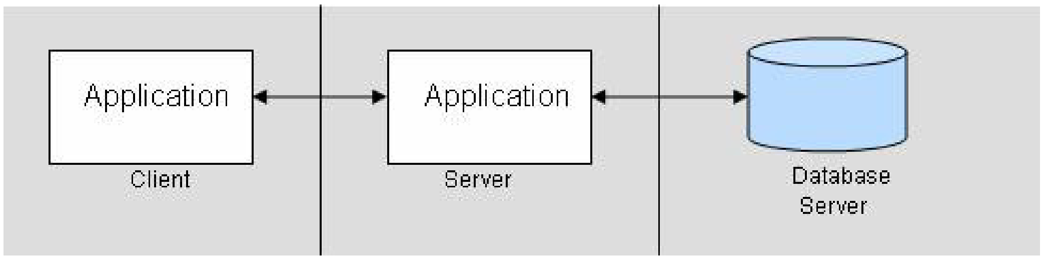


Drawbacks:

- Client send request to server to clear the traffic and most of the time is wasted in request-response processing
- Easy to deploy but difficult to enhance.
- Not scalable, not suitable for internet.

(ii) Three-Tier Architecture:

This application is broken up into three separate logical layers, each with a well defined set of interfaces.



- **Presentation Layer:**

It consists of a graphical user interface of some kind. It displays the information related to services as browsing, purchasing etc. For communicating with other tiers, it outputs the result to the browser and other tiers in the network.

- **Business/Application/Middle/Logic Layer:**

It consists of application or business logic. The business layer controls the functionality by performing the detailed processing. It is basically the code that the user calls upon to retrieve the data. The presentation layer then receives the data and formats for display.

- **Data Layer:**

  It contains the data needed for the application. This data can consists of any source of information including enterprise database such as Oracle, set of XML documents, or a directory service such as LDAP server. It keeps the data neutral and independent from the other two layers.

Advantages and Disadvantages:

The three-tier Web application architecture offers the following advantages:

- High performance, lightweight persistent objects
- High degree of flexibility in deployment platform and configuration

The disadvantage of this architecture is it is less standard than EJB.

(iii) N-Tier Architecture:

This application is responsible for representing user interface to end users for communicating business logic tier. The application logic is logically divided by function rather than physically. It is broken down into:

- **User Interface:**

It handles the user's interaction with the application. It can be a web browser running through a firewall, a heavier desktop application or even a wireless device.

- **Presentation Logic:**

  Defines what the user interface displays and how a user's requests are handled.

- **Business Logic:**

  It models the application's business rules, often through interaction with the application's data.

- **Infrastructure Services:**

Provides additionally functionality required by the application components, such as messaging, transactional support.

- **Data Layer:**

  It stores the enterprise data which can also be retrieved to the business layer.

J2EE Architecture Overview

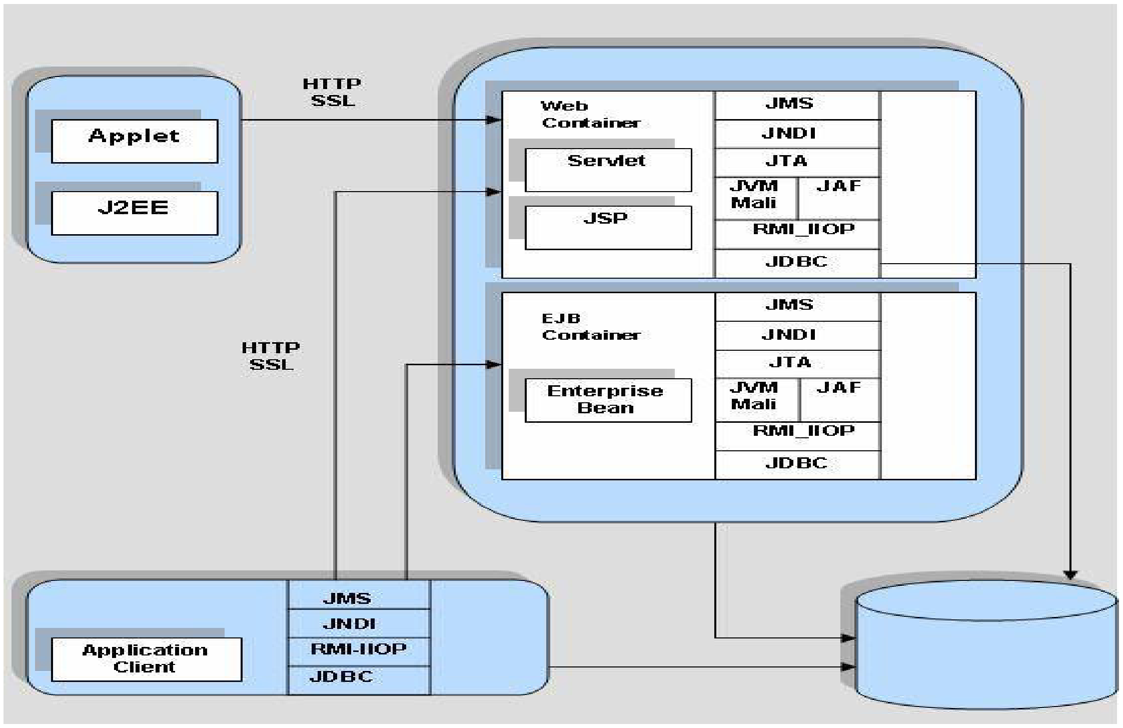
J2EE architecture is essentially a distributed architecture for developing multi-tier enterprise applications. Here, applications are divided into multiple tiers: the client-tier to provide a user interface, middle-tier modules to provide services and business logic, and backend EIS to provide data management. Figure illustrates the typical J2EE environment.

N-tier application architecture provides a model for developers to create a flexible and reusable application. By breaking up an application into tiers, developers only have to modify or add a specific layer, rather than have to rewrite the entire application over, if they decide to change technologies or scale up. In the term "N-tier," "N" implies any number -- like 2-tier, or 4-tier; basically, any number of distinct tiers used in your architecture.

Applications based on this architecture are employing the Model-View-Controller Pattern. Model is the data separated from how the information is presented. View represents data presentation. Controller is the application/business logic that controls the flow of information. Thus an application is designed based on these three functional components.

J2EE Architecture

The complete J2EE architecture is illustrated below.



Enterprise Architecture:

Enterprise Architecture is basically just n-tier architecture where we need to simply extend the middle tier by allowing for multiple application objects rather than just a single application. These application objects must each have an interface that allows them to work together. The interfaces used should be **generic** and only the objects are modified, not the interfaces, making it simple and quick.

With enterprise architecture, we can have multiple applications using a common set of components across an organization. This promotes the standardization of business practices. If a business rule changes, then the changes have to be made to the relevant “business object” and if necessary to the associated interface and the subsequent objects accessing the interface.

J2EE Platform:

The J2EE platform is essentially a distributed application-server environment- a java environment that provides the following:

- A set of java extension APIs to build applications. These APIs define a programming model for J2EE applications.
- A run-time infrastructure for hosting and managing applications. This is the server runtime in which application resides.

J2EE Runtime:

- Applications use interfaces. There is a clear demarcation (limit) between applications and runtime infrastructure.
- J2EE is more flexible to build applications including long-term and short-term demands. The long term demands are maintained and reusable. The short term demands are short lived internet designs.
- The layers in J2EE architecture are loosely coupled.
- It is highly extendable implementation.
- J2EE architecture provides uniform means of accessing platform-level services via its runtime environment.
- Before J2EE, distributed computing was client-server based.
  Server-implements interface, Client-connects to server.
- CORBA is an example of distributed application. It has interfaces using IDL(Interface Defining Language) to generate stubs and skeletons.

- Enterprise services need the following services- Transaction processing, database access, messaging and multithreading.
- To access the above services, we need plumbing code or middleware solutions using APIs.
- J2EE does not specify nature and structure of runtime, instead it introduces container.

J2EE APIs:

A typical commercial J2EE platform includes one or more containers, and access to the enterprise APIs is specified by the J2EE. The platform includes a set of Java standard extensions that each J2EE platform must support.

- **JDBC 2.0**
  Improves the standard JDBC 2.0 by adding more efficient means of obtaining connections, connection pooling distributed transactions etc.
- **Enterprise JavaBeans (EJB) 2.0**
  Component framework for multi-tier distributed applications.
- **Java Servlets 2.3**
  Object oriented abstractions to build dynamic web applications.
- **JavaServer Pages (JSP) 1.2**
  Provides template driven web application development
- **Java Message Service (JMS) 1.0**
  Provides message queuing and types of message oriented middleware services.
- **Java Transaction API (JTA) 1.0**
  Implements distributed transactional applications.
- **JavaMail 1.2**
  Platform-independent and protocol-independent framework to build java based applications
- **JavaBeans Activation Framework (JAF) 1.0**
  JavaMail uses JAF to determine contents of MIME (Multipurpose Internet Mail Extension) messages.
- **Java API for XML Parsing (JAXP) 1.1**

Parses XML Document Object Model (DOM) and XSLT Transformations (Style sheet Transform)

- **The Java Connector Architecture (JCA) 1.0**

  Integrates J2EE to legacy information systems.
- **Java Authentication and Authorization Service (JAAS) 1.0**

  Provides authentication and authorization mechanisms.

J2SE APIs:

- **Java Interface Definition Language (IDL) API**

  Invokes CORBA (Common Object Request Broker Architecture) objects via IIOP (Internet Inter-ORB Protocol).
- **JDBC Core API**

  Provides basic database programming facilities.
- **Java Remote Method Invocation (RMI)-IIOP API**

  Bridges the gap between RMI & CORBA Applications.
- **Java Naming and Directory Interface (JNDI) API**

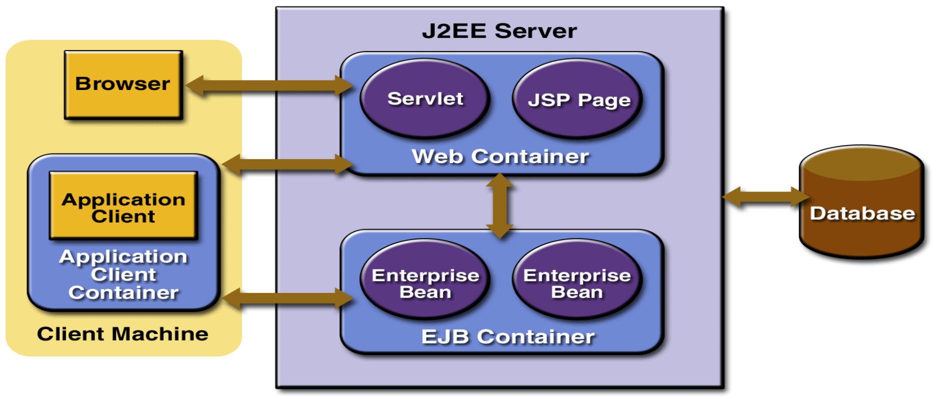
  Acts as SPI (Service Provider Interface) to J2SE.

J2EE Architecture

Containers:

A J2EE Container is a runtime to manage application components developed according to the API specifications, and to provide access to the J2EE APIs.

J2EE components are also called Managed Objects since they are created and managed within container runtime.



The J2EE Architecture contains various types of containers:

- A **web container** foe hosting Java Servlets and JSP pages
- An **EJB container** for hosting Enterprise JavaBean components
- An **applet container** for hosting Java applets
- An **application client container** for hosting standard Java applications

(i)Applet Container:

- Container consists of Java program which is static and embedded within the HTML pages.
- Used to manage the construction and the execution of applets.
- Consists of web browser and java plug in the runtime environment.

(ii)Application Container:

- Interface between J2EE application client and J2EE server.
- Runs on client machine.
- Acts as the gateway between the client application and java double EE application

(iii)EJB Container (Enterprise Java Bean):

- Interface between the enterprise bean which provides the business logic in J2EE application and J2EE server.
- Mainly runs on J2EE server.
- Manages the execution of application enterprise beans.

(iv)Web Container:

- Acts as the interface between the web components and web server.
- The main components of web containers are JSP and Servlet.
  JSP- It is a mechanism for server sides scripting between the web pages.
  Servlet - It is one of the paths of java programming that executes on web server in response to http request similar to CGI (Computer Graphical Interface). The servlets produced HTML outputs in the http response and returns by invoking java object methods on server side without returning any text.

In this architecture, there are primarily two types of clients:

(i)Web Clients:

- Executed in the Web browsers.

- For these clients, UI generated on server side as HTML/XML.
- Use http to communicate with web containers.
- Application components in web containers include Java Servlets. JSP which implements functionality required by web clients.
- Web containers are responsible for accepting requests from web clients & generating response with help of application components.

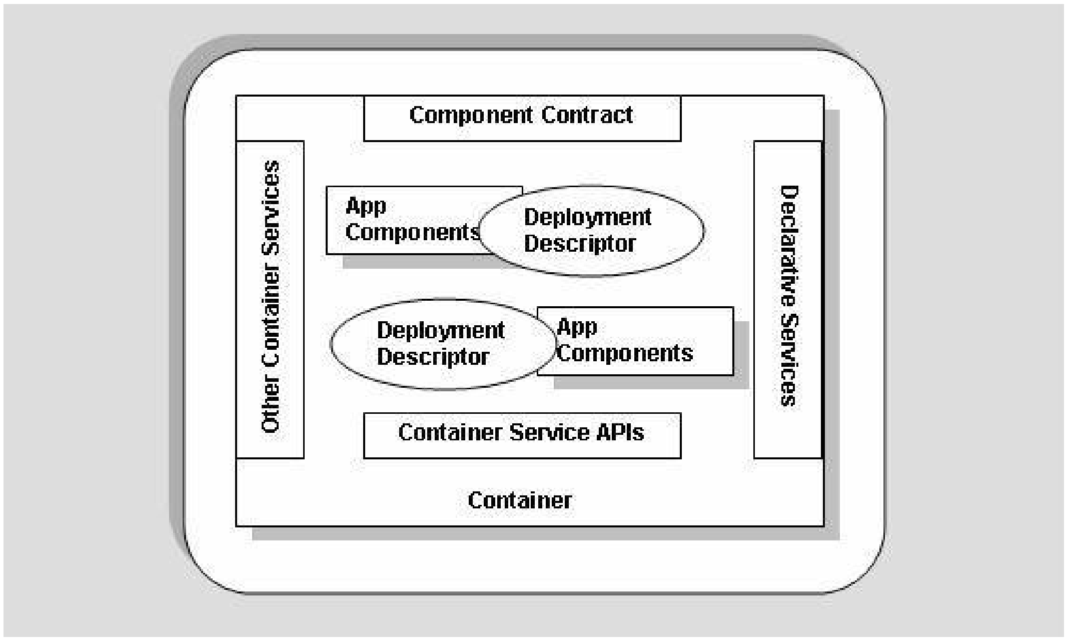
(ii)EJB Clients:

- Applications that access EJB components or EJB containers
- There are three types of EJB Clients
    1. Application Clients that are stand alone applications accessing EJB components using RMI-IIOP Protocol
    2. Components in web container- E.g. Java servlets, JSP also access EJB components using RMI-IIOP Protocol
    3. Other EJBs running within the EJB container-communicate via standard Java method calls via local interface.

Container Architecture:

The J2EE specification defines a container as being responsible for providing the runtime support for the application components. The container acts as an intermediary between the J2EE application and the services provided by the J2EE server, to give a federated view of APls for the application components. The container provides the APls to application components for accessing services. It also handles security, resource pooling, and state management.

Figure shown below illustrates the container architecture.



(i)Application Components:

Application components include Servlets, JSP, and EJB. These can be packaged into archive files.

(ii)Deployment Descriptors:

Descriptor is an XML file that describes application components. It also Includes additional information required by containers.

Architecture of a container is divided into four parts:

(i)Component Contract:

- Set of APIs specified by the container, that application components are required to extend or implement.
- Basic purpose of container is to provide a runtime for application
- Instance of application components are created and invoked within JVM of the container.
- This makes container responsible for managing life cycle of application components& they are required to abide by certain contracts specified by the container.
- J2EE application components are remote to client. Clients can't directly call methods on components. It makes request to application server and it is container that actually invokes methods. Application components are required to follow contract specified by the container.

- All J2EE application components are managed which includes location, instantiation, pooling, initializing, service invocation and removal of components from service.

(ii)Container Service APIs:

- Additional services provided by the container, which are commonly required for all applications in the container.
- A container in the J2EE architecture provides a consolidated view of various enterprise APIs specified in the J2EE platform.
- Application components can access APIs via appropriate objects created a published in JNDI service or implementation.
- Loosely coupling between implementation and client.
- Uses delegation (code reusability) instead of inheritance

(iii)Declarative Services:

- Services that the container interposes on applications, based on the deployment description provided for each application component, such as security, transactions etc.
- A deployment descriptor defines contract between container and component. As application developers, we are required to specify a deployment descriptor for each group of application components
- Two methods for invoking services:
  1. Explicit invocation
     Standard method. Eg. In DBMS, we explicitly commit/rollback queries.
  2. Declarative invocation
     These are not explicitly invoked. Instead we specify business transactions (Start, Stop etc) in deployment descriptor & the container will automatically start a transaction. Declarative Service is a service performed on the container on our behalf.

Advantage:

We can place new services without changing application components. So decisions can be postponed to runtime instead of design time.

(iv)Other container services:

- Lifecycle Management of application components

- Resource pooling
- Populating JNDI namespace based on deployment names associated with EJB components
- Populating JNDI namespaces with objects necessary for utilizing container service APIs
- Clustering-enhances availability and scalability of applications.

J2EE Technologies:

This large collection, of quite disparate technologies, can be divided according to use as follows:

- **The Component Technologies:** There are three types of components: JSP pages, servlets and Enterprise JavaBeans.
- **The Service Technologies:** These technologies provide the application's component with supported services to function efficiently.
- **The Communication Technologies:** These technologies provide the mechanism for communication among different parts of the application, whether they are local or remote

(i)Component Technologies:

It is further divided into two types:

- Web components
  - (i) Components respond to http request
  - (ii) Extends the functionality of web server to enable dynamic content in HTML
  - (iii) JSP- Extension of Java Servlet programming. When user request JSP pages, web container compiles JSP page into servlet.JSP page provide powerful & dynamic page assemble mechanism.
- EJB components
  - (i) EJB 2.0 is a distributed model for developing secure, scalable, transactional and multi-users components.
  - (ii) EJBs are reusable software units containing business logic
  - (iii) EJB allow separation of application logic from system level services.
  - (iv) Three types of beans: Entity Beans, Session Beans and Message Driven Beans

(ii)The Service Technologies:

- JDBC- Database connectivity, RDBMS
- JTA (Java Transaction API) & JTS (Java Transaction Services)-for transaction in distributed applications. They are container controlled so developers need not be concerned about it
- JNDI-provides means to perform standard operations on directory service resources such as LDAP (Lightweight Directory Access Protocol), Novell Directory Services and Netscape Directory Services.
- JMS-Mechanism of sending message asynchronously. It provides such functionality with the help of MOM (Message Oriented Middleware)
- JavaMail-e-mail sending and receiving. It supports internet protocols like SMTP, IMAP (Interactive Mail Access Protocol), POP3 (Post Office Protocol)
- JCA (Java Connector Architecture)-Standardised means by which J2EE applications can access variety of legacy applications mostly ERP systems. Plug and play components allow us to access legacy system without having to know too much specific about how to work with it.
- JAAS-means to grant permission based on who is executing the code. It utilizes pluggable architecture of authentication modules, such as Kerberos or PKI.

(iii)The Communication Technologies:

It is further divided into:

- Internet Protocol
    - (i) HTTP-stateless application level protocol
    - (ii) TCP/IP-separate modules combined into single entity. IP takes care of data received on both endpoints over internet. TCP takes care of error checking on packets.
    - (iii) SSL (Secure Socket Layer)-uses cryptography to encrypt information across client/server and provides authentication.
- Remote Object Protocols
    - (i) RMI-One of the primary mechanisms and allows us to use interfaces to define remote objects.

(ii) RMI-IIOP-extension of RMI. Allows us to use interface to define remote object that can be implemented in any language that supports OMG mapping

- JavaIDL

  Java client can invoke method calls on CORBA objects. They need not be written in java but implement an IDL defined interface.

Java Naming and Directory Services:

(i)Naming Services:

A service that enables creation of standard name for given set of data. In internet, each host has a FQDN

FQDN (Fully Qualified Domain Name).

(ii)Directory Services:

It is a special type of database that is optimised for read access by using various indexing, caching and disk access technique. The information in directory service is described using a hierarchial information model.

Advantage: Allows more flexible searching

One commonly used directory service is the Domain Naming Service (DNS) used in the internet.

General Purpose directory services used by Enterprise:

- Novell Directory Services (NDS)
- Network Information Services (NIS/NIS+)
- Active Directory Services (ADS)
- Windows NT Domains

LDAP (Lightweight Directory Access Protocol):

In 1990, it was the standard directory protocol. JNDI can access LDAP. LDAP defines how clients should access data in server. It does not specify how data is stored in server.

LDAP Data:

Data in LDAP is organised in a hierarchical tree called DIT (Directory Information Tree). Each leaf is called an entry and the first entry is called the root entry. An entry comprises a DN (Distinguished Name) and any number of attribute-value pair. The left most part of a DN is called a Relative Distinguished Name (RDN).

LDIF (LDAP Data Interchange Format):

It is human-readable. Attributes also have matching rules.

Matching Rules:

- DN-Attribute is in the form of a Distinguished Name
- Case Insensitive String (CIS)-Attribute can match if the value of the query equals the attribute's value, regardless of case.
- Case Sensitive String (CSS)-Attribute can match if the value of the query equals the attribute's value including the case.
- Telephone-Is the same as CIS except that the characters like "-" and "()" are ignored when determining the match.
- Integer-Attribute match is determined using only numbers.
- Binary-Attribute matches if the value of the query and the value of the attribute are the same binary values.

Database Programming with JDBC/ODBC Bridge:

Database Driver:

Provides set of APIs for accessing data managed by data server. eg. Vendors: Oracle

JDBC Driver:

JDBC-ODBC Bridge:

Open Database Connectivity was originally created to provide an API standard for SQL on Microsoft Windows platform.

JDBC drivers are divided into four types or levels. The **different types of jdbc drivers** are:

Type 1: JDBC-ODBC Bridge driver (Bridge)

Type 2: Part Native-API/partly Java driver (Native)

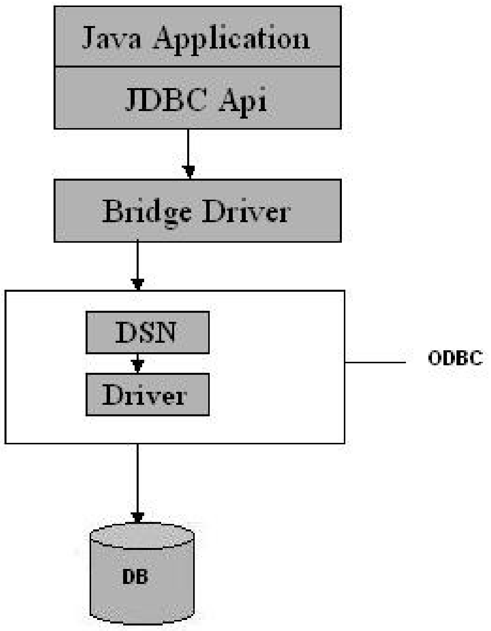
Type 3: Intermediate Database Access Server (Middleware)

Type 4: Pure Java/Native-protocol driver (Pure)

Type 1 JDBC Driver

<u>JDBC-ODBC Bridge driver</u>

The Type 1 driver translates all JDBC calls into ODBC calls and sends them to the ODBC driver. ODBC is a generic API. The JDBC-ODBC Bridge driver is recommended only for experimental use or when no other alternative is available.



Advantage

The JDBC-ODBC Bridge allows access to almost any database, since the database's ODBC drivers are already available.

Disadvantages

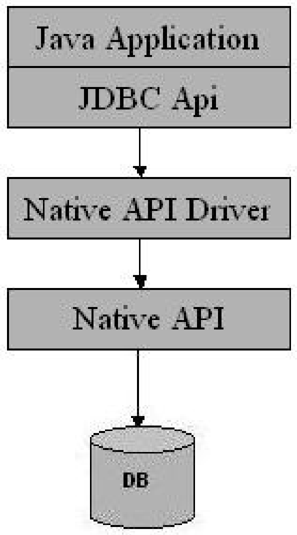
1. Since the Bridge driver is not written fully in Java, Type 1 drivers are not portable.
2. A performance issue is seen as a JDBC call goes through the bridge to the ODBC driver, then to the database, and this applies even in the reverse process. They are the slowest of all

driver types.
3. The client system requires the ODBC Installation to use the driver.
4. Not good for the Web.

Type 2 JDBC Driver

<u>Part Native-API/partly Java driver</u>

The distinctive characteristic of type 2 JDBC drivers is that Type 2 drivers convert JDBC calls into database-specific calls i.e. this driver is specific to a particular database. Some distinctive characteristic of type 2 JDBC drivers are shown below. Example: Oracle will have oracle native API.



Advantage

The distinctive characteristic of type 2 JDBC drivers are that they are typically offer better performance than the JDBC-ODBC Bridge as the layers of communication (tiers) are less than that of Type 1 and also it uses Native API which is Database specific.

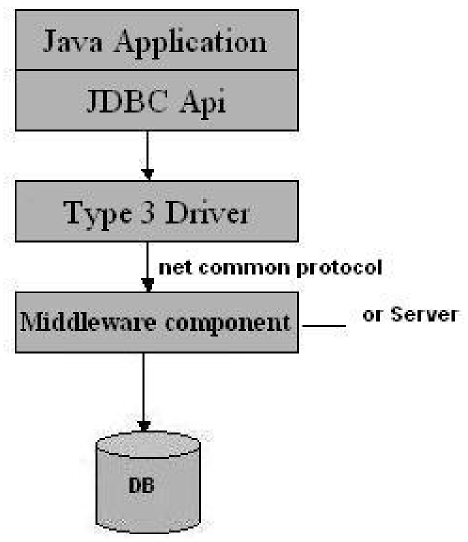
Disadvantage

1. Native API must be installed in the Client System and hence type 2 drivers cannot be used for the Internet.
2. Like Type 1 drivers, it's not written in Java Language which forms a portability issue.
3. If we change the Database we have to change the native api as it is specific to a database
4. Mostly obsolete now
5. Usually not thread safe.

Type 3 JDBC Driver

<u>Intermediate Database Access Server</u>

Type 3 database requests are passed through the network to the middle-tier server. The middle-tier then translates the request to the database. If the middle-tier server can in turn use Type1, Type 2 or Type 4 drivers.



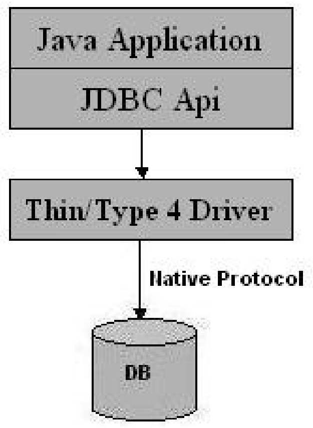
Advantage

1. This driver is server-based, so there is no need for any vendor database library to be present on client machines.
2. This driver is fully written in Java and hence Portable. It is suitable for the web.
3. There are many opportunities to optimize portability, performance, and scalability.
4. The net protocol can be designed to make the client JDBC driver very small and fast to load.
5. The type 3 driver typically provides support for features such as caching (connections, query results, and so on), load balancing, and advanced system administration such as logging and auditing.
6. This driver is very flexible allows access to multiple databases using one driver.
7. They are the most efficient amongst all driver types.

Disadvantage: It requires another server application to install and maintain. Traversing the record set may take longer, since the data comes through the backend server.

Type 4 JDBC Driver

Pure Java/Native-protocol driver The Type 4 uses java networking libraries to communicate directly with the database server.



Advantage

1. The major benefit of using a type 4 jdbc drivers are that they are completely written in Java to achieve platform independence and eliminate deployment administration issues. It is most suitable for the web.
2. Number of translation layers is very less i.e. type 4 JDBC drivers don't have to translate database requests to ODBC or a native connectivity interface or to pass the request on to another server, performance is typically quite good.
3. You don’t need to install special software on the client or server. Further, these drivers can be downloaded dynamically.

Disadvantage

With type 4 drivers, the user needs a different driver for each database.

The java.sql Package

The classes in the java.sql package can be divided into the following groups based on their functionality.

Connection Management

Database Access

Data Types

Database Metadata

Exceptions and Warnings.

JDBC Connection:

java.sql.Connection

JDBC 2.0 Connection Methods Supported

Returns	**Signature**
Statement	createStatement(int resultSetType, int resultSetConcurrency)
PreparedStatement	prepareStatement(String sql, int resultSetType, int resultSetConcurrency)
CallableStatement	prepareCall(String sql, int resultSetType, int resultSetConcurrency

Before you can create a java jdbc connection to the database, you must first import the java.sql package.

import java.sql.*; The star (*) indicates that all of the classes in the package java.sql are to be imported.

1. Loading a database driver

In this step of the jdbc connection process, we load the driver class by calling Class.forName() with the Driver class name as an argument. Once loaded, the Driver class creates an instance of itself. A client can connect to Database Server through JDBC Driver. Since most of the Database servers support ODBC driver therefore JDBC-ODBC Bridge driver is commonly used.
The return type of the Class.forName (String ClassName) method is "Class". Class is a class in
java.lang package.

```
 try {
          Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); //Or any other driver
}
catch(Exception x){
```

```
            System.out.println( “Unable to load the driver class!” );

}
```

2.Creating a oracle jdbc Connection

The JDBC DriverManager class defines objects which can connect Java applications to a JDBC driver. DriverManager is considered the backbone of JDBC architecture. DriverManager class manages the JDBC drivers that are installed on the system. Its getConnection() method is used to establish a connection to a database. It uses a username, password, and a jdbc url to establish a connection to the database and returns a connection object. A jdbc Connection represents a session/connection with a specific database. Within the context of a Connection, SQL, PL/SQL statements are executed and results are returned. An application can have one or more connections with a single database, or it can have many connections with different databases. A Connection object provides metadata i.e. information about the database, tables, and fields. It also contains methods to deal with transactions.

JDBC URL Syntax:: jdbc: <subprotocol>: <subname>

JDBC URL Example:: jdbc: <subprotocol>: <subname>•Each driver has its own subprotocol •Each subprotocol has its own syntax for the source. We’re using the jdbc odbc subprotocol, so the DriverManager knows to use the sun.jdbc.odbc.JdbcOdbcDriver.

```
try{
 Connection dbConnection=DriverManager.getConnection(url,”loginName”,”Password”)
}
catch( SQLException x ){
            System.out.println( “Couldn’t get connection!” );

}
```

3. Creating a jdbc Statement object,

Once a connection is obtained we can interact with the database. Connection interface defines methods for interacting with the database via the established connection. To execute SQL statements, you need to instantiate a Statement object from your connection object by using the createStatement() method.

Statement statement = dbConnection.createStatement();

A statement object is used to send and execute SQL statements to a database.

Three kinds of Statements

Statement: Execute simple sql queries without parameters.
Statement createStatement()
Creates an SQL Statement object.

Prepared Statement: Execute precompiled sql queries with or without parameters.
PreparedStatement prepareStatement(String sql)
returns a new PreparedStatement object. PreparedStatement objects are precompiled SQL statements.

Callable Statement: Execute a call to a database stored procedure.
CallableStatement prepareCall(String sql)
returns a new CallableStatement object. CallableStatement objects are SQL stored procedure call statements.

4. Executing a SQL statement with the Statement object, and returning a jdbc resultSet.

Statement interface defines methods that are used to interact with database via the execution of SQL statements. The Statement class has three methods for executing statements: executeQuery(), executeUpdate(), and execute(). For a SELECT statement, the method to use is executeQuery . For statements that create or modify tables, the method to use is executeUpdate. Note: Statements that create a table, alter a table, or drop a table are all examples of DDL
statements and are executed with the method executeUpdate. execute() executes an SQL statement that is written as String object.

ResultSet provides access to a table of data generated by executing a Statement. The table rows are retrieved in sequence. A ResultSet maintains a cursor pointing to its current row of data. The next() method is used to successively step through the rows of the tabular results.

ResultSetMetaData Interface holds information on the types and properties of the columns in a ResultSet. It is constructed from the Connection object.

/*JDBC CONNECTIVITY*/

```
import java.io.*;

import java.sql.*;

public class jdbcc

{

public static void main(String args[]) throws IOException

{

try

{

        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

        Connection cn=DriverManager.getConnection("jdbc:odbc:mydsn");

        System.out.println("Connected...");

        Statement st=cn.createStatement();

        st.executeUpdate("Insert into emp values(3,'arun',22,10000,'chennai')");

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

        while(rs.next())

        {

        System.out.print(rs.getInt(1)+rs.getString(2)+"\t"+rs.getInt(3)+"\t"+rs.getInt(4)+"\t"+r
s.getString(5)+"\t");

        System.out.println();

        }

        cn.close();

}
catch(Exception e)

{

        System.out.println(e);

}  }  }

/*  OutPut : */

Z:\j2ee>java jdbcc

Connected...

1 akash 20      200     hydr

2 raj    23      200     mysore

3 arun   22      10000   chennai

Z:\j2ee>          */
```

Scrollable result sets, updatable result sets

```
package com.jdbctest.ScrollResult;
import java.sql.*;

import com.jdbctest.Util.JDBCUtil;

public class ScrollResult {

    /** Scrollable result set
     * @param args
     */
    public static void main(String[] args) {

        Connection conn = null;
        Statement stmt = null;
        ResultSet rs = null;

        conn = JDBCUtil.getConnection();
        try {// Set to become a scrollable, updatable
            stmt     =     conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);
            String sql = "SELECT id,sno,sname,score  FROM score";
            rs = stmt.executeQuery(sql);

   /*********** The scroll of the result set  **********************/
            // Navigate to the second record
            rs.absolute(1);
```

```
                print(rs);
                // Backward scrolling a
                //rs.previous();
                print(rs);
                // Navigate to the last record
                rs.last();
                // Navigate to the first
                print(rs);
                rs.first();

                rs.beforeFirst();
                while(rs.next()){
                    print(rs);
                }

/************ Update the data in the result set  ******************/
              System.out.println(" Update data  -----------------");
                rs.absolute(1);
                rs.updateString("score", "86");
                rs.updateRow();

                rs.beforeFirst();
                while(rs.next()){
                    print(rs);
                }

              System.out.println(" Insert data  -----------------");

                              rs.moveToInsertRow();

                rs.updateString("sname", " Wu rain  ");
                rs.updateString("sno", "200608016");
                rs.updateDouble("score", 87);
                //rs.updateString("birthday", "1996-1-26");
                rs.insertRow();
                rs.beforeFirst();
                while(rs.next()){
                    print(rs);
                }
        System.out.println(" Delete data  -----------------------------");
                rs.absolute(3);
                rs.deleteRow();

                // The way to the front
                rs.beforeFirst();
                while(rs.next()){
                    print(rs);
                }
            } catch (SQLException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            }

        private static void print(ResultSet rs) throws SQLException {
            System.out.println(rs.getString("sno"));
            System.out.println(rs.getString("sname"));
            System.out.println(rs.getDouble("score"));
            System.out.println("****************************");
        }
    }
```

Executing SQL Statements:

JDBC Execute Statement is used to execute SQL Statement, The Execute Statement accept SQL object as parameter and return you the result set from a database.

st.execute ( ): This method is used to execute the SQL query. This may return you a set of row into your table of database.

```
String sql = "create table Abc(no integer,name varchar(10))"
        st.execute(sql);
```

St.executeQuery ( ): This is used to return the result set obtained from the record set of the database. The select statement is used to retrieve the result set from the database.

```
String sql = "select * from person";
   rs = st.executeQuery(sql);
```

```
import java.sql.*;

public class JdbcExecuteStatement {

   public static void main(String args[]) {

      Connection con = null;

      Statement st = null;

      String url = "jdbc:mysql://localhost:3306/";

      String db = "komal";

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

      String user = "root";

      String pass = "root";

      try {

         Class.forName(driver);
```

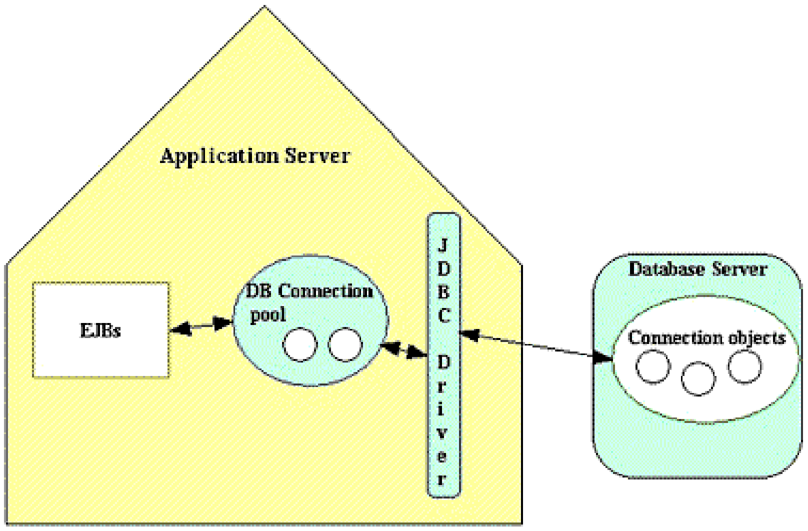
```
            con = DriverManager.getConnection(url + db, user, pass);

           st = con.createStatement();

            String sql = "create table Abc(no integer,name varchar(10))";

            st.execute(sql);

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

CONNECTION POOLING

Connection pooling being normally used in web-based and enterprise applications is usually handled by an application server. Any dynamic web page can be coded to open a connection and close it normally but behind the scenes when a new connection is requested, one is returned from the connection pool maintained by the application server. Similarly, when a connection is closed it is actually returned to the connection pool.

Connection pooling is a collection of database connections that is maintained by memory that can be reused. Once an application has finished its physical connection the connection is recycled rather than being destroyed.

The JDCConnectionPool.java class makes connections available to calling program in its getConnection method. This method searches for an available connection in the connection pool. If no connection is available from the pool, a new connection is created. If a connection is available from the pool, the getConnection method leases the connection and returns it to the calling program.


Application Server
EJBs
DB Connection pool
JDBC Driver
Database Server
Connection objects

<u>**SCSX1022_J2EE**</u>

UNIT II

Introduction to Web Containers:

In J2EE architecture the basic purpose of the container is to provide the runtime environment for the components. The web container is a Java runtime environment which is responsible for managing life cycle of JSP pages and Servlets. Servlets and JSP pages are collectively called web components. A web container is responsible for instantiating, initializing and invoking Servlets and JSP pages. The web container implements Servlet and JSP API and provide infrastructure for deploying and managing web components. The web container is part of web server or application server that provides the network services over which request and response are sent. All web containers must support HTTP protocol however it may support addition protocols like HTTPS. Web container is also called Servlet container or Servlet engine.

<u>The HTTP Protocol</u>

The Hypertext Transfer Protocol (HTTP) is an application-level TCP/IP based protocol with the lightness and speed necessary for distributed, collaborative, hypermedia information systems (internet). HTTP is a request-response standard typical of client-server computing. The HTTP protocol is a stateless protocol.

HTTP is a request/response protocol, which means your computer sends a request for some file (e.g. "Get me the file 'home.html'"), and the web server sends back a response ("Here's the file", followed by the file itself).

HTTP Request Methods

HTTP/1.0 allows an open-ended set of *methods* to be used to indicate the purpose of a request. The three most often used methods are GET, HEAD, and POST.

The <u>GET</u> Method

Information from a form using the **GET** method is appended onto the end of the action URI being requested. Your CGI program will receive the encoded form input in the environment variable **QUERY_STRING**.

The GET method is used to ask for a specific document - when you click on a hyperlink, GET is being used. GET should probably be used when a URL access will not change the state of a database (by, for example, adding or deleting information) and POST should be used when an access *will* cause a change. Many database searches have no visible side-effects and make ideal applications of query forms using GET.

The HEAD method

The HEAD method is used to ask only for information *about* a document, not for the document itself. HEAD is much faster than GET, as a much smaller amount of data is transferred. It's often used by clients who use caching, to see if the document has changed since it was last accessed. If it was not, then the local copy can be reused, otherwise the updated version must be retrieved with a GET.

The POST Method

This method transmits all form input information immediately after the requested URI. The POST method is commonly used for accessing dynamic resources. POST requests are typically used to transmit information that is request-dependent, or when large amount of complex information must be sent to the server.

HTTP Response

After receiving and interpreting a request message, a server responds with an HTTP response message.

```
Response      = Status-Line

                *(( general-header
                | response-header
                | entity-header ) CRLF)
                CRLF
```

[message-body]

Status-Line:

The first line of a Response message is the Status-Line, consisting of the protocol version followed by a numeric status code

Status Code and Reason Phrase:

The Status-Code element is a 3-digit integer result code of the attempt to understand and satisfy the request

Response Header Fields:

The response-header fields allow the server to pass additional information about the response which cannot be placed in the Status- Line. These header fields give information about the server and about further access to the resource identified by the Request-URI.

A Simple Web Application:

Servlet

A *servlet* is a Java programming language class that is used to extend the capabilities of servers that host applications access via a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers. For such applications, Java Servlet technology defines HTTP-specific servlet classes.

A simple Servlet Program

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

public class HelloWorldServlet extends HttpServlet
{


 public void doGet(HttpServletRequest req, HttpServletResponse res)
```

```
   throws ServletException, IOException
 {
  doPost(req,res);     //  Pass all GET request to the the doPost method
 }


 public void doPost(HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException
 {
  res.setContentType("text/html");   // Set the content type of the
                                        response
  PrintWriter out=res.getWriter();    //  PrintWriter to write text to
                                         the response
  out.println("Hello World");         //  Write Hello World
  out.close();                  //  Close the PrintWriter
 }
}
```

This servlet will produce a simple text response every time it is called by the webserver.

Servlet Exceptions

public class **UnavailableException**
extends ServletException

This exception indicates that a servlet is unavailable. Servlets may report this exception at any time, and the network service running the servlet should behave appropriately. There are two types of unavailability, and sophisticated services will to deal with these differently:

- *Permanent* unavailability. The servlet will not be able to handle client requests until some administrative action is taken to correct a servlet problem.
- *Temporary* unavailability. The servlet can not handle requests at this moment due to a system-wide problem. For example, a third tier server might not be accessible, or there may be insufficient memory or disk storage to handle requests. The problem may be self correcting, such as those due to excessive load, or corrective action may need to be taken by an administrator.

Network services may safely treat both types of exceptions as "permanent", but good treatment of temporary unavailability leads to more robust network services.

Constructors :

1. `public UnavailableException(Servlet servlet,
                              String msg)`

   Constructs a new exception with the specified descriptive message, indicating that the servlet is permanently unavailable.
   Parameters:
   servlet - the servlet which is unavailable
   msg - the descriptive message

vailableException

```
public UnavailableException(int seconds,
                            Servlet servlet,
                            String msg)
```

Constructs a new exception with the specified descriptive message, indicating that the servlet is temporarily unavailable and giving an estimate of how long it will be unavailable. In some cases, no estimate can be made; this is indicated by a non-positive time. For example, the servlet might know a server it needs is "down", but not be able to report how long it will take to restore it to an adequate level of functionality.

Parameters:

seconds - number of seconds that the servlet is anticipated to be unavailable. If negative or zero, no estimate is available.

servlet - the servlet which is unavailable

msg - the descriptive message

Methods

● **isPermanent**

```
public boolean isPermanent()
```

Returns true if the servlet is "permanently" unavailable, indicating that the service administrator must take some corrective action to make the servlet be usable.

● **getServlet**

```
public Servlet getServlet()
```

Returns the servlet that is reporting its unavailability.

● **getUnavailableSeconds**

```
public int getUnavailableSeconds()
```

Returns the amount of time the servlet expects to be temporarily unavailable. If the servlet is permanently unavailable, or no estimate was provided, returns a negative number. No effort is made to correct for the time elapsed since the exception was first reported.

Servlet Request Interface:

This interface defines an object that provides client request information to a servlet. The servlet container creates a request object and passes it as an argument to the servlet's service() method.

Public interface ServletRequest

A **Servlet::ServletRequest** object provides data including parameter name and values, attributes, and an input handle. Interfaces that extend ServletRequest can provide additional protocol-specific data (for example, HTTP data is provided by **Servlet::Http::HttpServletRequest**.

Methods:

Methods for Request Parameters:

getParameter($name)

Returns the value of a request parameter, or *undef* if the parameter does not exist. Request parameters are extra information sent with the request. For HTTP servlets, parameters are contained in the query string or posted form data.

getParameterValues($name)

Returns an array containing all of the values of the given request parameter, or *undef* if the parameter does not exist. If the parameter has a single value, the array has a length of 1. If the parameter has no value, the array is empty.

Methods for Request Attributes:

getAttribute($name)

Returns the value of the named attribute, or *undef* if no attribute of the given name exists.

getAttributeNames()

Returns an array containing the names of the attributes available to this request, or an empty array if the request has no attributes available to it.

setAttribute($name, $object)

Stores an attribute in this request. Attributes are reset between requests. This method is most often used in conjunction with **Servlet::RequestDispatcher**.

removeAttribute($name)

Removes an attribute from this request. This method is not generally needed as attributes only persist as long as the request is being handled.

Note: $name: The name of the attribute /parameter being passed.

$object: The object to be stored. Can be a scalar or a reference to an arbitrary data structure.

Methods for Input:

getReader()

Retrieves the body of the request as character data. The reader translates the character data according to the character encoding used on the body.

Servlet Response Interface:

This interface defines an object that assists a servlet in sending a response to the client. The servlet container creates the object and passes it as an argument to the servlet's service() method.

Public interface ServletResponse

Methods for Content Type and Length

setContentType($type)

Sets the content type of the response. The content type may include the type of character encoding used, for example *text/html; charset=ISO-8859-4*.

Parameters: ***$type:*** The MIME type of the content

setContentLength($len)

Sets the length of the content body in the response. In HTTP servlets, this method sets the HTTP *Content-Length* header.

Parameters: ***$len:*** The length of the content being returned to the client

Methods for Output

getWriter()

Returns a object that can send character text to the client. The character encoding used is the one specified in the *charset* parameter of setContentType(), which must be called before calling this method for the charset to take effect.

Methods for Buffered Output

getBufferSize()

Returns the actual buffer size used for the response, or 0 if no buffering is used.

resetBuffer()

Clears the content of the underlying buffer in the response without clearing headers or status code.

setBufferSize($size)

Sets the preferred buffer size for the body of the response. The servlet container will use a buffer at least as large as the size requested.

Parameters: ***$size:*** The preferred buffer size

Servlet Security

Security is the science of keeping sensitive information in the hands of authorized users. On the web, this boils down to three important issues:

Authentication

Being able to verify the identities of the parties involved

Confidentiality

Ensuring that only the parties involved can understand the communication

Integrity

Being able to verify that the content of the communication is not changed during transmission

A servlet can retrieve information about the server's authentication using two methods namely : getRemoteUser() and getAuthType(). The below example shows a simple servlet that tells the client its name and what kind of authentication has been performed (basic, digest, or some alternative). To see this servlet in action, web server should support with a basic or digest security scheme.

Snooping the authorization information:

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

public class AuthorizationSnoop extends HttpServlet {
  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    out.println("<HTML><HEAD><TITLE>Authorization
Snoop</TITLE></HEAD><BODY>");

    out.println("<H1>This is a password protected resource</H1>");
    out.println("<PRE>");
    out.println("User Name: " + req.getRemoteUser());
    out.println("Authorization Type: " + req.getAuthType());
    out.println("</PRE>");
    out.println("</BODY></HTML>");
  }
}
```

The below example shows a servlet that performs custom authorization, receiving an Authorization header and sending the SC_UNAUTHORIZED status code and WWW-Authenticate header when necessary. The servlet restricts access to its "top-secret stuff" to those users (and passwords) it recognizes in its user list. For this example, the list is kept in a simple Hashtable and its contents are hard-coded; this would, of course, be replaced with some other mechanism, such as an external relational database, for a production servlet.

Security in a servlet

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

public class CustomAuth extends HttpServlet {
```

```
Hashtable users = new Hashtable();

public void init(ServletConfig config) throws ServletException {
  super.init(config);
  users.put("Wallace:cheese",     "allowed");
  users.put("Gromit:sheepnapper", "allowed");
  users.put("Penguin:evil",       "allowed");
}

public void doGet(HttpServletRequest req, HttpServletResponse res)
                             throws ServletException, IOException {
      PrintWriter out = res.getWriter();

  // Get Authorization header
  String auth = req.getHeader("Authorization");

  // Do we allow that user?
  if (!allowUser(auth)) {
    // Not allowed, so report he's unauthorized
    res.setHeader("WWW-Authenticate", "BASIC realm=\"users\"");
    res.sendError(res.SC_UNAUTHORIZED);
    // Could offer to add him to the allowed user list
  }
  else {
    // Allowed, so show him the secret stuff
    out.println("Top-secret stuff");
  }
}
// This method checks the user information sent in the Authorization
// header against the database of users maintained in the users Hashtable.
protected boolean allowUser(String auth) throws IOException {
  if (auth == null) return false;  // no auth
```

```
    if (!auth.toUpperCase().startsWith("BASIC "))
      return false;  // we only do BASIC

    // Get encoded user and password, comes after "BASIC "
    String userpassEncoded = auth.substring(6);

    // Decode it, using any base 64 decoder
    sun.misc.BASE64Decoder dec = new sun.misc.BASE64Decoder();
    String userpassDecoded = new String(dec.decodeBuffer(userpassEncoded));

    // Check our user list to see if that user and password are "allowed"
    if ("allowed".equals(users.get(userpassDecoded)))
      return true;
    else
      return false;
  }
}
```

Servlet Communication

Servlet to Servlet Communication

Servlets are not alone in a Web Server. They have access to other Servlets in the same Servlet Context (usually a Servlet directory), represented by an instance of javax.servlet.ServletContext. The ServletContext is available through the ServletConfig object's getServletContext method.A Servlet can get a list of all other Servlets in the Servlet Context by calling getServletNames on the ServletContext object. A Servlet for a known name (probably obtained through getServletNames) is returned by getServlet. Note that this method can throw a ServletException because it may need to load and initialize the requested Servlet if this was not already done.After obtaining the reference to another Servlet that Servlet's methods can be called. Methods which are not declared in javax.servlet.Servlet but in a subclass thereof can be called by casting the returned object to the required class type. Note that in Java the identity of a class is not only defined by the class name but also by the ClassLoader by which it was loaded. Web servers usually load each Servlet with a different class loader. This is necessary to reload Servlets on the fly because single classes cannot be replaced in the running JVM. Only a ClassLoader object with all loaded classes can be replaced.

Servlet to applet Communication

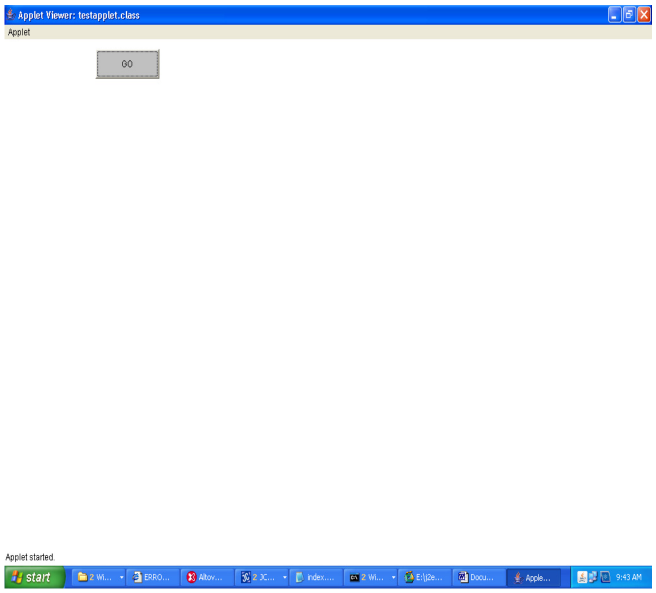
<u>Tespapplet.java</u>

```
import java.io.*;
import java.awt.*;
import java.applet.*;
/*<applet code="testapplet.class" width=426 height=266></applet>*/
public class testapplet extends Applet
{
public void init()
{
setLayout(null);
setSize(426,266);
goButton.setLabel("GO");
add(goButton);
goButton.setBackground(java.awt.Color.lightGray);
goButton.setBounds(144,12,101,39);
SymMouse asm=new SymMouse();
goButton.addMouseListener(asm);
}
java.awt.Button goButton=new java.awt.Button();
class SymMouse extends java.awt.event.MouseAdapter
{
public void mouseClicked(java.awt.event.MouseEvent event)
{
Object object=event.getSource();
}
}
```

```
void goButton_MouseClicked(java.awt.event.MouseEvent event)
{
try{
System.out.println("Attempting to connect to http://localhost:7001/appserv/testservlet");
java.net.URL url=new java.net.URL("http://localhost:7001/appserv/testservlet");
java.net.URLConnection c=url.openConnection();
c.setUseCaches(false);
c.setDoOutput(true);
c.setDoInput(true);
ByteArrayOutputStream byteout=new ByteArrayOutputStream();
DataOutputStream outdata=new DataOutputStream(byteout);
System.out.println("writing test data");
outdata.writeByte(1);
outdata.writeChar(2);
outdata.writeInt(3);
outdata.writeUTF("test message");
outdata.flush();
byte buf[]=byteout.toByteArray();
c.setRequestProperty("Content-type","application/octet-stream");
c.setRequestProperty ("Content-length", "" + buf.length);
DataOutputStream dataout = new DataOutputStream(c.getOutputStream());
dataout.write(buf);
dataout.flush();
dataout.close();
System.out.println("reading response");
DataInputStream inData=new DataInputStream(c.getInputStream());
byte byteval=inData.readByte();
char charval=inData.readChar();
int intval =inData.readInt();
String stringval=inData.readUTF();
inData.close ();
System.out.println("Data read:  " + byteval + " " + ((int) charval) + " " + intval + " " +
stringval);
}
catch(Exception e)
{
  e.printStackTrace ();
}
}
}
```

testservlet.java

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

public class testservlet extends HttpServlet
{
public void service(HttpServletRequest req, HttpServletResponse resp)throws
ServletException,IOException
{
DataInputStream br=new DataInputStream(req.getInputStream());
resp.setContentType("application/octect-stream");
ByteArrayOutputStream byteout=new ByteArrayOutputStream();
DataOutputStream outdata=new DataOutputStream(byteout);
byte byteval=br.readByte();
char charval=br.readChar();
int intval=br.readInt();
String stringval=br.readUTF();
outdata.writeByte(byteval);
outdata.writeChar(charval);
outdata.writeInt(intval);
outdata.writeUTF(stringval);
outdata.flush();
byte[] buf=byteout.toByteArray();
resp.setContentLength(buf.length);
ServletOutputStream servout=resp.getOutputStream();
servout.write(buf);
servout.close();
}
}
```



Simple Servlet example

You have learned about J2EE web containers,J2EE web application structure and basics of Java Servlets in previous **Servlet tutorials**. This **Servlet tutorial** provides a very simple **servlet example**.

What is covered in this Servlet example

1. How to write the servlet class.
2. How to compile the Servlet class.
3. How to extract the HTML form parameters from HttpServletRequest.
4. web.xml deployment descriptor file.

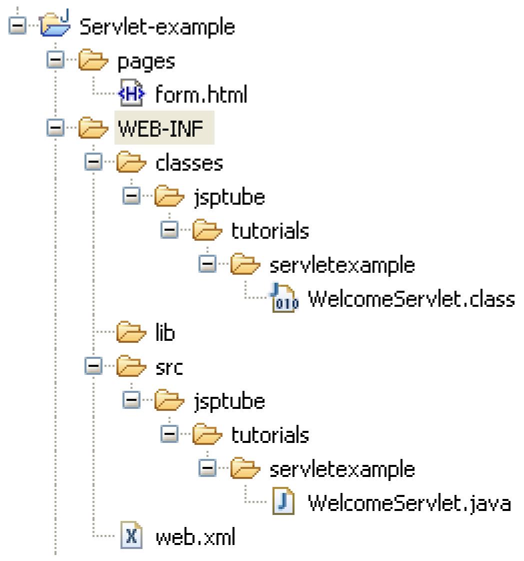
5. How to create the war (web application archive) file.
6. How to deploy and run the sample web application in tomcat web container.

You need the Java SDK and tomcat web server installed, to run the example Servlet application developed in this tutorial. Setting up the Servlet development environment tutorial explains how to setup the development environment.

If you don't have the basic understanding of the J2EE web containers and web application structure, read the previous Servlet tutorials which explain these basic things.

This is a very simple web application containing a HTML file and a servlet. The HTML document has a form which allows the user to enter the name, when the form is submitted application displays the welcome message to the user.

First of all we need to create the directory structure for our sample web application as explained in the tutorial Understanding the directory structure of web applications. Create the directory structure as shown in below figure.



```
Servlet-example
    pages
        form.html
    WEB-INF
        classes
            jsptube
                tutorials
                    servletexample
                        WelcomeServlet.class
        lib
        src
            jsptube
                tutorials
                    servletexample
                        WelcomeServlet.java
        web.xml
```

Create the HTML file.

Copy the following code into form.html file and save it under servlet-example/pages directory.

```
<html>
```

```
        <head>
        <title>The servlet example </title>
        </head>
        <body>
                <h1>A simple web application</h1>
                <form method="POST" action="WelcomeServlet">
                        <label for="name">Enter your name </label>
                        <input type="text" id="name" name="name"/><br><br>
                        <input type="submit" value="Submit Form"/>
                        <input type="reset" value="Reset Form"/>
                </form>
        </body>
</html>
```

The welcome Servlet class

Copy the following code into WelcomeServlet.java file and save it under servlet-example/WEB-INF/src/jsptube/tutorials/servletexample directory.

Note: it is not necessary to crate /src directory under WEB-INF directory and you can safely exclude WEB-INF/src directory when creating WAR file. You can put the source files any where you want, but don’t forget to put the compiled classes into WEB-INF/classes directory before creating the WAR file.

```
package jsptube.tutorials.servletexample;
 import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class WelcomeServlet extends HttpServlet {
```

```
public void init(ServletConfig config) throws ServletException {
super.init(config);
}

protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
/*
* Get the value of form parameter
*/
String name = request.getParameter("name");
String welcomeMessage = "Welcome "+name;
/*
* Set the content type(MIME Type) of the response.
*/
response.setContentType("text/html");
 PrintWriter out = response.getWriter();
/*
* Write the HTML to the response
*/
out.println("<html>");
out.println("<head>");
out.println("<title> A very simple servlet example</title>");
out.println("</head>");
out.println("<body>");
out.println("<h1>"+welcomeMessage+"</h1>");
out.println("<a href="/servletexample/pages/form.html">"+"Click here to go back to
input page "+"</a>");
out.println("</body>");
out.println("</html>");
out.close();
 }
 public void destroy() {
 }
}
```

Now compile the servlet class as explained below.
Open the command prompt and change the directory to the servlet-example/WEB-INF/src/jsptub/tutorials/servletexample directory. Compile the WelcomeServlet.java using the following command. javac WelcomeServlet.java It will create the file WelcomeServlet.class in the same directory. Copy the class file to classes directory. All the Servlets and other classes used in a web application must be kept under WEB-INF/classes directory.

Note: to compile a servlet you need to have servlet-api.jar file in the class path.

The deployment descriptor (web.xml) file.

Copy the following code into web.xml file and save it directly under servlet-example/WEB-INF directory.

```
<web-app version="2.4"
xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
		<servlet>
			<servlet-name>WelcomeServlet</servlet-name>
			<servlet-
class>jsptube.tutorials.servletexample.WelcomeServlet</servlet-class>
		</servlet>
		<servlet-mapping>
			<servlet-name>WelcomeServlet</servlet-name>
			<url-pattern>/WelcomeServlet</url-pattern>
		</servlet-mapping>
		<welcome-file-list>
```

```
                    <welcome-file> /pages/form.html </welcome-file>
          </welcome-file-list>
</web-app>
```

Deploying the application to tomcat web container.

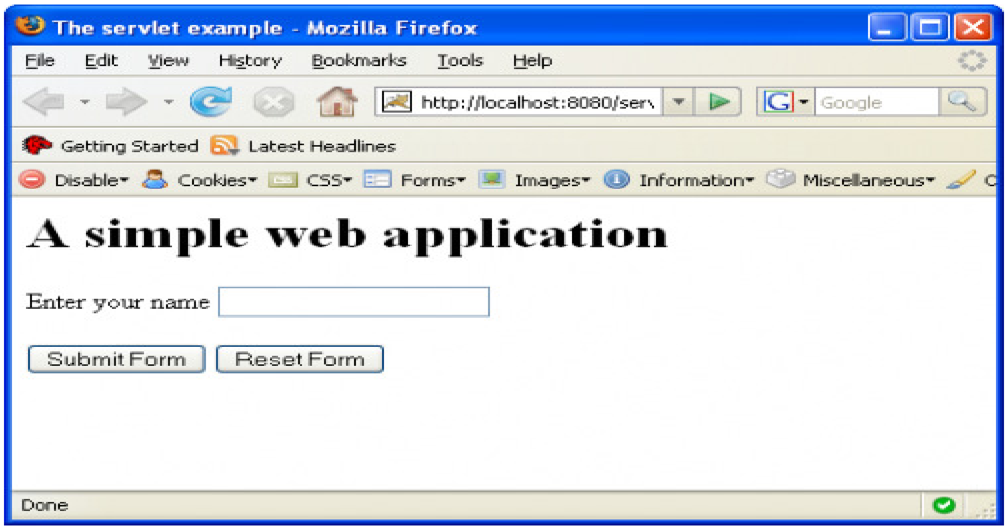
Deployment steps are different for different J2EE servers. This tutorial explains how to deploy the sample web application to tomcat web container. If you are using any other J2EE server, consult the documentation of the server.

A web application can be deployed in tomcat server simply by copying the war file to <TOMCAT_HOME>/webapp directory.
Copy servletexample.war to <TOMCAT_HOME>/webapp directory. That's it! You have successfully deployed the application to tomcat web server. Once you start the server, tomcat will extract the war file into the directory with the same name as the war file.

To start the server, open the command prompt and change the directory to <TOMCAT_HOME/bin> directory and run the startup.bat file.

Our sample application can be accessed at http://localhost:8080/servletexample/. If tomcat server is running on port other than 8080 than you need to change the URL accordingly.

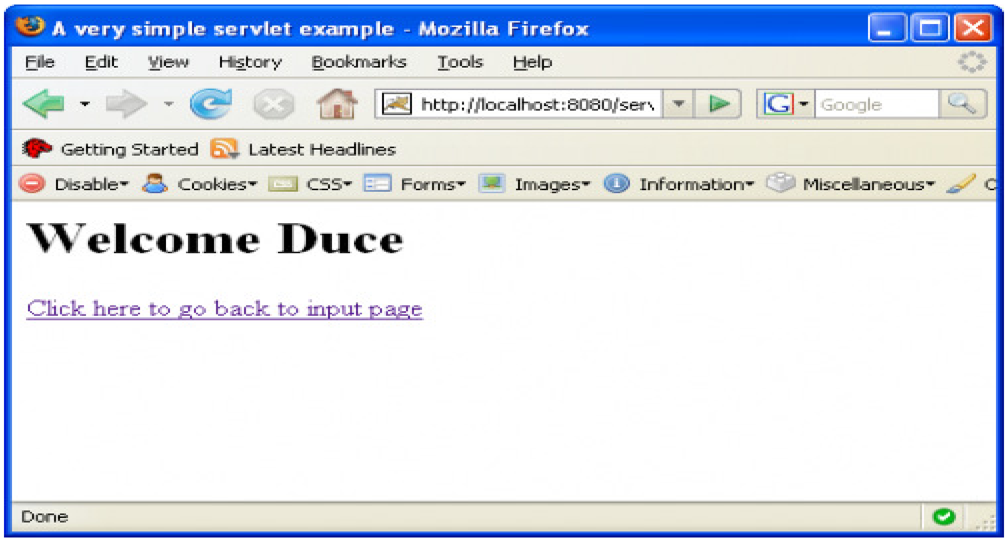
If the application has been deployed properly, you should see the screen similar to below when you open the application in browser.


The servlet example - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/serv
Getting Started Latest Headlines
Disable Cookies CSS Forms Images Information Miscellaneous
Submit Form
Reset Form
Done

A simple web application

Enter your name

Enter your name in text box and click on submit button. It will display welcome message with your name as shown in below figure.


A very simple servlet example - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/serv
Getting Started Latest Headlines
Disable Cookies CSS Forms Images Information Miscellaneous
Done

Welcome Duce

Click here to go back to input page

A simple program for Session Tracking

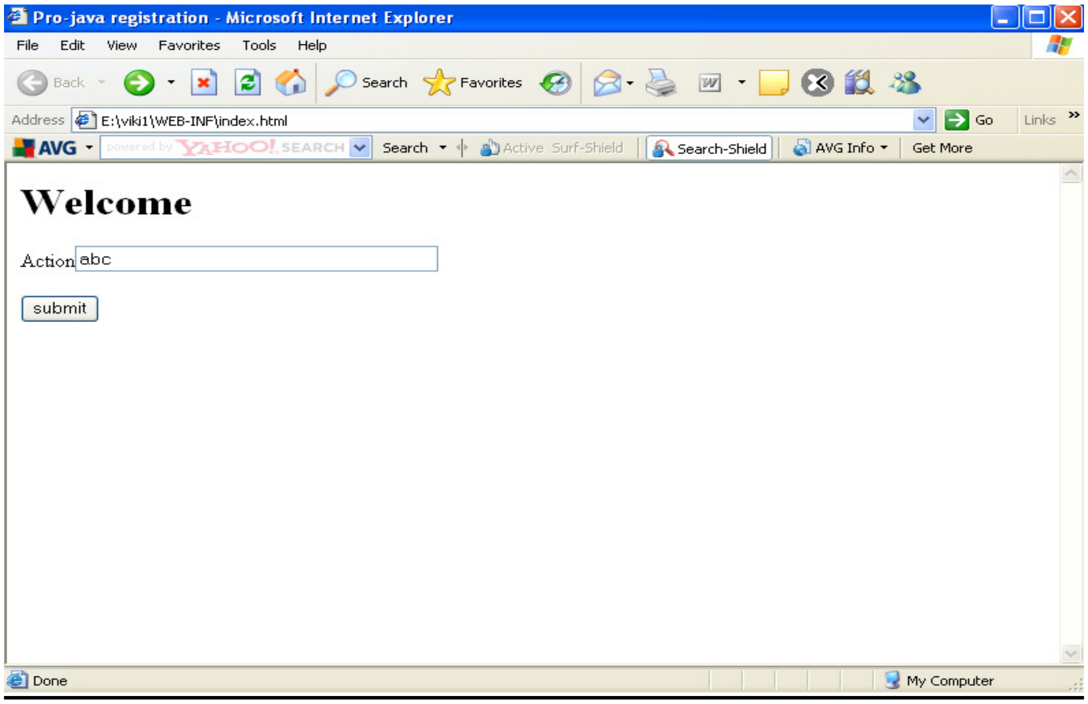
E:\viki1\WEB-INF\classes\SessionLifeCycleServlet.java

```
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

import java.io.PrintWriter;
import java.io.IOException;
import java.util.Date;

public class SessionLifeCycleServlet extends HttpServlet
{
protected void doPost(HttpServletRequest req,HttpServletResponse resp) throws
ServletException,IOException
{
String action=req.getParameter("action");
if(action!=null && action.equals("invalidate"))
{
        HttpSession ses=req.getSession();
        ses.invalidate();
        resp.setContentType("text/html");
        PrintWriter out=resp.getWriter();

        out.println("<html>");
        out.println("<head><title>Session Lifecycle</title></head>");
        out.println("<body>");
        out.println("<p> Wer session has been invalidated</p>");
String lfcyurl="/session/servlet/lifecycle";
        out.println("<a href=\" "+lfcyurl+ "?action=newSession\">");
        out.println("Create a new session</a>");
        out.println("</body></html>");
}
else
{
        HttpSession ses=req.getSession();
        resp.setContentType("text/html");
        PrintWriter out=resp.getWriter();

        out.println("<html>");
        out.println("<meta http-equiv=\"Pragma\" content=\"no-cache\">");
        out.println("<head><title>Session Life Cycle</title></head>");
        out.println("<body bgcolor=\"#FFFFFF\">");
        out.println("<h1> Session Lifecycle</center></h1>");

        out.print("<br> Session Status: ");
        if(ses.isNew())
        {
        out.println("New Session.");
        }
        else
        {
        out.println("Old Session. ");
        }
        out.println("<br> Session ID: ");

        out.println(ses.getId());
        out.println("<br>Creation Time: ");
        out.println(new Date(ses.getCreationTime()));
        out.println("<br> Last Accessed Time: ");
        out.println(new Date(ses.getLastAccessedTime()));
        out.println("<br> Max inactive interval(seconds): ");
        out.println(ses.getMaxInactiveInterval());

        String lfcyurl="/session/servlet/lifeCycle";
        out.print("<br><a href=\"" + lfcyurl+"\">");
out.println("Reload this pg</a>");
out.println("</body></html>");
out.close();

}
}
}
```

E:\viki1\WEB-INF\index.html

```
<html>
<head>
<title>Pro-java registration</title>
</head>
<body>
<h1>Welcome</h1>
<form action="http://localhost:7001/viki1/SessionLifeCycleServlet" method="POST">
<p>Action<input type="text" size="40" name="action"/></p>

<input type="submit" value="submit"/></p>
</form>
</body>
</html>
```

Input Screen

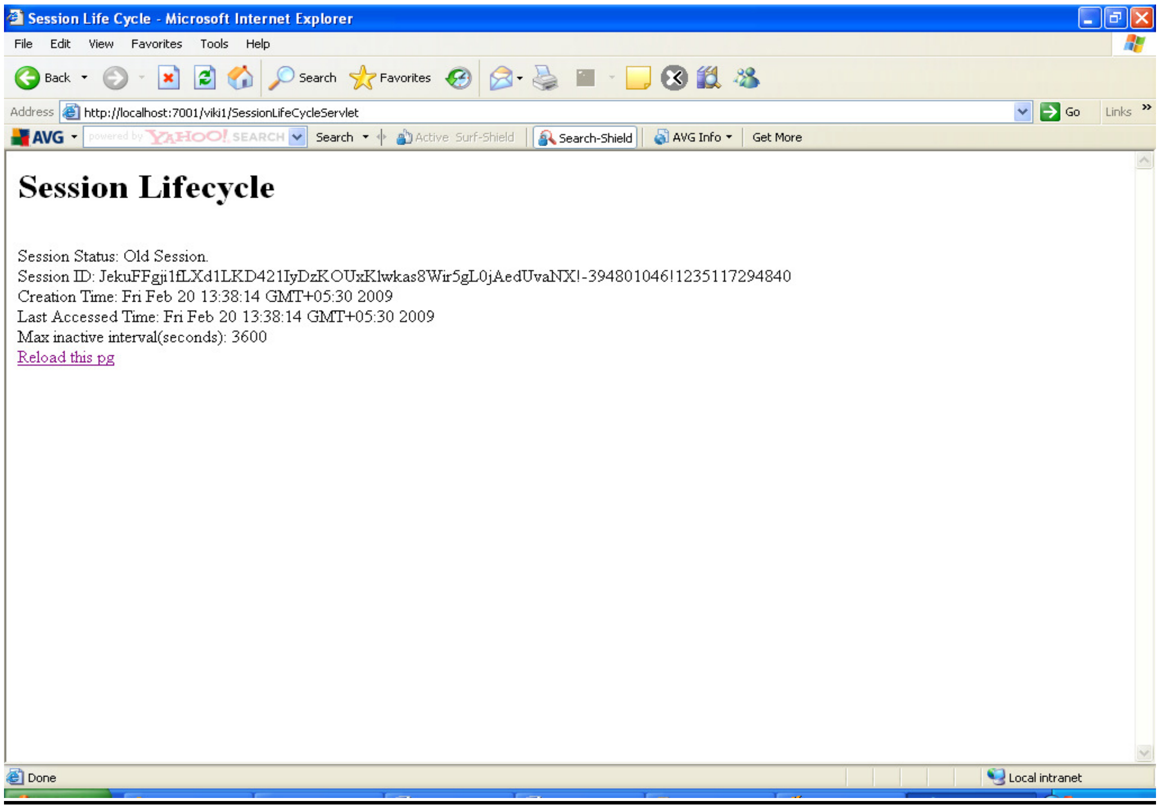

Pro-java registration - Microsoft Internet Explorer
Address E:\viki1\WEB-INF\index.html

Welcome

Action abc

submit

Output Screen


Session Life Cycle - Microsoft Internet Explorer
Address http://localhost:7001/viki1/SessionLifeCycleServlet

Session Lifecycle

```
Session Status: Old Session.
Session ID: JekuFFgji1fLXd1LKD421IyDzKOUxKlwkas8Wir5gL0jAedUvaNX!-394801046!1235117294840
Creation Time: Fri Feb 20 13:38:14 GMT+05:30 2009
Last Accessed Time: Fri Feb 20 13:38:14 GMT+05:30 2009
Max inactive interval(seconds): 3600
Reload this pg
```

Web.xml for sessionlifecycle servlet

```
    <!DOCTYPE web-app (View Source for full doctype...)>
- <web-app>
- <servlet>
  <servlet-name>SessionLifeCycleServlet</servlet-name>
  <servlet-class>SessionLifeCycleServlet</servlet-class>
    </servlet>
- <servlet-mapping>
  <servlet-name>SessionLifeCycleServlet</servlet-name>
  <url-pattern>/SessionLifeCycleServlet/*</url-pattern>
    </servlet-mapping>
- <welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
  <welcome-file>index.html</welcome-file>
  <welcome-file>index.htm</welcome-file>
    </welcome-file-list>
- <security-constraint>
- <web-resource-collection>
  <web-resource-name>My secure resources</web-resource-name>
  <description>Resources to be placed under security control.</description>
  <url-pattern>/private/*</url-pattern>
  <url-pattern>/registered/*</url-pattern>
    </web-resource-collection>
- <auth-constraint>
  <role-name>guest</role-name>
    </auth-constraint>
- <user-data-constraint>
  <transport-guarantee>NONE</transport-guarantee>
    </user-data-constraint>
    </security-constraint>
- <login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>WebApp</realm-name>
    </login-config>
- <security-role>
  <description>The role allowed to access our content</description>
  <role-name>guest</role-name>
    </security-role>
    </web-app>
```

A Simple Shopping Cart using Sessions

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

import java.io.*;
import java.util.ArrayList;

public class catalog extends HttpServlet
{
protected void doGet(HttpServletRequest req,HttpServletResponse res) throws
ServletException,IOException
{
HttpSession ses=req.getSession();
int itemcount=0;
ArrayList cart=(ArrayList)ses.getAttribute("cart");
if(cart!=null)
{
itemcount=cart.size();
}

res.setContentType("text/html");
PrintWriter out=res.getWriter();
out.println("<html><head><title>Shopping cart Example</title></head>");
out.println("<body><table border=\"0\" width=\"100%\"><tr>");
out.println("<td valign=\"top\">");
out.println("<td align=\"left\" valign=\"bottom\" >");
out.println("<h1>Apress Book store</h1></td></tr></table><hr>");
out.println("<p>We've "+itemcount+" items in wer cart.</p>");
out.println("<form action=\"");
out.println(res.encodeURL("/shopping/cart"));
out.println("\"method=\"post\">");
out.println("<table cellspacing=\"5\" cellpadding=\"5\"><tr>");
out.println("<td align=\"center\"><b>Add to cart</b></td>");
out.println("<td align=\"center\"></td></tr><tr>");
out.println("<td align=\"center\">");
out.println("<input type=\"checkbox\" name=\"item\" value=\"Pro Java
programming\"></td>");
out.println("<td align=\"left\">Item 1:"+"Projava programming</td></tr><tr>");
out.println("<td align=\"center\">");
out.println("<input type=\"checkbox\" name=\"item\" value=\"Beginning java
objects\"></td>");
out.println("<td align=\"left\">Item 2:"+"Beginning java objects</td></tr><tr>");
out.println("<td align=\"center\">");
out.println("<input type=\"checkbox\" name=\"item\" value=\"Professional java server
programming\"></td>");
out.println("<td align=\"left\">Item 3:"+"Professional java server
programming</td></tr><tr>");
out.println("</table><hr>");
out.println("<input type=\"submit\" name=\"btn_submit\" value=\"Add to cart\">");
out.println("</form></body></html>");
out.close();
}
}
```

//shoppingcart servlet

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

import java.io.*;
import java.util.*;

public class shoppingcart extends HttpServlet
{
protected void doPost(HttpServletRequest req,HttpServletResponse res) throws
ServletException,IOException
{
HttpSession ses=req.getSession(true);
int itemcount=0;
ArrayList cart=(ArrayList)ses.getAttribute("cart");
if(cart!=null)
{
cart=new ArrayList();
ses.setAttribute("cart",cart);
}

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

String[] itemselected;
String itemname;
itemselected=req.getParameterValues("item");

if(itemselected!=null)
{
for(int i=0;i<itemselected.length;i++)
{
itemname=itemselected[i];
cart.add(itemname);
}
}

//print the contents of the cart
out.println("<html><head><title>Shopping cart contents</title></head>");
out.println("<body>");

out.println("<h1>Items currently in wer cart</h1>");
out.println("<hr>");

Iterator iterator=cart.iterator();
while(iterator.hasNext())
{
out.println("<p>"+iterator.next()+"</p>");
}

out.println("<hr><p><a href=\"");
out.println(res.encodeURL("/shopping/catalog"));
out.println("\">Back to the shop</a></p>");

out.close();
}
}
```

//Web.xml

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
```

```
<servlet>
<servlet-name>catalog</servlet-name>
  <display-name>catalog</display-name>
  <servlet-class>catalog</servlet-class>
</servlet>

<servlet>
<servlet-name>cart</servlet-name>
  <display-name>cart</display-name>
  <servlet-class>shoppingcart</servlet-class>
</servlet>



<servlet-mapping>
  <servlet-name>catalog</servlet-name>
  <url-pattern>/catalog/*</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>cart</servlet-name>
  <url-pattern>/cart/*</url-pattern>
</servlet-mapping>


<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
  <welcome-file>index.html</welcome-file>
  <welcome-file>index.htm</welcome-file>
</welcome-file-list>

<security-constraint>
```

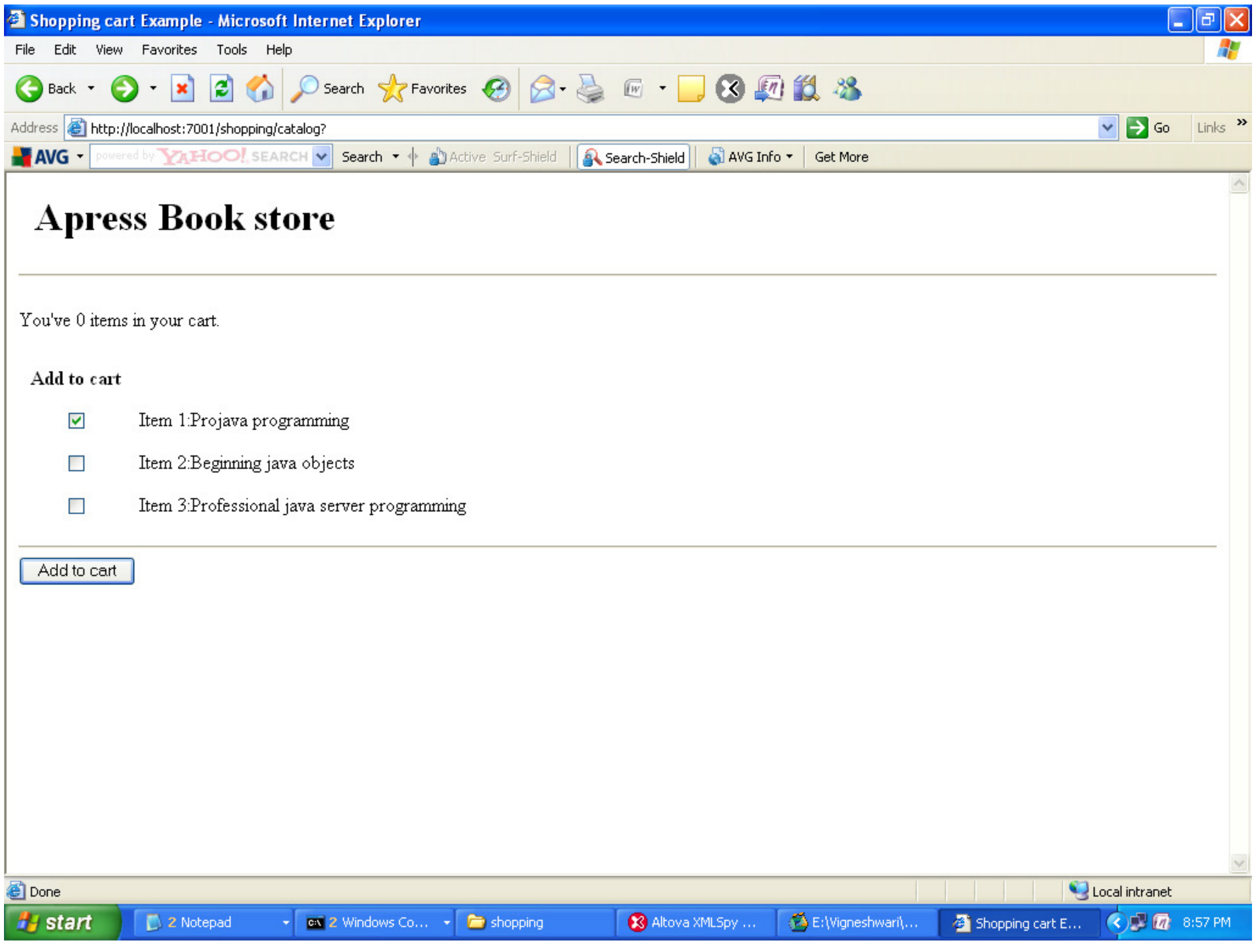
```
    <web-resource-collection>
      <web-resource-name>My secure resources</web-resource-name>
      <description>Resources to be placed under security control.</description>
      <url-pattern>/private/*</url-pattern>
      <url-pattern>/registered/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>guest</role-name>
    </auth-constraint>
    <user-data-constraint>
      <transport-guarantee>NONE</transport-guarantee>
    </user-data-constraint>
  </security-constraint>

  <login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>WebApp</realm-name>
  </login-config>

  <security-role>
    <description>The role allowed to access our content</description>
    <role-name>guest</role-name>
  </security-role>

</web-app>
```

//weblogic.xml

```
<!DOCTYPE weblogic-web-app PUBLIC "-//BEA Systems, Inc.//DTD Web Application
8.1//EN" "http://www.bea.com/servers/wls810/dtd/weblogic810-web-jar.dtd">

<weblogic-web-app>
```

```
  <jsp-descriptor>
    <jsp-param>
      <param-name>compileCommand</param-name>
      <param-value>javac</param-value>
    </jsp-param>
    <jsp-param>
      <param-name>workingDir</param-name>
      <param-value>C:\DOCUME~1\TEJASW~1\LOCALS~1\Temp\</param-value>
    </jsp-param>
    <jsp-param>
      <param-name>keepgenerated</param-name>
      <param-value>true</param-value>
    </jsp-param>
    <jsp-param>
      <param-name>pageCheckSeconds</param-name>
      <param-value>5</param-value>
    </jsp-param>
  </jsp-descriptor>
</weblogic-web-app>
```

//output screen


Shopping cart Example - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
Address http://localhost:7001/shopping/catalog?
Go
Links
Done
Local intranet
start
8:57 PM

Apress Book store

You've 0 items in your cart.

Add to cart

- [✓] Item 1:Projava programming
- [ ] Item 2:Beginning java objects
- [ ] Item 3:Professional java server programming

Add to cart

Session Tracking

An important feature required by most web-applications is **Session-Tracking**.Its used to

maintain state across a series of requests from the same user over a period of time.

Ex: Online Shopping Cart

STATELESSNESS AND SESSIONS

HTTP is a stateless protocol. A client open a connection and requests some resource or information. The server responds with the requested resources(if available), of sends an HTTP error status.after closing the connection, the server does not remember any information about the client. So, the server considers the next request from the same client as a fresh request, with no relation to the previous request. This is what makes HTTP a stateless protocol.

A protocol is stateful if the response to a given request may depend not only on the current request, but also on the outcome of previous requests.

SESSION: A session is a conversion between the server and a client. A conversion consists series of continuous request and response.By associating a specific request to belong to a specific working session, the shopping cart or the online banking application can distinguish one user from another.

STATE: The server should be able to remember information related to previous requests and other business decisions that are made for requests. That is, the application should be able to associate state with each session. In the case of the shopping cart application, possible state could include the user's favorite item categories, user profile or even the shopping cart itself.

Approaches to Session Tracking

There are essentially four approaches to session tracking :

- Hidden Form Field
- URL Rewriting
- Using Cookies
- Sessions using the Secure Socket Layer (SSL)

Hidden Form Field

We can use hidden form fields t maintain the session information of a user while the user interacts with the Web application. A hidden form field is embedded in an HTML page and is not visible when viewed in a browser. The following code snipped shows a hidden form field in an HTML page:

```
<HTML>
<FORM METHOE=”GET” Action=”/servlet/TestServlet”>
<input type=”hidden” name=”id” value=”l123”>-
 - - - - - - - - - - - - -
 - - - - - - - - - - - - -
</HTML>
```

URL Rewriting

URL rewriting is a session management technique that manages user session by modifying a URL. Usually, technique is used when information that is to be transferred is not very critical because the URL can be intercepted easily during transfer. For example, in an online shopping portal, a servlet can modify a URL to include user information, such as username. The servlet can then display the URL. When the user clicks the URL hyper link, The information is sent to another servlet that retrieves the user information aand displays a welcome message. We can use the following code to create the servlet, RewriteServletURL, that modifies and displays a URL.

```
/*import  the java packages*/
Import java.io.*;
Import javax.servlet.*;
Import javax.servlet.http.*;
Public class RewriteServletURL extends HttpServlet
{
Public void doGet(HttpServletRequest req, HttpServletResponse res)throws
ServletException, IOException
{
 dpPost(req,res);
}
Public void doPost(HttpServletRequest req, HttpServletResponse res)throws
servletException, IOException
{
/*retrieve the parameters bound to user, password and login from the request
object*/
        String username=req.getparameter("user");
        PrintWriter pw=res.getwriter();
/*verify the login status*/
        res.serContentType("text/html");
        pw.println("hello<a
href=\http://localhost:8080/rewrite_cntxt/servlet/SecondServlet?uname="+userna
me  +"\">click hear </a>to proceed");
```

```
        }
      }
```

Using Cookies

Cookies are small text files that are stored by an application server in the client browser to keep track of all the user. A cookies has values in the form of name/value pairs. For example, a cookie can hava a name, user with the value, Michael. They are created by the server and sent to the client with the HTTP response headers. The client saves the cookies in the local disk and sends them along with the HTTP request headers to the server. A Web browser is expected to support 20 cookies per characteristics of cookies are:

1. Cookies can only be read by the application server that had written them in the client browser. For example, if a cookie is created by the application server of www.macromedia .com and sent to the client or to the browser, the cookie can be sent back to the same server only.
2. Cookies can be used by the server to find out the computer name, IP address any other details of the client computer by retrieving the remote host address of the client where the cookies are stored.

The Servlet API provides support for cookies. The cookie class of javax.servlet.http package represents a cookie. The cookie class provides a constructor that accepts the name and value to create a cookie. The following code snippet shows the constructor for creating a cookie with name, name and value.

Value:

Public Cookie(String name, String value);

The HttpServletResponse class provides the getCookie() method to add a cookie to the response object, for sending it to the client. The following code snippet shows how to add a cookie:

resp.addCookie(Cookie cookie);

The HttpServletRequest class provides the getCookie() method that returns an array of cookies that the request object contains. The following code snippet shows how to add a cookie:

Cookie cookie[]=req.getCookies();

The Cookie class provides several methods to set and retrieve various properties of a cookie. The following table describes the various methods of the Cookie class:

METHODS	**DESCTIPTION**
Public String getName()	Returns the name of the cookie .
Public void setMaxAge(int expiry)	Sers the maximum time for which the client browser retains the cookie value.
Public int getMaxAge()	Returns the maximum age of the cookie in seconds.

Secure socket layer(ssl) sessions :

SSL is an encrypting technology that runs on top of TCP/IP and below application-level protocols such as HTTP. SSL is the technology used in the HTTPS protocols. SSL-enabled servers can authenticate SSL-enabled client, and use an encrypted connection between the client and server. In the process of establishing an encrypt connection between the client and server generate what are called ‘session keys’, which are symmetric keys used for encrypting and decrypting messages. Server based on the HTTPS protocols can use the client’s symmetric keys to establish a session.

SERVLET API

The classes and interfaces defined in the Servlet Session API is used to create and manage user sessions. Various interfaces provided by Servlet Session API to create and manage user session are, javax.servlet.http.HttpSession, javax.servlet.http.HttpSessionListener, and javax.servlet.http.HttpSessionBindingListener.

The javax.servlet.http.HttpSession interfaces provides methods for tracking the session of a user. An object of HttpSession interface is created to store session information as name/value pairs, which can be later retrieved to manage user sessions.

The following table describes the various methods defined in the HttpSession interface:

METHODS	**DESCTIPTION**
Public void setAttribute(String name, Object value)	Binds the attribute session object with a unique name and stores the name/value

	pair in the current session. If an object is already bound with the same attribute, then the new object replaces the existing object
Public getAttribute(String name)	Retrieves the object bound with the attribute name specified in the method, from the session object. If no object is found for the specified attribute, then the getAttribute() method returns null
Public void setMaxInactiveInterval(int interval)	Sets the maximum time for which the session will remain active. The time is specified in seconds. A negative value in this method signifies that the session should always remain active.
Public int getMaxInactiveInterval()	Returns the maximum time in seconds for which the server will not invalidate the session even if there is no client request.
Public void invalidate()	Invalidates a session. All time objects bound to the session are automatically unbound.

SERVLET CONTEXT

In the Java Servlet API, the servlet context defines a servlet's view of the web application, and provides access to resources and facilities(such as login) common to all servlets in the application. Unlike a session, wich is specify to a client, the servlet context in specific to a particular web application runnin in JVM. That is each web application in a container will have a single servlet context associated with it.

Servlet context is yet another useful concept provide by the Servlet API. In the previous section, we saw how we coule use Httpsession objects to maintain state relating ot a single client on the server. The servlet context complements this facility by letting we maintain application-level state-that is state information common to all servelets and clients in that application.

The Java Servlet API provides a Servletcontext interface to represent a context: the resources shared by a group of servlets. In the 1.0 and 2.0 vdrsions of the servlet API the SERVLET CONTEXT interface onlu provided access to information about the servlet's environment, such as the name of the server MIME type mappings, etc., and a log() method for writing log messages to the server's log file.

THE SERVLETCONTEXT INTERFACES

The ServeletContext interfaces specifies the following methods; web container provide an implementation.

```
 public String getMimeType(String filename)
 public RequestDispatcher getRequestDispatcher(String path)
public String getserverinfo()
public set getResourcePaths(String path)
public void log(java.lang.String msg)
public void log(String messages,Throwable throwable)
```

The getMimeType() method

```
public String getMimeType(String filename)
```

This method returns the MIME type of a file by its extension. As we shall see in chapter 9,we can specify the MIME types that wer application can handle in the deployment descriptor.

The getResource() method

```
public RequestDispatcher getRequestDispatcher(String path)
```

This method returns a RequestDispatcher object associated with the resource located at the current path. We can use the RequestDispatcher to delegate request/response processing to other resources within the application.

The getServerInfo() method

```
public String getserverinfo()
```

This method is returns the name and version of the servlet container on which the servlet is running. This information may be used for logging purpose.

The getResourcePaths() method

```
public set getResourcePaths(String path)
```

This method is functionally similar to executing a directory listing command. Given a path name relative to the root of the web application, this method returns all sub-paths.
The ServletContext interface provides two methods for logging purpose. Of these, the second method can be used to log exceptions, while the first method can be used for general-purpose logging.
The log() method

```
  public void log(java.lang.String msg)
 public void log(String message, Throwable  throwable)
```

These four methods let we maintain the state of a web application. Any servlet can set an attribute, which can then be obtained by any other servlet within the same application, irrespective of whether these servlets are serving the same client or not. Using these methods, servlets can share information common to all servlets.

SERVLETCONTEXT LIFECYCLE EVENT HANDLING

The container creates the servlet context before attempting to serve any contents of a web application. Similarly, the container may delete the context before stopping to serve the contents of a web application. In order to handle these two events, the Servlet API specifies the following interface:

THE SERVLET CONTEXT LISTENER INTERFACE

```
   public interface ServletContextListner extends java.util.EventListener
```

An object of this interface receives events about the creation or destruction of a servlet context. This interface has the following methods:

```
  public void contextDestroyed(ServletContextEvent event)
```

This method will be called when a new context is created.

```
Public void contextDestroyed(ServletContextEvent event)
```

This method will be called when an existing context is destroyed.

We can register a class implementing this interface using the <listener> element in the deployment descriptor:

```
    <listener>
           <listener-class> MyContextListener</listener>
</listener>
```

SERVLET CONTEXT ATTRIBUTE LISTENER

public interface ServletContextAttributeListener extends java.util.EventListener

This interface is similar to the HttpSessionAttributeListner and gets notifies when there is change in the state of a context. This interface has the following methods:

public void attributeAdded(ServletContextAttributeEvent event).The container invokes this method on the listener when a new attribute is added to the context.

public void attributeReplaced(ServketContextAttributeEvent event)

This container invokes this method on the listener when the value of an attribute is changed, but not the attribute itself. This happens when we call the setAttribute() method on the context with the name of a previously added attribute as the name.

public void attributeRemoved(ServletContextAttributeEvent event)

The container invokes this method on the listener when an attribute is removed from the context.

SERVLET COLLABORATION

In the typical servlet model, a servlet receives an HTTP request,execute some application logic, and prepares the response. This completes one request-response trip for the client. However, there are several scenarios in which this basic model is not adequate:

- A servlet receives an HTTP request from a client, processes application logic, and a JavaServerPage drives the response. In this case the servlet is not responsible for response generation. Instead, the JSP page is responsible for dynamic content.
- A servlet receives an HTTP request from client, processes application logic partially, and hands over the request to another servlet. The second servlet completes the application logic, and either prepares the response, or request a JSP page to drive the response.

In both the scenarios, the servlet is not completely responsible for processing a request. Instead , it delegates the processing to another servlet(or a JSP page, which equivalent to a servlet at run time)

1.**SERVLET CHAINING**

This was once a very widely used approach, and supported by some of the servlet engine vendors. Although this is not supported by the Java Servlet API specification, a short description below.

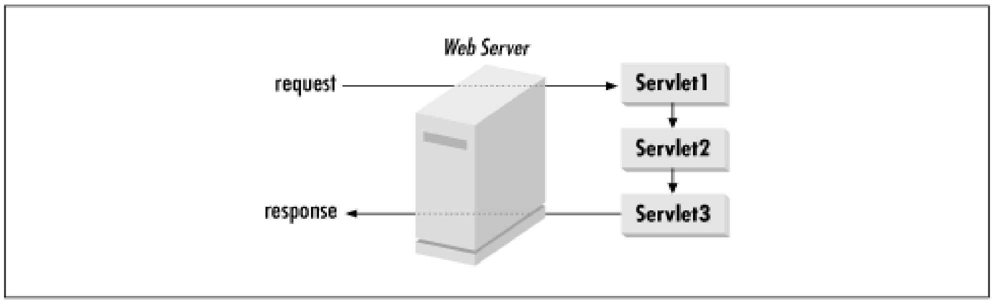
2.**REQUEST DISPATCHING**

Request dispatching allows one servlet to dispatch the request to another resource(a servlet, a JSP page ,or any other resources). Prior to version2.2 of the Servlet API, this approach used to be called 'inter-servlet communication'. The API used to provide a method to get an instance of another servlet using a name. see the documentation of the now deprecated getServlet() method of the javax.servlet.ServletContext interface. From version 2.2 of the API request dispatchers replace this functionality.

SERVLET CHAINING

Servlet chaining predates J2EE and its component model. The idea of servlet chaining is very simple: we design a set of servlets, each of which does a single task. After developing these servlets, we configure the servlet engine to specify chain of servlets for a given URL path alias. Once the servlet engine receives a request fof this alias , it invokes the servlets in the specified in the order. This is similar to piping on Unix, where output of one program become input for another program in the pipe.

Suppose we've servlets 1,2,&3 executing three parts of a request-response proves for single customer service. Let's assume that/custService is the alias giben to this chain(/custService=1,2,3). Consider a browser sending a request to the URL path pointing to this alias. The servlet engine sends all request for this alias to servlet 1. after executing servlet 1's service method, the servlet engine invokes servlet 2's service method, followed by servlets 3's service method. The final response is then sending to the client. Briefly, this is servlet chaining. Refer to the fig below for an overview of the approach.



This J2EE tip demostrates chaining method in servlets. Servlet Chaining means the output of one servlet act as a input to another servlet. Servlet Aliasing allows us to invoke more than one servlet in sequence when the URL is opened with a common servlet alias. The output from first Servlet is sent as input to other Servlet and so on. The Output from the last Servlet is sent back to the browser. The entire process is called Servlet Chaining.

```
// FirstServlet

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

public class FirstServlet extends HttpServlet {
  String name ;
  ServletConfig config;

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

    response.setContentType("text/plain");

    PrintWriter out = response.getWriter();
    name = request.getParameter("name");
    RequestDispatcher rd = config.
     getServletContext().getRequestDispatcher("SecondServlet");

    if(name!=null) {
        request.setAttribute("UserName",name);
        rd.forward(request , response);
        // Forward the value to another Secondservlet
      } else {
        response.sendError(response.SC_BAD_REQUEST,
          "UserName Required");
      }
    }
}


// SecondServlet

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

    response.setContentType("text/plain");
    PrintWriter out = response.getWriter();
    String UserName = (String)request.getAttribute("UserName");

    // Extracting the value which is set in FirstServlet
    out.println("The UserName is "+ UserName);
  }
}
```

REQUEST DESPATCHING

Request dispatching allows a servlet or a JSP page to dispatch a request to another servlet,(or a JSP page, or even a plain HTML page), which will then be responsible for any further processing and for generating the response.

When the web container receives a request,it constructs request and response objects and invokes one of the servlet service methods with the request and response objects. This is a process of the web container dispatching a request to a servlet. What if this servlet wants to dispatch the same request to another servlet after some preliminary processing? For this purpose, the first servlet should be able to obtain a reference to the second servlet. Using this reference,the first servlet can dispatch a request to the second servlet. In simple terms this is request dispatching.

The java servlet API has a special interface called ‘ javax.servlet.ReqiestDispacher for this purpose. REQUEST DISPATCHER INTERFACE

public Interface RequestDispatcher

This interface encpasulates a reference to another web resources at a specified path within the scope of the same servlet context. A javax.servlet.RequestDispatcher object can be used to despatch reques to other servlets and JSP pages.

This interface has two methods, which allow we to delegate the request-response processing to another resource, after the calling servlet has finishyed any preliminary processing.

The forward() METHOD

Public void forward(servletRequest request, ServletResponse response) throws ServeltException, java.io.IOException

This method lets we forward the request to another servlet or a JSP page, or an HTML file on the server;

This resource then takes over responsibility for processing the response.

The include()Method

Public void include(ServletRequest request,ServletResponse response) throws ServletExcepton, java.io.IOException

This method lets we include the content producedby another resource in the calling servlet's response.

OBTAINING A REQUESTDISPATCHER OBJECT

There are three ways in which we can obtain a RequestDispatcher object for a resource:

1. Public RequestDispatcher getRequestDispatcher(String path)
2. Public RequestDispatcher getNamedDispatcher(String name)
3. Public RequestDispatcher getRequestDispatcher(String path)

Although these methods serve the same purpose, the usage depends on what information is available to we.

The two getRequestDispatcher()methods accept a URL path referring to the target respurce. However, the getRequestDispatcher()methods on javax.servlet.ServletContext requires the absolute path(that is , the path name should be begin with a/). For example, if we have a servlet /myWebApp/servlet/servlet1, and want to get the RequestDispatcher object for /myWebApp/servlet/servlet2, we should specify the complete path relative to the root context. Hear the root context is myWebApp ,and the absolute path is /servlet/servlet2.

The same method on javax.servlet.servletRequst accepts both absolute and relative paths. In the above example, we could now also use Servlet2 as the paths.

A javax.servlet.ServletRequest is associated with a URL path, and the web container can use this to resolve relative paths into absolute paths.

The getNamedDispatcher() method is a convenience method that acceps a name associated with the servlet. This is the same name that we specify in the deployemt descriptor in the <servlet-name> element.

Example 5-3 demonstrates a basic servlet, derived from `HttpServlet`, that examines incoming text for a `<DATE>` tag and replaces the tag with the current date. This servlet is never called on its own, but instead after another servlet (such as, an HTML generator) has produced the actual content.

Example 5-3. Date Filtering Servlet

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

public class DateFilter extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {

    PrintWriter out = resp.getWriter();

    String contentType = req.getContentType();
    if (contentType == null)
      return; // No incoming data
    // Note that if we were using MIME filtering we would have to set this to
    // something different to avoid an infinite loop
    resp.setContentType(contentType);

    BufferedReader br = new BufferedReader(req.getReader());

    String line = null;
    Date d = new Date();
    while ((line = br.readLine()) != null) {
      int index;
      while ((index=line.indexOf("<DATE>")) >= 0)
        line = line.substring(0, index) + d + line.substring(index + 6);
      out.println(line);
    }



    br.close();
  }
}
```

The `DateFilter` servlet works by reading each line of input, scanning for the text `<DATE>`, and replacing it with the current date. This example introduces the `getReader()` method of `HttpServletRequest`, which returns a `PrintReader` that points to the original request body. When we call `getReader()` in an `HttpServlet`, we can read the original HTTP form variables, if any. When this method is used within a filtering servlet, it provides access to the output of the previous servlet in the chain.

UNIT III JAVA SERVER PAGES

Introduction to JSP:

The goal of the java server pages is to simplify the creation and management of dynamic web pages. JSP page combine static markup, like HTML, XML, with special scripting tags. JSP page resemble markup documents, but each page is translated into a servlet the first time it is invoked. The resulting servlet is a combination of the markup from the JSP file and embedded dynamic content specified by the scripting tags.

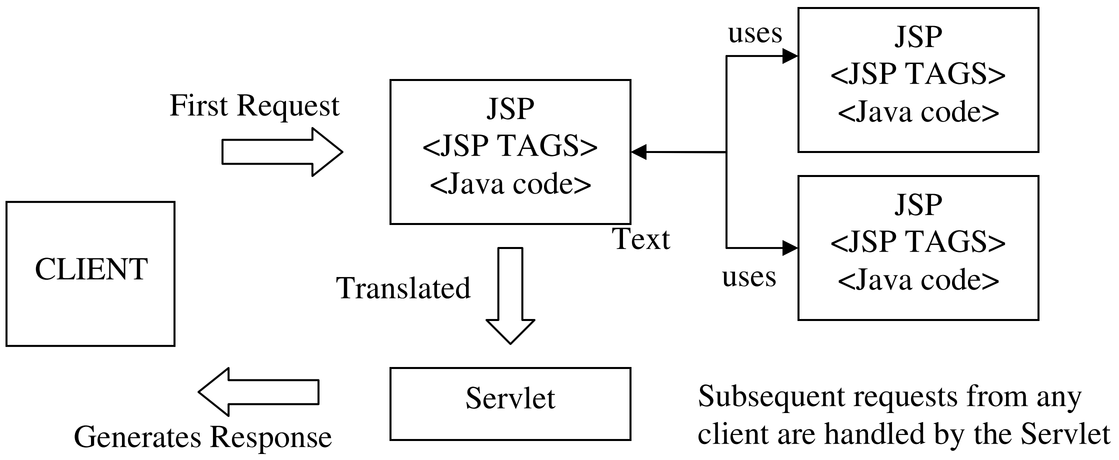
A simple JSP that includes the current date in an HTML page would like this:

```
<% @ page import=”java.util.Date” %>
<html>
<body>    The content time is < % = (new Date()).toString() %>    </body>
</html>
```

Save the example as ddate.jsp and deploy it.After deploying navigate to http://localhost:8000/JSPExamples/dddate.jsp and we obtain the output as follows:

The JSP page and the dependent files are together known as a translation unit.The first time the JSP engine intercepts a request for JSP,it compiles the translation uint into a servlet.

The distinction between translation time and run time is very important in understanding how JSP pages work. This figure summarizes the process:



The first time a JSP is loaded by the JSP container, the servlet code necessary to implement the JSP tags is automatically generated, compiled and loaded into the servlet container. this occurs at translation time. It is important to note that this occurs only the first time a JSP page is requested. There will be a slow response the first time a JSP page is accessed, but on subsequent requests the previously compiled servlet simply process the requests. This occurs at run time.

Three major life event methods working together in JSP are:

```
public void  jspInit()            }
                                  }     javax.servlet.jsp.JspPage   interface
public void  jspDestroy()         }

public void  jspService (HttpServletRequest req,     }
                         HttpServletResponse rep)    }  javax.servlet.jsp.HttpJspPage (I/f)
          throws ServletException,IOException        }
```

- The page is first initialized by invoking the jspInit() method, which may be defined by the page author. This initializes the JSP in the same way as Servlets.

- Every time a request is made to the JSP, the container generated JSP pageservice() method is invoked, the request is processed and the response is generated.

- When the JSP is destroyed by the server, the JSP destroy() method is invoked to perform any cleanup.

The Nuts and Bolts

There are three categories of JSP tags:

- Directives: These affect the overall structure of the servlet that results from translation, but produces no output themselves.
- Scripting Elements: These let us inside java code into the JSP page.
- Actions: These are special tags available to affect the runtime behavior of the JSP page. Two types of actions: Standard Actions and Custom Actions.

General rule that apply to JSP pages: As for sending mail from webserver, using JavaMail API, the following code shows how the required data

such as 'from', 'to', 'subject' and 'message' are collected by the servlet and then processed for sending the mail.

----

mailservlet.htm

```
<html>

<body>

<form   method=post   action="http://localhost:8080/servlet/mailservlet">

                sender           <input type=text name=text1><br>

                 Reciever        <input type=text name=text2><br>
                Subject        <input type=text name=text3><br>

                  Message          <textarea    name='area1'    rows=5    cols=30>
</textarea>   <input    type=submit>

</form>

</body>

</html>
```

mailservlet.java

```
import javax.servlet.*;

import javax.servlet.http.*;

import java.io.*;

import javax.mail.*;

import javax.mail.internet.*;   // important

import javax.mail.event.*;      // important

import java.net.*;

import java.util.*;

public class servletmail extends HttpServlet

{

   public  void doPost(HttpServletRequest request,HttpServletResponse response)

                              throws ServletException, IOException

   {
    PrintWriter out=response.getWriter();

    response.setContentType("text/html");

    try

    {

      Properties props=new Properties();

      props.put("mail.smtp.host","localhost");   //  'localhost' for testing

Session   session1  =  Session.getDefaultInstance(props,null);

      String s1 = request.getParameter("text1"); //sender (from)

      String s2 = request.getParameter("text2");

      String s3 = request.getParameter("text3");

      String s4 = request.getParameter("area1");

  Message message =new MimeMessage(session1);

   message.setFrom(new InternetAddress(s1));

   message.setRecipients

        (Message.RecipientType.TO,InternetAddress.parse(s2,false));

      message.setSubject(s3);

      message.setText(s4);

      Transport.send(message);

      out.println("mail has been sent");

    }
```

```
        catch(Exception ex)

        {

          System.out.println("ERROR....."+ex);

        }

    }

}
```

Using javamail, requires classpath to **mail.jar** & **activation.jar.** These should have been already installed in our machine. Otherwise, we will not be able to compile the servlet. For testing the servlet, we should have installed some mail server in our machine. For compiling the servlet, we have to set classpath to c:\jsdk2.0\src (java servlet development kit).

JSP DIRECTIVES

JSP directives provide directions and instructions to the container, telling it how to handle certain aspects of JSP processing.

A JSP directive affects the overall structure of the servlet class. It usually has the following form:

```
<%@ directive attribute="value" %>
```

Directives can have a number of attributes which you can list down as key-value pairs and separated by commas.

The blanks between the @ symbol and the directive name, and between the last attribute and the closing %>, are optional.

There are three types of directive tag:

Directive	Description
<%@ page ... %>	Defines page-dependent attributes, such as scripting language, error page, and buffering requirements.
<%@ include ... %>	Includes a file during the translation phase.
<%@ taglib ... %>	Declares a tag library, containing custom actions, used in the page

The page Directive:

The **page** directive is used to provide instructions to the container that pertain to the current JSP page. You may code page directives anywhere in your JSP page. By convention, page directives are coded at the top of the JSP page.

Following is the basic syntax of page directive:

```
<%@ page attribute="value" %>
```

You can write XML equivalent of the above syntax as follows:

```
<jsp:directive.page attribute="value" />
```

Attributes:

Following is the list of attributes associated with page directive:

Attribute	Purpose
buffer	Specifies a buffering model for the output stream.

autoFlush	Controls the behavior of the servlet output buffer.
contentType	Defines the character encoding scheme.
errorPage	Defines the URL of another JSP that reports on Java unchecked runtime exceptions.
isErrorPage	Indicates if this JSP page is a URL specified by another JSP page's errorPage attribute.
extends	Specifies a superclass that the generated servlet must extend
import	Specifies a list of packages or classes for use in the JSP as the Java import statement does for Java classes.
info	Defines a string that can be accessed with the servlet's getServletInfo() method.
isThreadSafe	Defines the threading model for the generated servlet.
language	Defines the programming language used in the JSP page.
session	Specifies whether or not the JSP page participates in HTTP sessions
isELIgnored	Specifies whether or not EL expression within the JSP page will be ignored.

isScriptingEnabled	Determines if scripting elements are allowed for use.

Check more detail related to all the above attributes at Page Directive.

The include Directive:

The **include** directive is used to includes a file during the translation phase. This directive tells the container to merge the content of other external files with the current JSP during the translation phase. You may code *include* directives anywhere in your JSP page.

The general usage form of this directive is as follows:

```
<%@ include file="relative url" >
```

The filename in the include directive is actually a relative URL. If you just specify a filename with no associated path, the JSP compiler assumes that the file is in the same directory as your JSP.

You can write XML equivalent of the above syntax as follows:

```
<jsp:directive.include file="relative url" />
```

Check more detail related to include directive at Include Directive.

The taglib Directive:

The JavaServer Pages API allows you to define custom JSP tags that look like HTML or XML tags and a tag library is a set of user-defined tags that implement custom behavior.

The **taglib** directive declares that your JSP page uses a set of custom tags, identifies the location of the library, and provides a means for identifying the custom tags in your JSP page.

The taglib directive follows the following syntax:

```
<%@ taglib uri="uri" prefix="prefixOfTag" >
```

JSP actions

use constructs in XML syntax to control the behavior of the servlet engine. You can dynamically insert a file, reuse JavaBeans components, forward the user to another page, or generate HTML for the Java plugin.

There is only one syntax for the Action element, as it conforms to the XML standard:

```
<jsp:action_name attribute="value" />
```

Action elements are basically predefined functions and there are following JSP actions available:

Syntax	**Purpose**
jsp:include	Includes a file at the time the page is requested
jsp:useBean	Finds or instantiates a JavaBean
jsp:setProperty	Sets the property of a JavaBean
jsp:getProperty	Inserts the property of a JavaBean into the output
jsp:forward	Forwards the requester to a new page
jsp:plugin	Generates browser-specific code that makes an OBJECT or EMBED tag for the Java plugin
jsp:element	Defines XML elements dynamically.
jsp:attribute	Defines dynamically defined XML element's attribute.

jsp:body	Defines dynamically defined XML element's body.
jsp:text	Use to write template text in JSP pages and documents.

Common Attributes:

There are two attributes that are common to all Action elements: the **id**attribute and the **scope** attribute.

- **Id attribute:** The id attribute uniquely identifies the Action element, and allows the action to be referenced inside the JSP page. If the Action creates an instance of an object the id value can be used to reference it through the implicit object PageContext

- **Scope attribute:** This attribute identifies the lifecycle of the Action element. The id attribute and the scope attribute are directly related, as the scope attribute determines the lifespan of the object associated with the id. The scope attribute has four possible values: (a) page, (b)request, (c)session, and (d) application.

The <jsp:include> Action

This action lets you insert files into the page being generated. The syntax looks like this:

```
<jsp:include page="relative URL" flush="true" />
```

Unlike the **include** directive, which inserts the file at the time the JSP page is translated into a servlet, this action inserts the file at the time the page is requested.

Following is the list of attributes associated with include action:

Attribute	Description
page	The relative URL of the page to be included.
flush	The boolean attribute determines whether the included resource has its

	buffer flushed before it is included.

Example:

Let us define following two files (a)date.jsp and (b) main.jsp as follows:

Following is the content of date.jsp file:

```
<p>
   Today's date: <%= (new java.util.Date()).toLocaleString()%>
</p>
```

Here is the content of main.jsp file:

```
<html>
<head>
<title>The include Action Example</title>
</head>
<body>
<center>
<h2>The include action Example</h2>
<jsp:include page="date.jsp" flush="true" />
</center>
</body>
</html>
```

Now let us keep all these files in root directory and try to access main.jsp. This would display result something like this:

```
The include action Example

   Today's date: 12-Sep-2010 14:54:22
```

The <jsp:useBean> Action

The **useBean** action is quite versatile. It first searches for an existing object utilizing the id and scope variables. If an object is not found, it then tries to create the specified object.

The simplest way to load a bean is as follows:

```
<jsp:useBean id="name" class="package.class" />
```

Once a bean class is loaded, you can use **jsp:setProperty** and**jsp:getProperty** actions to modify and retrieve bean properties.

Following is the list of attributes associated with useBean action:

Attribute	Description
class	Designates the full package name of the bean.
type	Specifies the type of the variable that will refer to the object.
beanName	Gives the name of the bean as specified by the instantiate () method of the java.beans.Beans class.

Let us discuss about **jsp:setProperty** and **jsp:getProperty** actions before giving a valid example related to these actions.

The <jsp:setProperty> Action

The **setProperty** action sets the properties of a Bean. The Bean must have been previously defined before this action. There are two basic ways to use the setProperty action:

You can use jsp:setProperty after, but outside of, a jsp:useBean element, as below:

```
<jsp:useBean id="myName" ... />

...

<jsp:setProperty name="myName" property="someProperty" .../>
```

In this case, the jsp:setProperty is executed regardless of whether a new bean was instantiated or an existing bean was found.

A second context in which jsp:setProperty can appear is inside the body of a jsp:useBean element, as below:

```
<jsp:useBean id="myName" ... >

...

   <jsp:setProperty name="myName" property="someProperty" .../>

</jsp:useBean>
```

Here, the jsp:setProperty is executed only if a new object was instantiated, not if an existing one was found.

Following is the list of attributes associated with setProperty action:

Attribute	**Description**
name	Designates the bean whose property will be set. The Bean must have been previously defined.
property	Indicates the property you want to set. A value of "*" means that all request parameters whose names match bean property names will be passed to the appropriate setter methods.
value	The value that is to be assigned to the given property. The the parameter's value is null, or the parameter does not exist, the setProperty action is ignored.

param	The param attribute is the name of the request parameter whose value the property is to receive. You can't use both value and param, but it is permissible to use neither.

The <jsp:getProperty> Action

The **getProperty** action is used to retrieve the value of a given property and converts it to a string, and finally inserts it into the output.

The getProperty action has only two attributes, both of which are required ans simple syntax is as follows:

```
<jsp:useBean id="myName" ... />

...

<jsp:getProperty name="myName" property="someProperty" .../>
```

Following is the list of required attributes associated with setProperty action:

Attribute	Description
name	The name of the Bean that has a property to be retrieved. The Bean must have been previously defined.
property	The property attribute is the name of the Bean property to be retrieved.

Example:

Let us define a test bean which we will use in our example:

```
/* File: TestBean.java */

package action;


public class TestBean {

   private String message = "No message specified";
   public String getMessage() {
      return(message);
   }
   public void setMessage(String message) {
      this.message = message;
   }
}
```

Compile above code to generated TestBean.class file and make sure that you copied TestBean.class in C:\apache-tomcat-7.0.2\webapps\WEB-INF\classes\action folder and CLASSPATH variable should also be set to this folder:

Now use the following code in main.jsp file which loads the bean and sets/gets a simple String parameter:

```
<html>
<head>
<title>Using JavaBeans in JSP</title>
</head>
<body>
<center>
<h2>Using JavaBeans in JSP</h2>

<jsp:useBean id="test" class="action.TestBean" />

<jsp:setProperty name="test"
                    property="message"
                    value="Hello JSP..." />

<p>Got message....</p>
```

```
<jsp:getProperty name="test" property="message" />


</center>

</body>

</html>
```

Now try to access main.jsp, it would display following result:

```

```

Using JavaBeans in JSP

```
Got                                                                      message....

Hello JSP...
```

The <jsp:forward> Action

The **forward** action terminates the action of the current page and forwards the request to another resource such as a static page, another JSP page, or a Java Servlet.

The simple syntax of this action is as follows:

```
<jsp:forward page="Relative URL" />
```

Following is the list of required attributes associated with forward action:

Attribute	**Description**
page	Should consist of a relative URL of another resource such as a static page, another JSP page, or a Java Servlet.

Example:

Let us reuse following two files (a) date.jsp and (b) main.jsp as follows:

Following is the content of date.jsp file:

```
<p>
   Today's date: <%= (new java.util.Date()).toLocaleString()%>
</p>
```

Here is the content of main.jsp file:

```
<html>
<head>
<title>The include Action Example</title>
</head>
<body>
<center>
<h2>The include action Example</h2>
<jsp:forward page="date.jsp" />
</center>
</body>
</html>
```

Now let us keep all these files in root directory and try to access main.jsp. This would display result something like as below. Here it discarded content from main page and displayed content from forwarded page only.

```
   Today's date: 12-Sep-2010 14:54:22
```

The <jsp:plugin> Action

The **plugin** action is used to insert Java components into a JSP page. It determines the type of browser and inserts the <object> or <embed> tags as needed.

If the needed plugin is not present, it downloads the plugin and then executes the Java component. The Java component can be either an Applet or a JavaBean.

The plugin action has several attributes that correspond to common HTML tags used to format Java components. The <param> element can also be used to send parameters to the Applet or Bean.

Following is the typical syntax of using plugin action:

```
<jsp:plugin type="applet" codebase="dirname" code="MyApplet.class"
                  width="60" height="80">
  <jsp:param name="fontcolor" value="red" />
  <jsp:param name="background" value="black" />

  <jsp:fallback>
    Unable to initialize Java Plugin
  </jsp:fallback>

</jsp:plugin>
```

You can try this action using some applet if you are interested. A new element, the <fallback> element, can be used to specify an error string to be sent to the user in case the component fails.

The <jsp:element> Action

The <jsp:attribute> Action

The <jsp:body> Action

The <jsp:element>, lt;jsp:attribute> and <jsp:body> actions are used to define XML elements dynamically. The word dynamically is important, because it means that the XML elements can be generated at request time rather than statically at compile time.

Following is a simple example to define XML elements dynamically:

```
<%@page language="java" contentType="text/html"%>
<html xmlns="http://www.w3c.org/1999/xhtml"
    xmlns:jsp="http://java.sun.com/JSP/Page">
```

```
<head><title>Generate XML Element</title></head>

<body>

<jsp:element name="xmlElement">

<jsp:attribute name="xmlElementAttr">

   Value for the attribute

</jsp:attribute>

<jsp:body>

   Body for XML element

</jsp:body>

</jsp:element>

</body>

</html>
```

This would produce following HTML code at run time:

```
<html xmlns="http://www.w3c.org/1999/xhtml"

      xmlns:jsp="http://java.sun.com/JSP/Page">



<head><title>Generate XML Element</title></head>

<body>

<xmlElement xmlElementAttr="Value for the attribute">

   Body for XML element

</xmlElement>

</body>

</html>
```

The <jsp:text> Action

The <jsp:text> action can be used to write template text in JSP pages and documents. Following is the simple syntax for this action:

```
<jsp:text>Template data</jsp:text>
```

The body fo the template cannot contain other elements; it can only contain text and EL expressions (Note: EL expressions are explained in subsequent chapter). Note that in XML files, you cannot use expressions such as ${whatever > 0}, because the greater than signs are illegal. Instead, use the gt form, such as ${whatever gt 0} or an alternative is to embed the value in a CDATA section.

```
<jsp:text><![CDATA[<br>]]></jsp:text>
```

If you need to include a DOCTYPE declaration, for instance

Where the **uri** attribute value resolves to a location the container understands and the **prefix** attribute informs a container what bits of markup are custom actions.

You can write XML equivalent of the above syntax as follows:

```
<jsp:directive.taglib uri="uri" prefix="prefixOfTag" />
```

JSP with XML tags

Write a JSP Document (XML-based document) that uses the correct syntax.

What is a JSP document? (JSP6.1)

The key word here is document as opposed to page, though sometimes the two terms are used interchangeably. Everywhere else we have been considering JSP pages which are not fully formed xml documents. To allow JSP documents to be manipulated by standard XML programs the tags need to be in standardised XML structure. Thus for example you might have a JSP page where you are using a Java XML parser to transform all of one tag occurrence to another.

When I first discovered the existence of XML equivalents of JSP tags I tried converting existing pages on the mistaken that “XML is good”. This was an unrewarding experience and I suspect that the XML tag equivalents are only used where there is a definite need to automatically parse or generate valid JSP pages. This is probably a reason to pay additional attention to this topic, as most JSP you come across will not use the XML format tags. Note that in earlier versions of JSP a document had to begin with the <jsp:root element but this is not true of JSP 2.

Throughout this book I have tried to include the XML equivalent for each of the tags, thus for the import page tag I had

```
<%@ page import=”java.util.*,java.io.*” %>
```

The xml equivalent tag is

```
<jsp:directive.page import="java.util.*"/>
```

<jsp:expression

The <jsp:expression tag is the equivalent of the <%= tag. When translated it generates Java code that has an output terminated by a semi-colon. It is very similar to generating a System.out.print(“somestring”); in traditional Java programming. The following two lines of code are functionally equivalent.

```
<%= request.getParameter("username") %>

<jsp:expression>request.getParameter("username")</jsp:expression>
```

<jsp:scriptlet

The <jsp:scriptlet tag is the equivalent of the <% tag. Thus the following lines of code are equivalent.

```
<html>
<%
  for(int i =0; i <2; i++){
       out.print(i);
  }
```

```
%>

<jsp:scriptlet>
   <![CDATA[
  for(int i =0; i < 2; i++){
      out.print(i);
  }
  ]]>
</jsp:scriptlet>
```

Note the use of the <![CDATA[tags which effectively “escapes” the following code. The reason for this is so that the less than symbol < does not get interpreted as the start of the closure of the <jsp:scriptlet tag

<jsp:declaration

The servlet code generated for declarations is external to the *service* method and the tag can be used to create class fields and for the creation of methods.

```
<jsp:declaration>

   public String amethod() {
   return "amethod";
   }
</jsp:declaration>
```

Which is the equivalent of

```
<%!
  public String amethod() {
   return "amethod";
   }
%>
```

<jsp:directive.directivetext

JSP directives, i.e. Those tags that start with <%@ characters can be created with XML syntax in the form <jsp:directive.directivetext. The following are examples of JSP directives with the page syntax followed by the document (XML) syntax.

```
<%@ include file="standac.jsp"%>
<jsp:directive.include file="standac.jsp"/>

<%@ page import="java.util.*" %>
<jsp:directive.page import="java.util.*" />

<%@ page errorPage="errorPage" %>
<jsp:directive.page errorPage = "errorPage.jsp"/>
<%@ page isErrorPage="true" %>
<jsp:directive.page isErrorPage="true"/>
```

<jsp:text

The <jsp:text tag is designed for outputting template text, so it can be considered the equivalent of plain text within a jsp page. According to the Sun J2EE tutorial at

http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSPX3.html

“If you use jsp:text, all white space is preserved. “

However in my tests white space is preserved without the <jsp:text tag. I suspect that this tag is mainly for the convenience of parsers to flag up what is template text.

<!--

The <!-- is the standard XML comment tag and is the equivalent of the JSP <%-- tag. Unfortunately it does not work quite correctly as a JSP document comment tag as there is a clash with the meaning of the same tag in standard HTML. The contents of the <!-- tag are sent to the output stream, and thus appear in the final HTML page. I'm not quite sure

if this should be considered a JSP document tag, but I feel it is unlikely to come up as a topic in the exam.

Other sources

Writing JSPs in XML by Stephanie Fesler http://www.onjava.com/pub/a/onjava/2001/11/28/jsp_xml.html

Carl Trusiak of Javaranch on JSP xml tags http://www.javaranch.com/newsletter/Feb2002/xmljsp.html

According to Sun http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSPX3.html

More JSP best practices by Dustin Marx http://www.javaworld.com/javaworld/jw-07-2003/jw-0725-morejsp.html

This objective according to Mikalai Zaikin http://java.boot.by/wcd-guide/ch06s03.html

JSP TAGS

These elements are used to identify the tag library, and are nested within the <taglib> ... </taglib> element.

<tlibversion>version_number</tlibversion>

(Required) The version number of the tag library. The current version number is 1.0

<jspversion>version_number</jspversion>

(Optional) The JSP version that this tag library is designed to work with. WebLogic supposrts JSP version 1.1 from this release.

<shortname>*TagLibraryName*</shortname>

(Required) Assigns a short name to this tag library. Currently, this elements is not used by WebLogic.

<uri>*unique_string*</uri>

(Optional) Currently, this element is not used by WebLogic.

<info>*...text...*</info>

(Optional) Use this element to provide a description of the tag library. Currently, this element is not used by WebLogic.

Defining a tag

Use a separate <tag element to define each new tag in the tag library. The <tag> element takes the following nested tags.

<tag>

<name>tag_name</name>

(Required) Defines the name of the tag. This is used in the referencing JSP file after the : symbol, such as:

<mytaglib:tag_name>

For more details, see Using custom tags in a JavaServer page.

<tagclass>package.class.name</tagclass>

(Required) Declares the tag handler class that implements the functionality of this tag. You must specify the fully qualified package name of the class. The class files should be located under the WEB-INF/classes directory, under a directory structure reflecting the package name. For more detail on writing the tag handler class, see Implementing the tag handler.

<teiclass>package.class.name</teiclass>

(Optional) Declares the subclass of TagExtraInfo that describes the scripting variables introduced by this tag. If your tag does not define new scripting variables, it does not use this element.

You must specify the fully qualified package name of the class. The class files should be located under the WEB-INF/classes directory, under a directory structure reflecting the package name. For more detail on writing the tag handler class, see Defining new scripting variables.

<bodycontent>tagdependent | JSP | empty</bodycontent>

(Optional) Defines the content of the tag body. If you specify empty, then the tag must appear in the *empty-tag* format <taglib:tagname\> in the JSP page. If you specify JSP, the contents of the tag can be interpreted as JSP, and the tag must be used in the *body-tag* format <taglib:tagname>. You should specify tagdependent if your tag expects to interpret the contents of the body as non-JSP, for example an SQL statement.

If the <bodycontent> element is not defined, the default value is JSP.

<attribute>

Use a separate <attribute> element to define each attribute that the tag can take. Tag attributes are useful for allowing the JSP author to alter the behavior of your tags.

<name>attr_name</name>

(Required) Defines the name of the attribute as it appears in the tag element in the JSP page. Such as:
<taglib:mytag **attr_name**="xxxxxx">

<required>true | false</required>

(Optional) Defines whether this attribute has optional use in the JSP page. If not defined here, the default is false -- that is, the attribute is optional by default. If true is specified, and the attribute is not used in a JSP page, a translation-time error will occur.

<rtexprvalue>true | false</rtexprvalue>

(Optional) Defines whether this attribute can take a scriptlet expression as a value, allowing it to be dynamically calculated at request time.

The default value false is assumed if this element is not specified.

</attribute>

</tag>

Sample JSP program for <jsp:forward page=”url”>

<u>E:\j2ee\myjsp\forward.html</u>

```
<html>
<head><title>Welcome to my JSP</title></head>
<body>
</h1>Forward action test</h1>
<form method="post" action="http://localhost:7001/myjsp/forward.jsp">
<p>Please enter your username:
<input type="text" name="username">
<br>and password:
<input type="password" name="password">
</p>
<p><input type="submit" value="Log in">
</form>
</body>
</html>
```

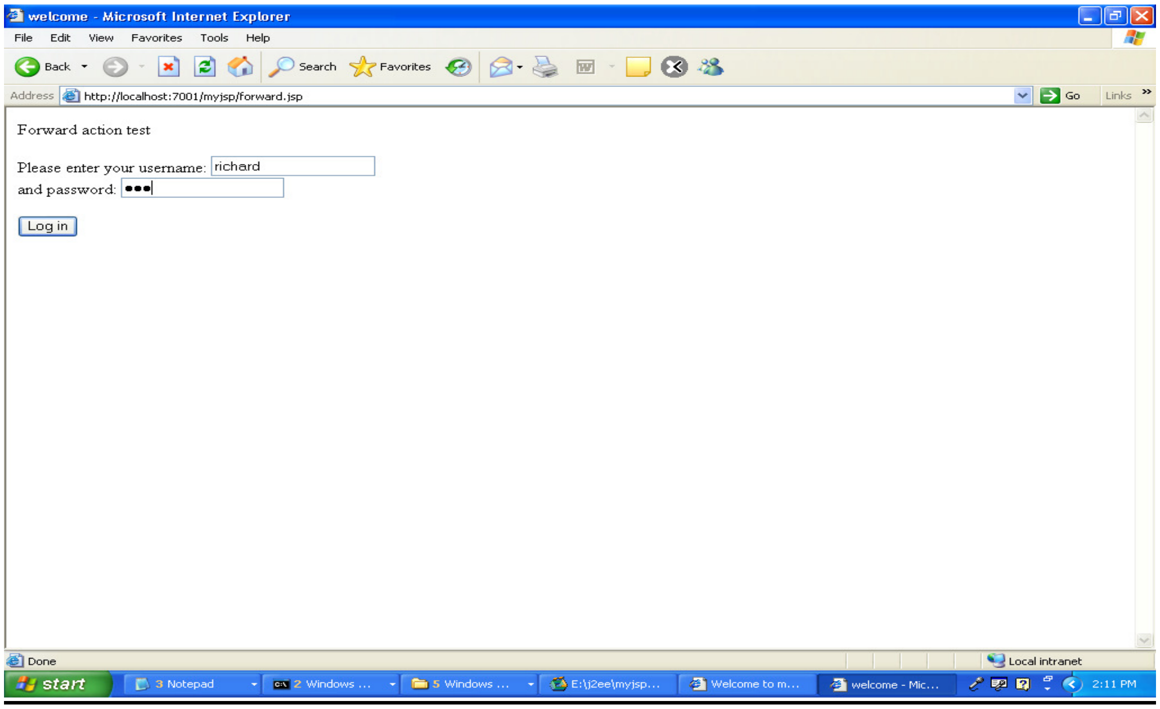
<u>E:\j2ee\myjsp\forward.jsp</u>

```
<html>
<head><title>welcome</title></head>
</body>
<%
if((request.getParameter("username").equals("richard"))&&
(request.getParameter("password").equals("xyz"))){
%>
<jsp:forward page="forward2.jsp"/>
<% } else { %>
<%@ include file="forward.html"%>
<%} %>
</body>
</html>
```

<u>E:\j2ee\myjsp\forward2.jsp</u>

```
<html>
<head><title>Forward action test: </title></head>
<body>
<h1>Forward action test:Login successful</h1>
<form method="post" action="forward.jsp">
<p><b>Welcome, <%= request.getParameter("username") %> !</b>

</body>
</html>
```

<u>Input screen:</u>

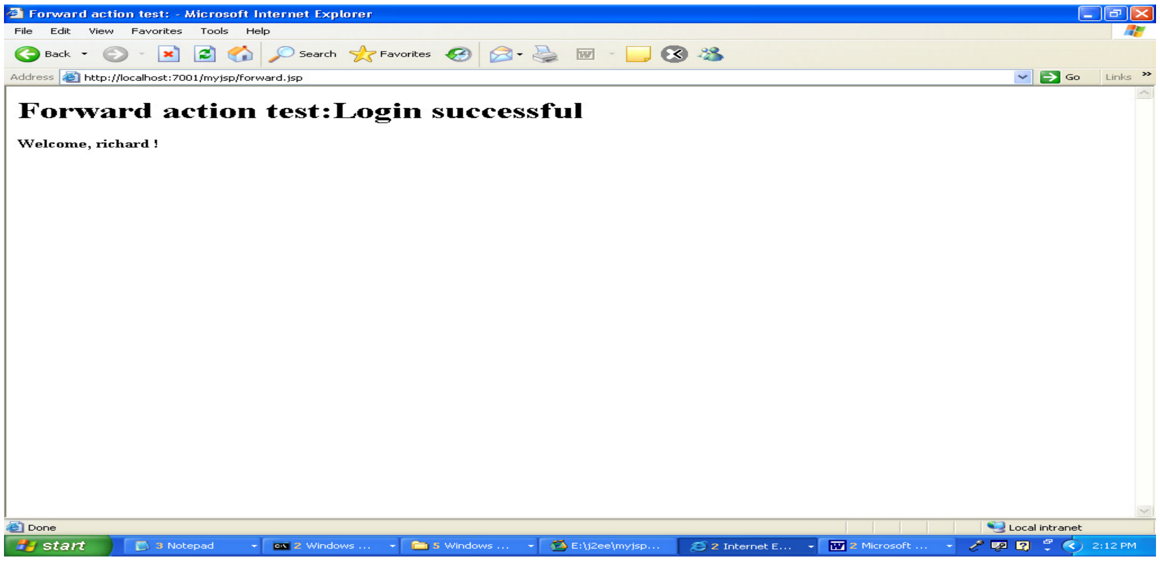

welcome - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:7001/myjsp/forward.jsp
Go Links
Done
Local intranet
start 3 Notepad 2 Windows ... 5 Windows ... E:\j2ee\myjsp... Welcome to m... welcome - Mic... 2:11 PM

Forward action test

Please enter your username: richard
and password: ●●●

Log in

<u>output screen:</u>


Forward action test: - Microsoft Internet Explorer
http://localhost:7001/myjsp/forward.jsp

Forward action test:Login successful

Welcome, richard !

JAVA MAIL

The JavaMail^{TM} API provides classes that model a mail system. The javax.mail package defines classes that are common to all mail systems. Thejavax.mail.internet package defines classes that are specific to mail systems based on internet standards such as MIME, SMTP, POP3, and IMAP. The JavaMail API includes the javax.mail package and subpackages.

For an overview of the JavaMail API, read the JavaMail specification.

The code to send a plain text message can be as simple as the following:

```
Properties props = new Properties();
props.put("mail.smtp.host", "my-mail-server");
Session session = Session.getInstance(props, null);

try {
    MimeMessage msg = new MimeMessage(session);
    msg.setFrom("me@example.com");
    msg.setRecipients(Message.RecipientType.TO,
                      "you@example.com");
```

```
        msg.setSubject("JavaMail hello world example");
        msg.setSentDate(new Date());
        msg.setText("Hello, world!\n");
        Transport.send(msg, "me@example.com", "my-password");
    } catch (MessagingException mex) {
        System.out.println("send failed, exception: " + mex);
    }
```

The JavaMail download bundle contains many more complete examples in the "demo" directory.

Don't forget to see the JavaMail API FAQ for answers to the most common questions. The JavaMail web site contains many additional resources.

Properties

The JavaMail API supports the following standard properties, which may be set in the Session object, or in the Properties object used to create the Sessionobject. The properties are always set as strings; the Type column describes how the string is interpreted. For example, use

```
        props.put("mail.debug", "true");
```

to set the mail.debug property, which is of type boolean.

Name	**Type**	**Description**
mail.debug	boolean	The initial debug mode. Default is false.
mail.from	String	The return email address of the current user, used by the InternetAddress method getLocalAddress.
mail.mime.address.strict	boolean	The MimeMessage class uses the InternetAddress method parseHeader to parse headers in messages. This property controls the strict flag passed to the parseHeader method. The default is true.
mail.host	String	The default host name of the mail server for both Stores and Transports. Used if the mail.*protocol*.host property

		isn't set.
mail.store.protocol	String	Specifies the default message access protocol. The Session method getStore() returns a Store object that implements this protocol. By default the first Store provider in the configuration files is returned.
mail.transport.protocol	String	Specifies the default message transport protocol. The Session method getTransport() returns a Transport object that implements this protocol. By default the first Transport provider in the configuration files is returned.
mail.user	String	The default user name to use when connecting to the mail server. Used if the mail.*protocol*.user property isn't set.
mail.*protocol*.class	String	Specifies the fully qualified class name of the provider for the specified protocol. Used in cases where more than one provider for a given protocol exists; this property can be used to specify which provider to use by default. The provider must still be listed in a configuration file.
mail.*protocol*.host	String	The host name of the mail server for the specified protocol. Overrides the mail.host property.
mail.*protocol*.port	int	The port number of the mail server for the specified protocol. If not specified the protocol's default port number is used.
mail.*protocol*.user	String	The user name to use when connecting to mail servers using the specified protocol. Overrides the mail.user property.

The following properties are supported by the reference implementation (RI) of JavaMail, but are not currently a required part of the specification. The names, types, defaults, and semantics of these properties may change in future releases.

Name	**Type**	**Description**
mail.debug.auth	boolean	Include protocol authentication commands (including usernames and passwords) in the debug output. Default is false.
mail.transport.protocol.*address-type*	String	Specifies the default message transport protocol for the specified address type. The Sessionmethod getTransport(Address) returns a Transport object that implements this protocol when the address is of the specified type (e.g., "rfc822" for standard internet addresses). By default the first Transport configured for that address type is used. This property can be used to override the behavior of the send method of the Transport class so that (for example) the "smtps" protocol is used instead of the "smtp" protocol by setting the propertymail.transport.protocol.rfc822 to "smtps".
mail.event.scope	String	Controls the scope of events. (See the javax.mail.event package.) By default, a separate event queue and thread is used for events for each Store, Transport, or Folder. If this property is set to "session", all such events are put in a single event queue processed by a single thread for the current session. If this property is set to "application", all

		such events are put in a single event queue processed by a single thread for the current application. (Applications are distinguished by their context class loader.)
mail.event.executor	java.util.concurrent.Executor	By default, a new Thread is created for each event queue. This thread is used to call the listeners for these events. If this property is set to an instance of an Executor, the Executor.execute method is used to run the event dispatcher for an event queue. The event dispatcher runs until the event queue is no longer in use.

The JavaMail API also supports several System properties; see the javax.mail.internet package documentation for details.

The JavaMail reference implementation includes protocol providers in subpackages of com.sun.mail. Note that the APIs to these protocol providers are not part of the standard JavaMail API. Portable programs will not use these APIs.

Nonportable programs may use the APIs of the protocol providers by (for example) casting a returned Folder object to a com.sun.mail.imap.IMAPFolderobject. Similarly for Store and Message objects returned from the standard JavaMail APIs.

The protocol providers also support properties that are specific to those providers. The package documentation for the IMAP, POP3, and SMTP packages provide details.

In addition to printing debugging output as controlled by the Session configuration, the current implementation of classes in this package log the same information using Logger as described in the following table:

Logger Name	Logging Level	Purpose
javax.mail	CONFIG	Configuration of the Session
javax.mail	FINE	General debugging output

JAVA MAIL PROGRAM

```
import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class SendEmail
{
 public static void main(String [] args){
      String to = "sonoojaiswal1988@gmail.com";//change accordingly
      String from = "sonoojaiswal1987@gmail.com";change accordingly
      String host = "localhost";//or IP address

     //Get the session object
      Properties properties = System.getProperties();
      properties.setProperty("mail.smtp.host", host);
      Session session = Session.getDefaultInstance(properties);

     //compose the message
      try{
         MimeMessage message = new MimeMessage(session);
         message.setFrom(new InternetAddress(from));
         message.addRecipient(Message.RecipientType.TO,new InternetAddress(to)
);
         message.setSubject("Ping");
```

```
        message.setText("Hello, this is example of sending email  ");

        // Send message
        Transport.send(message);
        System.out.println("message sent successfully....");

      }catch (MessagingException mex) {mex.printStackTrace();}
   }
}
```

SCSX1022_J2EE

UNIT IV

Enterprise Java Beans

Enterprise JavaBeans (EJB) is a managed, server software for modular construction of enterprise software, and one of several Java APIs. EJB is a server-side software component that encapsulates the business logic of an application. The EJB specification is a subset of the Java EE specification. An EJB web container provides a runtime environment for web related software components, including computer security, Java servlet lifecycle management, transaction processing, and other web services.

Types of Enterprise Beans

Table summarizes the three types of enterprise beans. The following sections discuss each type in more detail.

Table : Enterprise Bean Types

Enterprise Bean Type	**Purpose**
Session	Performs a task for a client; implements a web service
Entity	Represents a business entity object that exists in persistent storage
Message-Driven	Acts as a listener for the Java Message Service API, processing messages asynchronously

SESSION BEAN

A *session bean* represents a single client inside the Application Server. To access an application that is deployed on the server, the client invokes the session bean's methods. The session bean performs work for its client, shielding the client from complexity by executing business tasks inside the server.

As its name suggests, a session bean is similar to an interactive session. A session bean is not shared; it can have only one client, in the same way that an interactive session can have only one user. Like an interactive session, a session bean is not persistent. (That is, its data is not saved to a database.) When the client terminates, its session bean appears to terminate and is no longer associated with the client.

State Management Modes

There are two types of session beans: stateless and stateful.

Stateless Session Beans

A *stateless* session bean does not maintain a conversational state for the client. When a client invokes the method of a stateless bean, the bean's instance variables may contain a state, but only for the duration of the invocation. When the method is finished, the state is no longer retained. Except during method invocation, all instances of a stateless bean are equivalent, allowing the EJB container to assign an instance to any client.

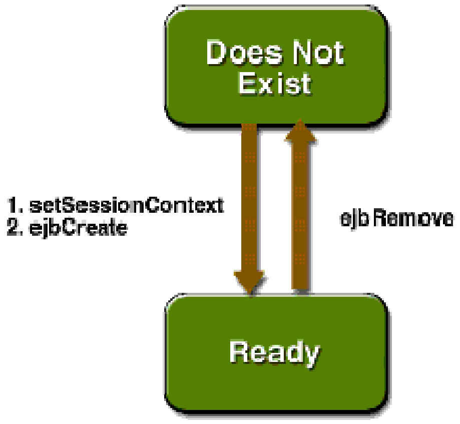
Because stateless session beans can support multiple clients, they can offer better scalability for applications that require large numbers of clients. Typically, an application requires fewer stateless session beans than stateful session beans to support the same number of clients.

At times, the EJB container may write a stateful session bean to secondary storage. However, stateless session beans are never written to secondary storage. Therefore, stateless beans may offer better performance than stateful beans.

A stateless session bean can implement a web service, but other types of enterprise beans cannot.

The Life Cycle of a Stateless Session Bean

Because a stateless session bean is never passivated, its life cycle has only two stages: nonexistent and ready for the invocation of business methods. Figure illustrates the stages of a stateless session bean.



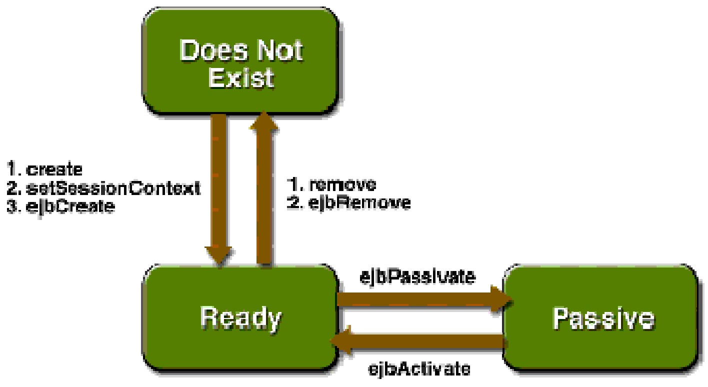
Stateful Session Beans

The state of an object consists of the values of its instance variables. In a *stateful* session bean, the instance variables represent the state of a unique client-bean session. Because the client interacts ("talks") with its bean, this state is often called the *conversational state*.

The state is retained for the duration of the client-bean session. If the client removes the bean or terminates, the session ends and the state disappears. This transient nature of the state is not a problem, however, because when the conversation between the client and the bean ends there is no need to retain the state.

The Life Cycle of a Stateful Session Bean

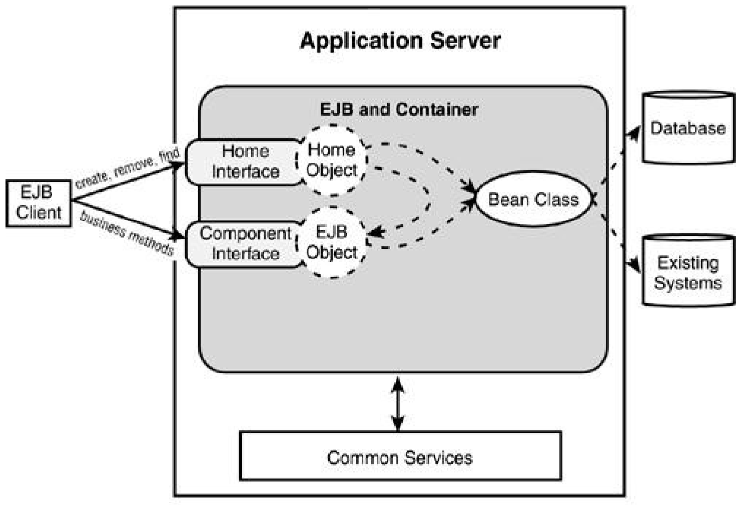
Figure illustrates the stages that a session bean passes through during its lifetime. The client initiates the life cycle by invoking the create method. The EJB container instantiates the bean and then invokes the setSessionContext and ejbCreate methods in the session bean. The bean is now ready to have its business methods invoked.



While in the ready stage, the EJB container may decide to deactivate, or *passivate*, the bean by moving it from memory to secondary storage. (Typically, the EJB container uses a least-recently-used algorithm to select a bean for passivation.) The EJB container invokes the bean's ejbPassivate method immediately before passivating it. If a client invokes a business method on the bean while it is in the passive stage, the EJB container activates the bean, calls the bean's ejbActivate method, and then moves it to the ready stage.

At the end of the life cycle, the client invokes the remove method, and the EJB container calls the bean's ejbRemove method. The bean's instance is ready for garbage collection.

EJB Architechture



EJB and Container

EJB CONTAINERS

Starting with EJB 3.1, the EJB specification defines two variants of the EJB container; a full version and a limited version. The limited version adheres to a proper subset of the specification called EJB 3.1 Lite [35][36] and is part of Java EE 6's web profile (which is itself a subset of the full Java EE 6 specification).

EJB 3.1 Lite excludes support for the following features:

- Remote interfaces
- RMI-IIOP Interoperability
- JAX-WS Web Service Endpoints
- EJB Timer Service (@Schedule, @Timeout)
- Asynchronous session bean invocations (@Asynchronous)
- Message-driven beans

EJB 3.2 Lite excludes less features. Particularly it no longer excludes @Asynchronous and @Schedule/@Timeout, but for @Schedule it does not support the "persistent" attribute that full EJB 3.2 does support.

The complete excluded list for EJB 3.2 Lite is:

- Remote interfaces
- RMI-IIOP Interoperability
- JAX-WS Web Service Endpoints
- Persistent timers ("persistent" attribute on @Schedule)
- Message-driven beans
- Container Services
- The container provides various services for the EJB to relieve the developer from having to implement such services, namely
- *Distribution via proxies*—The container will generate a client-side stub and server-side skeleton for the EJB. The stub and skeleton will use RMI over IIOP to communicate.

ENTITY BEANS

An *entity bean* represents a business object in a persistent storage mechanism. Some examples of business objects are customers, orders, and products. In the Application Server, the persistent storage mechanism is a relational database. Typically, each entity bean has an underlying table in a relational database, and each instance of the bean corresponds to a row in that table.

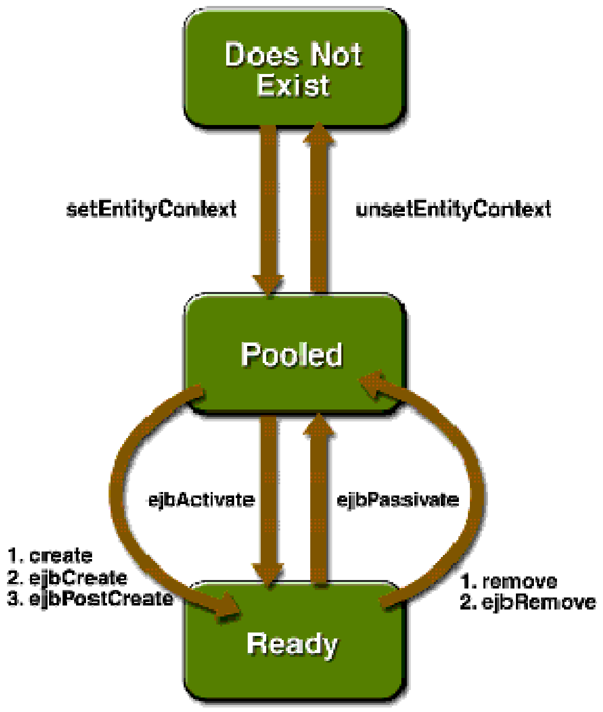
The Life Cycle of an Entity Bean

Figure shows the stages that an entity bean passes through during its lifetime. After the EJB container creates the instance, it calls the setEntityContext method of the entity bean class. The setEntityContext method passes the entity context to the bean.

After instantiation, the entity bean moves to a pool of available instances. While in the pooled stage, the instance is not associated with any particular EJB object identity. All instances in the pool are identical. The EJB container assigns an identity to an instance when moving it to the ready stage.

There are two paths from the pooled stage to the ready stage. On the first path, the client invokes the create method, causing the EJB container to call the ejbCreate and ejbPostCreate methods. On the second path, the EJB container invokes the ejbActivate method. While an entity bean is in the ready stage, and its business methods can be invoked.

There are also two paths from the ready stage to the pooled stage. First, a client can invoke the remove method, which causes the EJB container to call the ejbRemove method. Second, the EJB container can invoke the ejbPassivate method.



At the end of the life cycle, the EJB container removes the instance from the pool and invokes the unsetEntityContext method.

What Makes Entity Beans Different from Session Beans?

Entity beans differ from session beans in several ways. Entity beans are persistent, allow shared access, have primary keys, and can participate in relationships with other entity beans.

Persistence

Because the state of an entity bean is saved in a storage mechanism, it is persistent. *Persistence* means that the entity bean's state exists beyond the lifetime of the application or the Application Server process. If you've worked with databases, you're familiar with persistent data. The data in a database is persistent because it still exists even after you shut down the database server or the applications it services.

There are two types of persistence for entity beans: bean-managed and container-managed. With *bean-managed* persistence, the entity bean code that you write contains the calls that access the database. If your bean has *container-managed* persistence, the EJB container automatically generates the necessary database access calls.

MESSAGE DRIVEN BEAN

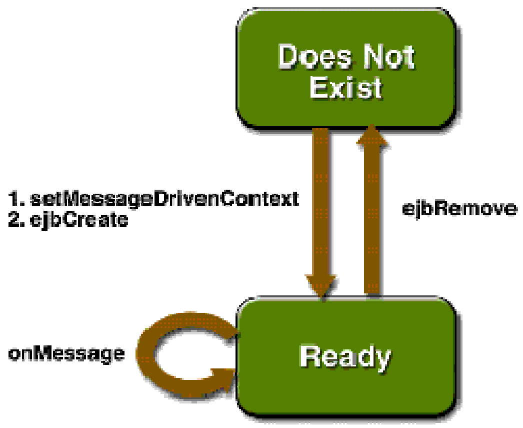
A *message-driven bean* is an enterprise bean that allows J2EE applications to process messages asynchronously. It normally acts as a JMS message listener, which is similar to an event listener except that it receives JMS messages instead of events. The messages can be sent by any J2EE component--an application client, another enterprise bean, or a web component--or by a JMS application or system that does not use J2EE technology. Message-driven beans can process either JMS messages or other kinds of messages.

The Life Cycle of a Message-Driven Bean

Figure illustrates the stages in the life cycle of a message-driven bean.

The EJB container usually creates a pool of message-driven bean instances. For each instance, the EJB container instantiates the bean and performs these tasks:

1. It calls the setMessageDrivenContext method to pass the context object to the instance.
2. It calls the instance's ejbCreate method.



Like a stateless session bean, a message-driven bean is never passivated, and it has only two states: nonexistent and ready to receive messages.

At the end of the life cycle, the container calls the ejbRemove method. The bean's instance is then ready for garbage collection.

What Makes Message-Driven Beans Different from Session and Entity Beans?

The most visible difference between message-driven beans and session and entity beans is that clients do not access message-driven beans through interfaces. Unlike a session or entity bean, a message-driven bean has only a bean class.

In several respects, a message-driven bean resembles a stateless session bean.

- A message-driven bean's instances retain no data or conversational state for a specific client.
- All instances of a message-driven bean are equivalent, allowing the EJB container to assign a message to any message-driven bean instance. The container can pool these instances to allow streams of messages to be processed concurrently.
- A single message-driven bean can process messages from multiple clients.

Message-driven beans have the following characteristics:

- They execute upon receipt of a single client message.
- They are invoked asynchronously.
- They are relatively short-lived.
- They do not represent directly shared data in the database, but they can access and update this data.
- They can be transaction-aware.
- They are stateless.

When a message arrives, the container calls the message-driven bean's onMessage method to process the message. The onMessage method normally casts the message to one of the five JMS message types and handles it in accordance with the application's business logic. The onMessage method can call helper methods, or it can invoke a session or entity bean to process the information in the message or to store it in a database.

A message can be delivered to a message-driven bean within a transaction context, so all operations within the onMessage method are part of a single transaction. If message processing is rolled back, the message will be redelivered.

When to Use Message-Driven Beans

Session beans and entity beans allow you to send JMS messages and to receive them synchronously, but not asynchronously. To avoid tying up server resources, you may prefer not to use blocking synchronous receives in a server-side component. To receive messages asynchronously, use a message-driven bean.

EJB CONTAINER SERVICES

- *Lifecycle management*—Bean initialization, state management, and destruction is driven by the container, all the developer must do is implement the appropriate methods.
- *Naming and registration*—The EJB container and server will provide the EJB with access to naming services. These services are used by local and remote clients to look up the EJB and by the EJB itself to look up resources it may need.
- *Transaction management*—Declarative transactions provide a means for the developer to easily delegate the creation and control of transactions to the container.
- *Security and access control*—Again, declarative security provides a means for the developer to easily delegate the enforcement of security to the container.

- *Persistence (if you want)*—Using the Entity EJB's container-managed persistence mechanism, state can be saved and restored without having to write a single line of code.

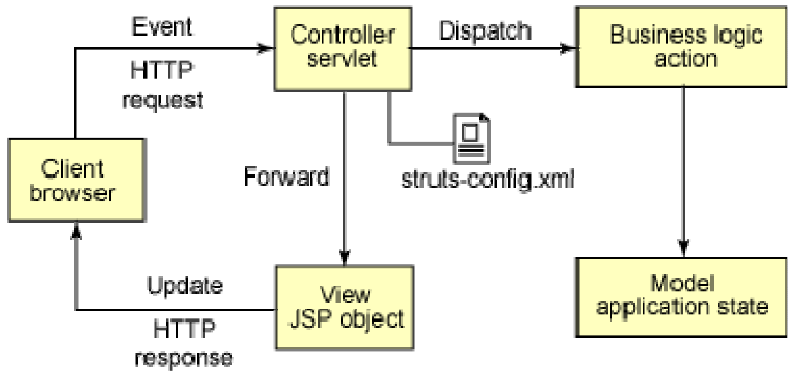
UNIT V

STRUTS:

A brief introduction

Struts, an open source framework you can use to build Web applications, is based on the popular Model-View-Controller (MVC2) design paradigm. The framework is built upon standard technologies like Java Servlets, JavaBeans, ResourceBundles, and XML, and it provides flexible and extensible components. Struts implements the Controller layer in the form of ActionServlet and recommends building the View layer using JSP tag libraries. Struts also provides a wrapper around the Model layer through Action classes. Figure 1 illustrates the Struts framework based on the Model-View-Controller design.

Figure 1.



Overview of the MVC design pattern

The model-view-controller design pattern, also known as Model 2 in J2EE application programming, is a well-established design pattern for programming. Table 1 summarizes the three main components of MVC.

Table 1. Summary of MVC components

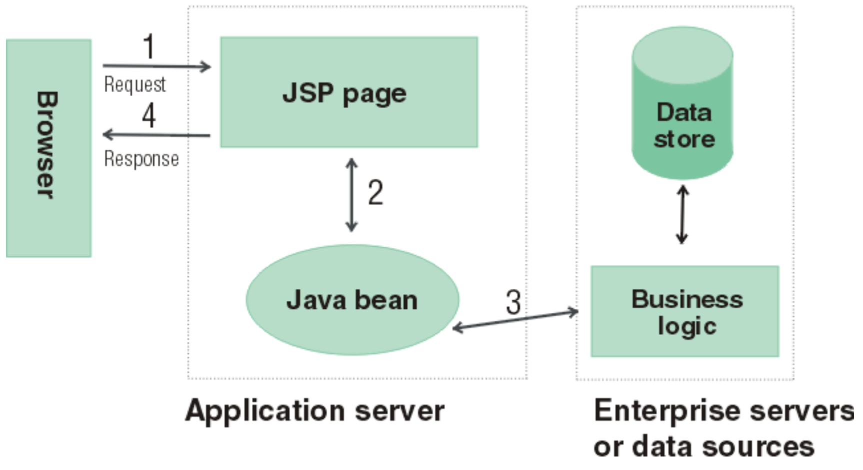
	Purpose	**Description**
Model	Maintain data	Business logic plus one or more data sources such as a relational database
View	Display all or a portion of the data	The user interface that displays information about the model to the user
Controller	Handle events that affect the model or view	The flow-control mechanism means by which the user interacts with the application

Model 1 versus Model 2

The Model 1 and Model 2 architectures both separate content generation (business logic) from the content presentation (HTML formatting). Model 2 differs from Model 1 in the location where the bulk of the request processing is performed: by a controller rather than in the JSP pages.

In the JSP Model 1 architecture, the JSP page alone processes the incoming request and replies to the client, as shown in Figure 1.

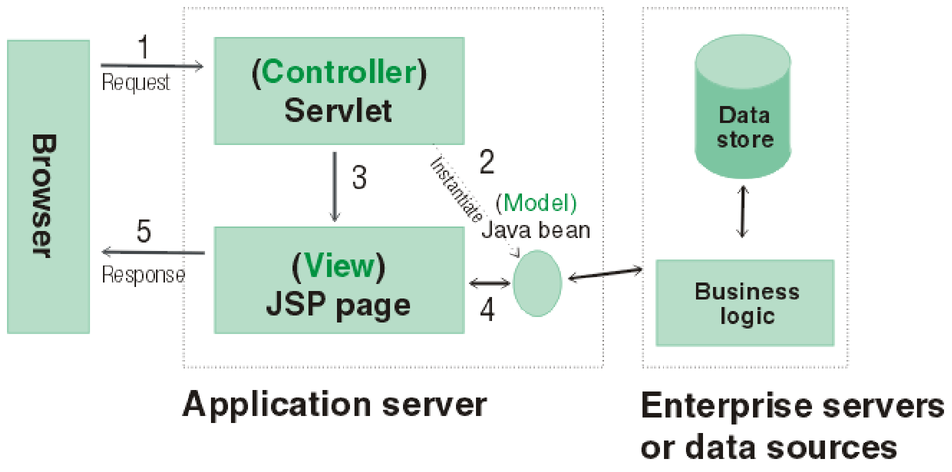
Figure 1. JSP Model 1 architecture



1. The browser sends a request to a JSP page.
2. The JSP page communicates with a Java bean.
3. The Java bean is connected to a database.
4. The JSP page responds to the browser.

In the JSP Model 2 architecture, the servlet processes the request, creates any beans or objects used by the JSP file, and forwards the request, as shown in Figure 2.

Figure 2. JSP Model 2 architecture



1. The browser sends a request to a servlet.
2. The servlet instantiates a Java bean that is connected to a database.
3. The servlet communicates with a JSP page.
4. The JSP page communicates with the Java bean.
5. The JSP page responds to the browser.

Table 2 presents criteria to help you determine when Model 1 or Model 2 is likely to be more appropriate:

Table 2. Guidelines for using Model 1 or Model 2

Criterion	**Model 1**	**Model 2**

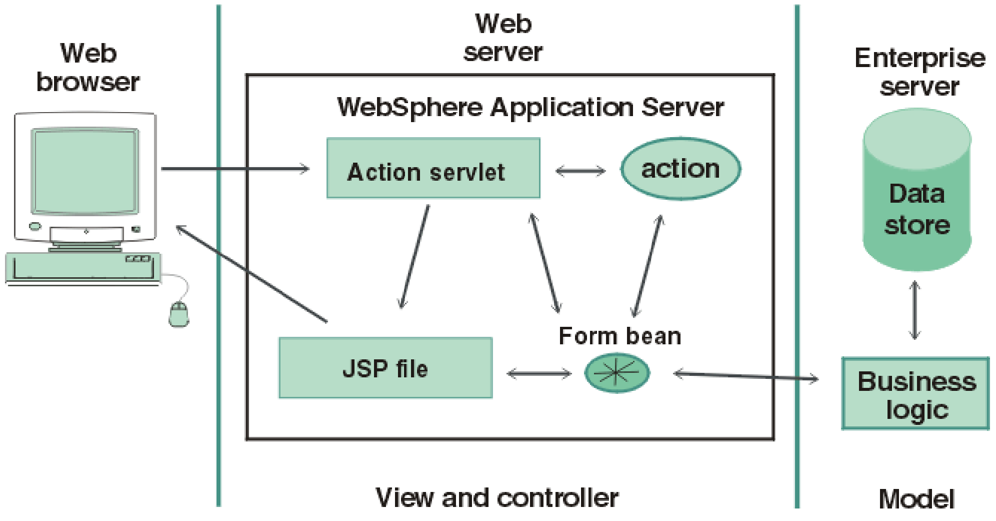
Table 2. Guidelines for using Model 1 or Model 2

Criterion	**Model 1**	**Model 2**
Type of Web application	Simple	Complex
Nature of developer's task	Quick prototyping	Creating an application to be modified and maintained
Who is doing the work	View and controller being done by the same team	View and controller being done by different teams

The Struts implementation of Model 2

The Struts implementation of Model 2 uses a specific type of servlet, called on action servlet, and one or more actions and action mappings to implement the controller. It also uses a specific type of Java bean, called a form bean. As illustrated in Figure 3, the Web server at run time contains both the view and controller components of a Model 2 Web application, while a third tier (which is usually outside of the Web server) contains the model.

Figure 3. Struts framework: a Model 2 architecture



Struts contribution to MVC components as shown in Table 3.

Table 3. Struts's contributions to model, view, and controller

Component	**Contribution**
Model	None directly. However, the Struts actions and configuration file provide an elegant way to control the circumstances under which the model components are invoked.
View	• Java™ class org.apache.struts.action.ActionForm, which you subclass to create a form bean that is used in two ways at run time: o When a JSP page prepares the related HTML form for display, the JSP page accesses the bean, which holds values to be placed into the form. Those values are provided from business logic or from previous user input. o When user input is returned from a Web browser, the bean validates and holds that input either for use by business logic or (if validation failed) for subsequent redisplay. • Numerous custom JSP tags that are simple to use but are powerful in the sense that they hide information. Page Designer does not need to know much about form beans, for example, beyond the bean names and the names of each field in a given bean.
Controller	• The Struts action servlet handles run-time events in accordance with a set of rules that are provided at deployment time. Those rules are contained in a Struts configuration file and specify how the servlet responds to every outcome received from the business logic. Changes to the flow of control require changes only to the configuration file. • Struts also provides the Java class org.apache.struts.action.Action, which a Java developer subclasses to create an "action class". At run time, the

Table 3. Struts's contributions to model, view, and controller

Component	Contribution
	action servlet is said to "execute actions," which means that the servlet invokes the execute method of each of the instantiated action classes. The object returned from the execute method directs the action servlet as to what action or JSP file to access next. We recommend that you promote reuse by invoking business logic from the action class rather than including business logic in that class.

Benefits of the MVC design pattern

The application of the model-view-controller division to the development of dynamic Web applications has several benefits:

- You can distribute development effort to some extent, so that implementation changes in one part of the Web application do not require changes to another. The developers responsible for writing the business logic can work independently of the developers responsible for the flow of control, and Web-page designers can work independently of the developers.
- You can more easily prototype your work. You might do as follows, for example:
    1. Create a prototype Web application that accesses several workstation-based programs.
    2. Change the application in response to user feedback.
    3. Implement the production-level programs on the same or other platforms.

  Outside of the work you do on the programs themselves, your only adjustments are to configuration files or name-server content, not to other source code.

- You can more easily migrate legacy programs, because the view is separate from the model and the control and can be tailored to platform and user category.

- You can maintain an environment that comprises different technologies across different locations.
- The MVC design has an organizational structure that better supports scalability (building bigger applications) and ease of modification and maintenance (due to the cleaner separation of tasks).

BASIC COMPONENTS OF STRUTS

The Struts framework is a rich collection of Java libraries and can be broken down into the following major pieces:

- Base framework
- JSP tag libraries
- Tiles plugin
- Validator plugin

A brief description of each follows.

Base Framework

The base framework provides the core MVC functionality and is comprised of the building blocks for your application. At the foundation of the base framework is the Controller servlet: **ActionServlet**. The rest of the base framework is comprised of base classes that your application will extend and several utility classes. Most prominent among the base classes are the **Action** and **ActionForm** classes. These two classes are used extensively in all Struts applications. **Action** classes are used by **ActionServlet** to process specific requests. **ActionForm** classes are used to capture data from HTML forms and to be a conduit of data back to the View layer for page generation.

JSP Tag Libraries

Struts comes packaged with several JSP tag libraries for assisting with programming the View logic in JSPs. JSP tag libraries enable JSP authors to use HTML-like tags to represent functionality that is defined by a Java class.

Following is a listing of the libraries and their purpose:

- **HTML** Used to generate HTML forms that interact with the Struts APIs.
- **Bean** Used to work with Java bean objects in JSPs, such as accessing bean values.
- **Logic** Used to cleanly implement simple conditional logic in JSPs.
- **Nested** Used to allow arbitrary levels of nesting of the HTML, Bean, and Logic tags that otherwise do not work.

Tiles Plugin

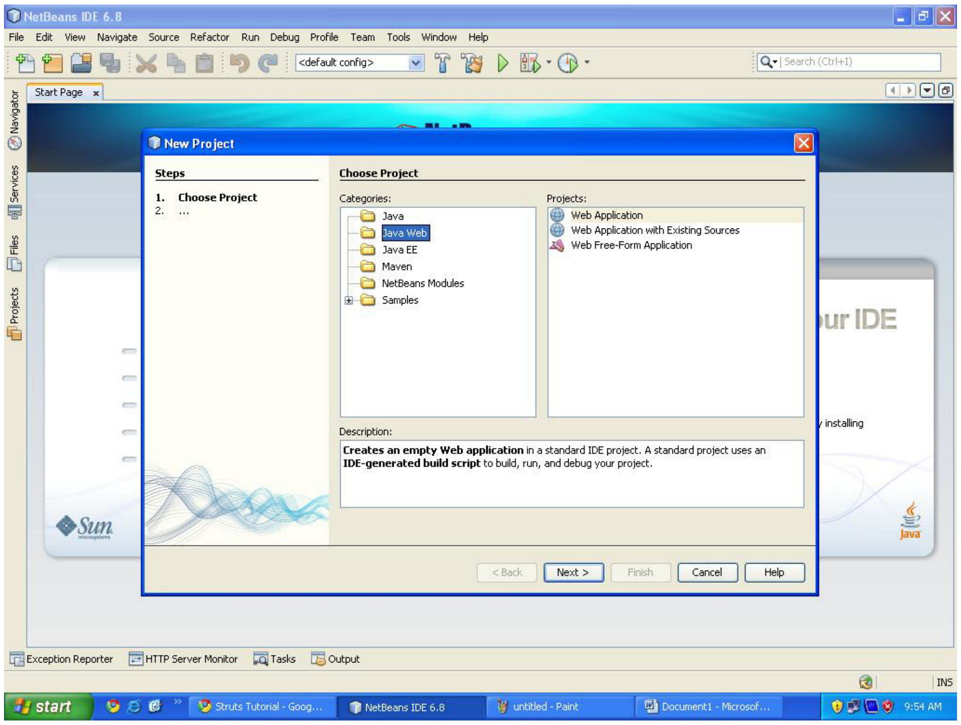
Struts comes packaged with the Tiles subframework. Tiles is a rich JSP templating framework that facilitates the reuse of presentation (HTML) code. With Tiles, JSP pages can be broken up into individual 'tiles' or pieces and then glued together to create one cohesive page. Similar to the design principles that the core Struts framework is built on, Tiles provides excellent reuse of View code. As of Struts 1.1, Tiles is part of and packaged with the core Struts download. Prior to Struts 1.1, Tiles was a third-party add-on, but has since been contributed to the project and is now more tightly integrated.

Validator Plugin

Struts comes packaged, as of version 1.1, with the Validator subframework for performing data validation. Validator provides a rich framework for performing data validation on both the server side and client side (browser). Each validation is configured in an outside XML file so that validations can easily be added to and removed from an application declaratively versus being hard-coded into the application. Similar to Tiles, prior to Struts 1.1, Validator was a third-party add-on, but has since been included in the project and is more tightly integrated.

BUILDING A SIMPLE STRUTS APPLICATION

1). Click on File Menu and choose New Project. A Dialog Box will be opened,choose Java Web from catagoeries and Web application from Projects list.



Steps

1. Choose Project
2. ...

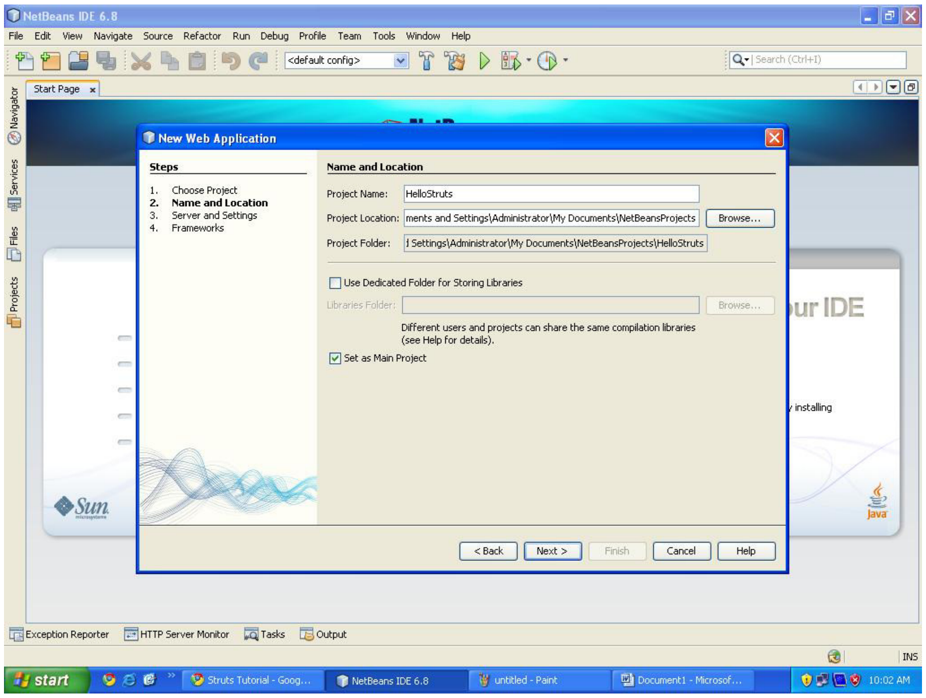
Choose Project

Categories: Java, Java Web, Java EE, Maven, NetBeans Modules, Samples

Projects: Web Application, Web Application with Existing Sources, Web Free-Form Application

Description:
Creates an empty Web application in a standard IDE project. A standard project uses an **IDE-generated build script** to build, run, and debug your project.

Click on Next button, type HelloStruts in the Project name and click on



Steps

1. Choose Project
2. Name and Location
3. Server and Settings
4. Frameworks

Name and Location

Project Name: HelloStruts

Project Location: ments and Settings\Administrator\My Documents\NetBeansProjects

Project Folder: | Settings\Administrator\My Documents\NetBeansProjects\HelloStruts

☐ Use Dedicated Folder for Storing Libraries

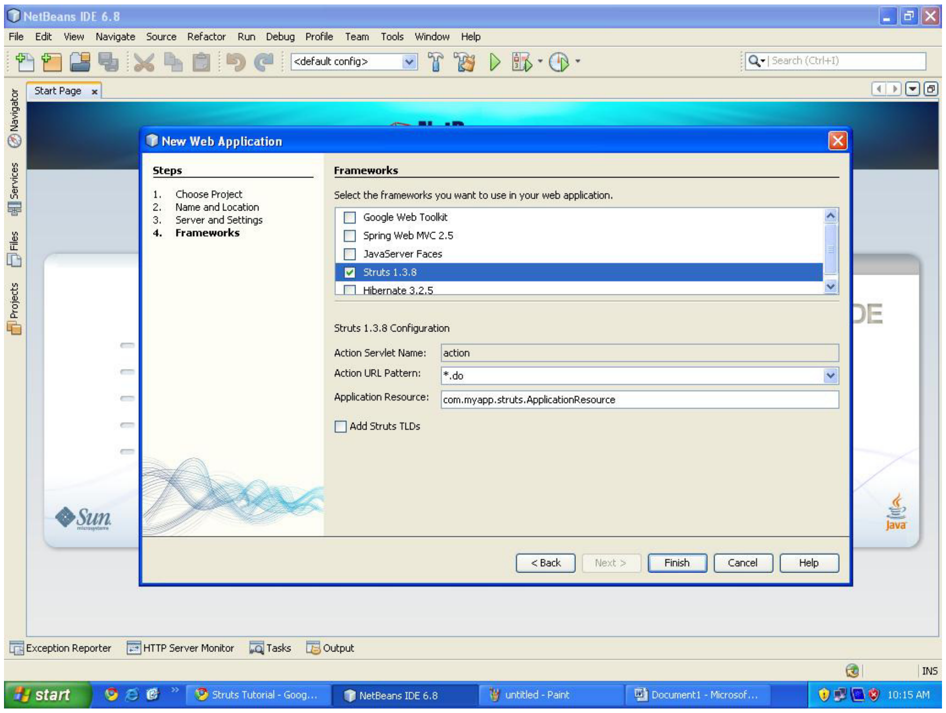
Libraries Folder:

Different users and projects can share the same compilation libraries (see Help for details).

☑ Set as Main Project

Choose Apache Tomcat 6.0 or GlassFish V3 Domain from Server and press Next.

From Frameworks click on struts 1.3.8 checkbox, and finally click on Finish button.


NetBeans IDE 6.8
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Start Page
New Web Application
< Back
Next >
Finish
Cancel
Help

Steps

1. Choose Project
2. Name and Location
3. Server and Settings
4. **Frameworks**

Frameworks

Select the frameworks you want to use in your web application.

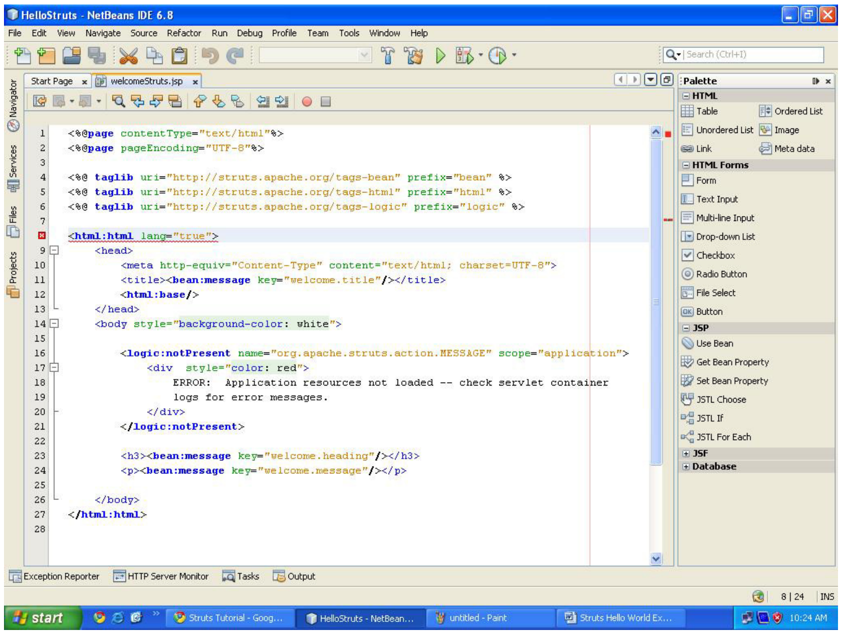
- [ ] Google Web Toolkit
- [ ] Spring Web MVC 2.5
- [ ] JavaServer Faces
- [✓] Struts 1.3.8
- [ ] Hibernate 3.2.5

Struts 1.3.8 Configuration

Action Servlet Name:	action
Action URL Pattern:	*.do
Application Resource:	com.myapp.struts.ApplicationResource

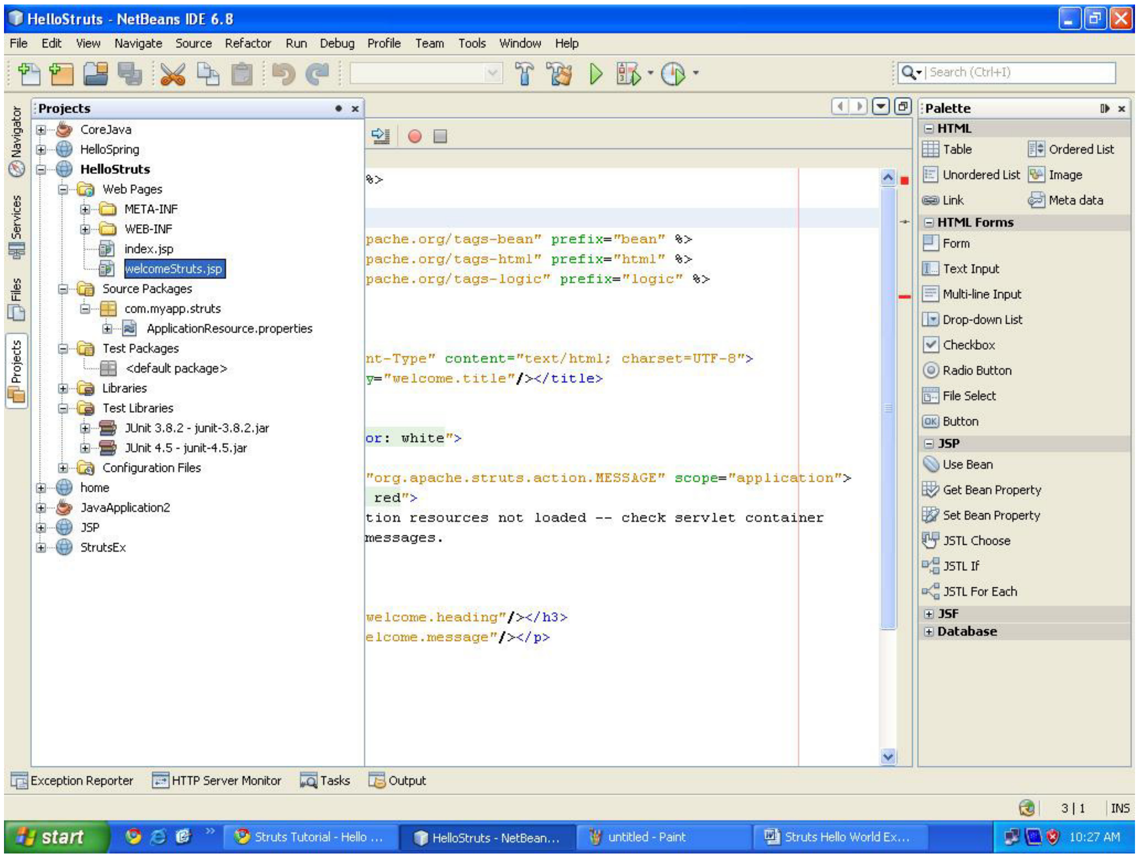
- [ ] Add Struts TLDs

It will open welcomeStruts.jsp page by default.



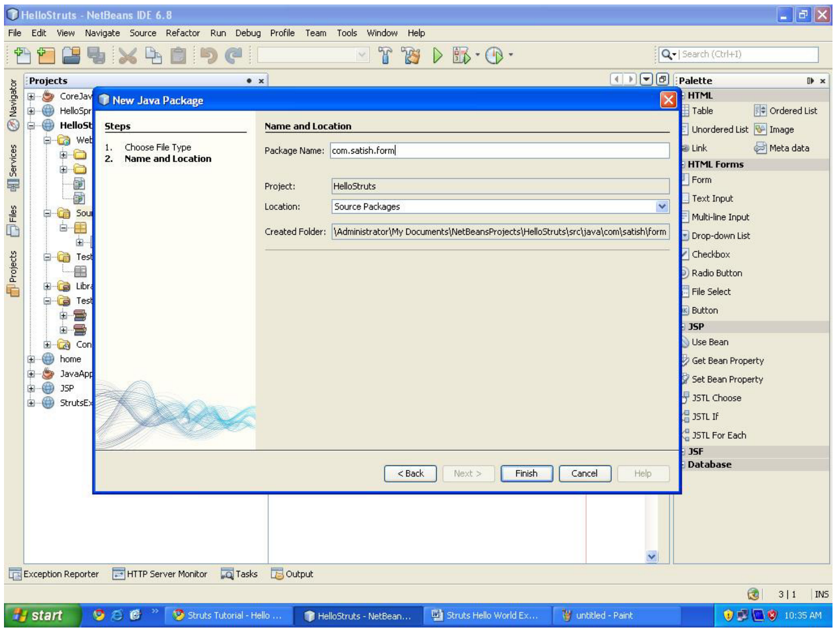
```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>

<html:html lang="true">
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title><bean:message key="welcome.title"/></title>
        <html:base/>
    </head>
    <body style="background-color: white">

        <logic:notPresent name="org.apache.struts.action.MESSAGE" scope="application">
            <div  style="color: red">
                ERROR:  Application resources not loaded -- check servlet container
                logs for error messages.
            </div>
        </logic:notPresent>

        <h3><bean:message key="welcome.heading"/></h3>
        <p><bean:message key="welcome.message"/></p>

    </body>
</html:html>
```

This is the Directory Structure of HelloStruts application



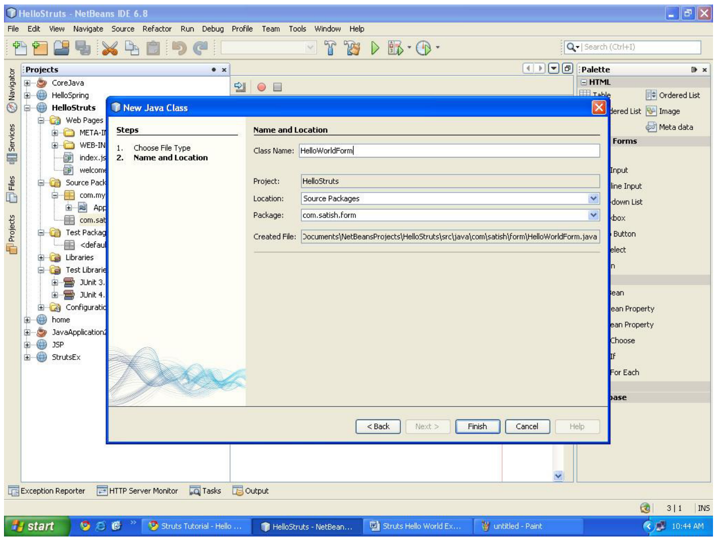
```
CoreJava
HelloSpring
HelloStruts
    Web Pages
        META-INF
        WEB-INF
        index.jsp
        welcomeStruts.jsp
    Source Packages
        com.myapp.struts
            ApplicationResource.properties
    Test Packages
        <default package>
    Libraries
    Test Libraries
        JUnit 3.8.2 - junit-3.8.2.jar
        JUnit 4.5 - junit-4.5.jar
    Configuration Files
home
JavaApplication2
JSP
StrutsEx
```

Right click on HelloStruts application and choose Java Package. Enter the package name as *com.satish.form* and click Finish.



Package Name: com.satish.form
Project: HelloStruts
Location: Source Packages
Created Folder: \Administrator\My Documents\NetBeansProjects\HelloStruts\src\java\com\satish\form

Right click on the newly created Package New->Java Class. Enter the class name as *HelloWorldForm and click on Finish.*

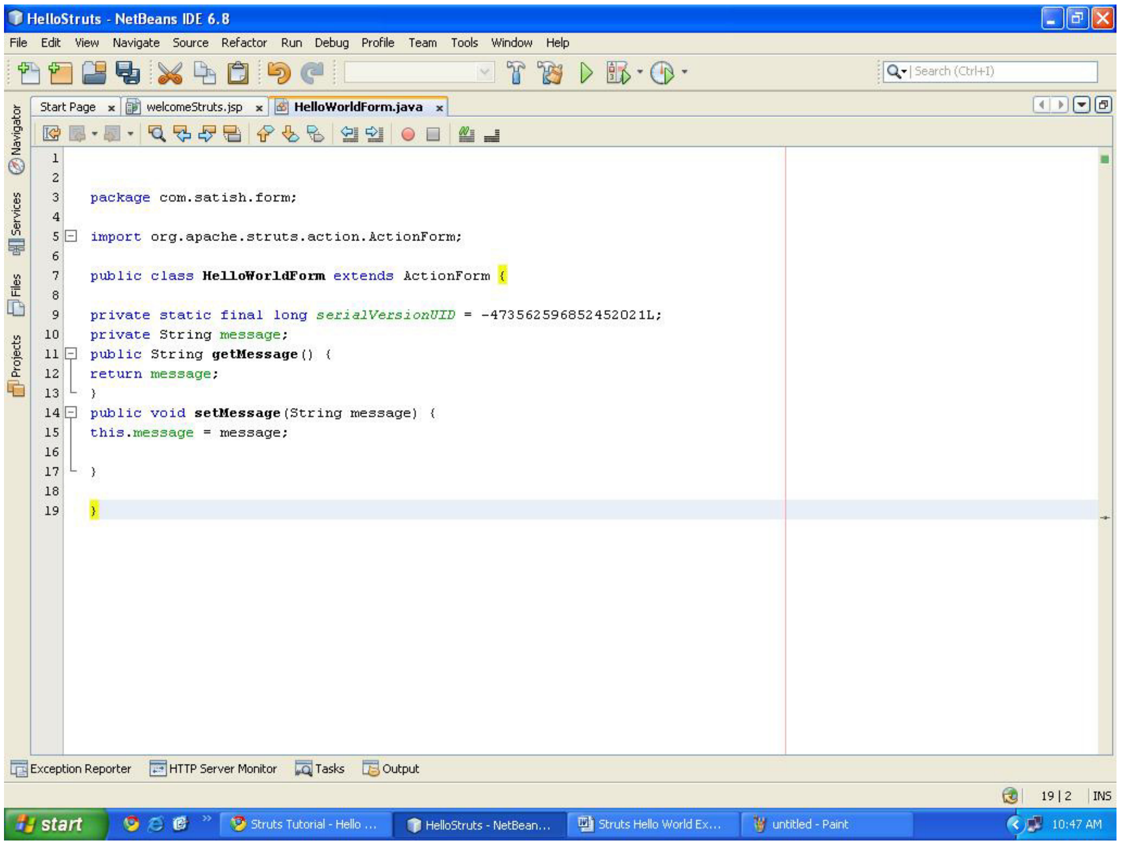


Class Name: HelloWorldForm
Project: HelloStruts
Location: Source Packages
Package: com.satish.form
Created File: Documents\NetBeansProjects\HelloStruts\src\java\com\satish\form\HelloWorldForm.java

In the HelloWorldForm class add the following code.

```
package com.satish.form;
```

```
import org.apache.struts.action.ActionForm;

public class HelloWorldForm extends ActionForm {

private static final long serialVersionUID = -473562596852452021L;

private String message;

public String getMessage() {

return message;

}

public void setMessage(String message) {

this.message = message;

}

}
```



```
package com.satish.form;

import org.apache.struts.action.ActionForm;

public class HelloWorldForm extends ActionForm {

private static final long serialVersionUID = -473562596852452021L;
private String message;
public String getMessage() {
return message;
}
public void setMessage(String message) {
this.message = message;

}

}
```

In the same way create a new package *com.arpit.action* and create a *HelloWorldAction* class, extending *org.apache.struts.action.Action*. And add following code in action class.

package com.arpit.action;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.Action;

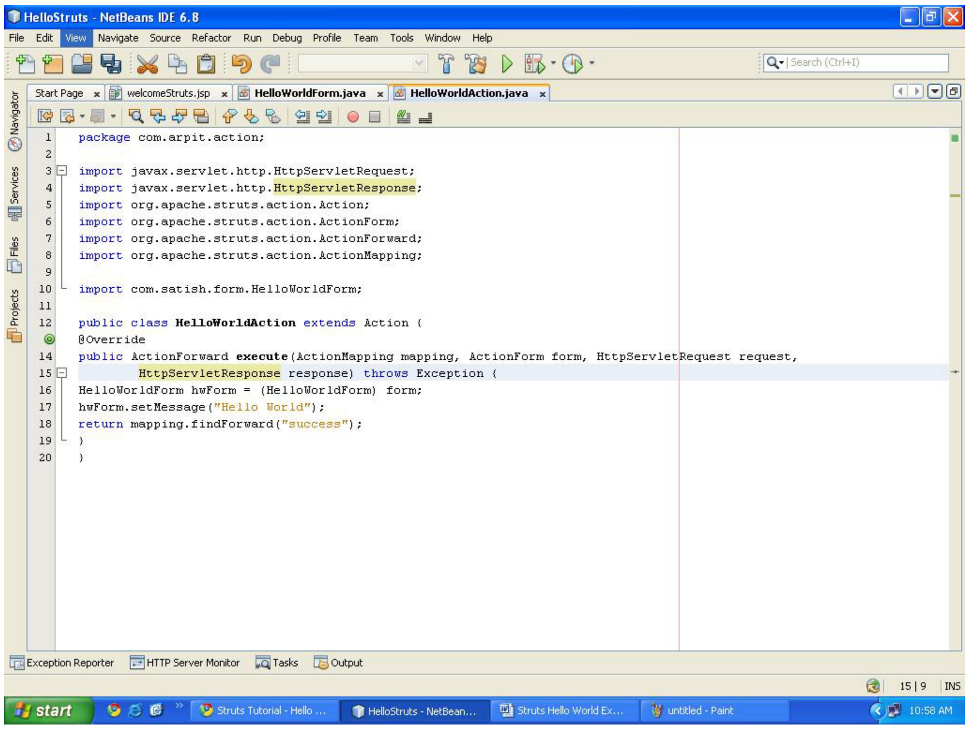
import org.apache.struts.action.ActionForm;

import org.apache.struts.action.ActionForward;

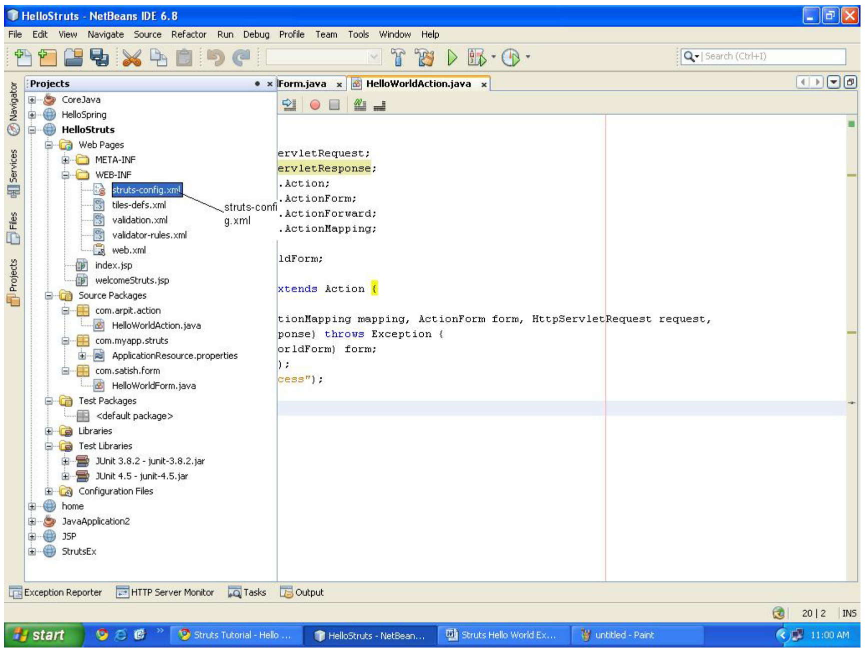
import org.apache.struts.action.ActionMapping;

import com.satish.form.HelloWorldForm;

```
public class HelloWorldAction extends Action {

@Override

public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest
request,

HttpServletResponse response) throws Exception {

HelloWorldForm hwForm = (HelloWorldForm) form;

hwForm.setMessage(“Hello World”);

return mapping.findForward(“success”);

}

}
```



```
package com.arpit.action;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

import com.satish.form.HelloWorldForm;

public class HelloWorldAction extends Action {
@Override
public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest request,
        HttpServletResponse response) throws Exception {
HelloWorldForm hwForm = (HelloWorldForm) form;
hwForm.setMessage("Hello World");
return mapping.findForward("success");
}
}
```



```
ervletRequest;
ervletResponse;
.Action;
.ActionForm;
.ActionForward;
.ActionMapping;

ldForm;

xtends Action {

tionMapping mapping, ActionForm form, HttpServletRequest request,
ponse) throws Exception {
orldForm) form;
);
cess");
```

Add the following entries in the *struts-config.xml* file.

<?xml version=”1.0″ encoding=”UTF-8″ ?>

<!DOCTYPE struts-config PUBLIC

“-//Apache Software Foundation//DTD Struts Configuration 1.3//EN”

“http://jakarta.apache.org/struts/dtds/struts-config_1_3.dtd”>

<struts-config>

<form-beans>

<form-bean/>

</form-beans>

<global-exceptions>

</global-exceptions>

<global-forwards>

<forward path="/helloWorld.do"/>

</global-forwards>

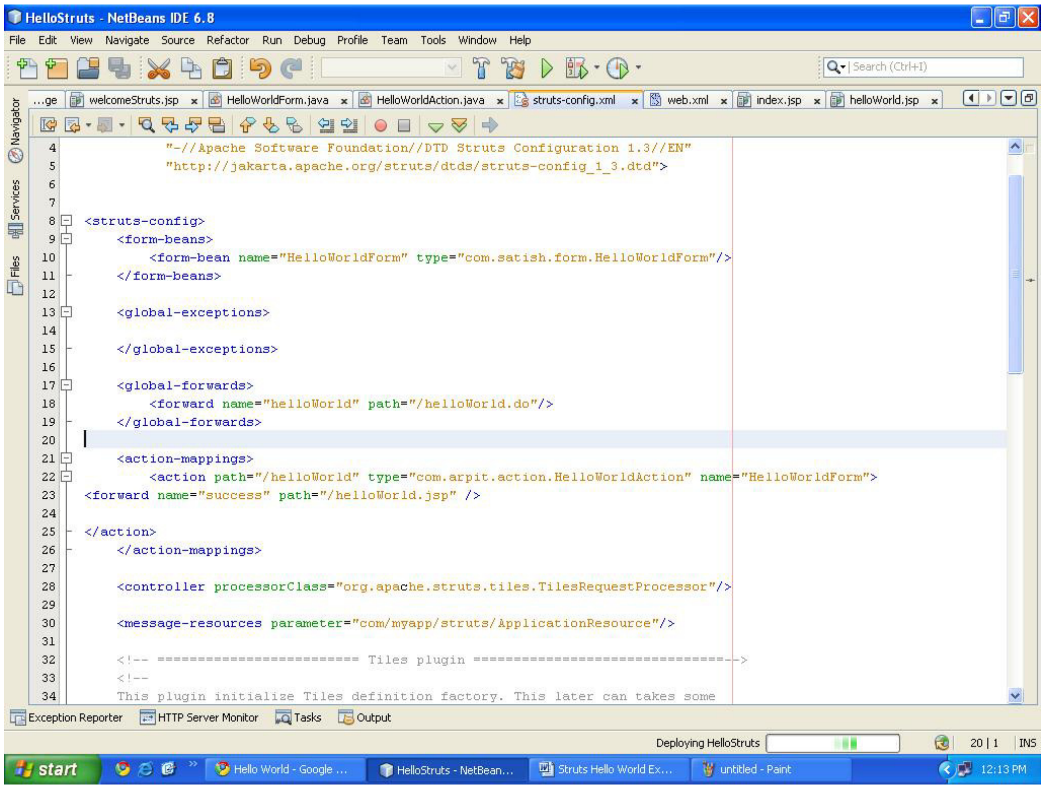
<action-mappings>

<action path="/helloWorld">

<forward path="/helloWorld.jsp" />

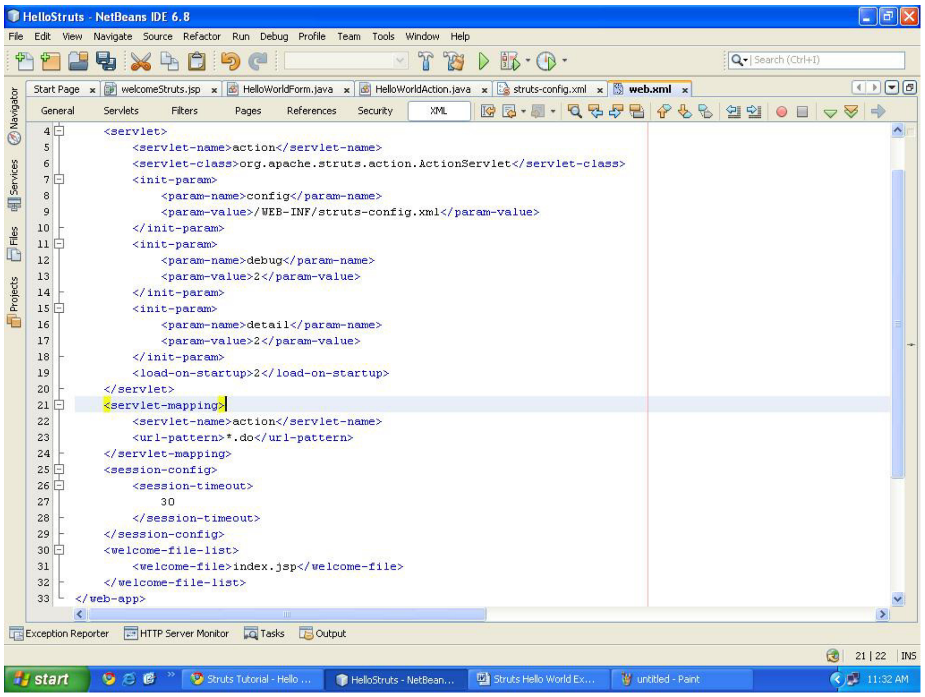
</action>

</action-mappings>


HelloStruts - NetBeans IDE 6.8
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
...ge welcomeStruts.jsp HelloWorldForm.java HelloWorldAction.java struts-config.xml web.xml index.jsp helloWorld.jsp
Navigator Services Files
Exception Reporter HTTP Server Monitor Tasks Output
Deploying HelloStruts 20 | 1 INS
start Hello World - Google ... HelloStruts - NetBean... Struts Hello World Ex... untitled - Paint 12:13 PM

```
            "-//Apache Software Foundation//DTD Struts Configuration 1.3//EN"
            "http://jakarta.apache.org/struts/dtds/struts-config_1_3.dtd">


<struts-config>
    <form-beans>
        <form-bean name="HelloWorldForm" type="com.satish.form.HelloWorldForm"/>
    </form-beans>

    <global-exceptions>

    </global-exceptions>

    <global-forwards>
        <forward name="helloWorld" path="/helloWorld.do"/>
    </global-forwards>

    <action-mappings>
        <action path="/helloWorld" type="com.arpit.action.HelloWorldAction" name="HelloWorldForm">
<forward name="success" path="/helloWorld.jsp" />

</action>
    </action-mappings>

    <controller processorClass="org.apache.struts.tiles.TilesRequestProcessor"/>

    <message-resources parameter="com/myapp/struts/ApplicationResource"/>

    <!-- ======================== Tiles plugin ===============================-->
    <!--
    This plugin initialize Tiles definition factory. This later can takes some
```

Now configure the deployment descriptor. Add the following configuration information in the *web.xml* file.


HelloStruts - NetBeans IDE 6.8
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Start Page
welcomeStruts.jsp
HelloWorldForm.java
HelloWorldAction.java
struts-config.xml
web.xml
General Servlets Filters Pages References Security XML
Exception Reporter
HTTP Server Monitor
Tasks
Output

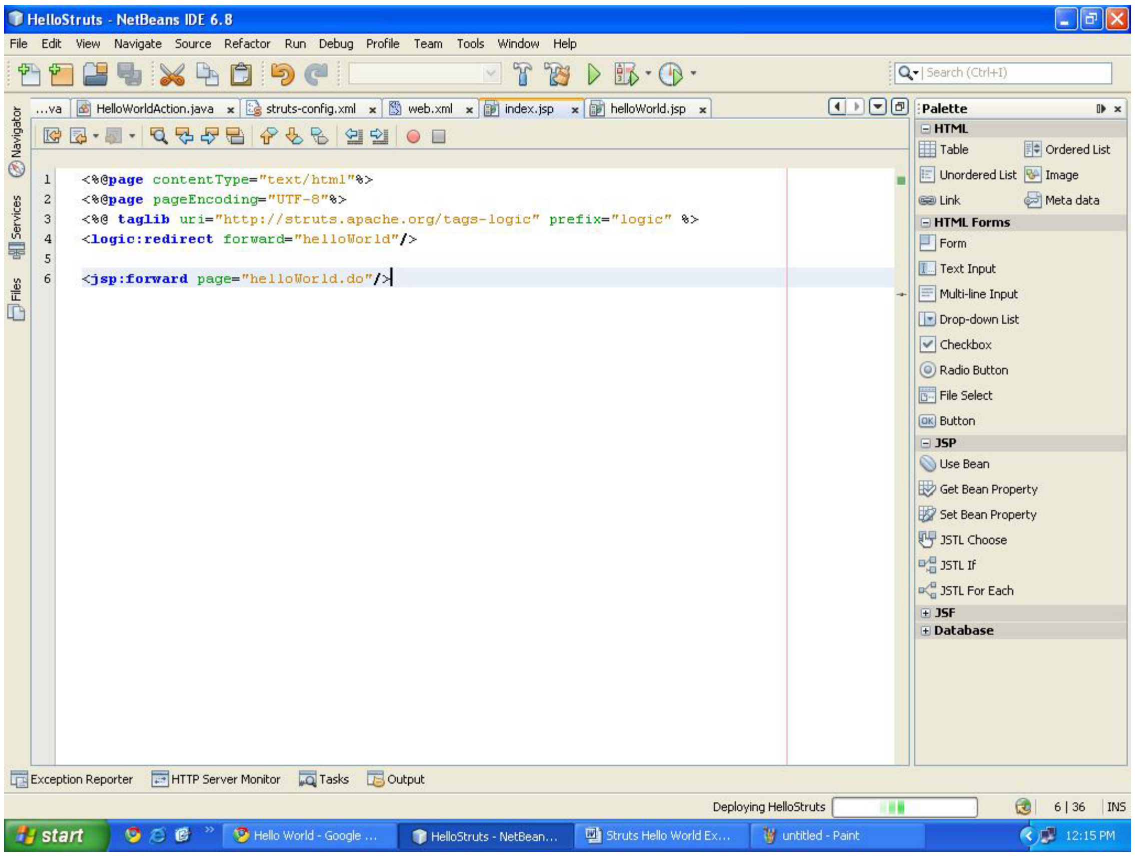
```
    <servlet>
        <servlet-name>action</servlet-name>
        <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
        <init-param>
            <param-name>config</param-name>
            <param-value>/WEB-INF/struts-config.xml</param-value>
        </init-param>
        <init-param>
            <param-name>debug</param-name>
            <param-value>2</param-value>
        </init-param>
        <init-param>
            <param-name>detail</param-name>
            <param-value>2</param-value>
        </init-param>
        <load-on-startup>2</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>action</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

When we run the application the index.jsp page will be executed first. In the *index.jsp* page we redirect the request to the *helloWorld.do* URI, which inturn invokes the *HelloWorldAction*.

<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>

<logic:redirect forward="helloWorld"/>

Add these lines of code in index.jsp page.



```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>
<logic:redirect forward="helloWorld"/>

<jsp:forward page="helloWorld.do"/>
```

In the action class we return the ActionForward "*success*" which is mapped to the*helloWorld.jsp* page. In the *helloWorld.jsp* page we display the "Hello World" message.

<%@taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-">

<title>Hello World</title>

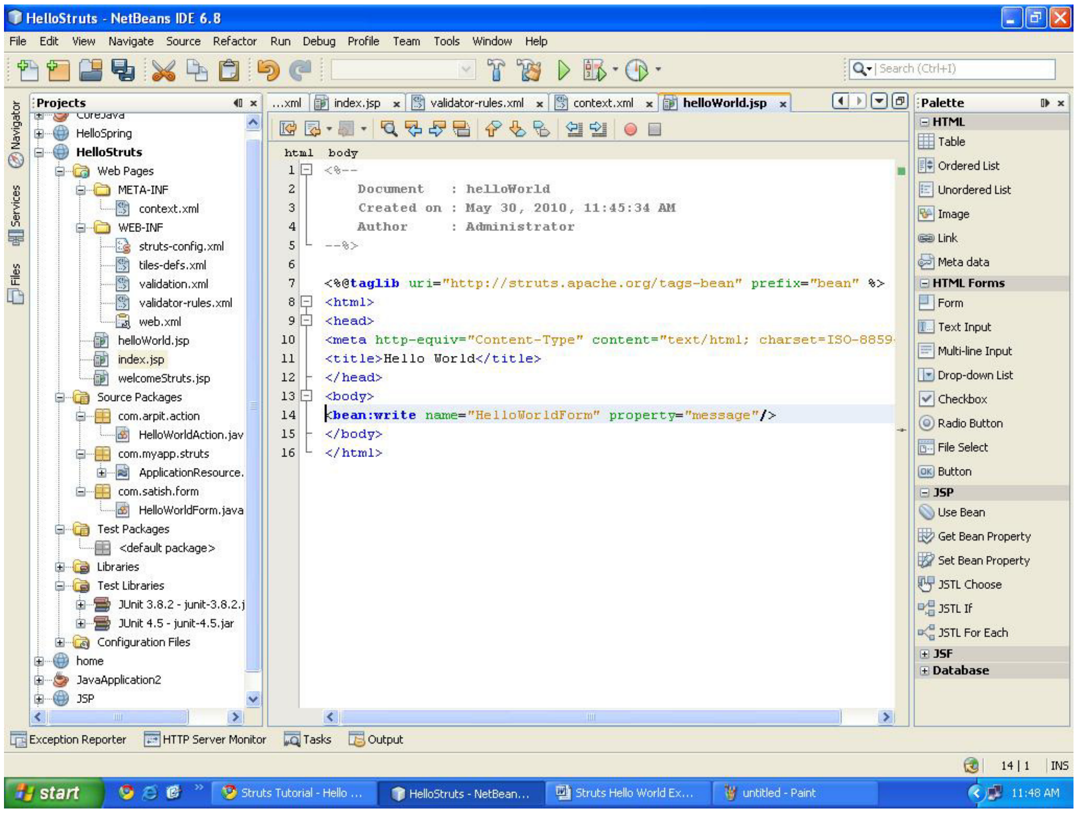
</head>

<body>

<bean:write name="HelloWorldForm" property="message"/>

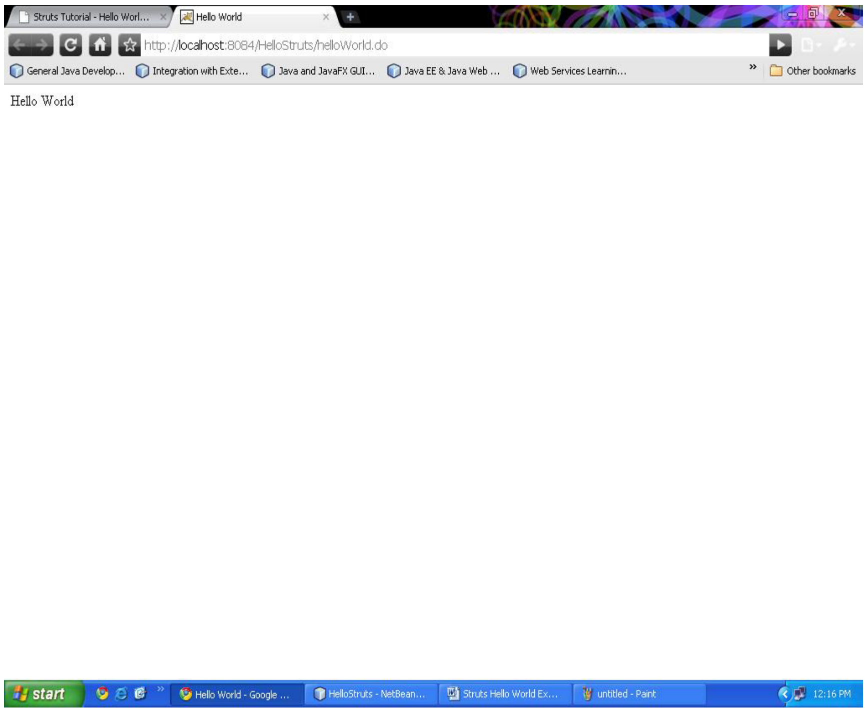
</body>

</html>



```
<%--
    Document   : helloWorld
    Created on : May 30, 2010, 11:45:34 AM
    Author     : Administrator
--%>

<%@taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859
<title>Hello World</title>
</head>
<body>
<bean:write name="HelloWorldForm" property="message"/>
</body>
</html>
```

Output:



Hello World

INTRODUCTION TO WEB SERVICE TECHNOLOGIES

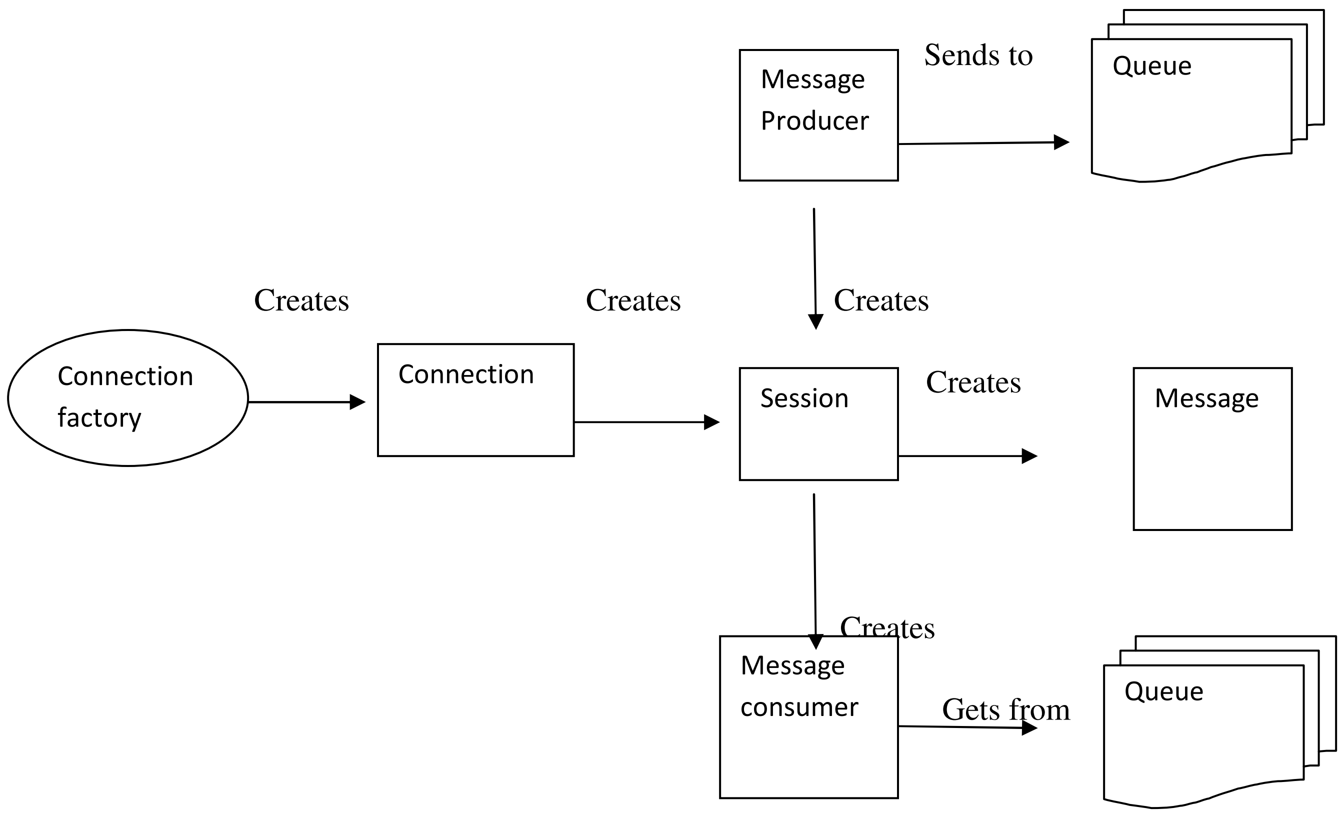
Java Message Service (JMS)

The **Java Message Service** API is a Java Message Oriented Middleware (MOM) API for sending messages between two or more clients.

It is a messaging standard that allows application components based on the Java 2 Platform, Enterprise Edition (J2EE) to create, send, receive, and read messages.

It allows the communication between different components of a distributed application to be loosely coupled, reliable, and asynchronous

JMS Architecture



There are six principal building blocks in the JMS architecture. They are used one by one to build the JMS application. The JMS API is provided in the Java package javax.jms.

1. **Administered objects**

- An administered object is that a client uses to create a connection to the JMS provider.
- JMS clients access the connectionfactory through portable interfaces so the code does not need to be changed if the underlying implementation changes.

- Administrators configure the connection factory in the Java Naming and Directory Interface (JNDI) namespace so that JMS clients can look them up.

2. Connection

- A Connection represents an open communication channel between an application and the messaging system, and is used to create a Session for producing and consuming messages.
- A connection creates server-side and client-side objects that manage the messaging activity between an application and JMS.
- A Connection is created by a ConnectionFactory, obtained through a JNDI lookup.

3. Session

- It represents a single-threaded context used for sending and receiving messages.
- The benefit of a session is that it supports transactions
- A session allows users to create message producers to send messages, and message consumers to receive messages.
- A Session is created by a Connection

4. Message Producers

- A Message Producers sends messages to a queue or topic.
- A Session creates the Message Producers that are attached to queues and topics.

.

5. **Message Consumers**

- A Message Consumers receives messages from a queue or topic.
- A Session creates the Message Consumers that are attached to queues and topics.

6. **Messages**

An object that is sent between consumers and producers; that is, from one application to another. A message has three main parts:

1. A message header (required): Contains operational settings to identify and route messages.
2. A set of message properties (optional): Contains additional properties to support compatibility with other providers or users
3. A message body (optional): Allows users to create five types of messages (text message, map message, bytes message, stream message, and object message).

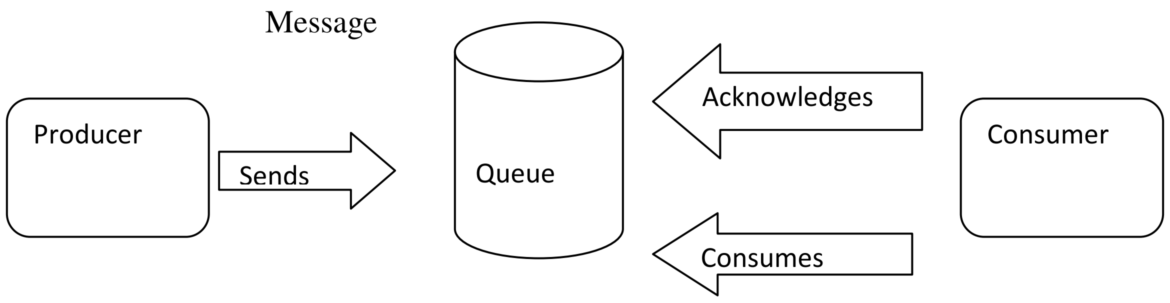
The JMS API Supports two models:

- point-to-point or queuing model
- publish and subscribe model

1. **Point-to-point or queuing model:**

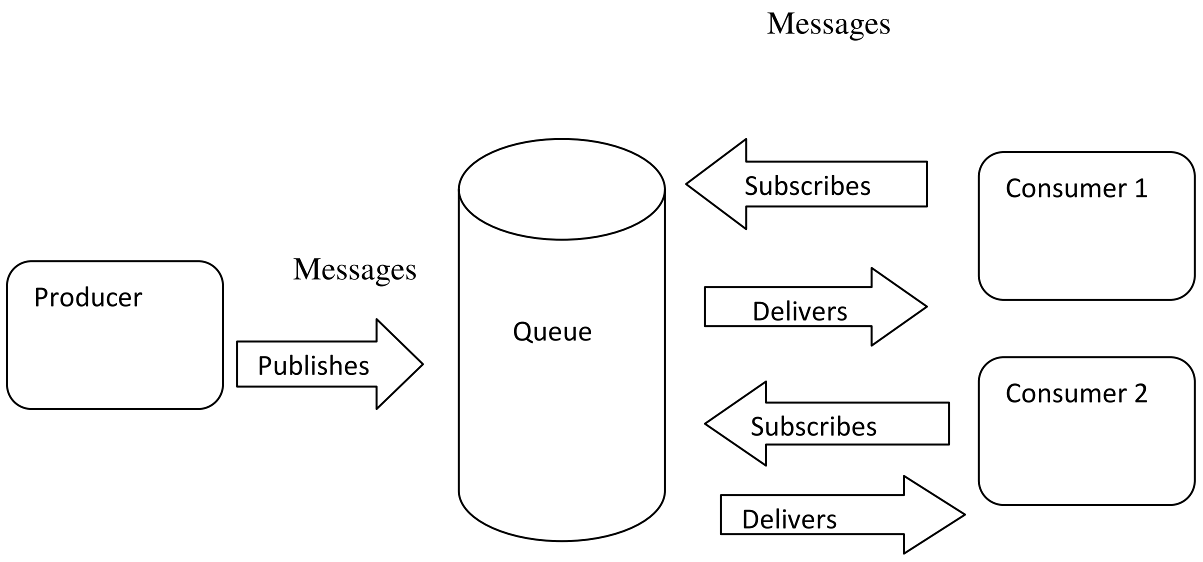
    - In this model, a producer sends messages to a particular queue and a consumer gets messages from the queue.

- Only one consumer will get the message and producer knows the consumer of the message.



2. **Publish and subscribe model:**

- In this model, many subscribers can receive messages on a particular message topic.
- Publisher and subscriber both are unaware of each other.



WEB SERVICES

* **Web Services** is a technology that allows applications to communicate with each other in a platform- and programming language-independent manner.

* It is a software interface that describes a collection of operations that can be accessed over the network through standardized XML messaging.

* It uses protocols based on the XML language to describe an operation to execute or data to exchange with another Web service.

The following are the technologies available for the creation of web services

- SOAP (Simple Object Access Protocol)
- UDDI (Universal Description, Discovery and Integration)
- WSDL (Web Services Description Language)

SOAP Simple Object Access Protocol

The SOAP is a lightweight, XML-based protocol for exchanging information in a decentralized, distributed environment. SOAP supports different styles of information exchange, including:

- Remote Procedure Call style (RPC), which allows for request-response processing, where an endpoint receives a procedure oriented message and replies with a correlated response message.

- Message-oriented information exchange, which supports organizations and applications that need to exchange business or other types of documents where a message is sent but the sender may not expect or wait for an immediate response.

A SOAP message is an ordinary XML document containing the following elements:

- An Envelope element is the root element of a SOAP message that identifies the XML document as a SOAP message
- A Header element that contains header information
- A Body element that contains call and response information
- A Fault element containing errors and status information

Here are some important syntax rules:

- A SOAP message MUST be encoded using XML
- A SOAP message MUST use the SOAP Envelope namespace
- A SOAP message MUST use the SOAP Encoding namespace

Syntax:

```
<?xml                                                                    version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Header>
...
</soap:Header>

<soap:Body>
...
                                                                           <soap:Fault>
                                                                                   ...
                                                                          </soap:Fault>
```

```
</soap:Body>

</soap:Envelope>
```

WSDL Web Services Description Language

WSDL describes network services by using an XML grammar. It provides documentation for distributed systems and has the goal to enable applications to communicate with each other in an automated way.

- WSDL v1.0, 9/2000
- WSDL v1.1 submitted to W3C 3/2001.

WSDL is an XML grammar for describing web services. The specification itself is divided into six major elements:

definitions

The definitions element must be the root element of all WSDL documents. It defines the name of the web service, declares multiple namespaces used throughout the remainder of the document, and contains all the service elements described here.

types

The types element describes all the data types used between the client and server.

message

The message element describes a one-way message, whether it is a single message request or a single message response.

port

The port element combines multiple message elements to form a complete one-way or round-trip operation

binding

The binding element describes the concrete specifics of how the service will be implemented on the wire. WSDL includes built-in extensions for defining SOAP services, and SOAP-specific information therefore goes here.

service

The service element defines the address for invoking the specified service. Most commonly, this includes a URL for invoking the SOAP service.

The structure of a WSDL document is:

```
<definitions>

<types>
  definition of types........
</types>

<message>
  definition of a message....
</message>
```

```
<portType>
  definition of a port.......
</portType>


<binding>
  definition of a binding....
</binding>

</definitions>
```

UDDI Universal Description, Discovery and Integration

UDDI is a platform-independent framework for describing services, discovering businesses, and integrating business services by using the Internet.

- UDDI is a directory for storing information about web services
- UDDI is a directory of web service interfaces described by WSDL
- UDDI communicates via SOAP
- UDDI is built into the Microsoft .NET platform

A Web-based distributed directory that enables businesses to list themselves on the Internet and discover each other, similar to a traditional phone book's yellow and white pages.

JAXP

The Java API for XML Processing (JAXP) is for processing XML data using applications written in the Java programming language. JAXP leverages the parser standards Simple API for XML Parsing (SAX) and Document Object Model (DOM) so that you can choose to parse your data as a stream of events or to build an object representation of it. JAXP also supports the Extensible Stylesheet Language Transformations (XSLT) standard, giving you control over the presentation of the data and enabling you to convert the data to other XML documents or to other formats, such as HTML. JAXP also provides namespace support, allowing you to work with DTDs that might otherwise have naming conflicts. Finally, as of version 1.4, JAXP implements the Streaming API for XML (StAX) standard.

Designed to be flexible, JAXP allows you to use any XML-compliant parser from within your application. It does this with what is called a pluggability layer, which lets you plug in an implementation of the SAX or DOM API. The pluggability layer also allows you to plug in an XSL processor, letting you control how your XML data is displayed.

The JAXP APIs

The main JAXP APIs are defined in the javax.xml.parsers package. That package contains vendor-neutral factory classes, SAXParserFactory, DocumentBuilderFactory, and TransformerFactory, which give you a SAXParser, a DocumentBuilder, and an XSLT transformer, respectively. DocumentBuilder, in turn, creates a DOM-compliant Document object.

The factory APIs let you plug in an XML implementation offered by another vendor without changing your source code. The implementation you get depends on the setting of the javax.xml.parsers.SAXParserFactory, javax.xml.parsers.DocumentBuilderFactory, and javax.xml.transform.TransformerFactory system properties, using System.setProperties() in the code, <sysproperty key="..." value="..."/> in an Ant build script, or -DpropertyName="..." on the command line. The default values (unless overridden at runtime on the command line or in the code) point to Sun's implementation.

Overview of the Packages

The SAX and DOM APIs are defined by the XML-DEV group and by the W3C, respectively. The libraries that define those APIs are as follows:

- javax.xml.parsers: The JAXP APIs, which provide a common interface for different vendors' SAX and DOM parsers.
- org.w3c.dom: Defines the Document class (a DOM) as well as classes for all the components of a DOM.
- org.xml.sax: Defines the basic SAX APIs.
- javax.xml.transform: Defines the XSLT APIs that let you transform XML into other forms.
- javax.xml.stream: Provides StAX-specific transformation APIs.

The Simple API for XML (SAX) is the event-driven, serial-access mechanism that does element-by-element processing. The API for this level reads and writes XML to a data repository or the web. For server-side and high-performance applications, you will want to fully understand this level. But for many applications, a minimal understanding will suffice.

The DOM API is generally an easier API to use. It provides a familiar tree structure of objects. You can use the DOM API to manipulate the hierarchy of application objects it encapsulates. The DOM API is ideal for interactive applications because the entire object model is present in memory, where it can be accessed and manipulated by the user.

On the other hand, constructing the DOM requires reading the entire XML structure and holding the object tree in memory, so it is much more CPU- and memory-intensive. For that reason, the SAX API tends to be preferred for server-side applications and data filters that do not require an in-memory representation of the data.

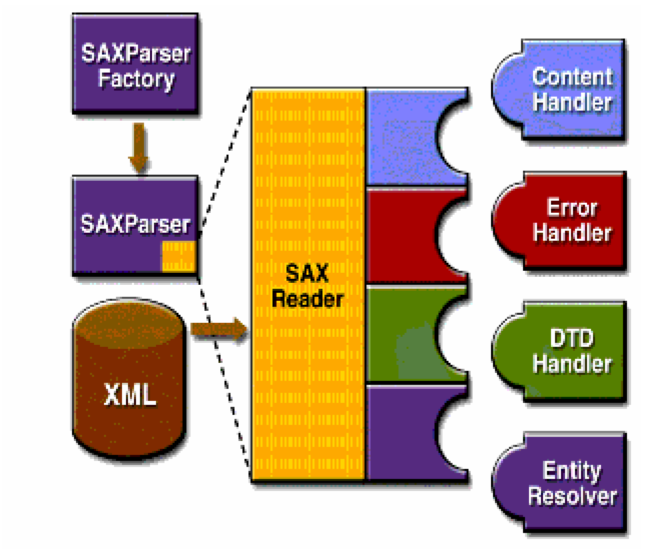
The XSLT APIs defined in javax.xml.transform let you write XML data to a file or convert it into other forms. As shown in the XSLT section of this tutorial, you can even use it in conjunction with the SAX APIs to convert legacy data to XML.

Finally, the StAX APIs defined in javax.xml.stream provide a streaming Java technology-based, event-driven, pull-parsing API for reading and writing XML documents. StAX offers a simpler programming model than SAX and more efficient memory management than DOM.

Simple API for XML APIs

The basic outline of the SAX parsing APIs is shown in Figure 1-1. To start the process, an instance of the SAXParserFactory class is used to generate an instance of the parser.

Figure 1-1 SAX APIs



The parser wraps a SAXReader object. When the parser's parse() method is invoked, the reader invokes one of several callback methods implemented in the application. Those methods are defined by the interfaces ContentHandler, ErrorHandler, DTDHandler, and EntityResolver.

Here is a summary of the key SAX APIs:

SAXParserFactory

A SAXParserFactory object creates an instance of the parser determined by the system property, javax.xml.parsers.SAXParserFactory.

SAXParser

The SAXParser interface defines several kinds of parse() methods. In general, you pass an XML data source and a DefaultHandler object to the parser, which processes the XML and invokes the appropriate methods in the handler object.

SAXReader

The SAXParser wraps a SAXReader. Typically, you do not care about that, but every once in a while you need to get hold of it using SAXParser's getXMLReader() so that you can configure it. It is the SAXReader that carries on the conversation with the SAX event handlers you define.

DefaultHandler

Not shown in the diagram, a DefaultHandler implements the ContentHandler, ErrorHandler, DTDHandler, and EntityResolver interfaces (with null methods), so you can override only the ones you are interested in.

ContentHandler

Methods such as startDocument, endDocument, startElement, and endElement are invoked when an XML tag is recognized. This interface also defines the methods characters() and processingInstruction(), which are invoked when the parser encounters the text in an XML element or an inline processing instruction, respectively.

ErrorHandler

Methods error(), fatalError(), and warning() are invoked in response to various parsing errors. The default error handler throws an exception for fatal errors and ignores other errors (including validation errors). This is one reason you need to know something about the SAX parser, even if you are using the DOM. Sometimes, the application may be able to recover from a validation error. Other times, it may need to generate an exception. To ensure the correct handling, you will need to supply your own error handler to the parser.

DTDHandler

Defines methods you will generally never be called upon to use. Used when processing a DTD to recognize and act on declarations for an unparsed entity.

EntityResolver

The resolveEntity method is invoked when the parser must identify data identified by a URI. In most cases, a URI is simply a URL, which specifies the location of a document, but in some cases the document may be identified by a URN - a public identifier, or name, that is unique in the web space. The public identifier may be specified in addition to the URL. The EntityResolver can then use the public identifier instead of the URL to find the document-for example, to access a local copy of the document if one exists.

A typical application implements most of the ContentHandler methods, at a minimum. Because the default implementations of the interfaces ignore all inputs except for fatal errors, a robust implementation may also want to implement the ErrorHandler methods.

SAX Packages

The SAX parser is defined in the packages listed in Table 1-1.

Table 1-1 SAX Packages

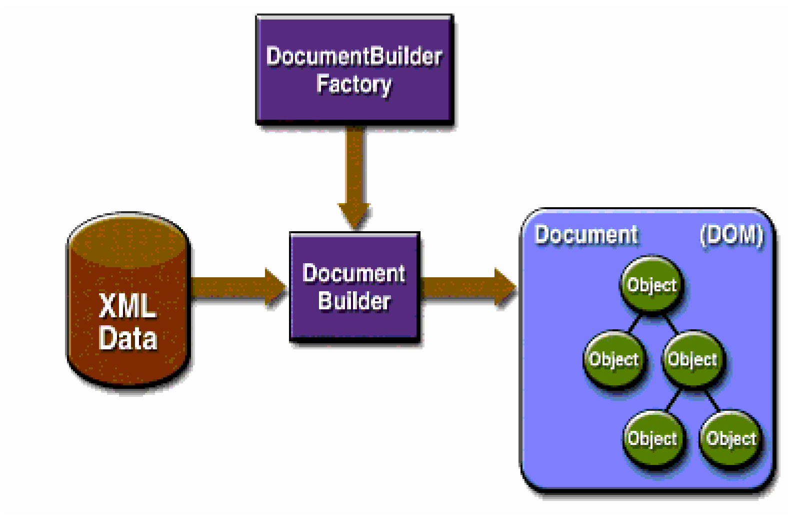
Packages	**Description**
org.xml.sax	Defines the SAX interfaces. The name org.xml is the package prefix that was settled on by the group that defined the SAX API.
org.xml.sax.ext	Defines SAX extensions that are used for doing more sophisticated SAX processing-for example, to process a document type definition (DTD) or to see the detailed syntax for a file.
org.xml.sax.helpers	Contains helper classes that make it easier to use SAX-for example, by defining a default handler that has null methods for all the interfaces, so that you only need to override the ones you actually want to implement.
javax.xml.parsers	Defines the SAXParserFactory class, which returns the SAXParser. Also

defines exception classes for reporting errors.

Document Object Model APIs

Figure 1-2 shows the DOM APIs in action.

Figure 1-2 DOM APIs



You use the javax.xml.parsers.DocumentBuilderFactory class to get a DocumentBuilder instance, and you use that instance to produce a Document object that conforms to the DOM specification. The builder you get, in fact, is determined by the system property javax.xml.parsers.DocumentBuilderFactory, which selects the factory implementation that is used to produce the builder. (The platform's default value can be overridden from the command line.)

You can also use the DocumentBuilder newDocument() method to create an empty Document that implements the org.w3c.dom.Document interface. Alternatively, you can use one of the builder's parse methods to create a Document from existing XML data. The result is a DOM tree like that shown in Figure 1-2.

Note - Although they are called objects, the entries in the DOM tree are actually fairly low-level data structures. For example, consider this structure: <color>blue</color>. There is an element node for the color tag, and under that there is a text node that contains the data, blue! This issue will be explored at length in the DOM chapter of this tutorial, but developers who are expecting objects are usually surprised to find that invoking getNodeValue() on the element node returns nothing. For a truly object-oriented tree, see the JDOM API at http://www.jdom.org.

DOM Packages

The Document Object Model implementation is defined in the packages listed in Table 1-2.

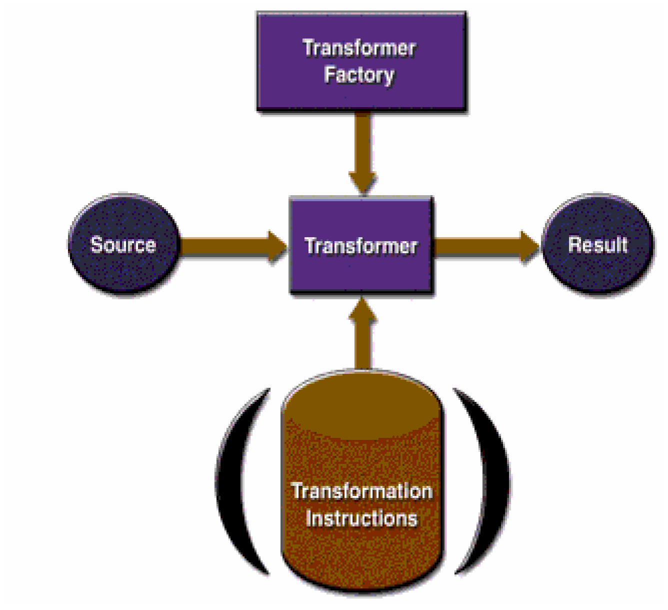
Table 1-2 DOM Packages

Package	**Description**
org.w3c.dom	Defines the DOM programming interfaces for XML (and, optionally, HTML) documents, as specified by the W3C.
javax.xml.parsers	Defines the DocumentBuilderFactory class and the DocumentBuilder class, which returns an object that implements the W3C Document interface. The factory that is used to create the builder is determined by the javax.xml.parsers system property, which can be set from the command line or overridden when invoking the new Instance method. This package also defines the ParserConfigurationException class for reporting errors.

Extensible Stylesheet Language Transformations APIs

Figure 1-3 shows the XSLT APIs in action.

Figure 1-3 XSLT APIs



A TransformerFactory object is instantiated and used to create a Transformer. The source object is the input to the transformation process. A source object can be created from a SAX reader, from a DOM, or from an input stream.

Similarly, the result object is the result of the transformation process. That object can be a SAX event handler, a DOM, or an output stream.

When the transformer is created, it can be created from a set of transformation instructions, in which case the specified transformations are carried out. If it is created without any specific instructions, then the transformer object simply copies the source to the result.

XSLT Packages

The XSLT APIs are defined in the packages shown in Table 1-3.

Table 1-3 XSLT Packages

Package	**Description**
javax.xml.transform	Defines the TransformerFactory and Transformer classes, which you use to get an object capable of doing transformations. After creating a transformer object, you invoke its transform() method, providing it

	with an input (source) and output (result).
javax.xml.transform.dom	Classes to create input (source) and output (result) objects from a DOM.
javax.xml.transform.sax	Classes to create input (source) objects from a SAX parser and output (result) objects from a SAX event handler.
javax.xml.transform.stream	Classes to create input (source) objects and output (result) objects from an I/O stream.

Streaming API for XML APIs

StAX is the latest API in the JAXP family, and provides an alternative to SAX, DOM, TrAX, and DOM for developers looking to do high-performance stream filtering, processing, and modification, particularly with low memory and limited extensibility requirements.

To summarize, StAX provides a standard, bidirectional **pull parser** interface for streaming XML processing, offering a simpler programming model than SAX and more efficient memory management than DOM. StAX enables developers to parse and modify XML streams as events, and to extend XML information models to allow application-specific additions. More detailed comparisons of StAX with several alternative APIs are provided in Chapter 5, Streaming API for XML, in Comparing StAX to Other JAXP APIs.

StAX Packages

The StAX APIs are defined in the packages shown in Table 1-4.

Table 1-4 StAX Packages

Package	**Description**
javax.xml.stream	Defines the XMLStreamReader interface, which is used to iterate over the elements of an XML document. The XMLStreamWriter interface specifies how the XML should be written.
javax.xml.transform.stax	Provides StAX-specific transformation APIs.

Introduction to JAXB

Java Architecture for XML Binding (JAXB) provides a fast and convenient way to bind XML schemas and Java representations, making it easy for Java developers to incorporate XML data and processing functions in Java applications. As part of this process, JAXB provides methods for unmarshalling (reading) XML instance documents into Java content trees, and then marshalling (writing) Java content trees back into XML instance documents. JAXB also provides a way to generate XML schema from Java objects.

JAXB 2.0 includes several important improvements to JAXB 1.0:

- Support for all W3C XML Schema features. (JAXB 1.0 did not specify bindings for some of the W3C XML Schema features.)
- Support for binding Java-to-XML, with the addition of the javax.xml.bind.annotation package to control this binding. (JAXB 1.0 specified the mapping of XML Schema-to-Java, but not Java-to-XML Schema.)
- A significant reduction in the number of generated schema-derived classes.
- Additional validation capabilities through the JAXP 1.3 validation APIs.
- Smaller runtime libraries.

This lesson describes the JAXB architecture, functions, and core concepts, and provides examples with step-by-step procedures for using JAXB.

JAXB Architecture

This section describes the components and the interactions in the JAXB processing model.

Architectural Overview

The following figure shows the components that make up a JAXB implementation.

Figure: JAXB Architectural Overview

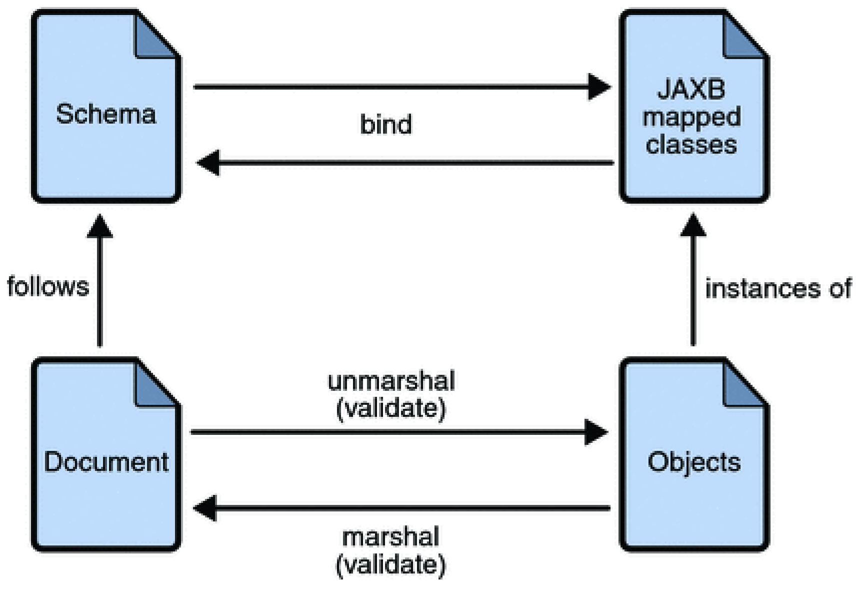
A JAXB implementation consists of the following architectural components:

- Schema compiler: Binds a source schema to a set of schema-derived program elements. The binding is described by an XML-based binding language.
- Schema generator: Maps a set of existing program elements to a derived schema. The mapping is described by program annotations.
- Binding runtime framework: Provides unmarshalling (reading) and marshalling (writing) operations for accessing, manipulating, and validating XML content using either schema-derived or existing program elements.

The JAXB Binding Process

The following figure shows what occurs during the JAXB binding process.

Figure: Steps in the JAXB Binding Process



The general steps in the JAXB data binding process are:

1. Generate classes: An XML schema is used as input to the JAXB binding compiler to generate JAXB classes based on that schema.
2. Compile classes: All of the generated classes, source files, and application code must be compiled.
3. Unmarshal: XML documents written according to the constraints in the source schema are unmarshalled by the JAXB binding framework. Note that JAXB also supports unmarshalling XML data from sources other than files and documents, such as DOM nodes, string buffers, SAX sources, and so forth.
4. Generate content tree: The unmarshalling process generates a content tree of data objects instantiated from the generated JAXB classes; this content tree represents the structure and content of the source XML documents.
5. Validate (optional): The unmarshalling process involves validation of the source XML documents before generating the content tree. Note that if you modify the content tree in Step 6, you can also use the JAXB Validate operation to validate the changes before marshalling the content back to an XML document.
6. Process content: The client application can modify the XML data represented by the Java content tree by using interfaces generated by the binding compiler.

7. Marshal: The processed content tree is marshalled out to one or more XML output documents. The content may be validated before marshalling.

More About Unmarshalling

Unmarshalling provides a client application the ability to convert XML data into JAXB-derived Java objects.

Representing XML Content

This section describes how JAXB represents XML content as Java objects.

Java Representation of XML Schema

JAXB supports the grouping of generated classes in Java packages. A package consists of the following:

- A Java class name that is derived from the XML element name or specified by a binding customization.
- An ObjectFactory class, which is a factory that is used to return instances of a bound Java class.

Binding XML Schemas

This section describes the default XML-to-Java bindings used by JAXB. All of these bindings can be overridden globally or case-by-case by using a custom binding declaration. See the JAXB Specification for complete information about the default JAXB bindings.

Simple Type Definitions

A schema component using a simple type definition typically binds to a Java property. Because there are different kinds of schema components, the following Java property attributes (common to the schema components) include:

- Base type

- Collection type, if any
- Predicate

The rest of the Java property attributes are specified in the schema component using the simple type definition.

Default Data Type Bindings

The following sections explain the default schema-to-Java, JAXBElement, and Java-to-schema data type bindings.

Schema-to-Java Mapping

The Java language provides a richer set of data types than the XML schema. The following table provides a mapping of XML data types to Java data types in JAXB.

Table: JAXB Mapping of XML Schema Built-in Data Types

XML Schema Type	**Java Data Type**
xsd:string	java.lang.String
xsd:integer	java.math.BigInteger
xsd:int	int
xsd.long	long
xsd:short	short
xsd:decimal	java.math.BigDecimal
xsd:float	float
xsd:double	double

xsd:boolean	boolean
xsd:byte	byte
xsd:QName	javax.xml.namespace.QName
xsd:dateTime	javax.xml.datatype.XMLGregorianCalendar
xsd:base64Binary	byte[]
xsd:hexBinary	byte[]
xsd:unsignedInt	long
xsd:unsignedShort	int
xsd:unsignedByte	short
xsd:time	javax.xml.datatype.XMLGregorianCalendar
xsd:date	javax.xml.datatype.XMLGregorianCalendar
xsd:g	javax.xml.datatype.XMLGregorianCalendar
xsd:anySimpleType	java.lang.Object
xsd:anySimpleType	java.lang.String
xsd:duration	javax.xml.datatype.Duration
xsd:NOTATION	javax.xml.namespace.QName

JAXBElement Object

When XML element information cannot be inferred by the derived Java representation of the XML content, a JAXBElement object is provided. This object has methods to get and set the object name and object value.

Java-to-Schema Mapping

The following table shows the default mapping of Java classes to XML data types.

Table: JAXB Mapping of XML Data Types to Java Classes

Java Class	**XML Data Type**
java.lang.String	xs:string
java.math.BigInteger	xs:integer
java.math.BigDecimal	xs:decimal
java.util.Calendar	xs:dateTime
java.util.Date	xs:dateTime
javax.xml.namespace.QName	xs:QName
java.net.URI	xs:string
javax.xml.datatype.XMLGregorianCalendar	xs:anySimpleType
javax.xml.datatype.Duration	xs:duration
java.lang.Object	xs:anyType
java.awt.Image	xs:base64Binary
javax.activation.DataHandler	xs:base64Binary

javax.xml.transform.Source	xs:base64Binary
java.util.UUID	xs:string

Customizing Generated Classes and Java Program Elements

The following sections describe how to customize generated JAXB classes and Java program elements.

Schema-to-Java

Custom JAXB binding declarations enable you to customize your generated JAXB classes beyond the XML-specific constraints in an XML schema to include Java-specific refinements, such as class and package name mappings.

JAXB provides two ways to customize an XML schema:

- As inline annotations in a source XML schema
- As declarations in an external binding customization file that is passed to the JAXB binding compiler

Code examples that show how to customize JAXB bindings are provided later in this document.

Java-to-Schema

The JAXB annotations defined in the javax.xml.bind.annotation package can be used to customize Java program elements to XML schema mapping. The following table summarizes the JAXB annotations that can be used with a Java package.

Table: JAXB Annotations Associated with a Java Package

Annotation	**Description and Default Setting**
@XmlSchema	Maps a package to an XML target namespace.

	Default settings:
	
	@XmlSchema (
	    xmlns = {},
	    namespace = "",
	    elementFormDefault = XmlNsForm.UNSET,
	    attributeFormDefault = XmlNsForm.UNSET
	)

@XmlAccessorType	Controls default serialization of fields and properties. Default settings: @XmlAccessorType (value = AccessType.PUBLIC_MEMBER)
@XmlAccessorOrder	Controls the default ordering of properties and fields mapped to XML elements. Default settings: @XmlAccessorOrder (value = AccessorOrder.UNDEFINED)
@XmlSchemaType	Allows a customized mapping to an XML Schema built-in type. Default settings: @XmlSchemaType (namespace = "http://www.w3.org/2001/XMLSchema", type = DEFAULT.class)
@XmlSchemaTypes	A container annotation for defining multiple @XmlSchemaType annotations.

	Default settings:None

The following table summarizes JAXB annotations that can be used with a Java class.

Table: JAXB Annotations Associated with a Java Class

Annotation	**Description and Default Setting**

@XmlType	Maps a Java class to a schema type.
	Default settings:
	
	@XmlType (
	    name = "##default",
	    propOrder = {""},
	    namespace = "##default",
	    factoryClass = DEFAULT.class,
	    factoryMethod = ""
	)

@XmlRootElement	Associates a global element with the schema type to which the class is mapped. Default settings: @XmlRootElement (name = "##default", namespace = "##default")

The following table summarizes JAXB annotations that can be used with a Java enum type.

Table: JAXB Annotations Associated with a Java enum Type

Annotation	**Description and Default Setting**
@XmlEnum	Maps a Java type to an XML simple type. Default settings: @XmlEnum (value = String.class)
@XmlEnumValue	Maps a Java type to an XML simple type. Default settings:None

@XmlType	Maps a Java class to a schema type. Default settings:
	
	@XmlType (
	    name = "##default",
	    propOrder = {""},
	    namespace = "##default",
	    factoryClass = DEFAULT.class,
	    factoryMethod = ""
	)

@XmlRootElement	Associates a global element with the schema type to which the class is mapped. Default settings: @XmlRootElement (name = "##default", namespace = "##default")

The following table summarizes JAXB annotations that can be used with Java properties and fields.

Table: JAXB Annotations Associated with Java Properties and Fields

Annotation	**Description and Default Setting**

@XmlElement	Maps a JavaBeans property or field to an XML element derived from a property or field name.
	Default settings:
	
	@XmlElement (
	    name = "##default",
	    nillable = false,
	    namespace = "##default",
	    type = DEFAULT.class,
	    defaultValue = "\u0000"
	)

@XmlElements	A container annotation for defining multiple @XmlElement annotations. Default settings:None
@XmlElementRef	Maps a JavaBeans property or field to an XML element derived from a property or field’s type. Default settings: @XmlElementRef (name = "##default", namespace = "##default", type = DEFAULT.class)
@XmlElementRefs	A container annotation for defining multiple @XmlElementRef annotations. Default settings:None

@XmlElementWrapper	Generates a wrapper element around an XML representation. It is typically used as a wrapper XML element around collections. Default settings: @XmlElementWrapper (name = "##default", namespace = "##default", nillable = false)
@XmlAnyElement	Maps a JavaBeans property to an XML infoset representation or a JAXB element. Default settings: @XmlAnyElement (lax = false, value = W3CDomHandler.class)
@XmlAttribute	Maps a JavaBeans property to an XML attribute. Default settings: @XmlAttribute (name = ##default, required = false, namespace = "##default")
@XmlAnyAttribute	Maps a JavaBeans property to a map of wildcard attributes. Default settings:None
@XmlTransient	Prevents the mapping of a JavaBeans property to an XML representation. Default settings:None

@XmlValue	Defines mapping of a class to an XML Schema complex type with a simpleContent or an XML Schema simple type. Default settings:None
@XmlID	Maps a JavaBeans property to an XML ID. Default settings:None
@XmlIDREF	Maps a JavaBeans property to an XML IDREF. Default settings:None
@XmlList	Maps a property to a list simple type. Default settings:None
@XmlMixed	Marks a JavaBeans multi-valued property to support mixed content. Default settings:None
@XmlMimeType	Associates the MIME type that controls the XML representation of the property. Default settings:None
@XmlAttachmentRef	Marks a field/property that its XML form is a URI reference to mime content. Default settings:None
@XmlInlineBinaryData	Disables consideration of XOP encoding for data types that are bound to base64-encoded binary data in XML. Default settings:None

The following table summarizes the JAXB annotation that can be used with object factories.

Table: JAXB Annotations Associated with Object Factories

Annotation	Description and Default Setting

@XmlElementDecl	Maps a factory method to an XML element.
	Default settings:
	
	@XmlElementDecl (
	    scope = GLOBAL.class,
	    namespace = "##default",
	    substitutionHeadNamespace = "##default",
	    substitutionHeadName = ""
	)

The following table summarizes JAXB annotations that can be used with adapters.

Table: JAXB Annotations Associated with Adapters

Annotation	Description and Default Setting
@XmlJavaTypeAdapter	Use the adapter that implements the @XmlAdapter annotation for custom marshalling. Default settings: @XmlJavaTypeAdapter (type = DEFAULT.class)
@XmlJavaTypeAdapters	A container annotation for defining multiple @XmlJavaTypeAdapter annotations at the package level. Default settings: None