

SRINIVAS UNIVERSITY



INSTITUTE OF COMPUTER & INFORMATION SCIENCE

CITY CAMPUS, PANDESHWAR, MANGALORE-575 001.

BACGROUND STUDY MATERIAL

JAVA AND J2EE

IV SEMESTER

CONTENTS

Java & J2EE Syllabus

Java & J2EE Teaching Plan

UNIT-I JAVA DATABASE CONNECTIVITY (JDBC)

- 1.1 Introduction to JDBC
- 1.2 JDBC Driver
- 1.3 Database Connectivity Steps
- 1.4 Connectivity with MySQL
- 1.5 Connectivity with Access without DSN
- 1.6 DriverManager, Connection Interface, Statement interface, ResultSet Interface
- 1.7 PreparedStatement
- 1.8 Java CallableStatement Interface
- 1.9 JDBC Data Types and Transactions
- 1.10 Batch Processing
- 1.11 Assignment-1

UNIT-II SERVLETS

- 2.1 Introduction to Servlets
- 2.2 Life Cycle of a Servlet
- 2.3 Using Tomcat for Servlet Development
- 2.4 The Servlet API
- 2.5 Handling HTTP request and Response
- 2.6 Using Cookies in Servlets
- 2.7 Session Tracking
- 2.8 Servlet Database
- 2.9 Assignment-2

UNIT-III JAVA SERVER PAGES

- 3.1 Introduction
- 3.2 Need and Importance of JSP
- 3.3 The Lifecycle of a JSP Page
- 3.4 MVC Architecture in JSP
- 3.5 JSP Environments and Directives
- 3.6 JSP Actions and Implicit Objects
- 3.7 JSP Session and Cookies Handling
- 3.8 Database Access and JSP Standard Tag Libraries
- 3.9 JSP Custom Tag, JSP Expression Language, and JSP Exception Handling
- 3.10 Assignment-3

UNIT-IV ANNOTATIONS AND JAVA BEANS

- 4.1 Creating Packages, and Interfaces with examples
- 4.2 Creating JAR files and Annotations and New java
- 4.3 Lang Sub Package, Built-in Annotations
- 4.4 Java Beans, Introspection, Customizers, Creating a Java Bean
- 4.5 Bean Manifest File, Creating a Bean JAR file
- 4.6 How to create new Bean, Adding controls to Beans, and setting bean properties
- 4.7 Design Patterns for Properties, Simple Properties, and Design Pattern for Events
- 4.8 Creating Bound Properties, and giving a Bean Methods
- 4.9 Bean an Icon, and creating a Bean Info Class
- 4.10 Persistence and JavaBeans API
- 4.11 Assignment-4

UNIT-V INTRODUCTION TO SPRING FRAMEWORK

- 5.1 Springing into Action - Simplifying Java development
 - 5.2 Containing your Beans
 - 5.3 Surveying the Spring landscape
 - 5.4 The Spring Portfolio
 - 5.5 Wiring Beans – Declaring Beans, and Injecting with JavaConfig
 - 5.6 Using Spring’s Java-based configuration
 - 5.7 Building web applications with Spring Model View Controller (MVC)- Getting started with Spring MVC, and Writing a basic controller
 - 5.8 Handling controller input, Processing forms
 - 5.9 Spring-MVC File Upload
 - 5.10 Assignment-5
- Blueprint with Question Bank
- Sample Question Paper

JAVA AND J2EE Syllabus

Sub. Code: 19MCA43

Hours/ Week: 04

Total Hours: 50

IA Marks: 50

Exam Hours: 02

Exam Marks: 50

Course Objective:

To learn advanced concepts of Java and to provide the knowledge of using J2EE (Java 2 Enterprise Edition) APIs.

UNIT – I**- 10 Hrs**

JDBC: Introduction to JDBC, JDBC Driver types, JDBC `database connections, JDBC Statements, PreparedStatement, CallableStatement, ResultSet, JDBC data types, transactions, Batch Processing, Stored Procedure

UNIT – II**- 10 Hrs**

Servlet: Servlet structure, Life Cycle of a Servlet, Using Tomcat for Servlet Development, The Servlet API, Handling Client Request: Form data, Handling client HTTP request and server HTTP Response, HTTP status codes, Handling Cookies, Session tracking, Database Access

UNIT – III**- 10 Hrs**

JSP: Overview of JSP Technology, Need of JSP, Advantages of JSP, Life Cycle of JSP Page, JSP Processing, JSP Application Design with MVC, Setting Up the JSP Environment, JSP Directives, JSP Action, JSP Implicit Objects, JSP Form Processing, JSP Session and Cookies Handling, JSP Session Tracking JSP Database Access, JSP Standard Tag Libraries, JSP Custom Tag, JSP Expression Language, JSP Exception Handling

UNIT – IV**- 10 Hrs**

Annotations and Java Beans: Creating Packages, Interfaces, JAR files and Annotations, New java. Lang Sub Package, Built-in Annotations, Working with Java Beans, Introspection, Customizers, Creating a Java Bean, Creating a Bean Manifest File, Creating a Bean JAR file, Using a new Bean, Adding controls to Beans, Giving a bean properties, Design Patterns for Properties, Simple Properties, Design Pattern for Events, Creating Bound Properties, Giving a Bean Methods, Giving a Bean an Icon, Creating a Bean Info Class, Persistence, The JavaBeans API

UNIT – V**- 10 Hrs**

Introduction to Spring Framework: Springing into Action - Simplifying Java development, Containing your beans, Surveying the Spring landscape - Spring Modules, The Spring Portfolio, Wiring Beans – Declaring Beans, Injecting into bean properties, Using Spring's Java-based configuration, Building web applications with Spring MVC - Getting started with Spring MVC, Writing a basic controller, Handling controller input, Processing forms, Handling file uploads

Reference Books:

1. Java - The Complete Reference – Herbert Schildt, 7th Edition, Tata McGraw Hill.
2. J2EE - The Complete Reference – Jim Keogh, Tata McGraw Hill.

Teaching Plan

JAVA & J2EEE

(A Computer Language with Platform Independence and having applications in Mobile and Desktop platforms for Standalone and Web based programming)

Instructor: Dr. Krishna Prasad K

Sub. Code: 19MCA43

Hours/ Week: 04

Total Hours: 50

IA Marks: 50

Exam Hours: 02

Exam Marks: 50

Learning Objective:

- To understand the importance of extension JDBC package in Enterprise Java applications.
- To understand and use the Java Servlet Programming with its lifecycle and different methods.
- To acquire knowledge of Java Server Programming with its lifecycle and different methods.
- To become familiarize in the concepts of Annotations and Java Beans.
- To understand fundamental concepts of Spring Framework.

Learning Outcome: Upon Successful completion of the course students can able to understand;

- JDBC package in Enterprise Java Applications with different procedures.
- Java Servlet Programming with its lifecycle and different methods.
- Java Server Programming with its lifecycle and different methods.
- Concepts of Annotations and Java Beans.
- Fundamental concepts of Spring Framework.

Teaching Pedagogy:

Lecturing Method, Video Lecturing, Group Discussion, Practical Learning, Student Presentation, and Assignment based learning.

UNIT-I JDBC

Session 1: Introduction to Java Database Connectivity (JDBC) with its importance.

Session 2: Java Database Connectivity (JDBC)-Drivers and Different types of Drivers.

Session 3: Video Lecturing on Java Database Connectivity (JDBC) and its Drivers.

Session 4: Using Java Database Connectivity (JDBC) connecting java program to Databases.

Session 5: To understand the concepts Java Database Connectivity (JDBC) with its Statements

Session 6: Java Database Connectivity (JDBC) prepared statements and example for the same.

Session 7: Java Database Connectivity (JDBC) callabelstatement concept with examples.

Session 8: Java Database Connectivity (JDBC)-data types and various transactions.

Session 9: Java Database Connectivity (JDBC)-Batch Processing concepts with its importance.

Session 10: Java Database Connectivity (JDBC)-Stored Procedure concepts and examples.

Assignment I

UNIT-II

Servlets

Session 11: Background- Introduction to Servlets, Introduction to web application, Common Gateway Interface (CGI), Advantages of Servlets.

Session 12: Life Cycle of a Servlet-Loading Servlet class, Creating Instance of Servlet class, Invoking init method, invoking service method, invoking destroy method.

Session 13: Using Tomcat for Servlet Development-Creating a directory structure, Creating servlet, compiling the servlet, creating a deployment descriptor, Start the server and deploy the project, Access the Servlet.

Session 14: The Servlet API- The javax.servlet package, Classes in javax.servlet package, Interfaces in javax.servlet.http package, Classes in javax.servlet.http package.

Session 15: Handling HTTP request and Response-HTTP Client Request, Server Response.

Session 16: Using Cookies-Anatomy of a cookie, Cookies Methods, Setting Cookies with Servlets.

Session 17: Session tracking-Hidden Form Field, URL writing, The HTTP session Object.

Session 18: Database Access-Create Table, Create Data Records, Accessing a database.

Session 19: Video Lecturing on Servlets Basics and its lifecycle.

Session 20: Student Presentations on various concepts of Servlets and few example program demonstrations. Assignment II

UNIT-III

JSP

Session 21: Overview of Java Server Programming (JSP) Technology with fundamental concepts.

Session 22: Need and importance of Java Server Programming (JSP) technology with its Architecture.

Session 23: Java Server Programming (JSP) Page Life Cycle.

Session 24: Java Server Programming (JSP) processing and Applications Design with Model View and Controller (MVC).

Session 25: Video Lecturing Java Server Programming (JSP)-Fundamentals and its life cycle.

Session 26: Setting up of JSP programming Environments and Directives.

Session 27: Discussion on the concepts of JSP Actions and Implicit objects.

Session 28: Discussion on the topics JSP Session and Cookies Handling.

Session 29: Discussion on the topics JSP Session Tracking, Database Access and JSP Standard Tag Libraries.

Session 30: Discussion on the topics JSP Custom Tag, JSP Expression Language, and JSP Exception Handling. Assignment III

UNIT-IV

ANNOTATIONS AND JAVA BEANS

Session 31: Discussion on the basic concepts of Creating Packages, Interfaces with examples.

Session 32: Discussion on the basic concepts of **creating** JAR files and Annotations and New java.

Session 33: Discussion on the topics Lang Sub Package, Built-in Annotations.

Session 34: How to work with Java Beans, Introspection, Customizers, Creating a Java Bean.

Session 35: Discussion on the topics creating a Bean Manifest File, Creating a Bean JAR file.

Session 36: Discussion on how to create new Bean, Adding controls to Beans, and setting bean properties.

Session 37: Discussion on design Patterns for Properties, Simple Properties, and Design Pattern for Events

Session 38: Discussion on Creating Bound Properties, and giving a Bean Methods.

Session 39: Explanation on the topic giving a Bean an Icon, and creating a Bean Info Class.

Session 40: Explanation on the topic Persistence, The JavaBeans API. Assignment IV

UNIT-V

INTRODUCTION TO SPRING FRAMEWORK

Session 41: Introduction to spring, springing into Actions-Springs Bean container, Spring Score modules.

Session 42: Discussion on the topic simplifying Java development, and containing your beans.

Session 43: Explanation on the topic surveying the Spring landscape, and Spring Modules.

Session 44: Explanation on the topic the Spring Portfolio, Wiring Beans – Declaring Beans.

Session 45: Discussion on the topic Injecting into bean properties, Using Spring's Java-based configuration

Session 46: Discussion on the topic Building web applications with Spring Model View Controller (MVC)

Session 47: Discussion on the topic getting started with Spring MVC with examples.

Session 48: A demo program on writing a basic controller with spring properties.

Session 49: Discussion on the topic handling controller input, Processing forms.

Session 50: Discussion on the topic handling file uploads with examples. Assignment V

Reference Books

1. Java - The Complete Reference – Herbert Schildt, 7th Edition, Tata McGraw Hill.
2. J2EE - The Complete Reference – Jim Keogh, Tata McGraw Hill.

UNIT-I

JAVA DATABASE CONNECTIVITY (JDBC)

1.1 Introduction to JDBC

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

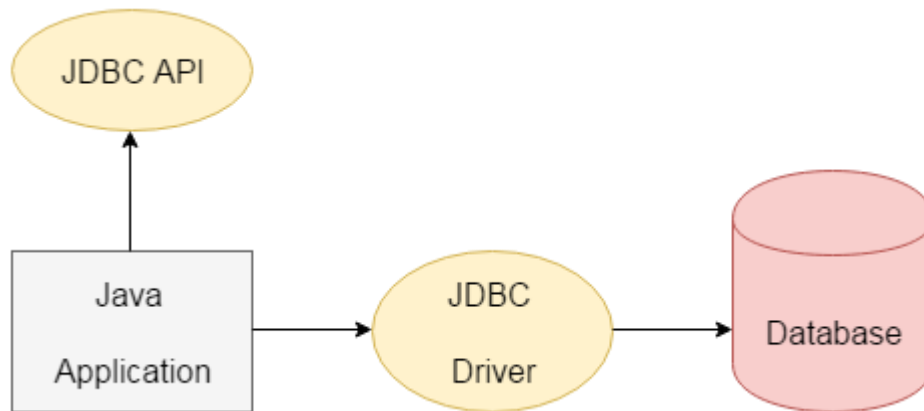


Figure: JDBC Driver

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

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

1.2 JDBC Driver

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

JDBC-ODBC bridge driver

Native-API driver (partially java driver)

Network Protocol driver (fully java driver)

Thin driver (fully java driver)

JDBC-ODBC bridge driver

The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged because of thin driver.

Advantages:

- easy to use.
- can be easily connected to any database.

Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.

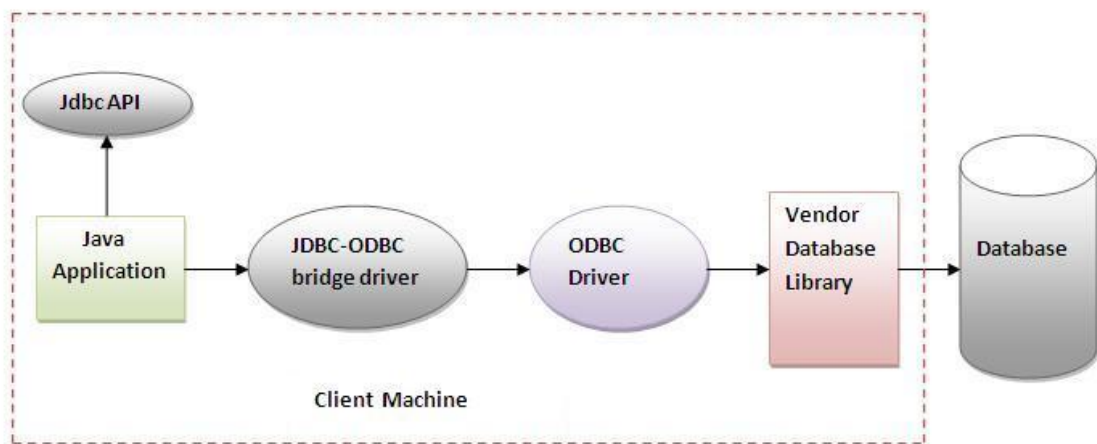


Figure- JDBC-ODBC Bridge Driver

Native-API driver

The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java.

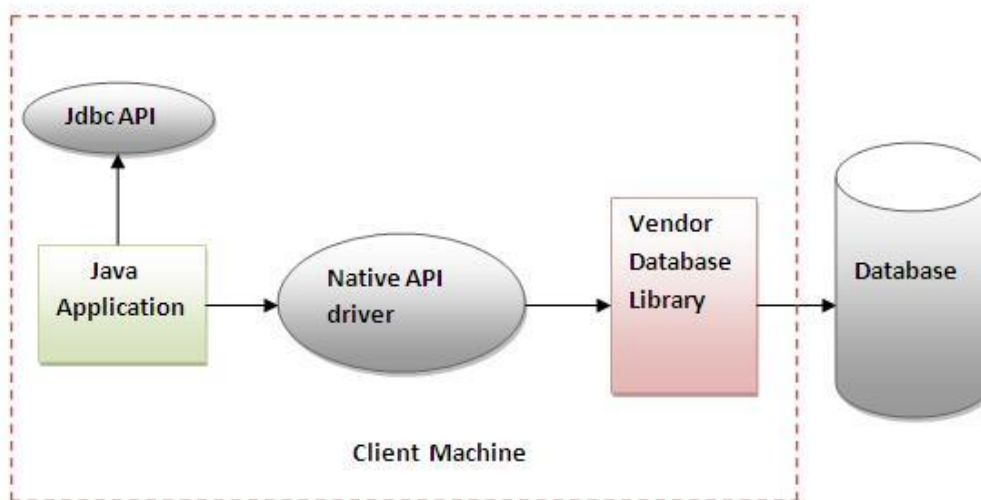


Figure- Native API Driver

Advantage:

- performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

- The Native driver needs to be installed on the each client machine.
- The Vendor client library needs to be installed on client machine.

Network Protocol Driver

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.

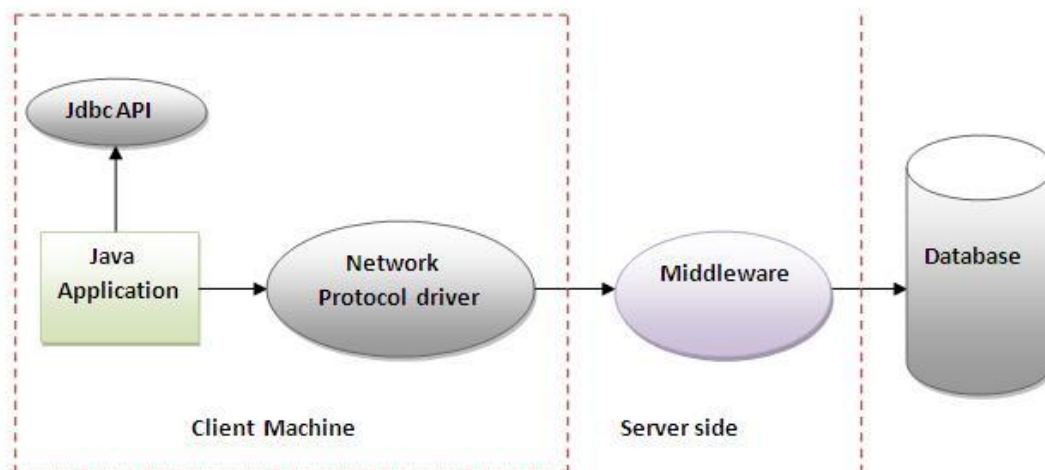


Figure- Network Protocol Driver

Advantage:

- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantages:

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

Thin Driver

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.

Advantage:

- Better performance than all other drivers.
- No software is required at client side or server side.

Disadvantage:

- Drivers depends on the Database.

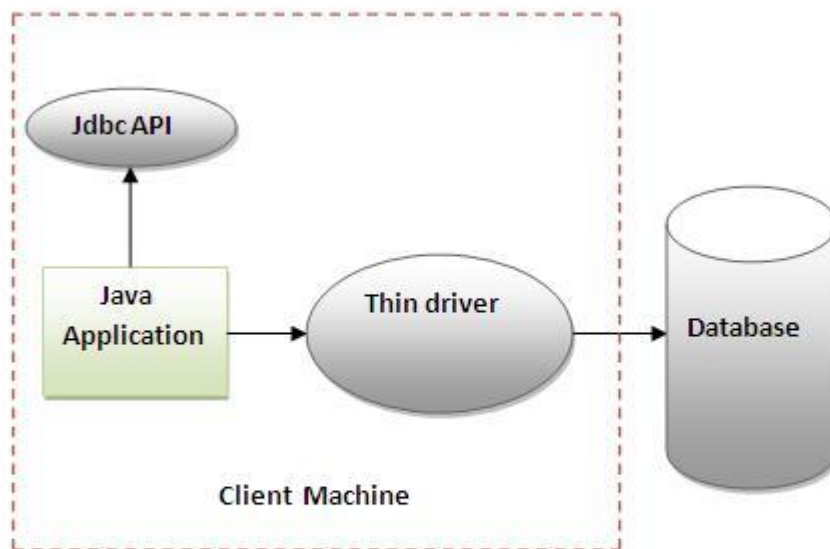


Figure- Thin Driver

1.3 Database Connectivity Steps

There are 5 steps to connect any java application with the database in java using JDBC. They are as follows:

- Register the driver class
- Creating connection
- Creating statement
- Executing queries
- Closing connection

1) Register the driver class

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

Syntax of `forName()` method

`public static void forName(String className) throws ClassNotFoundException`

Example to register the `OracleDriver` class

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

2) Create the connection Object

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

Syntax for `getConnection()` method

1) `public static Connection getConnection(String url) throws SQLException`

2) `public static Connection getConnection(String url, String name, String password)`

`throws SQLException`

Example to establish connection with Oracle database

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

3) Create the statement object

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

Syntax for createStatement() method

```
public Statement createStatement()throws SQLException
```

Example to create the statement object

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

4) Execute the query

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

Syntax of executeQuery() method

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

Example to wxwcute query

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

5) Close the connection object

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

Syntax of close() method

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

Example to close connection

```
con.close()
```

This sample example can serve as a template when you need to create your own JDBC application.

//STEP 1. Import required packages

```
import java.sql.*;
```

```
public class FirstExample {
```

```
    // JDBC driver name and database URL
```

```
    static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
```

```
    static final String DB_URL = "jdbc:mysql://localhost/EMP";
```

```
    // Database credentials
```

```
    static final String USER = "username";
```

```
    static final String PASS = "password";
```

```
    public static void main(String[] args) {
```

```
        Connection conn = null;
```

```
        Statement stmt = null;
```

```
        try{
```

```
            //STEP 2: Register JDBC driver
```

```
Class.forName("com.mysql.jdbc.Driver");

//STEP 3: Open a connection
System.out.println("Connecting to database...");
conn = DriverManager.getConnection(DB_URL,USER,PASS);

//STEP 4: Execute a query
System.out.println("Creating statement...");
stmt = conn.createStatement();
String sql;
sql = "SELECT id, first, last, age FROM Employees";
ResultSet rs = stmt.executeQuery(sql);

//STEP 5: Extract data from result set
while(rs.next()){
    //Retrieve by column name
    int id = rs.getInt("id");
    int age = rs.getInt("age");
    String first = rs.getString("first");
    String last = rs.getString("last");

    //Display values
    System.out.print("ID: " + id);
    System.out.print(", Age: " + age);
    System.out.print(", First: " + first);
    System.out.println(", Last: " + last);
}
//STEP 6: Clean-up environment
rs.close();
stmt.close();
conn.close();
}catch(SQLException se){
    //Handle errors for JDBC
    se.printStackTrace();
}catch(Exception e){
    //Handle errors for Class.forName
    e.printStackTrace();
}finally{
    //finally block used to close resources
    try{
        if(stmt!=null)
            stmt.close();
    }catch(SQLException se2){
    }// nothing we can do
    try{
        if(conn!=null)
            conn.close();
    }catch(SQLException se){
        se.printStackTrace();
    }//end finally try
```

```

    }//end try
    System.out.println("Goodbye!");
} //end main
} //end FirstExample

```

1.4 Connectivity with MySQL

For connecting java application with the mysql database, you need to follow 5 steps to perform database connectivity.

In this example we are using MySql as the database. So we need to know following informations for the mysql database:

- **Driver class:** The driver class for the mysql database is **com.mysql.jdbc.Driver**.
- **Connection URL:** The connection URL for the mysql database is **jdbc:mysql://localhost:3306/sonoo** where jdbc is the API, mysql is the database, localhost is the server name on which mysql is running, we may also use IP address, 3306 is the port number and sonoo is the database name. We may use any database, in such case, you need to replace the sonoo with your database name.
- **Username:** The default username for the mysql database is **root**.

Password: Password is given by the user at the time of installing the mysql database. In this example, we are going to use root as the password.

Let's first create a table in the mysql database, but before creating table, we need to create database first.

create a table in the mysql database, but before creating table, we need to create database first.

```

create database sonoo;
use sonoo;
create table emp(id int(10),name varchar(40),age int(3));

```

In this example, sonoo is the database name, root is the username and password.

```

import java.sql.*;
class MysqlCon{
public static void main(String args[]){
try{
Class.forName("com.mysql.jdbc.Driver");
Connection con=DriverManager.getConnection(
"jdbc:mysql://localhost:3306/sonoo","root","root");
//here sonoo is database name, root is username and password
Statement stmt=con.createStatement();
ResultSet rs=stmt.executeQuery("select * from emp");
while(rs.next())
System.out.println(rs.getInt(1)+" "+rs.getString(2)+" "+rs.getString(3));
con.close();
}catch(Exception e){ System.out.println(e);}
}
}

```

Two ways to load the jar file:

1. paste the mysqlconnector.jar file in jre/lib/ext folder
2. set classpath
 - 1) paste the mysqlconnector.jar file in JRE/lib/ext folder:

Download the mysqlconnector.jar file. Go to jre/lib/ext folder and paste the jar file here.

- 2) set classpath:

There are two ways to set the classpath:

- temporary
- permanent

How to set the temporary classpath

open command prompt and write:

C:>set classpath=c:\folder\mysql-connector-java-5.0.8-bin.jar;;

How to set the permanent classpath

Go to environment variable then click on new tab. In variable name write **classpath** and in variable value paste the path to the mysqlconnector.jar file by appending mysqlconnector.jar;; as C:\folder\mysql-connector-java-5.0.8-bin.jar;;

1.5 Connectivity with Access without DSN

There are two ways to connect java application with the access database.

1. Without DSN (Data Source Name)
2. With DSN

Java is mostly used with Oracle, mysql, or DB2 database. So you can learn this topic only for knowledge.

Example to Connect Java Application with access without DSN

In this example, we are going to connect the java program with the access database. In such case, we have created the login table in the access database. There is only one column in the table named name. Let's get all the name of the login table.

```
import java.sql.*;
class Test{
    public static void main(String ar[]){
        try{
            String database="student.mdb";//Here database exists in the current directory

            String url="jdbc:odbc:Driver={Microsoft Access Driver (*.mdb)};
                DBQ=" + database + ";DriverID=22;READONLY=true";

            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            Connection c=DriverManager.getConnection(url);
            Statement st=c.createStatement();
            ResultSet rs=st.executeQuery("select * from login");

            while(rs.next()){
                System.out.println(rs.getString(1));
            }

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

    }
}
```

Example to Connect Java Application with access with DSN

Connectivity with type1 driver is not considered good. To connect java application with type1 driver, create DSN first, here we are assuming your dsn name is mydsn.

```
import java.sql.*;
class Test{
public static void main(String ar[]){
try{
String url="jdbc:odbc:mydsn";
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection c=DriverManager.getConnection(url);
Statement st=c.createStatement();
ResultSet rs=st.executeQuery("select * from login");

while(rs.next()){
System.out.println(rs.getString(1));
}
}catch(Exception ee){System.out.println(ee);}
}
}
```

1.6 DriverManager, Connection Interface, Statement interface, ResultSet Interface

The DriverManager class acts as an interface between user and drivers. It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver. The DriverManager class maintains a list of Driver classes that have registered themselves by calling the method DriverManager.registerDriver().

Method	Description
1) public static void registerDriver(Driver driver):	is used to register the given driver with DriverManager.
2) public static void deregisterDriver(Driver driver):	is used to deregister the given driver (drop the driver from the list) with DriverManager.
3) public static Connection getConnection(String url):	is used to establish the connection with the specified url.
4) public static Connection getConnection(String url,String userName,String password):	is used to establish the connection with the specified url, username and password.

Connection Interface

A Connection is the session between java application and database. The Connection interface is a factory of Statement, PreparedStatement, and DatabaseMetaData i.e. object of Connection can be used to get the object of Statement and DatabaseMetaData. The Connection interface provide many methods for transaction management like commit(), rollback() etc.

Commonly use methods of connection interface

1) public Statement createStatement(): creates a statement object that can be used to execute SQL queries.
2) public Statement createStatement(int resultSetType,int resultSetConcurrency): Creates a Statement object that will generate ResultSet objects with the given type and concurrency.

3) public void setAutoCommit(boolean status): is used to set the commit status. By default it is true.
4) public void commit(): saves the changes made since the previous commit/rollback permanent.
5) public void rollback(): Drops all changes made since the previous commit/rollback.
6) public void close(): closes the connection and Releases a JDBC resources immediately.

Statement interface

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

Commonly used methods of Statement interface:

The important methods of Statement interface are as follows:

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

Example of Statement interface

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

```
import java.sql.*;
class FetchRecord{
public static void main(String args[])throws Exception{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
Statement stmt=con.createStatement();
//stmt.executeUpdate("insert into emp765 values(33,'Irfan',50000)");
//int result=stmt.executeUpdate("update emp765 set name='Vimal',salary=10000 where id=33");
int result=stmt.executeUpdate("delete from emp765 where id=33");
System.out.println(result+" records affected");
con.close();
}}
```

ResultSet Interface

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

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

Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,

ResultSet.CONCUR_UPDATABLE);

Commonly used methods of ResultSet interface

1) public boolean next():	is used to move the cursor to the one row next from the current position.
2) public boolean previous():	is used to move the cursor to the one row previous from the current position.
3) public boolean first():	is used to move the cursor to the first row in result set object.
4) public boolean last():	is used to move the cursor to the last row in result set object.
5) public boolean absolute(int row):	is used to move the cursor to the specified row number in the ResultSet object.
6) public boolean relative(int row):	is used to move the cursor to the relative row number in the ResultSet object, it may be positive or negative.
7) public int getInt(int columnIndex):	is used to return the data of specified column index of the current row as int.
8) public int getInt(String columnName):	is used to return the data of specified column name of the current row as int.
9) public String getString(int columnIndex):	is used to return the data of specified column index of the current row as String.
10) public String getString(String columnName):	is used to return the data of specified column name of the current row as String.

Example of Scrollable ResultSet

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

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

Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
Statement stmt=con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_UPDATABLE);
ResultSet rs=stmt.executeQuery("select * from emp765");
//getting the record of 3rd row
rs.absolute(3);
System.out.println(rs.getString(1)+" "+rs.getString(2)+" "+rs.getString(3));

con.close();
}}
```

1.7 PreparedStatement

The PreparedStatement interface is a sub interface of Statement. It is used to execute parameterized query.

Let's see the example of parameterized query:

```
String sql="insert into emp values(?,?,?);"
```

As you can see, we are passing parameter (?) for the values. Its value will be set by calling the setter methods of PreparedStatement.

Why use PreparedStatement?

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

How to get the instance of PreparedStatement?

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

```
public PreparedStatement prepareStatement(String query)throws SQLException{ }
```

Methods of PreparedStatement interface

The important methods of PreparedStatement interface are given below:

Method	Description
public void setInt(int paramIndex, int value)	sets the integer value to the given parameter index.
public void setString(int paramIndex, String value)	Example of PreparedStatement interface that updates the record
public void setFloat(int paramIndex, float value)	sets the float value to the given parameter index.
This sample example can serve as a template when you need to create your own JDBC application in the future.	sets the double value to the given parameter index.
public int executeUpdate()	executes the query. It is used for create, drop, insert, update, delete etc.
public ResultSet executeQuery()	executes the select query. It returns an instance of ResultSet.

The first example shows how to create a prepared statement by using a string literal to supply the text of the statement:

```
mysql> PREPARE stmt1 FROM 'SELECT SQRT(POW(?,2) + POW(?,2)) AS hypotenuse';
```

```
mysql> SET @a = 3;
```

```
mysql> SET @b = 4;
```

```
mysql> EXECUTE stmt1 USING @a, @b;
```

```
+-----+
```

```
| hypotenuse |
```

```
+-----+
```

```
|      5      |
```

```
+-----+
```

```
mysql> DEALLOCATE PREPARE stmt1;
```

The second example is similar, but supplies the text of the statement as a user variable:

```
mysql> SET @s = 'SELECT SQRT(POW(?,2) + POW(?,2)) AS hypotenuse';
```

```
mysql> PREPARE stmt2 FROM @s;
```

```
mysql> SET @a = 6;
```

```
mysql> SET @b = 8;
```

```
mysql> EXECUTE stmt2 USING @a, @b;
```

```
+-----+  
| hypotenuse |  
+-----+  
|      10 |  
+-----+
```

```
mysql> DEALLOCATE PREPARE stmt2;
```

Here is an additional example that demonstrates how to choose the table on which to perform a query at runtime, by storing the name of the table as a user variable:

```
mysql> USE test;
```

```
mysql> CREATE TABLE t1 (a INT NOT NULL);
```

```
mysql> INSERT INTO t1 VALUES (4), (8), (11), (32), (80);
```

```
mysql> SET @table = 't1';
```

```
mysql> SET @s = CONCAT('SELECT * FROM ', @table);
```

```
mysql> PREPARE stmt3 FROM @s;
```

```
mysql> EXECUTE stmt3;
```

```
+-----+  
| a |  
+-----+  
| 4 |  
| 8 |  
| 11 |  
| 32 |  
| 80 |  
+-----+
```

```
mysql> DEALLOCATE PREPARE stmt3;
```

SQL Syntax Allowed in Prepared Statements

The following SQL statements can be used as prepared statements:

ALTER TABLE

ALTER USER

ANALYZE TABLE

CACHE INDEX

CALL

CHANGE MASTER

CHECKSUM {TABLE | TABLES}

COMMIT

{CREATE | DROP} INDEX

{CREATE | RENAME | DROP} DATABASE

{CREATE | DROP} TABLE

{CREATE | RENAME | DROP} USER

{CREATE | DROP} VIEW

DELETE

DO

FLUSH {TABLE | TABLES | TABLES WITH READ LOCK | HOSTS | PRIVILEGES

| LOGS | STATUS | MASTER | SLAVE | USER_RESOURCES}

GRANT

INSERT

INSTALL PLUGIN
KILL
LOAD INDEX INTO CACHE
OPTIMIZE TABLE
RENAME TABLE
REPAIR TABLE
REPLACE
RESET {MASTER | SLAVE}
REVOKE
SELECT
SET
SHOW {WARNINGS | ERRORS}
SHOW BINLOG EVENTS
SHOW CREATE {PROCEDURE | FUNCTION | EVENT | TABLE | VIEW}
SHOW {MASTER | BINARY} LOGS
SHOW {MASTER | SLAVE} STATUS
SLAVE {START | STOP}
TRUNCATE TABLE
UNINSTALL PLUGIN
UPDATE

1.8 Java CallableStatement Interface

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

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

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

How to get the instance of CallableStatement?

The prepareCall() method of Connection interface returns the instance of CallableStatement.

Syntax is given below:

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

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

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

It calls the procedure myprocedure that receives 2 arguments.

Full example to call the stored procedure using JDBC

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

```
create or replace procedure "INSERTR"
```

```
(id IN NUMBER,  
name IN VARCHAR2)
```

```
is
```

```
begin
```

```
insert into user420 values(id,name);
```

```
end;
```

```
/
```

The table structure is given below:

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

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

```
import java.sql.*;
public class Proc {
    public static void main(String[] args) throws Exception{
        Class.forName("oracle.jdbc.driver.OracleDriver");
        Connection con=DriverManager.getConnection(
            "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
        CallableStatement stmt=con.prepareCall("{ call insertR(?,?) }");
        stmt.setInt(1,1011);
        stmt.setString(2,"Amit");
        stmt.execute();
        System.out.println("success");
    }
}
```

Following is the example, which makes use of the CallableStatement along with the following **getEmpName()** MySQL stored procedure –

Make sure you have created this stored procedure in your EMP Database. You can use MySQL Query Browser to get it done.

DELIMITER \$\$

```
DROP PROCEDURE IF EXISTS `EMP`.`getEmpName` $$
CREATE PROCEDURE `EMP`.`getEmpName`
    (IN EMP_ID INT, OUT EMP_FIRST VARCHAR(255))
BEGIN
    SELECT first INTO EMP_FIRST
    FROM Employees
    WHERE ID = EMP_ID;
END $$
```

DELIMITER ;

Copy and past the following example in JDBCExample.java, compile and run as follows –

//STEP 1. Import required packages

```
import java.sql.*;
public class JDBCExample {
    // JDBC driver name and database URL
    static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
    static final String DB_URL = "jdbc:mysql://localhost/EMP";

    // Database credentials
    static final String USER = "username";
    static final String PASS = "password";

    public static void main(String[] args) {
        Connection conn = null;
```

```
CallableStatement stmt = null;
try{
    //STEP 2: Register JDBC driver
    Class.forName("com.mysql.jdbc.Driver");

    //STEP 3: Open a connection
    System.out.println("Connecting to database...");
    conn = DriverManager.getConnection(DB_URL,USER,PASS);

    //STEP 4: Execute a query
    System.out.println("Creating statement...");
    String sql = "{call getEmpName (?, ?)}";
    stmt = conn.prepareCall(sql);

    //Bind IN parameter first, then bind OUT parameter
    int empID = 102;
    stmt.setInt(1, empID); // This would set ID as 102
    // Because second parameter is OUT so register it
    stmt.registerOutParameter(2, java.sql.Types.VARCHAR);

    //Use execute method to run stored procedure.
    System.out.println("Executing stored procedure..." );
    stmt.execute();

    //Retrieve employee name with getXXX method
    String empName = stmt.getString(2);
    System.out.println("Emp Name with ID:" +
        empID + " is " + empName);
    stmt.close();
    conn.close();
}catch(SQLException se){
    //Handle errors for JDBC
    se.printStackTrace();
}catch(Exception e){
    //Handle errors for Class.forName
    e.printStackTrace();
}finally{
    //finally block used to close resources
    try{
        if(stmt!=null)
            stmt.close();
    }catch(SQLException se2){
    }// nothing we can do
    try{
        if(conn!=null)
            conn.close();
    }catch(SQLException se){
        se.printStackTrace();
    }//end finally try
}//end try
```

```

    System.out.println("Goodbye!");
} //end main
} //end JDBCExample
Now let us compile the above example as follows –
C:\>javac JDBCExample.java
C:\>

```

When you run **JDBCExample**, it produces the following result –

```

:\>java JDBCExample
Connecting to database...
Creating statement...
Executing stored procedure...
Emp Name with ID:102 is Zaid
Goodbye!
C:\>

```

1.9 JDBC Data Types and Transactions

The JDBC driver converts the Java data type to the appropriate JDBC type, before sending it to the database. It uses a default mapping for most data types. For example, a Java int is converted to an SQL INTEGER. Default mappings were created to provide consistency between drivers.

The following table summarizes the default JDBC data type that the Java data type is converted to, when you call the setXXX() method of the PreparedStatement or CallableStatement object or the ResultSet.updateXXX() method.

SQL	JDBC/Java	setXXX	updateXXX
VARCHAR	java.lang.String	setString	updateString
CHAR	java.lang.String	setString	updateString
LONGVARCHAR	java.lang.String	setString	updateString
BIT	boolean	setBoolean	updateBoolean
NUMERIC	java.math.BigDecimal	setBigDecimal	updateBigDecimal
TINYINT	byte	setByte	updateByte
SMALLINT	short	setShort	updateShort
INTEGER	int	setInt	updateInt
BIGINT	long	setLong	updateLong
REAL	float	setFloat	updateFloat
FLOAT	float	setFloat	updateFloat
DOUBLE	double	setDouble	updateDouble
VARBINARY	byte[]	setBytes	updateBytes

BINARY	byte[]	setBytes	updateBytes
DATE	java.sql.Date	setDate	updateDate
TIME	java.sql.Time	setTime	updateTime
TIMESTAMP	java.sql.Timestamp	setTimestamp	updateTimestamp
CLOB	java.sql.Clob	setClob	updateClob
BLOB	java.sql.Blob	setBlob	updateBlob
ARRAY	java.sql.Array	setARRAY	updateARRAY
REF	java.sql.Ref	SetRef	updateRef
STRUCT	java.sql.Struct	SetStruct	updateStruct

JDBC 3.0 has enhanced support for BLOB, CLOB, ARRAY, and REF data types. The `ResultSet` object now has `updateBLOB()`, `updateCLOB()`, `updateArray()`, and `updateRef()` methods that enable you to directly manipulate the respective data on the server.

The `setXXX()` and `updateXXX()` methods enable you to convert specific Java types to specific JDBC data types. The methods, `setObject()` and `updateObject()`, enable you to map almost any Java type to a JDBC data type.

`ResultSet` object provides corresponding `getXXX()` method for each data type to retrieve column value. Each method can be used with column name or by its ordinal position.

SQL	JDBC/Java	setXXX	getXXX
VARCHAR	java.lang.String	setString	getString
CHAR	java.lang.String	setString	getString
LONGVARCHAR	java.lang.String	setString	getString
BIT	boolean	setBoolean	getBoolean
NUMERIC	java.math.BigDecimal	setBigDecimal	getBigDecimal
TINYINT	byte	setByte	getByte
SMALLINT	short	setShort	getShort
INTEGER	int	setInt	getInt
BIGINT	long	setLong	getLong
REAL	float	setFloat	getFloat

FLOAT	float	setFloat	getFloat
DOUBLE	double	setDouble	getDouble
VARBINARY	byte[]	setBytes	getBytes
BINARY	byte[]	setBytes	getBytes
DATE	java.sql.Date	setDate	getDate
TIME	java.sql.Time	setTime	getTime
TIMESTAMP	java.sql.Timestamp	setTimestamp	getTimestamp
CLOB	java.sql.Clob	setClob	getClob
BLOB	java.sql.Blob	setBlob	getBlob
ARRAY	java.sql.Array	setARRAY	getARRAY
REF	java.sql.Ref	SetRef	getRef
STRUCT	java.sql.Struct	SetStruct	getStruct

Date & Time Data Types

The `java.sql.Date` class maps to the SQL DATE type, and the `java.sql.Time` and `java.sql.Timestamp` classes map to the SQL TIME and SQL TIMESTAMP data types, respectively.

Following example shows how the Date and Time classes format the standard Java date and time values to match the SQL data type requirements.

```
import java.sql.Date;
import java.sql.Time;
import java.sql.Timestamp;
import java.util.*;

public class SqlDateTime {
    public static void main(String[] args) {
        //Get standard date and time
        java.util.Date javaDate = new java.util.Date();
        long javaTime = javaDate.getTime();
        System.out.println("The Java Date is:" +
            javaDate.toString());

        //Get and display SQL DATE
        java.sql.Date sqlDate = new java.sql.Date(javaTime);
        System.out.println("The SQL DATE is: " +
            sqlDate.toString());

        //Get and display SQL TIME
        java.sql.Time sqlTime = new java.sql.Time(javaTime);
```

```

System.out.println("The SQL TIME is: " +
    sqlTime.toString());
//Get and display SQL TIMESTAMP
java.sql.Timestamp sqlTimestamp =
    new java.sql.Timestamp(javaTime);
System.out.println("The SQL TIMESTAMP is: " +
    sqlTimestamp.toString());
} //end main
} //end SqlDateTime

```

Now let us compile the above example as follows –

```
C:\>javac SqlDateTime.java
```

```
C:\>
```

When you run JDBCExample, it produces the following result –

```
C:\>java SqlDateTime
```

```
The Java Date is: Tue Aug 18 13:46:02 GMT+04:00 2009
```

```
The SQL DATE is: 2009-08-18
```

```
The SQL TIME is: 13:46:02
```

```
The SQL TIMESTAMP is: 2009-08-18 13:46:02.828
```

```
C:\>
```

Handling NULL Values

SQL's use of NULL values and Java's use of null are different concepts. So, to handle SQL NULL values in Java, there are three tactics you can use –

Avoid using getXXX() methods that return primitive data types.

Use wrapper classes for primitive data types, and use the ResultSet object's wasNull() method to test whether the wrapper class variable that received the value returned by the getXXX() method should be set to null.

Use primitive data types and the ResultSet object's wasNull() method to test whether the primitive variable that received the value returned by the getXXX() method should be set to an acceptable value that you've chosen to represent a NULL.

Here is one example to handle a NULL value –

```

Statement stmt = conn.createStatement();
String sql = "SELECT id, first, last, age FROM Employees";
ResultSet rs = stmt.executeQuery(sql);

```

```
int id = rs.getInt(1);
```

```
if( rs.wasNull() ) {
```

```
    id = 0;
```

```
}
```

1.10 Batch Processing

Batch Processing allows you to group related SQL statements into a batch and submit them with one call to the database. When you send several SQL statements to the database at once, you reduce the amount of communication overhead, thereby improving performance.

JDBC drivers are not required to support this feature. You should use the *DatabaseMetaData.supportsBatchUpdates()* method to determine if the target database supports batch update processing. The method returns true if your JDBC driver supports this feature.

The **addBatch()** method of *Statement*, *PreparedStatement*, and *CallableStatement* is used to add individual statements to the batch. The **executeBatch()** is used to start the execution of all the statements grouped together.

The **executeBatch()** returns an array of integers, and each element of the array represents the update count for the respective update statement.

Just as you can add statements to a batch for processing, you can remove them with the **clearBatch()** method. This method removes all the statements you added with the **addBatch()** method. However, you cannot selectively choose which statement to remove.

Batching with Statement Object

Here is a typical sequence of steps to use Batch Processing with Statement Object –

Create a Statement object using either *createStatement()* methods.

Set auto-commit to false using *setAutoCommit()*.

Add as many as SQL statements you like into batch using *addBatch()* method on created statement object.

Execute all the SQL statements using *executeBatch()* method on created statement object.

Finally, commit all the changes using *commit()* method.

Example

The following code snippet provides an example of a batch update using Statement object –

```
// Create statement object
Statement stmt = conn.createStatement();

// Set auto-commit to false
conn.setAutoCommit(false);

// Create SQL statement
String SQL = "INSERT INTO Employees (id, first, last, age) " +
    "VALUES(200,'Zia', 'Ali', 30)";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);

// Create one more SQL statement
String SQL = "INSERT INTO Employees (id, first, last, age) " +
    "VALUES(201,'Raj', 'Kumar', 35)";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);

// Create one more SQL statement
String SQL = "UPDATE Employees SET age = 35 " +
    "WHERE id = 100";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);

// Create an int[] to hold returned values
int[] count = stmt.executeBatch();

//Explicitly commit statements to apply changes
conn.commit();
```

For a better understanding, let us study the [Batching - Example Code](#).

Batching with PreparedStatement Object

Here is a typical sequence of steps to use Batch Processing with PreparedStatement Object –

Create SQL statements with placeholders.

Create PreparedStatement object using either *prepareStatement()* methods.

Set auto-commit to false using *setAutoCommit()*.

Add as many as SQL statements you like into batch using *addBatch()* method on created statement object.

Execute all the SQL statements using *executeBatch()* method on created statement object.

Finally, commit all the changes using *commit()* method.

The following code snippet provides an example of a batch update using PreparedStatement object –

```
// Create SQL statement
String SQL = "INSERT INTO Employees (id, first, last, age) " +
    "VALUES(?, ?, ?, ?)";

// Create PreparedStatement object
PreparedStatement pstmt = conn.prepareStatement(SQL);

//Set auto-commit to false
conn.setAutoCommit(false);

// Set the variables
pstmt.setInt( 1, 400 );
pstmt.setString( 2, "Pappu" );
pstmt.setString( 3, "Singh" );
pstmt.setInt( 4, 33 );
// Add it to the batch
pstmt.addBatch();

// Set the variables
pstmt.setInt( 1, 401 );
pstmt.setString( 2, "Pawan" );
pstmt.setString( 3, "Singh" );
pstmt.setInt( 4, 31 );
// Add it to the batch
pstmt.addBatch();

//add more batches
.
.
.
.

//Create an int[] to hold returned values
int[] count = stmt.executeBatch();
//Explicitly commit statements to apply changes
conn.commit();
```

JDBC and MYSQL Example Program

Create a new database called *feedback* and start using it with the following command.

```
create database feedback;
```

```
use feedback;
```

Create a user with the following command.

```
CREATE USER sqluser IDENTIFIED BY 'sqluserpw';
```

```
grant usage on *.* to sqluser@localhost identified by 'sqluserpw';
```

grant all privileges on feedback.* to sqluser@localhost;

Now create a sample database table with example content via the following SQL statement.

```
CREATE TABLE comments (
    id INT NOT NULL AUTO_INCREMENT,
    MYUSER VARCHAR(30) NOT NULL,
    EMAIL VARCHAR(30),
    WEBPAGE VARCHAR(100) NOT NULL,
    DATUM DATE NOT NULL,
    SUMMARY VARCHAR(40) NOT NULL,
    COMMENTS VARCHAR(400) NOT NULL,
    PRIMARY KEY (ID)
);
```

```
INSERT INTO comments values (default, 'lars',
'myemail@gmail.com','https://www.vogella.com/', '2009-09-14 10:33:11', 'Summary','My first
comment' );
```

Java Source Code

```
package de.vogella.mysql.first.test;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Date;

public class MySQLAccess {
    private Connection connect = null;
    private Statement statement = null;
    private PreparedStatement preparedStatement = null;
    private ResultSet resultSet = null;

    public void readDataBase() throws Exception {
        try {
            // This will load the MySQL driver, each DB has its own driver
            Class.forName("com.mysql.jdbc.Driver");
            // Setup the connection with the DB
            connect = DriverManager.getConnection("jdbc:mysql://localhost/feedback?"
                + "user=sqluser&password=sqluserpw");

            // Statements allow to issue SQL queries to the database
            statement = connect.createStatement();
            // Result set get the result of the SQL query
            resultSet = statement
                .executeQuery("select * from feedback.comments");
            writeResultSet(resultSet);

            // PreparedStatements can use variables and are more efficient
```

```

        preparedStatement = connect
            .prepareStatement("insert into feedback.comments values (default, ?, ?, ?, ? , ?,
?");
        // "myuser, webpage, datum, summary, COMMENTS from feedback.comments");
        // Parameters start with 1
        preparedStatement.setString(1, "Test");
        preparedStatement.setString(2, "TestEmail");
        preparedStatement.setString(3, "TestWebpage");
        preparedStatement.setDate(4, new java.sql.Date(2009, 12, 11));
        preparedStatement.setString(5, "TestSummary");
        preparedStatement.setString(6, "TestComment");
        preparedStatement.executeUpdate();

        preparedStatement = connect.prepareStatement("SELECT myuser, webpage, datum,
summary, COMMENTS from feedback.comments");
        resultSet = preparedStatement.executeQuery();
        writeResultSet(resultSet);

        // Remove again the insert comment
        preparedStatement = connect
            .prepareStatement("delete from feedback.comments where myuser= ? ; ");
        preparedStatement.setString(1, "Test");
        preparedStatement.executeUpdate();

        resultSet = statement
            .executeQuery("select * from feedback.comments");
        writeMetaData(resultSet);

    } catch (Exception e) {
        throw e;
    } finally {
        close();
    }
}

private void writeMetaData(ResultSet resultSet) throws SQLException {
    // Now get some metadata from the database
    // Result set get the result of the SQL query

    System.out.println("The columns in the table are: ");

    System.out.println("Table: " + resultSet.getMetaData().getTableName(1));
    for (int i = 1; i <= resultSet.getMetaData().getColumnCount(); i++){
        System.out.println("Column " + i + " " + resultSet.getMetaData().getColumnName(i));
    }
}

private void writeResultSet(ResultSet resultSet) throws SQLException {
    // ResultSet is initially before the first data set

```

```

while (resultSet.next()) {
    // It is possible to get the columns via name
    // also possible to get the columns via the column number
    // which starts at 1
    // e.g. resultSet.getString(2);
    String user = resultSet.getString("myuser");
    String website = resultSet.getString("webpage");
    String summary = resultSet.getString("summary");
    Date date = resultSet.getDate("datum");
    String comment = resultSet.getString("comments");
    System.out.println("User: " + user);
    System.out.println("Website: " + website);
    System.out.println("summary: " + summary);
    System.out.println("Date: " + date);
    System.out.println("Comment: " + comment);
}
}

// You need to close the resultSet
private void close() {
    try {
        if (resultSet != null) {
            resultSet.close();
        }

        if (statement != null) {
            statement.close();
        }

        if (connect != null) {
            connect.close();
        }
    } catch (Exception e) {
    }
}
}

```

Create the following main program to test your class.

```
package de.vogella.mysql.first.test;
```

```
import de.vogella.mysql.first.MySQLAccess;
```

```

public class Main {
    public static void main(String[] args) throws Exception {
        MySQLAccess dao = new MySQLAccess();
        dao.readDataBase();
    }
}

```


1.11 Assignment-1**Very Short Answer Questions****Knowledge/Remembering Level Questions**

1. What is full form of JDBC?
2. Which manages a list of database drivers in JDBC?
3. How would you abbreviate JDBC-ODBC?
4. Which method is used to register the driver class in Java Database Connectivity?
5. The getConnection() method of which class is used to establish connection with the database?
6. Which method of Statement interface is used to execute queries to the database?
7. Can you list the method which is used to close the connection in database connection programs?
8. Can you recall the method of a class act as an interface between user and driver?
9. Which interface is a sub interface of Statement?
10. Mention the connection URL for the MySQL database.
11. Which is the session between java application and database?
12. List out the interface used to call the stored procedures and functions.
13. Which converts the Java data type to the appropriate JDBC type?
14. Which allows grouping related SQL statements into a batch?
15. Can you mention the method used to set the commit status and it is by default true.

Long Answer Questions**Understanding Level Questions**

1. Can you explain the types of JDBC drivers with diagram?
2. Explain Database connectivity steps with syntax and example.
3. Describe Different types of JDBC Drivers.
4. Explain Database connectivity steps using JDBC
5. Explain Batch Processing with its importance.

Application Level Questions

6. Demonstrate the CallableStatement using a program.
7. Interpret the prepared statements with an example.
8. Implement 'INSERT' stored procedure that receives id and name as the parameter and inserts it into the table.
9. Demonstrate different types of JDBC datatypes.

Skill Level Questions

10. Create a JDBC and MYSQL program to demonstrate basic database Transactions.

UNIT-II SERVLETS

2.1 Introduction to Servlets

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

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

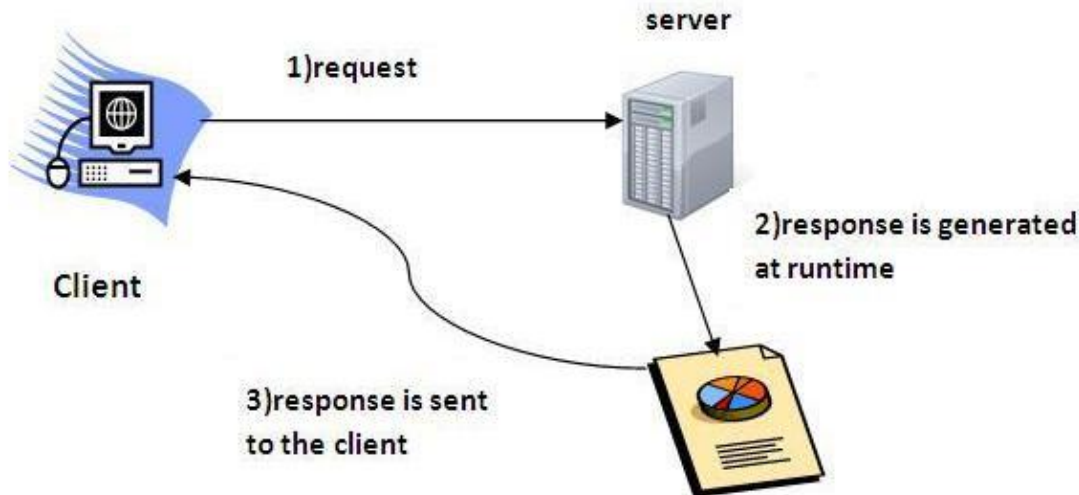


Figure 2.1: Servlet's basic components

Servlets perform the following major tasks:

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

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

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

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

Web Application

A web application is an application accessible from the web. A web application is composed of web components like Servlet, JSP, Filter etc. and other components such as HTML. The web components typically execute in Web Server and respond to HTTP request.

Common Gateway Interface (CGI)

CGI technology enables the web server to call an external program and pass HTTP request information to the external program to process the request. For each request, it starts a new process.

Disadvantages of CGI

There are many problems in CGI technology:

- If number of clients increases, it takes more time for sending response.
- For each request, it starts a process and Web server is limited to start processes.
- It uses platform dependent language e.g. C, C++, Perl.

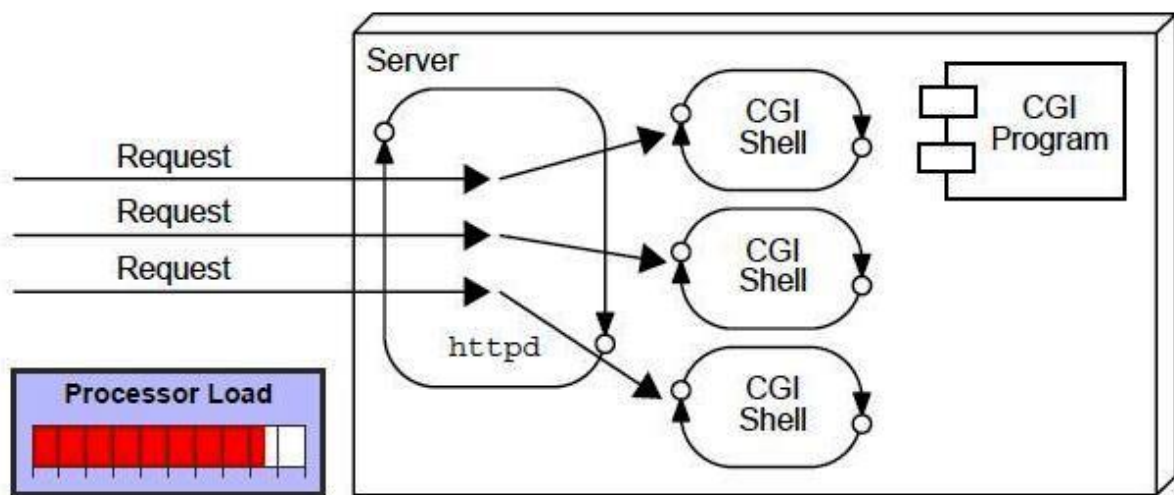


Figure 2.2: Common gateway Interface

Advantages of Servlets

Java Servlets often serve the same purpose as programs implemented using the Common Gateway Interface (CGI). Threads have a lot of benefits over the Processes such as they share a common memory area, lightweight, cost of communication between the threads are low. But Servlets offer several advantages in comparison with the CGI.

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

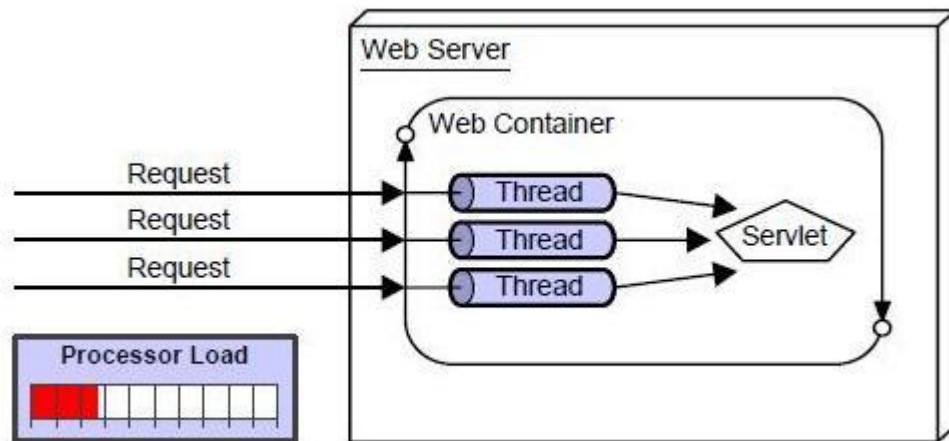


Figure 2.3: Servlet Web Container

2.2 Life Cycle of a Servlet

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

The servlet is initialized by calling the `init()` method.

The servlet calls `service()` method to process a client's request.

The servlet is terminated by calling the `destroy()` method.

Finally, servlet is garbage collected by the garbage collector of the JVM.

Now let us discuss the life cycle methods in details.

The `init()` method :

The `init` method is designed to be called only once. It is called when the servlet is first created, and not called again for each user request. So, it is used for one-time initializations, just as with the `init` method of applets. The servlet is normally created when a user first invokes a URL corresponding to the servlet, but you can also specify that the servlet be loaded when the server is first started. When a user invokes a servlet, a single instance of each servlet gets created, with each user request resulting in a new thread that is handed off to `doGet` or `doPost` as appropriate. The `init()` method simply creates or loads some data that will be used throughout the life of the servlet. The `init` method definition looks like this:

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

The `service()` method:

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

Each time the server receives a request for a servlet, the server spawns a new thread and calls `service`. The `service()` method checks the HTTP request type (GET, POST, PUT, DELETE, etc.) and calls `doGet`, `doPost`, `doPut`, `doDelete`, etc. methods as appropriate.

Here is the signature of this method:

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

```
}
```

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

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

The doGet() Method

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

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

The doPost() Method

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

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

The destroy() method:

The destroy() method is called only once at the end of the life cycle of a servlet. This method gives your servlet a chance to close database connections, halt background threads, write cookie lists or hit counts to disk, and perform other such cleanup activities.

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

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

Architecture Diagram:

The following figure depicts a typical servlet life-cycle scenario.

First the HTTP requests coming to the server are delegated to the servlet container.

The servlet container loads the servlet before invoking the service() method.

Then the servlet container handles multiple requests by spawning multiple threads, each thread executing the service() method of a single instance of the servlet.

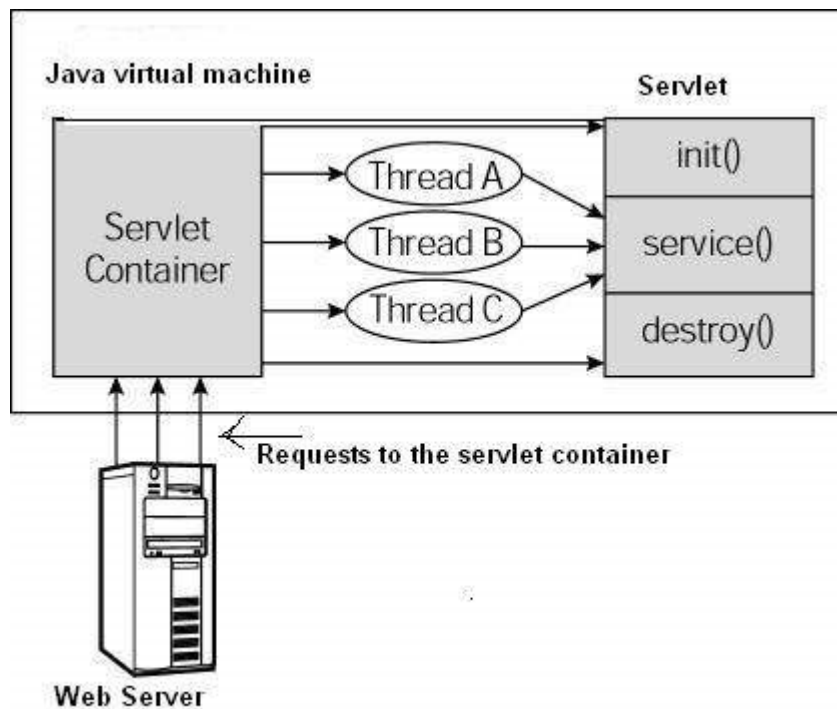


Figure 2.4: Servlet Life cycle

2.3 Using Tomcat for Servlet Development

To create a Servlet application you need to follow the below mentioned steps. These steps are common for all the Web server. In our example we are using Apache Tomcat server. Apache Tomcat is an open source web server for testing servlets and JSP technology. Download latest version of Tomcat Server and install it on your machine.

After installing Tomcat Server on your machine follow the below mentioned steps:

1. Create directory structure for your application.
2. Create a Servlet
3. Compile the Servlet
4. Create Deployment Descriptor for your application
5. Start the server and deploy the application

All these 5 steps are explained in details below; let's create our first Servlet Application.

Create directory structure for your application.

Sun Microsystem defines a unique directory structure that must be followed to create a servlet application.

Create the above directory structure inside Apache-Tomcat\webapps directory. All HTML, static files(images, css etc) are kept directly under Web application folder. While all the Servlet classes are kept inside `classes` folder.

The `web.xml` (deployment descriptor) file is kept under `WEB-INF` folder.

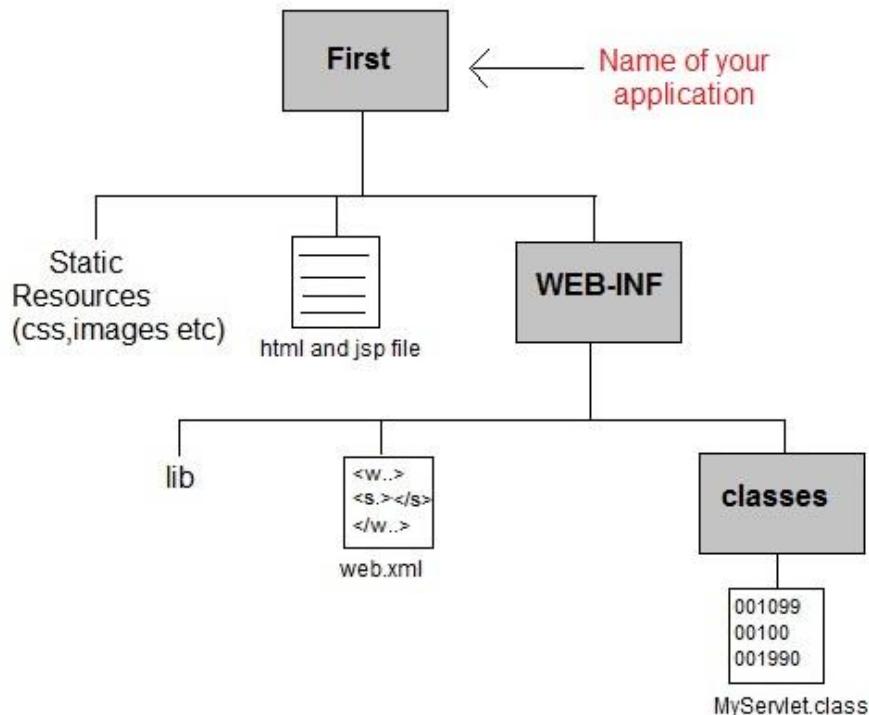


Figure 2.5: Servlet directory structure

Creating Servlet

There are three different ways to create a servlet.

- By implementing Servlet interface
- By extending GenericServlet class
- By extending HttpServlet class

But mostly a servlet is created by extending HttpServlet abstract class. As discussed earlier HttpServlet gives the definition of service() method of the Servlet interface. The servlet class that we will create should not override service() method. Our servlet class will override only doGet() or doPost() method.

When a request comes in for the servlet, the Web Container calls the servlet's service() method and depending on the type of request the service() method calls either the doGet() or doPost() method.

Note: By default a request is Get request.

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

```
public MyServlet extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><body>");
        out.println("<h1>Hello Readers</h1>");
        out.println("</body></html>");
    }
}
```



```
}
}
```

Write above code in a notepad file and save it as MyServlet.java anywhere on your PC. Compile it(explained in next step) from there and paste the class file into WEB-INF/classes/ directory that you have to create inside Tomcat/webapps directory.

Compiling a Servlet

To compile a Servlet a JAR file is required. Different servers require different JAR files. In Apache Tomcat server servlet-api.jar file is required to compile a servlet class.

Steps to compile a Servlet

- Set the Class Path.

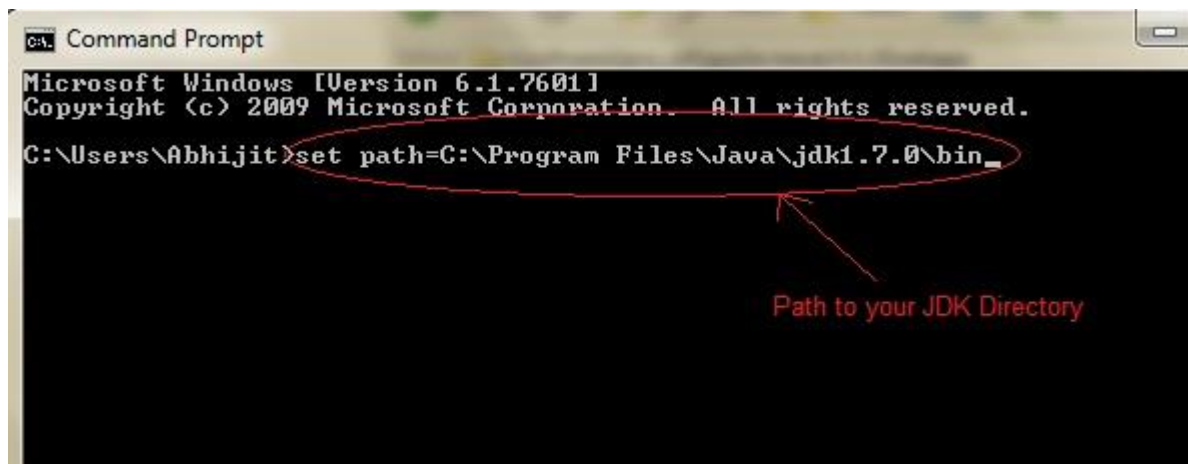


Figure 2.6: Setting Class Path

- Download **servlet-api.jar** file.
- Paste the servlet-api.jar file inside Java\jdk\jre\lib\ext directory.

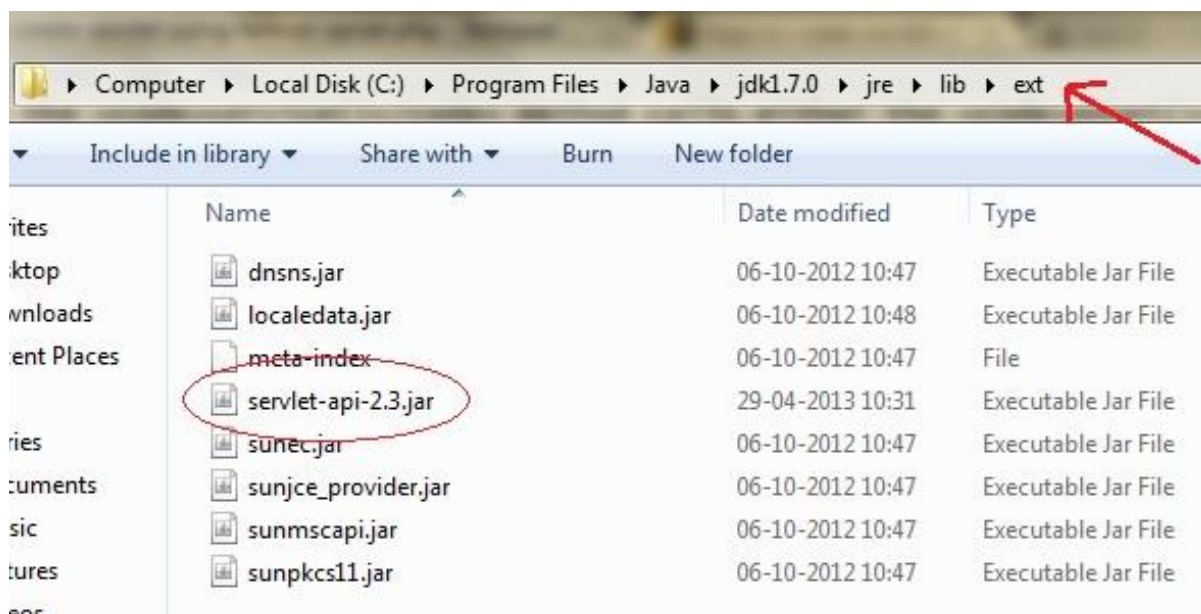


Figure 2.7: Adding Jar Files to Java Directory

- Compile the Servlet class.

NOTE: After compiling your Servlet class you will have to paste the class file into WEB-INF/classes/ directory.

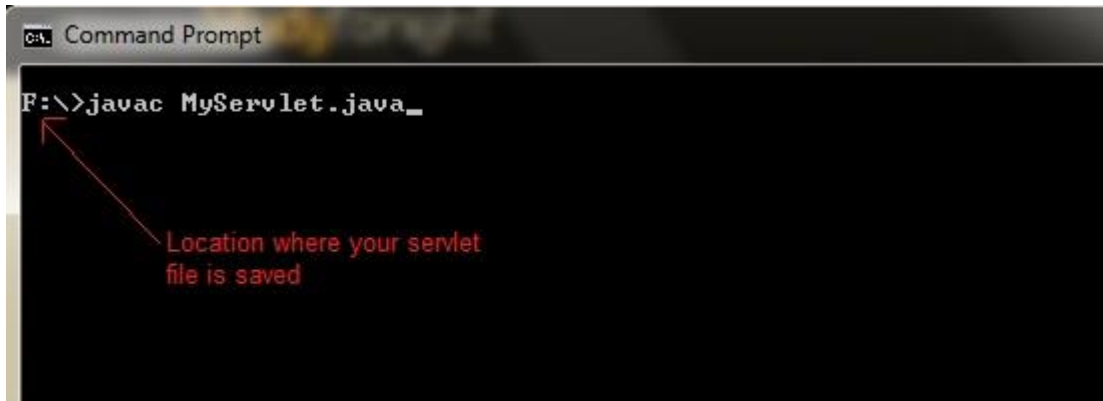


Figure 2.8: Compiling Servlet

Creating Deployment Descriptor

Deployment Descriptor(DD) is an XML document that is used by Web Container to run Servlets and JSP pages. DD is used for several important purposes such as:

- Mapping URL to Servlet class.
- Initializing parameters.
- Defining Error page.
- Security roles.
- Declaring tag libraries.

We will discuss about all these in details later. Now we will see how to create a simple web.xml file for our web application.

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
  <servlet>
    <servlet-name>hello</servlet-name>
    <servlet-class>MyServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>hello</servlet-name>
    <url-pattern>/hello</url-pattern>
  </servlet-mapping>
</web-app>

```

Annotations in the diagram:

- First line of any xml document (points to the XML declaration line).
- root tag of web.xml file. All other tag come inside it (points to the <web-app> tag).
- this tag maps internal name to fully qualified class name (points to the <servlet-class> tag).
- Give a internal name to your servlet (points to the <servlet-name> tag).
- this tag maps internal name to public URL name (points to the <url-pattern> tag).
- servlet class that you have created (points to the <servlet-class> tag).
- URL name. This is what the user will see to get to the servlet. (points to the <url-pattern> tag).

Figure 2.9 Web-XML file

Starting Tomcat Server for the first time

If you are starting Tomcat Server for the first time you need to set JAVA_HOME in the Enviroment variable. The following steps will show you how to set it.

- Right Click on My Computer, go to Properites
- Go to Advanced Tab and Click on Enviroment Variables... button.
- Click on New button, and enter JAVA_HOME inside Variable name text field and path of JDK inside Variable value text field. Click OK to save.

Run Servlet Application

Open Browser and type **http:localhost:8080/First/hello**



Figure 2.10: Running Servlet Application in the browser

16.4 The Servlet API

Servlet API consists of two important packages that encapsulate all the important classes and interface, namely:

javax.servlet
javax.servlet.http

Some Important Classes and Interfaces of javax.servlet

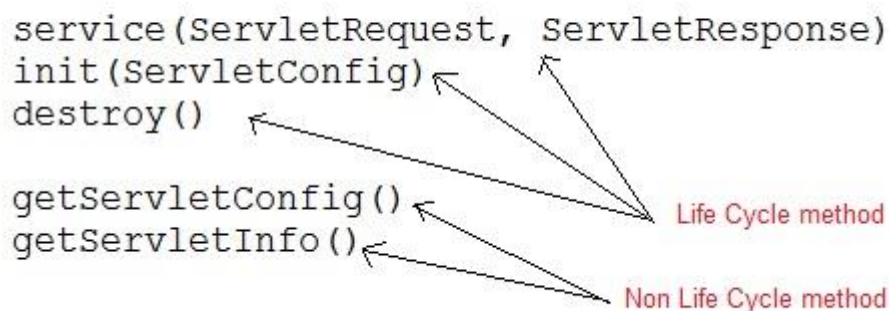
INTERFACES	CLASSES
Servlet	ServletInputStream
ServletContext	ServletOutputStream
ServletConfig	ServletRequestWrapper
ServletRequest	ServletResponseWrapper
ServletResponse	ServletRequestEvent
ServletContextListener	ServletContextEvent
RequestDispatcher	ServletRequestAttributeEvent
SingleThreadModel	ServletContextAttributeEvent
Filter	ServletException
FilterConfig	UnavailableException
FilterChain	GenericServlet
ServletRequestListener	

Some Important Classes and Interface of javax.servlet.http**CLASSES and INTERFACES**

HttpServlet	HttpServletRequest
HttpServletResponse	HttpSessionAttributeListener
HttpSession	HttpSessionListener
Cookie	HttpSessionEvent

Servlet Interface

Servlet Interface provides five methods. Out of these five methods, three methods are Servlet life cycle methods and rest two are non life cycle methods.

**2.5 Handling HTTP request and Response**

When a browser requests for a web page, it sends lot of information to the web server which cannot be read directly because this information travel as a part of header of HTTP request. You can check HTTP Protocol for more information on this.

Following is the important header information which comes from browser side and you would use very frequently in web programming:

Header	Description
Accept	This header specifies the MIME types that the browser or other clients can handle. Values of image/png or image/jpeg are the two most common possibilities.
Accept-Charset	This header specifies the character sets the browser can use to display the information. For example ISO-8859-1.
Accept-Encoding	This header specifies the types of encodings that the browser knows how to handle. Values of gzip or compress are the two most common possibilities.
Accept-Language	This header specifies the client's preferred languages in case the servlet can produce results in more than one language. For example en, en-us, ru, etc.
Authorization	This header is used by clients to identify themselves when accessing password-protected Web pages.
Connection	This header indicates whether the client can handle persistent HTTP connections. Persistent connections permit the client or

	other browser to retrieve multiple files with a single request. A value of Keep-Alive means that persistent connections should be used
Content-Length	This header is applicable only to POST requests and gives the size of the POST data in bytes.
Cookie	This header returns cookies to servers that previously sent them to the browser.
Host	This header specifies the host and port as given in the original URL.
If-Modified-Since	This header indicates that the client wants the page only if it has been changed after the specified date. The server sends a code, 304 which means Not Modified header if no newer result is available.
If-Unmodified-Since	This header is the reverse of If-Modified-Since; it specifies that the operation should succeed only if the document is older than the specified date.
Referer	This header indicates the URL of the referring Web page. For example, if you are at Web page 1 and click on a link to Web page 2, the URL of Web page 1 is included in the Referer header when the browser requests Web page 2.
User-Agent	This header identifies the browser or other client making the request and can be used to return different content to different types of browsers.

Methods to read HTTP Header:

There are following methods which can be used to read HTTP header in your servlet program. These methods are available with *HttpServletRequest* object.

S.N. Method & Description

- 1 `Cookie[] getCookies()`
Returns an array containing all of the Cookie objects the client sent with this request.
- 2 `Enumeration getAttributeNames()`
Returns an Enumeration containing the names of the attributes available to this request.
- 3 `Enumeration getHeaderNames()`
Returns an enumeration of all the header names this request contains.
- 4 `Enumeration getParameterNames()`
Returns an Enumeration of String objects containing the names of the parameters contained in this request.
- 5 `HttpSession getSession()`
Returns the current session associated with this request, or if the request does not have a session, creates one.
- 6 `HttpSession getSession(boolean create)`
Returns the current HttpSession associated with this request or, if there is no current session and create is true, returns a new session.

- Locale getLocale()
7 Returns the preferred Locale that the client will accept content in, based on the Accept-Language header.
- Object getAttribute(String name)
8 Returns the value of the named attribute as an Object, or null if no attribute of the given name exists.
- ServletInputStream getInputStream()
9 Retrieves the body of the request as binary data using a ServletInputStream.
- String getAuthType()
10 Returns the name of the authentication scheme used to protect the servlet, for example, "BASIC" or "SSL," or null if the JSP was not protected.
- String getCharacterEncoding()
11 Returns the name of the character encoding used in the body of this request.
- String getContentType()
12 Returns the MIME type of the body of the request, or null if the type is not known.
- String getContextPath()
13 Returns the portion of the request URI that indicates the context of the request.
- String getHeader(String name)
14 Returns the value of the specified request header as a String.
- String getMethod()
15 Returns the name of the HTTP method with which this request was made, for example, GET, POST, or PUT.
- String getParameter(String name)
16 Returns the value of a request parameter as a String, or null if the parameter does not exist.
- String getPathInfo()
17 Returns any extra path information associated with the URL the client sent when it made this request.
- String getProtocol()
18 Returns the name and version of the protocol the request.
- String getQueryString()
19 Returns the query string that is contained in the request URL after the path.
- String getRemoteAddr()
20 Returns the Internet Protocol (IP) address of the client that sent the request.
- String getRemoteHost()
21 Returns the fully qualified name of the client that sent the request.
- String getRemoteUser()
22 Returns the login of the user making this request, if the user has been authenticated, or null if the user has not been authenticated.
- String getRequestURI()
23 Returns the part of this request's URL from the protocol name up to the query string in the first line of the HTTP request.
- String getRequestedSessionId()
24 Returns the session ID specified by the client.
- String getServletPath()
25 Returns the part of this request's URL that calls the JSP.

- String[] getParameterValues(String name)
- 26 Returns an array of String objects containing all of the values the given request parameter has, or null if the parameter does not exist.
- boolean isSecure()
- 27 Returns a boolean indicating whether this request was made using a secure channel, such as HTTPS.
- int getContentLength()
- 28 Returns the length, in bytes, of the request body and made available by the input stream, or -1 if the length is not known.
- int getIntHeader(String name)
- 29 Returns the value of the specified request header as an int.
- int getServerPort()
- 30 Returns the port number on which this request was received.

HTTP Header Request Example:

Following is the example which uses **getHeaderNames()** method of `HttpServletRequest` to read the HTTP header information.

This method returns an Enumeration that contains the header information associated with the current HTTP request. Once we have an Enumeration, we can loop down the Enumeration in the standard manner, using *hasMoreElements()* method to determine when to stop and using *nextElement()* method to get each parameter name.

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

// Extend HttpServlet class
public class DisplayHeader extends HttpServlet {

    // Method to handle GET method request.
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        // Set response content type
        response.setContentType("text/html");

        PrintWriter out = response.getWriter();
        String title = "HTTP Header Request Example";

        String docType = "<!doctype html public \"-//w3c//dtd html 4.0 \" +
            \"transitional//en\">\n";
        out.println(docType +
            "<html>\n" +
            "<head><title>" + title + "</title></head>\n" +
            "<body bgcolor=\"#f0f0f0\">\n" +
            "<h1 align=\"center\">" + title + "</h1>\n" +
```

```

"<table width=\"100%\" border=\"1\" align=\"center\">\n" +
"<tr bgcolor=\"#949494\">\n" +
"<th>Header Name</th><th>Header Value(s)</th>\n" +
"</tr>\n");
Enumeration headerNames = request.getHeaderNames();
while(headerNames.hasMoreElements()) {
    String paramName = (String)headerNames.nextElement();
    out.print("<tr><td>" + paramName + "</td>\n");
    String paramValue = request.getHeader(paramName);
    out.println("<td> " + paramValue + "</td></tr>\n");
}

out.println("</table>\n</body></html>");
}
// Method to handle POST method request.
public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    doGet(request, response);
}
}

```

Server HTTP Response

When a Web server responds to a HTTP request to the browser, the response typically consists of a status line, some response headers, a blank line, and the document. A typical response looks like this:

```

HTTP/1.1 200 OK
Content-Type: text/html
Header2: ...
...
HeaderN: ...
(Blank Line)
<!doctype ...>
<html>
  <head>...</head>
  <body>
    ...
  </body>
</html>

```

The status line consists of the HTTP version (HTTP/1.1 in the example), a status code (200 in the example), and a very short message corresponding to the status code (OK in the example). Following is a summary of the most useful HTTP 1.1 response headers which go back to the browser from web server side and you would use them very frequently in web programming:

Header	Description
Allow	This header specifies the request methods (GET, POST, etc.) that the server supports.
Cache-Control	This header specifies the circumstances in which the response document can safely be cached. It can have values

	public, private or no-cache etc. Public means document is cacheable, Private means document is for a single user and can only be stored in private (nonshared) caches and no-cache means document should never be cached.
Connection	This header instructs the browser whether to use persistent in HTTP connections or not. A value of close instructs the browser not to use persistent HTTP connections and keep-alive means using persistent connections.
Content-Disposition	This header lets you request that the browser ask the user to save the response to disk in a file of the given name.
Content-Encoding	This header specifies the way in which the page was encoded during transmission.
Content-Language	This header signifies the language in which the document is written. For example en, en-us, ru, etc.
Content-Length	This header indicates the number of bytes in the response. This information is needed only if the browser is using a persistent (keep-alive) HTTP connection.
Content-Type	This header gives the MIME (Multipurpose Internet Mail Extension) type of the response document.
Expires	This header specifies the time at which the content should be considered out-of-date and thus no longer be cached.
Last-Modified	This header indicates when the document was last changed. The client can then cache the document and supply a date by an If-Modified-Since request header in later requests.
Location	This header should be included with all responses that have a status code in the 300s. This notifies the browser of the document address. The browser automatically reconnects to this location and retrieves the new document.
Refresh	This header specifies how soon the browser should ask for an updated page. You can specify time in number of seconds after which a page would be refreshed.
Retry-After	This header can be used in conjunction with a 503 (Service Unavailable) response to tell the client how soon it can repeat its request.
Set-Cookie	This header specifies a cookie associated with the page.

Methods to Set HTTP Response Header:

There are following methods which can be used to set HTTP response header in your servlet program. These methods are available with *HttpServletResponse* object.

S.N. Method & Description

- String `encodeRedirectURL(String url)`
Encodes the specified URL for use in the `sendRedirect` method or, if encoding is not needed, returns the URL unchanged.
- String `encodeURL(String url)`
Encodes the specified URL by including the session ID in it, or, if encoding is

- not needed, returns the URL unchanged.
- boolean containsHeader(String name)
- 3 Returns a boolean indicating whether the named response header has already been set.
- boolean isCommitted()
- 4 Returns a boolean indicating if the response has been committed.
- void addCookie(Cookie cookie)
- 5 Adds the specified cookie to the response.
- void addDateHeader(String name, long date)
- 6 Adds a response header with the given name and date-value.
- void addHeader(String name, String value)
- 7 Adds a response header with the given name and value.
- void addIntHeader(String name, int value)
- 8 Adds a response header with the given name and integer value.
- void flushBuffer()
- 9 Forces any content in the buffer to be written to the client.
- void reset()
- 10 Clears any data that exists in the buffer as well as the status code and headers.
- void resetBuffer()
- 11 Clears the content of the underlying buffer in the response without clearing headers or status code.
- void sendError(int sc)
- 12 Sends an error response to the client using the specified status code and clearing the buffer.
- void sendError(int sc, String msg)
- 13 Sends an error response to the client using the specified status.
- void sendRedirect(String location)
- 14 Sends a temporary redirect response to the client using the specified redirect location URL.
- void setBufferSize(int size)
- 15 Sets the preferred buffer size for the body of the response.
- void setCharacterEncoding(String charset)
- 16 Sets the character encoding (MIME charset) of the response being sent to the client, for example, to UTF-8.
- void setContentLength(int len)
- 17 Sets the length of the content body in the response In HTTP servlets, this method sets the HTTP Content-Length header.
- void setContentType(String type)
- 18 Sets the content type of the response being sent to the client, if the response has not been committed yet.
- void setDateHeader(String name, long date)
- 19 Sets a response header with the given name and date-value.
- void setHeader(String name, String value)
- 20 Sets a response header with the given name and value.
- void setIntHeader(String name, int value)
- 21 Sets a response header with the given name and integer value.

- 22 void setLocale(Locale loc)
Sets the locale of the response, if the response has not been committed yet.
- 23 void setStatus(int sc)
Sets the status code for this response.

HTTP Header Response Example:

You already have seen `setContentType()` method working in previous examples and following example would also use same method, additionally we would use **`setIntHeader()`** method to set Refresh header.

// Import required java libraries

import java.io.*;

import javax.servlet.*;

import javax.servlet.http.*;

import java.util.*;

// Extend HttpServlet class

public class Refresh extends HttpServlet {

// Method to handle GET method request.

public void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

// Set refresh, autoload time as 5 seconds

response.setIntHeader("Refresh", 5);

// Set response content type

response.setContentType("text/html");

// Get current time

Calendar calendar = new GregorianCalendar();

String am_pm;

int hour = calendar.get(Calendar.HOUR);

int minute = calendar.get(Calendar.MINUTE);

int second = calendar.get(Calendar.SECOND);

if(calendar.get(Calendar.AM_PM) == 0)

am_pm = "AM";

else

am_pm = "PM";

String CT = hour+":"+ minute + ":" + second + " " + am_pm;

PrintWriter out = response.getWriter();

String title = "Auto Refresh Header Setting";

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

out.println(docType +

<html>\n\" +

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

```
"<body bgcolor=\"#f0f0f0\">\n" +
"<h1 align=\"center\">" + title + "</h1>\n" +
"<p>Current Time is: " + CT + "</p>\n");
}

// Method to handle POST method request.
public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    doGet(request, response);
}
}
```

Now calling the above servlet would display current system time after every 5 seconds.

2.6 Using Cookies in Servlets

Cookies are text files stored on the client computer and they are kept for various information tracking purpose. Java Servlets transparently supports HTTP cookies.

There are three steps involved in identifying returning users:

- Server script sends a set of cookies to the browser. For example name, age, or identification number etc.
- Browser stores this information on local machine for future use.
- When next time browser sends any request to web server then it sends those cookies information to the server and server uses that information to identify the user.

The Anatomy of a Cookie:

Cookies are usually set in an HTTP header (although JavaScript can also set a cookie directly on a browser). A servlet that sets a cookie might send headers that look something like this:

```
HTTP/1.1 200 OK
Date: Fri, 04 Feb 2000 21:03:38 GMT
Server: Apache/1.3.9 (UNIX) PHP/4.0b3
Set-Cookie: name=xyz; expires=Friday, 04-Feb-07 22:03:38 GMT;
path=/; domain=tutorialspoint.com
Connection: close
Content-Type: text/html
```

As you can see, the Set-Cookie header contains a name value pair, a GMT date, a path and a domain. The name and value will be URL encoded. The expires field is an instruction to the browser to "forget" the cookie after the given time and date.

If the browser is configured to store cookies, it will then keep this information until the expiry date. If the user points the browser at any page that matches the path and domain of the cookie, it will resend the cookie to the server. The browser's headers might look something like this:

```
GET / HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.6 (X11; I; Linux 2.2.6-15apmac ppc)
Host: zink.demon.co.uk:1126
Accept: image/gif, */*
Accept-Encoding: gzip
```

Accept-Language: en
 Accept-Charset: iso-8859-1,*,utf-8
 Cookie: name=xyz

A servlet will then have access to the cookie through the request method `request.getCookies()` which returns an array of *Cookie* objects.

Servlet Cookies Methods:

Following is the list of useful methods which you can use while manipulating cookies in servlet.

S.N.	Method & Description
1	<code>public void setDomain(String pattern)</code> This method sets the domain to which cookie applies, for example <code>tutorialspoint.com</code> .
2	<code>public String getDomain()</code> This method gets the domain to which cookie applies, for example <code>tutorialspoint.com</code> .
3	<code>public void setMaxAge(int expiry)</code> This method sets how much time (in seconds) should elapse before the cookie expires. If you don't set this, the cookie will last only for the current session.
4	<code>public int getMaxAge()</code> This method returns the maximum age of the cookie, specified in seconds, By default, -1 indicating the cookie will persist until browser shutdown.
5	<code>public String getName()</code> This method returns the name of the cookie. The name cannot be changed after creation.
6	<code>public void setValue(String newValue)</code> This method sets the value associated with the cookie.
7	<code>public String getValue()</code> This method gets the value associated with the cookie.
8	<code>public void setPath(String uri)</code> This method sets the path to which this cookie applies. If you don't specify a path, the cookie is returned for all URLs in the same directory as the current page as well as all subdirectories.
9	<code>public String getPath()</code> This method gets the path to which this cookie applies.
10	<code>public void setSecure(boolean flag)</code> This method sets the boolean value indicating whether the cookie should only be sent over encrypted (i.e. SSL) connections.
11	<code>public void setComment(String purpose)</code> This method specifies a comment that describes a cookie's purpose. The comment is useful if the browser presents the cookie to the user.

12	<pre>public String getComment()</pre> <p>This method returns the comment describing the purpose of this cookie, or null if the cookie has no comment.</p>
----	---

Setting Cookies with Servlet:

Setting cookies with servlet involves three steps:

Creating a Cookie object: You call the Cookie constructor with a cookie name and a cookie value, both of which are strings.

```
Cookie cookie = new Cookie("key","value");
```

Keep in mind, neither the name nor the value should contain white space or any of the following characters:

```
[ ] ( ) = , " / ? @ : ;
```

(2) Setting the maximum age: You use `setMaxAge` to specify how long (in seconds) the cookie should be valid. Following would set up a cookie for 24 hours.

```
cookie.setMaxAge(60*60*24);
```

(3) Sending the Cookie into the HTTP response headers: You use `response.addCookie` to add cookies in the HTTP response header as follows:

```
response.addCookie(cookie);
```

Example:

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

```
// Extend HttpServlet class
public class HelloForm extends HttpServlet {

    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException
    {
        // Create cookies for first and last names.
        Cookie firstName = new Cookie("first_name",
            request.getParameter("first_name"));
        Cookie lastName = new Cookie("last_name",
            request.getParameter("last_name"));

        // Set expiry date after 24 Hrs for both the cookies.
        firstName.setMaxAge(60*60*24);
        lastName.setMaxAge(60*60*24);

        // Add both the cookies in the response header.
        response.addCookie( firstName );
        response.addCookie( lastName );

        // Set response content type
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Setting Cookies Example";
```

```
String docType =
"!doctype html public "-//w3c//dtd html 4.0 " +
"transitional//en">\n";
out.println(docType +
"<html>\n" +
"<head><title>" + title + "</title></head>\n" +
"<body bgcolor=\"#f0f0f0\">\n" +
"<h1 align=\"center\">" + title + "</h1>\n" +
"<ul>\n" +
" <li><b>First Name</b>: "
+ request.getParameter("first_name") + "\n" +
" <li><b>Last Name</b>: "
+ request.getParameter("last_name") + "\n" +
"</ul>\n" +
"</body></html>");
}
}
```

Compile above servlet HelloForm and create appropriate entry in web.xml file and finally try following HTML page to call servlet.

```
<html>
<body>
<form action="HelloForm" method="GET">
First Name: <input type="text" name="first_name">
<br />
Last Name: <input type="text" name="last_name" />
<input type="submit" value="Submit" />
</form>
</body>
</html>
```

Keep above HTML content in a file Hello.htm and put it in <Tomcat-installation-directory>/webapps/ROOT directory. When you would access <http://localhost:8080/Hello.htm>,

2.7 Session Tracking

HTTP is a "stateless" protocol which means each time a client retrieves a Web page, the client opens a separate connection to the Web server and the server automatically does not keep any record of previous client request.

Still there are following three ways to maintain session between web client and web server:

A web server can assign a unique session ID as a cookie to each web client and for subsequent requests from the client they can be recognized using the received cookie. This may not be an effective way because many time browser does not support a cookie, so I would not recommend to use this procedure to maintain the sessions.

Hidden Form Fields:

A web server can send a hidden HTML form field along with a unique session ID as follows:

```
<input type="hidden" name="sessionid" value="12345">
```

This entry means that, when the form is submitted, the specified name and value are automatically included in the GET or POST data. Each time when web browser sends request back, then session_id value can be used to keep the track of different web browsers.

This could be an effective way of keeping track of the session but clicking on a regular (<A HREF...>) hypertext link does not result in a form submission, so hidden form fields also cannot support general session tracking.

URL Rewriting:

You can append some extra data on the end of each URL that identifies the session, and the server can associate that session identifier with data it has stored about that session.

For example, with `http://tutorialspoint.com/file.htm;sessionid=12345`, the session identifier is attached as `sessionid=12345` which can be accessed at the web server to identify the client.

URL rewriting is a better way to maintain sessions and works for the browsers when they don't support cookies but here drawback is that you would have generate every URL dynamically to assign a session ID though page is simple static HTML page.

The HttpSession Object:

Apart from the above mentioned three ways, servlet provides HttpSession Interface which provides a way to identify a user across more than one page request or visit to a Web site and to store information about that user.

The servlet container uses this interface to create a session between an HTTP client and an HTTP server. The session persists for a specified time period, across more than one connection or page request from the user.

You would get HttpSession object by calling the public method **getSession()** of HttpServletRequest, as below:

```
HttpSession session = request.getSession();
```

You need to call `request.getSession()` before you send any document content to the client.

Here is a summary of the important methods available through HttpSession object:

S.N. Method & Description

- | | |
|---|---|
| | <code>public Object getAttribute(String name)</code> |
| 1 | This method returns the object bound with the specified name in this session, or null if no object is bound under the name. |
| | <code>public Enumeration getAttributeNames()</code> |
| 2 | This method returns an Enumeration of String objects containing the names of all the objects bound to this session. |
| | <code>public long getCreationTime()</code> |
| 3 | This method returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT. |
| | <code>public String getId()</code> |
| 4 | This method returns a string containing the unique identifier assigned to this session. |
| | <code>public long getLastAccessedTime()</code> |
| 5 | This method returns the last time the client sent a request associated with this session, as the number of milliseconds since midnight January 1, 1970 GMT. |
| | <code>public int getMaxInactiveInterval()</code> |
| 6 | This method returns the maximum time interval, in seconds, that the servlet container will keep this session open between client accesses. |
| 7 | <code>public void invalidate()</code>
This method invalidates this session and unbinds any objects bound to it. |
| 8 | <code>public boolean isNew()</code>
This method returns true if the client does not yet know about the session or if |

the client chooses not to join the session.

```
public void removeAttribute(String name)
```

- 9 This method removes the object bound with the specified name from this session.

```
public void setAttribute(String name, Object value)
```

- 10 This method binds an object to this session, using the name specified.

```
public void setMaxInactiveInterval(int interval)
```

- 11 This method specifies the time, in seconds, between client requests before the servlet container will invalidate this session.

Session Tracking Example:

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

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

```
import java.io.*;
```

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

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

```
import java.util.*;
```

```
// Extend HttpServlet class
```

```
public class SessionTrack extends HttpServlet {
```

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

```
{
```

```
    // Create a session object if it is already not created.
```

```
    HttpSession session = request.getSession(true);
```

```
    // Get session creation time.
```

```
    Date createTime = new Date(session.getCreationTime());
```

```
    // Get last access time of this web page.
```

```
    Date lastAccessTime =
```

```
        new Date(session.getLastAccessedTime());
```

```
    String title = "Welcome Back to my website";
```

```
    Integer visitCount = new Integer(0);
```

```
    String visitCountKey = new String("visitCount");
```

```
    String userIDKey = new String("userID");
```

```
    String userID = new String("ABCD");
```

```
    // Check if this is new comer on your web page.
```

```
    if (session.isNew()){
```

```
        title = "Welcome to my website";
```

```
        session.setAttribute(userIDKey, userID);
```

```
    } else {
```

```
        visitCount = (Integer)session.getAttribute(visitCountKey);
```

```
        visitCount = visitCount + 1;
```

```

        userID = (String)session.getAttribute(userIDKey);
    }
    session.setAttribute(visitCountKey, visitCount);
    // Set response content type
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String docType =
    "<!doctype html public "-//w3c//dtd html 4.0 " +
    "transitional//en">\n";
    out.println(docType +
        "<html>\n" +
        "<head><title>" + title + "</title></head>\n" +
        "<body bgcolor=\"#f0f0f0\">\n" +
        "<h1 align=\"center\">" + title + "</h1>\n" +
        "<h2 align=\"center\">Session Infomation</h2>\n" +
        "<table border=\"1\" align=\"center\">\n" +
        "<tr bgcolor=\"#949494\">\n" +
        " <th>Session info</th><th>value</th></tr>\n" +
        "<tr>\n" +
        " <td>id</td>\n" +
        " <td>" + session.getId() + "</td></tr>\n" +
        "<tr>\n" +
        " <td>Creation Time</td>\n" +
        " <td>" + createTime +
        " </td></tr>\n" +
        "<tr>\n" +
        " <td>Time of Last Access</td>\n" +
        " <td>" + lastAccessTime +
        " </td></tr>\n" +
        "<tr>\n" +
        " <td>User ID</td>\n" +
        " <td>" + userID +
        " </td></tr>\n" +
        "<tr>\n" +
        " <td>Number of visits</td>\n" +
        " <td>" + visitCount + "</td></tr>\n" +
        "</table>\n" +
        "</body></html>");
    }
}

```

Compile above servlet SessionTrack and create appropriate entry in web.xml file. Now running is <http://localhost:8080/SessionTrack>.

2.8 Servlet Database

To start with basic concept, let us create a simple table and create few records in that table as follows:

Create Table

To create the Employees table in TEST database, use the following steps:

Step 1:

Open a Command Prompt and change to the installation directory as follows:

```
C:\>
```

```
C:\>cd Program Files\MySQL\bin
```

```
C:\Program Files\MySQL\bin>
```

Step 2:

Login to database as follows

```
C:\Program Files\MySQL\bin>mysql -u root -p
```

```
Enter password: *****
```

```
mysql>
```

Step 3:

Create the table Employee in TEST database as follows:

```
mysql> use TEST;
```

```
mysql> create table Employees
```

```
(
    id int not null,
    age int not null,
    first varchar (255),
    last varchar (255)
);
```

```
Query OK, 0 rows affected (0.08 sec)
```

```
mysql>
```

Create Data Records

Finally you create few records in Employee table as follows:

```
mysql> INSERT INTO Employees VALUES (100, 18, 'Zara', 'Ali');
```

```
Query OK, 1 row affected (0.05 sec)
```

```
mysql> INSERT INTO Employees VALUES (101, 25, 'Mahnaz', 'Fatma');
```

```
Query OK, 1 row affected (0.00 sec)
```

```
mysql> INSERT INTO Employees VALUES (102, 30, 'Zaid', 'Khan');
```

```
Query OK, 1 row affected (0.00 sec)
```

```
mysql> INSERT INTO Employees VALUES (103, 28, 'Sumit', 'Mittal');
```

```
Query OK, 1 row affected (0.00 sec)
```

```
mysql>
```

Accessing a Database:

Here is an example which shows how to access TEST database using Servlet.

```
// Loading required libraries
```

```
import java.io.*;
```

```
import java.util.*;
```

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

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

```
import java.sql.*;
```

```
public class DatabaseAccess extends HttpServlet{
```

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

```

        throws ServletException, IOException
    {
        // JDBC driver name and database URL
        static final String JDBC_DRIVER="com.mysql.jdbc.Driver";
        static final String DB_URL="jdbc:mysql://localhost/TEST";

        // Database credentials
        static final String USER = "root";
        static final String PASS = "password";

        // Set response content type
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Database Result";
        String docType =
            "<!doctype html public \"-//w3c//dtd html 4.0 \" +
            \"transitional//en\">\n";
        out.println(docType +
            "<html>\n" +
            "<head><title>" + title + "</title></head>\n" +
            "<body bgcolor=\"#f0f0f0\">\n" +
            "<h1 align=\"center\">" + title + "</h1>\n");
        try{
            // Register JDBC driver
            Class.forName("com.mysql.jdbc.Driver");

            // Open a connection
            Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);

            // Execute SQL query
            Statement stmt = conn.createStatement();
            String sql;
            sql = "SELECT id, first, last, age FROM Employees";
            ResultSet rs = stmt.executeQuery(sql);

            // Extract data from result set
            while(rs.next()){
                //Retrieve by column name
                int id  = rs.getInt("id");
                int age = rs.getInt("age");
                String first = rs.getString("first");
                String last = rs.getString("last");

                //Display values
                out.println("ID: " + id + "<br>");
                out.println(", Age: " + age + "<br>");
                out.println(", First: " + first + "<br>");
                out.println(", Last: " + last + "<br>");
            }
        }
    }

```

```
out.println("</body></html>");

// Clean-up environment
rs.close();
stmt.close();
conn.close();
}catch(SQLException se){
    //Handle errors for JDBC
    se.printStackTrace();
}catch(Exception e){
    //Handle errors for Class.forName
    e.printStackTrace();
}finally{
    //finally block used to close resources
    try{
        if(stmt!=null)
            stmt.close();
    }catch(SQLException se2){
    }// nothing we can do
    try{
        if(conn!=null)
            conn.close();
    }catch(SQLException se){
        se.printStackTrace();
    }//end finally try
    } //end try
}
```

Now let us compile above servlet and create following entries in web.xml

```
....
<servlet>
    <servlet-name>DatabaseAccess</servlet-name>
    <servlet-class>DatabaseAccess</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>DatabaseAccess</servlet-name>
    <url-pattern>/DatabaseAccess</url-pattern>
</servlet-mapping>
....
```

Now call this servlet using URL <http://localhost:8080/DatabaseAccess> which would display following response:

Database Result

ID: 100, Age: 18, First: Zara, Last: Ali
ID: 101, Age: 25, First: Mahnaz, Last: Fatma
ID: 102, Age: 30, First: Zaid, Last: Khan
ID: 103, Age: 28, First: Sumit, Last: Mittal

2.9 Assignment Questions

Short Answer Questions

Knowledge/Remembering Level Questions

1. List out the two package used for servlet creation.
2. Which type of java programs run on a Web or Application server?
3. What is the method used by the servlet to initialize something?
4. Which method is used by the servlet to terminate its function?
5. Which method is the main method to perform the actual task in servlets?
6. What are the two most frequently used methods within each service request.
7. Which is an open source web server for testing servlets and JSP technology?
8. Can you recall a file required to compile a Servlet?
9. Which is an XML document that is used by Web Container to run Servlets?
10. To start Tomcat Server for the first time what you need to set in the Environment variable?
11. Which method returns an array containing all of the Cookie objects the client sent along with its request?
12. Which method returns an Enumeration containing the names of the attributes available to this request?
13. Which header instructs the browser whether to use persistent in HTTP connections or not.
14. By which method we can come to know how long (in seconds) the cookie should be valid.
15. Which is used to append some extra data on the end of each URL that identifies the session
16. Which Interface provides a way to identify a user across more than one page request or visit to a Web site.
17. What method is used to return the time when this session was created.
18. What are the two important packages that encapsulate all the important classes and interface?

Long Answer Questions

Understanding Level Questions

1. Can you explain the advantages of servlets?
2. Discuss the Life cycle of Servlet with an example.
3. Explain the Servlet API.
4. Explain the usage of Cookies with an example.

Application Level Questions

5. Demonstrate how tomcat is used for servlet development.
6. How would you apply the concept of handling HTTP request and Response using your own example?
7. Implement database access using servlet?
8. Differentiate Session tracking and HTTP Session object
9. Implement HTTP client Request with an example.

Skill Level Question

10. Create a program for database access using servlet

UNIT-III

JAVA SERVER PAGES

3.1 Introduction

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

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

Java Server Pages (JSP) is a technology for developing Webpages that supports dynamic content. This helps developers insert java code in HTML pages by making use of special JSP tags, most of which start with `<%` and end with `%>`.

A Java Server Pages component is a type of Java servlet that is designed to fulfill the role of a user interface for a Java web application. Web developers write JSPs as text files that combine HTML or XHTML code, XML elements, and embedded JSP actions and commands.

Using JSP, you can collect input from users through Webpage forms, present records from a database or another source, and create Webpages dynamically.

JSP tags can be used for a variety of purposes, such as retrieving information from a database or registering user preferences, accessing JavaBeans components, passing control between pages, and sharing information between requests, pages etc.

Advantages of JSP over Servlet

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

1) Extension to Servlet

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

2) Easy to maintain

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

3) Fast Development: No need to recompile and redeploy

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

4) Less code than Servlet

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

3.2 Need and Importance of JSP

Java Server Pages often serve the same purpose as programs implemented using the Common Gateway Interface (CGI). But JSP offers several advantages in comparison with the CGI.

Performance is significantly better because JSP allows embedding Dynamic Elements in HTML Pages itself instead of having separate CGI files.

JSP are always compiled before they are processed by the server unlike CGI/Perl which requires the server to load an interpreter and the target script each time the page is requested.

Java Server Pages are built on top of the Java Servlets API, so like Servlets, JSP also has access to all the powerful Enterprise Java APIs, including JDBC, JNDI, EJB, JAXP, etc.

JSP pages can be used in combination with servlets that handle the business logic, the model supported by Java servlet template engines.

Finally, JSP is an integral part of Java EE, a complete platform for enterprise class applications. This means that JSP can play a part in the simplest applications to the most complex and demanding.

Advantages of JSP

Following table lists out the other advantages of using JSP over other technologies –

vs. Active Server Pages (ASP)

The advantages of JSP are twofold. First, the dynamic part is written in Java, not Visual Basic or other MS specific language, so it is more powerful and easier to use. Second, it is portable to other operating systems and non-Microsoft Web servers.

vs. Pure Servlets

It is more convenient to write (and to modify!) regular HTML than to have plenty of `println` statements that generate the HTML.

vs. Server-Side Includes (SSI)

SSI is really only intended for simple inclusions, not for "real" programs that use form data, make database connections, and the like.

vs. JavaScript

JavaScript can generate HTML dynamically on the client but can hardly interact with the web server to perform complex tasks like database access and image processing etc.

vs. Static HTML

Regular HTML, of course, cannot contain dynamic information.

3.3 The Lifecycle of a JSP Page

The JSP pages follow these phases:

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

As depicted in the Figure 3.1, JSP page is translated into Servlet by the help of JSP translator. The JSP translator is a part of the web server which is responsible for translating the JSP page into Servlet. After that, Servlet page is compiled by the compiler and gets converted into the class file. Moreover, all the processes that happen in Servlet are performed on JSP later like initialization, committing response to the browser and destroy.

A JSP life cycle is defined as the process from its creation till the destruction. This is similar to a servlet life cycle with an additional step which is required to compile a JSP into servlet.

Paths Followed By JSP

The following are the paths followed by a JSP –

- Compilation
- Initialization
- Execution
- Cleanup

The four major phases of a JSP life cycle are very similar to the Servlet Life Cycle. The four phases have been described below –

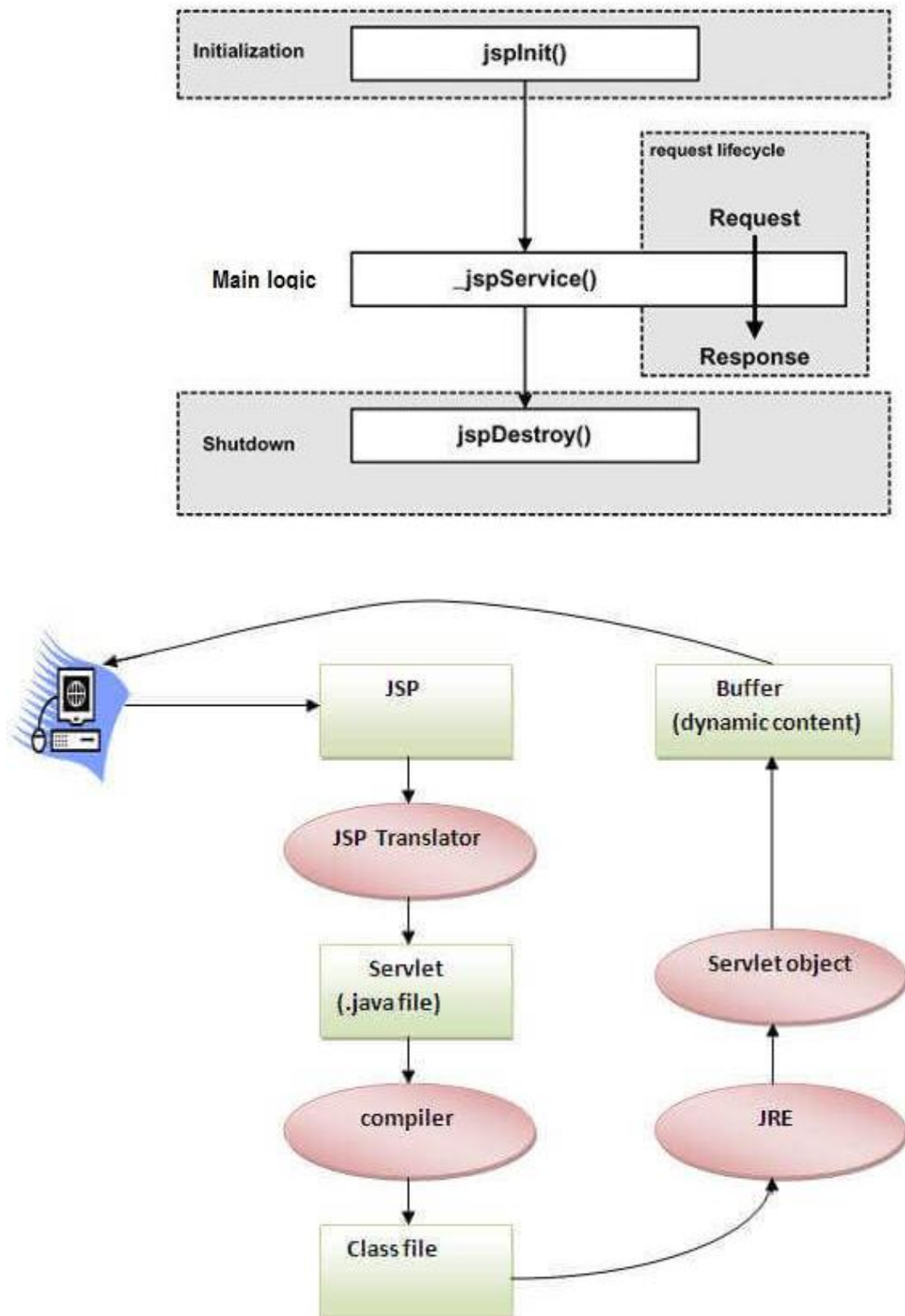


Figure 3.1: JSP Architecture

JSP Compilation

When a browser asks for a JSP, the JSP engine first checks to see whether it needs to compile the page. If the page has never been compiled, or if the JSP has been modified since it was last compiled, the JSP engine compiles the page.

The compilation process involves three steps –

- Parsing the JSP.

- Turning the JSP into a servlet.
- Compiling the servlet.

JSP Initialization

When a container loads a JSP it invokes the `jspInit()` method before servicing any requests. If you need to perform JSP-specific initialization, override the `jspInit()` method –

```
public void jspInit(){  
    // Initialization code...  
}
```

Typically, initialization is performed only once and as with the servlet `init` method, you generally initialize database connections, open files, and create lookup tables in the `jspInit` method.

JSP Execution

This phase of the JSP life cycle represents all interactions with requests until the JSP is destroyed.

Whenever a browser requests a JSP and the page has been loaded and initialized, the JSP engine invokes the `_jspService()` method in the JSP.

The `_jspService()` method takes an `HttpServletRequest` and an `HttpServletResponse` as its parameters as follows –

```
void _jspService(HttpServletRequest request, HttpServletResponse response) {  
    // Service handling code...  
}
```

The `_jspService()` method of a JSP is invoked on request basis. This is responsible for generating the response for that request and this method is also responsible for generating responses to all seven of the HTTP methods, i.e, GET, POST, DELETE, etc.

JSP Cleanup

The destruction phase of the JSP life cycle represents when a JSP is being removed from use by a container.

The `jspDestroy()` method is the JSP equivalent of the `destroy` method for servlets. Override `jspDestroy` when you need to perform any cleanup, such as releasing database connections or closing open files.

The `jspDestroy()` method has the following form –

```
public void jspDestroy() {  
    // Your cleanup code goes here.  
}
```

Creating a simple JSP Page

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

index.jsp

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

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

Follow the following steps to execute this JSP page:

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

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

The Directory structure of JSP

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

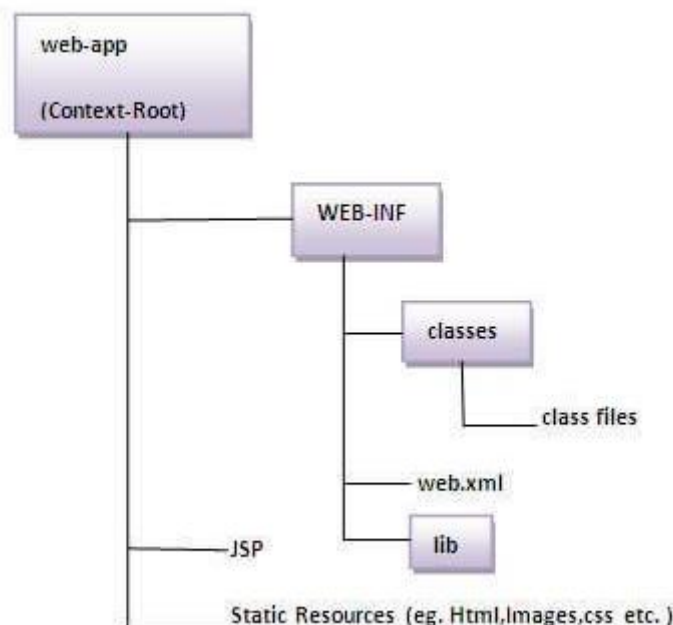


Figure 3.2: Director Structure of JSP

3.4 MVC Architecture in JSP

What is MVC?

MVC is an architecture that separates business logic, presentation and data. In MVC,

M stands for Model

V stands for View

C stands for controller.

MVC is a systematic way to use the application where the flow starts from the view layer, where the request is raised and processed in controller layer and sent to model layer to insert data and get back the success or failure message.

Model Layer:

This is the data layer which consists of the business logic of the system.

It consists of all the data of the application

It also represents the state of the application.

It consists of classes which have the connection to the database.

The controller connects with model and fetches the data and sends to the view layer.

The model connects with the database as well and stores the data into a database which is connected to it.

View Layer:

This is a presentation layer.

It consists of HTML, JSP, etc. into it.

It normally presents the UI of the application.

It is used to display the data which is fetched from the controller which in turn fetching data from model layer classes.

This view layer shows the data on UI of the application.

Controller Layer:

It acts as an interface between View and Model.

It intercepts all the requests which are coming from the view layer.

It receives the requests from the view layer and processes the requests and does the necessary validation for the request.

This request is further sent to model layer for data processing, and once the request is processed, it sends back to the controller with required information and displayed accordingly by the view.

The diagram is represented below:

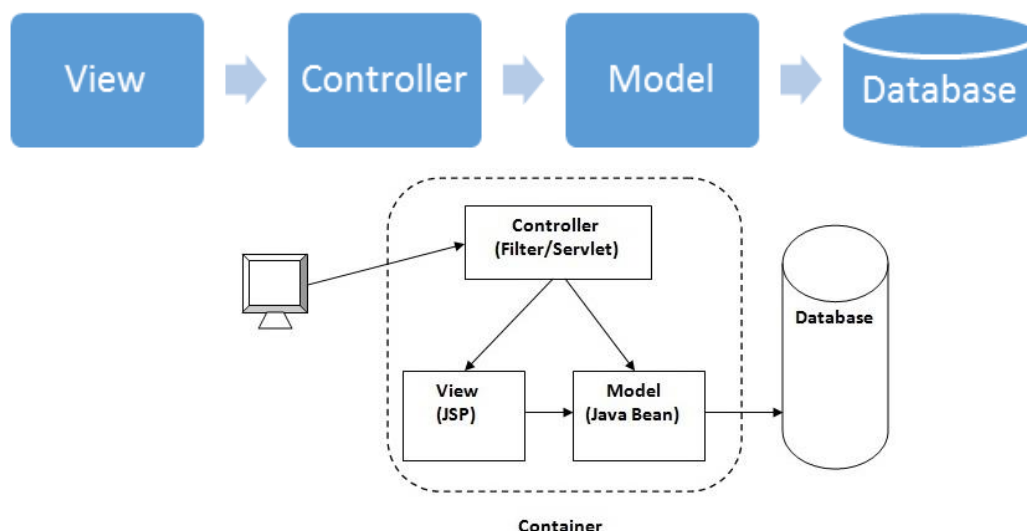


Figure 3.3: MVC Architecture of JSP

The advantages of MVC are:

Easy to maintain

Easy to extend

Easy to test

Navigation control is centralized

Example of MVC architecture

In this example, we are going to show how to use MVC architecture in JSP.

We are taking the example of a form with two variables "email" and "password" which is our view layer.

Once the user enters email, and password and clicks on submit then the action is passed in mvc_servlet where email and password are passed.

This mvc_servlet is controller layer. Here in mvc_servlet the request is sent to the bean object which act as model layer.

The email and password values are set into the bean and stored for further purpose.

From the bean, the value is fetched and shown in the view layer.

Mvc_example.jsp

```
<% @ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"% >
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>MVC Guru Example</title>
</head>
<body>
<form action="Mvc_servlet" method="POST">
Email: <input type="text" name="email">
<br />
Password: <input type="text" name="password" />
<input type="submit" value="Submit" />
</form>
</body>
</html>
```

Explanation of the code:

View Layer:

Code Line 10-15: Here we are taking a form which has two fields as parameter "email" and "password" and this request need to be forwarded to a controller Mvc_servlet.java, which is passed in action. The method through which it is passed is POST method.

Mvc_servlet.java

```
package demotest;
import java.io.IOException;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Servlet implementation class Mvc_servlet
 */
public class Mvc_servlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    /**
```

```

    * @see HttpServlet#HttpServlet()
    */
    public Mvc_servlet() {
        super();
        // TODO Auto-generated constructor stub
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        // TODO Auto-generated method stub
        String email=request.getParameter("email");
        String password=request.getParameter("password");

        TestBean testobj = new TestBean();
        testobj.setEmail(email);
        testobj.setPassword(password);
        request.setAttribute("gurubean",testobj);
        RequestDispatcher rd=request.getRequestDispatcher("mvc_success.jsp");
        rd.forward(request, response);
    }
}

```

Controller layer

Code Line 14: mvc_servlet is extending HttpServlet.

Code Line 26: As the method used is POST hence request comes into a doPost method of the servlet which process the requests and saves into the bean object as testobj.

Code Line 34: Using request object we are setting the attribute as gurubean which is assigned the value of testobj.

Code Line 35: Here we are using request dispatcher object to pass the success message to mvc_success.jsp

```

TestBean.java
package demotest;
import java.io.Serializable;
public class TestBean implements Serializable{
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    private String email="null";
    private String password="null";
}

```

Explanation of the code:

Model Layer:

Code Line 7-17: It contains the getters and setters of email and password which are members of Test Bean class

Code Line 19-20: It defines the members email and password of string type in the bean class. Mvc_success.jsp

```
<% @ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
    <% @page import="demotest.TestBean"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Guru Success</title>
</head>
<body>
<%
TestBean testguru=(TestBean)request.getAttribute("gurubean");
out.print("Welcome, "+testguru.getEmail());
%>
</body>
</html>
```

Explanation of the code:

Code Line 12: we are getting the attribute using request object which has been set in the doPost method of the servlet.

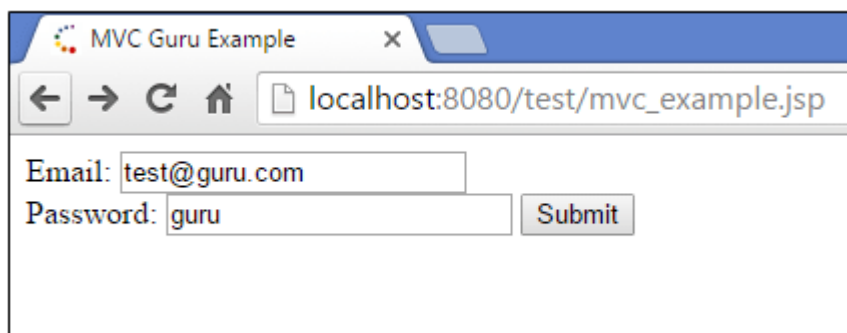
Code Line 13: We are printing the welcome message and email id of which have been saved in the bean object

Output:

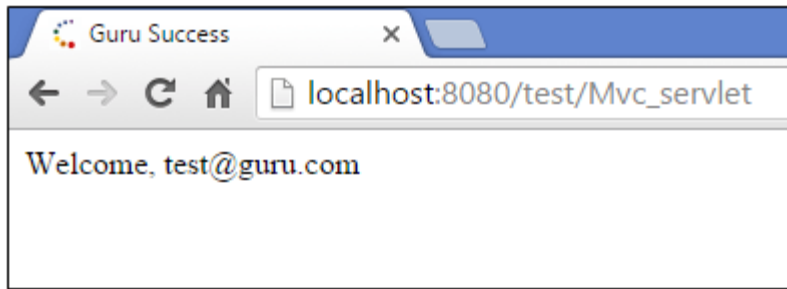
When you execute the above code, you get the following output:

When you click on mvc_example.jsp you get the form with email and password with the submit button.

Once you enter email and password to the form and then click on submit



After clicking on submit the output is shown as below

**Output:**

When you enter email and password in screen and click on submit then, the details are saved in TestBean and from the TestBean they are fetched on next screen to get the success message.

3.5 JSP Environments and Directives

A development environment is where you would develop your JSP programs, test them and finally run them.

This tutorial will guide you to setup your JSP development environment which involves the following steps –

Setting up Java Development Kit

This step involves downloading an implementation of the Java Software Development Kit (SDK) and setting up the PATH environment variable appropriately.

Once you download your Java implementation, follow the given instructions to install and configure the setup. Finally set the PATH and JAVA_HOME environment variables to refer to the directory that contains java and javac, typically java_install_dir/bin and java_install_dir respectively.

If you are running Windows and install the SDK in C:\jdk1.5.0_20, you need to add the following line in your C:\autoexec.bat file.

```
set PATH = C:\jdk1.5.0_20\bin;%PATH%
```

```
set JAVA_HOME = C:\jdk1.5.0_20
```

Alternatively, on Windows NT/2000/XP, you can also right-click on My Computer, select Properties, then Advanced, followed by Environment Variables. Then, you would update the PATH value and press the OK button.

On Unix (Solaris, Linux, etc.), if the SDK is installed in /usr/local/jdk1.5.0_20 and you use the C shell, you will put the following into your .cshrc file.

```
setenv PATH /usr/local/jdk1.5.0_20/bin:$PATH
```

```
setenv JAVA_HOME /usr/local/jdk1.5.0_20
```

Alternatively, if you use an Integrated Development Environment (IDE) like Borland JBuilder, Eclipse, IntelliJ IDEA, or Sun ONE Studio, compile and run a simple program to confirm that the IDE knows where you installed Java.

Setting up Web Server: Tomcat

A number of Web Servers that support Java Server Pages and Servlets development are available in the market. Some web servers can be downloaded for free and Tomcat is one of them.

Apache Tomcat is an open source software implementation of the Java Server Pages and Servlet technologies and can act as a standalone server for testing JSP and Servlets, and can be integrated with the Apache Web Server. Here are the steps to set up Tomcat on your machine –

Download the latest version of Tomcat from <https://tomcat.apache.org/>.

Once you downloaded the installation, unpack the binary distribution into a convenient location. For example, in C:\apache-tomcat-5.5.29 on windows, or /usr/local/apache-tomcat-5.5.29 on Linux/Unix and create CATALINA_HOME environment variable pointing to these locations.

Tomcat can be started by executing the following commands on the Windows machine –
%CATALINA_HOME%\bin\startup.bat

or

C:\apache-tomcat-5.5.29\bin\startup.bat

Tomcat can be started by executing the following commands on the Unix (Solaris, Linux, etc.) machine –

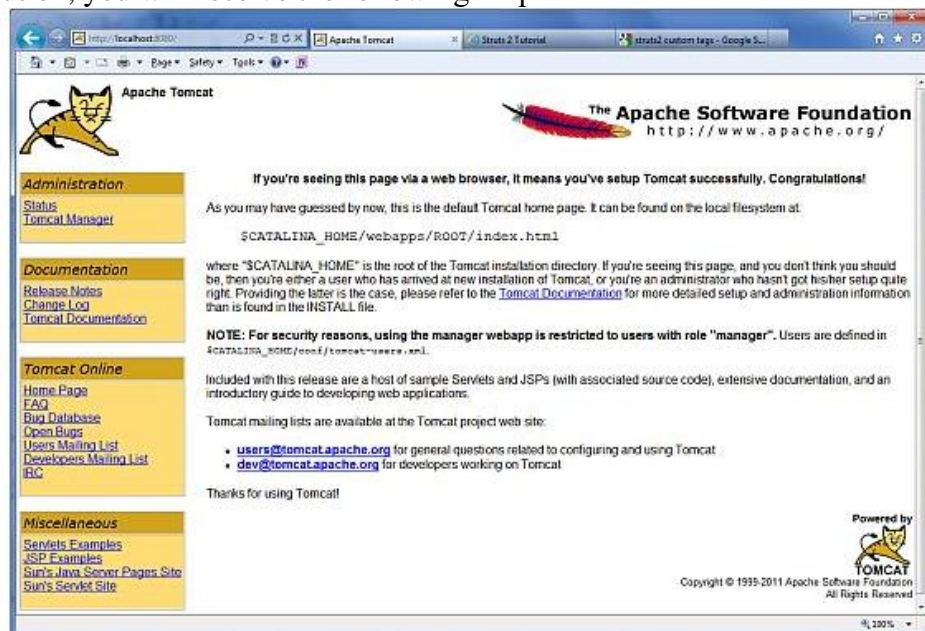
\$CATALINA_HOME/bin/startup.sh

or

/usr/local/apache-tomcat-5.5.29/bin/startup.sh

After a successful startup, the default web-applications included with Tomcat will be available by visiting <http://localhost:8080/>.

Upon execution, you will receive the following output –



Further information about configuring and running Tomcat can be found in the documentation included here, as well as on the Tomcat web site – <https://tomcat.apache.org/>.

Tomcat can be stopped by executing the following commands on the Windows machine –
%CATALINA_HOME%\bin\shutdown

or

C:\apache-tomcat-5.5.29\bin\shutdown

Tomcat can be stopped by executing the following commands on Unix (Solaris, Linux, etc.) machine –

\$CATALINA_HOME/bin/shutdown.sh

or

/usr/local/apache-tomcat-5.5.29/bin/shutdown.sh

Setting up CLASSPATH

Since servlets are not part of the Java Platform, Standard Edition, you must identify the servlet classes to the compiler.

If you are running Windows, you need to put the following lines in your C:\autoexec.bat file.
set CATALINA = C:\apache-tomcat-5.5.29

set CLASSPATH = %CATALINA%\common\lib\jsp-api.jar;%CLASSPATH%

Alternatively, on Windows NT/2000/XP, you can also right-click on My Computer, select Properties, then Advanced, then Environment Variables. Then, you would update the CLASSPATH value and press the OK button.

On Unix (Solaris, Linux, etc.), if you are using the C shell, you would put the following lines into your .cshrc file.

```
setenv CATALINA = /usr/local/apache-tomcat-5.5.29
```

```
setenv CLASSPATH $CATALINA/common/lib/jsp-api.jar:$CLASSPATH
```

NOTE – Assuming that your development directory is C:\JSPDev (Windows) or /usr/JSPDev (Unix), then you would need to add these directories as well in CLASSPATH.

JSP Directives

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 –

S.No.	Directive & Description
1	<pre><%@ page ... %></pre> Defines page-dependent attributes, such as scripting language, error page, and buffering requirements.
2	<pre><%@ include ... %></pre> Includes a file during the translation phase.
3	<pre><%@ taglib ... %></pre> Declares a tag library, containing custom actions, used in the page

JSP - The page Directive

The page directive is used to provide instructions to the container. These instructions 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 the page directive –

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

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

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

Attributes

Following table lists out the attributes associated with the page directive –

S.No.	Attribute & Purpose
1	Buffer Specifies a buffering model for the output stream.
2	autoFlush Controls the behavior of the servlet output buffer.

3	contentType Defines the character encoding scheme.
4	errorPage Defines the URL of another JSP that reports on Java unchecked runtime exceptions.
5	isErrorPage Indicates if this JSP page is a URL specified by another JSP page's errorPage attribute.
6	Extends Specifies a superclass that the generated servlet must extend.
7	Import Specifies a list of packages or classes for use in the JSP as the Java import statement does for Java classes.
8	Info Defines a string that can be accessed with the servlet's getServletInfo() method.
9	isThreadSafe Defines the threading model for the generated servlet.
10	Language Defines the programming language used in the JSP page.
11	Session Specifies whether or not the JSP page participates in HTTP sessions
12	isELIgnored Specifies whether or not the EL expression within the JSP page will be ignored.
13	isScriptingEnabled Determines if the scripting elements are allowed for use.

The include Directive

The include directive is used to include 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 the *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 the XML equivalent of the above syntax as follows –

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

Example

A good example of the include directive is including a common header and footer with multiple pages of content.

Let us define following three files (a) header.jsp, (b) footer.jsp, and (c) main.jsp as follows –

Following is the content of header.jsp –

```
<%!
    int pageCount = 0;
    void addCount() {
        pageCount++;
    }
%>
```

```
<% addCount(); %>
```

```
<html>
  <head>
    <title>The include Directive Example</title>
  </head>

  <body>
    <center>
      <h2>The include Directive Example</h2>
      <p>This site has been visited <%= pageCount %> times.</p>
    </center>
    <br/><br/>
```

Following is the content of footer.jsp –

```
<br/><br/>
  <center>
    <p>Copyright © 2010</p>
  </center>
</body>
</html>
```

Finally here is the content of main.jsp –

```
<% @ include file = "header.jsp" %>
<center>
  <p>Thanks for visiting my page.</p>
</center>
<% @ include file = "footer.jsp" %>
```

Let us now keep all these files in the root directory and try to access main.jsp. You will receive the following output –

The include Directive Example

This site has been visited 1 times.

Thanks for visiting my page.

Refresh main.jsp and you will find that the page hit counter keeps increasing.

You can design your webpages based on your creative instincts; it is recommended you keep the dynamic parts of your website in separate files and then include them in the main file. This makes it easy when you need to change a part of your webpage.

The taglib Directive

The JavaServer Pages API allow 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 means for identifying the custom tags in your JSP page.

The taglib directive follows the syntax given below –

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

Here, 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 the XML equivalent of the above syntax as follows –

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

Refresh main.jsp and you will find that the page hit counter keeps increasing.

You can design your webpages based on your creative instincts; it is recommended you keep the dynamic parts of your website in separate files and then include them in the main file. This makes it easy when you need to change a part of your webpage.

Example

For example, suppose the custlib tag library contains a tag called hello. If you wanted to use the hello tag with a prefix of mytag, your tag would be <mytag:hello> and it will be used in your JSP file as follows –

```
<%@ taglib uri = "http://www.example.com/custlib" prefix = "mytag" %>
```

```
<html>
  <body>
    <mytag:hello/>
  </body>
</html>
```

We can call another piece of code using <mytag:hello>.

3.6 JSP Actions and Implicit Objects

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 –

Action elements are basically predefined functions. The following table lists out the available JSP actions –

S. No.	Syntax & Purpose
1	jsp:include Includes a file at the time the page is requested.
2	jsp:useBean Finds or instantiates a JavaBean.
3	jsp:setProperty Sets the property of a JavaBean.
4	jsp:getProperty Inserts the property of a JavaBean into the output.
5	jsp:forward Forwards the requester to a new page.
6	jsp:plugin Generates browser-specific code that makes an OBJECT or EMBED tag for the Java plugin.
7	jsp:element Defines XML elements dynamically.
8	jsp:attribute Defines dynamically-defined XML element's attribute.
9	jsp:body Defines dynamically-defined XML element's body.
10	jsp:text Used to write template text in JSP pages and documents.

Example

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

Following is the content of the date.jsp file –

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 table lists out the attributes associated with the include action –

S.No.	Attribute & Description
1	page The relative URL of the page to be included.
2	flush The boolean attribute determines whether the included resource has its buffer flushed before it is included.

Example

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

Following is the content of the date.jsp file –

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

Following is the content of the 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>
```

Let us now keep all these files in the root directory and try to access main.jsp. You will receive the following output –

```
<html>
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 the bean properties.

Following table lists out the attributes associated with the useBean action –

.No.	Attribute & Description
1	class Designates the full package name of the bean.
2	type Specifies the type of the variable that will refer to the object.
3	beanName Gives the name of the bean as specified by the instantiate () method of the java.beans.Beans class.

Let us now discuss the jsp:setProperty and the 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 given 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 given 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 table lists out the attributes associated with the setProperty action –

S.No.	Attribute & Description
1	Name Designates the bean the property of which will be set. The Bean must have

	been previously defined.
2	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.
3	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.
4	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. The syntax of the getProperty action is as follows –

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

...

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

Following table lists out the required attributes associated with the getProperty action –

S.No.	Attribute & Description
1	name The name of the Bean that has a property to be retrieved. The Bean must have been previously defined.
2	property The property attribute is the name of the Bean property to be retrieved.

Example

Let us define a test bean that will further be used in our example –

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

```
package action;
```

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

Now use the following code in main.jsp file. This 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>
```

Let us now try to access main.jsp, it would display the 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.

Following is the syntax of the forward action –

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

Following table lists out the required attributes associated with the forward action –

S.No.	Attribute & Description
1	<p>page</p> <p>Should consist of a relative URL of another resource such as a static page, another JSP page, or a Java Servlet.</p>

Example

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

Following is the content of the date.jsp file –

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

Following is the content of the 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>

```

Let us now keep all these files in the root directory and try to access main.jsp. This would display result something like as below.

Here it discarded the content from the main page and displayed the 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 the 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>, <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:element>
  </body>
</html>

```

```

    </jsp:attribute>

    <jsp:body>
        Body for XML element
    </jsp:body>

    </jsp:element>
</body>
</html>

```

This would produce the 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 the template text in JSP pages and documents. Following is the simple syntax for this action –

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

The body of the template cannot contain other elements; it can only contain text and EL expressions (Note – EL expressions are explained in a 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 for XHTML, you must also use the <jsp:text> element as follows –

```
<jsp:text><![CDATA[<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"DTD/xhtml11-strict.dtd">]]></jsp:text>
```

```

<head><title>jsp:text action</title></head>

<body>
    <books><book><jsp:text>
        Welcome to JSP Programming
    </jsp:text></book></books>
</body>
</html>

```

JSP Implicit Object

These Objects are the Java objects that the JSP Container makes available to the developers in each page and the developer can call them directly without being explicitly declared. JSP Implicit Objects are also called pre-defined variables.

Following table lists out the nine Implicit Objects that JSP supports –

S.No.	Object & Description
1	Request This is the <code>HttpServletRequest</code> object associated with the request.
2	Response This is the <code>HttpServletResponse</code> object associated with the response to the client.
3	Out This is the <code>PrintWriter</code> object used to send output to the client.
4	Session This is the <code>HttpSession</code> object associated with the request.
5	Application This is the <code>ServletContext</code> object associated with the application context.
6	Config This is the <code>ServletConfig</code> object associated with the page.
7	pageContext This encapsulates use of server-specific features like higher performance <code>JspWriters</code> .
8	Page This is simply a synonym for this, and is used to call the methods defined by the translated servlet class.
9	Exception The <code>Exception</code> object allows the exception data to be accessed by designated JSP.

The request Object

The request object is an instance of a `javax.servlet.http.HttpServletRequest` object. Each time a client requests a page the JSP engine creates a new object to represent that request.

The request object provides methods to get the HTTP header information including form data, cookies, HTTP methods etc.

The response Object

The response object is an instance of a `javax.servlet.http.HttpServletResponse` object. Just as the server creates the request object, it also creates an object to represent the response to the client.

The response object also defines the interfaces that deal with creating new HTTP headers. Through this object the JSP programmer can add new cookies or date stamps, HTTP status codes, etc.

The out Object

The out implicit object is an instance of a `javax.servlet.jsp.JspWriter` object and is used to send content in a response.

The initial `JspWriter` object is instantiated differently depending on whether the page is buffered or not. Buffering can be easily turned off by using the `buffered = 'false'` attribute of the page directive.

The `JspWriter` object contains most of the same methods as the `java.io.PrintWriter` class. However, `JspWriter` has some additional methods designed to deal with buffering. Unlike the `PrintWriter` object, `JspWriter` throws `IOExceptions`.

Following table lists out the important methods that we will use to write boolean, char, int, double, object, String, etc.

S.No.	Method & Description
-------	----------------------

1	<code>out.print(dataType dt)</code> Print a data type value
2	<code>out.println(dataType dt)</code> Print a data type value then terminate the line with new line character.
3	<code>out.flush()</code> Flush the stream.

The session Object

The session object is an instance of `javax.servlet.http.HttpSession` and behaves exactly the same way that session objects behave under Java Servlets.

The session object is used to track client session between client requests.

The application Object

The application object is direct wrapper around the `ServletContext` object for the generated Servlet and in reality an instance of a `javax.servlet.ServletContext` object.

This object is a representation of the JSP page through its entire lifecycle. This object is created when the JSP page is initialized and will be removed when the JSP page is removed by the `jspDestroy()` method.

By adding an attribute to application, you can ensure that all JSP files that make up your web application have access to it.

The config Object

The config object is an instantiation of `javax.servlet.ServletConfig` and is a direct wrapper around the `ServletConfig` object for the generated servlet.

This object allows the JSP programmer access to the Servlet or JSP engine initialization parameters such as the paths or file locations etc.

The following config method is the only one you might ever use, and its usage is trivial – `config.getServletName()`;

This returns the servlet name, which is the string contained in the `<servlet-name>` element defined in the `WEB-INF/web.xml` file.

The pageContext Object

The `pageContext` object is an instance of a `javax.servlet.jsp.PageContext` object. The `pageContext` object is used to represent the entire JSP page.

This object is intended as a means to access information about the page while avoiding most of the implementation details.

This object stores references to the request and response objects for each request. The application, config, session, and out objects are derived by accessing attributes of this object.

The `pageContext` object also contains information about the directives issued to the JSP page, including the buffering information, the `errorPageURL`, and page scope.

The `PageContext` class defines several fields, including `PAGE_SCOPE`, `REQUEST_SCOPE`, `SESSION_SCOPE`, and `APPLICATION_SCOPE`, which identify the four scopes. It also supports more than 40 methods, about half of which are inherited from the `javax.servlet.jsp.JspContext` class.

One of the important methods is `removeAttribute`. This method accepts either one or two arguments. For example, `pageContext.removeAttribute("attrName")` removes the attribute from all scopes, while the following code only removes it from the page scope – `pageContext.removeAttribute("attrName", PAGE_SCOPE)`;

The page Object

This object is an actual reference to the instance of the page. It can be thought of as an object that represents the entire JSP page.

The page object is really a direct synonym for the this object.

The exception Object

The exception object is a wrapper containing the exception thrown from the previous page. It is typically used to generate an appropriate response to the error condition.

3.7 JSP Session and Cookies Handling

HTTP is a "stateless" protocol which means each time a client retrieves a Webpage, the client opens a separate connection to the Web server and the server automatically does not keep any record of previous client request.

Maintaining Session Between Web Client And Server

Let us now discuss a few options to maintain the session between the Web Client and the Web Server –

Cookies

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

This may not be an effective way as the browser at times does not support a cookie. It is not recommended to use this procedure to maintain the sessions.

Hidden Form Fields

A web server can send a hidden HTML form field along with a unique session ID as follows –
<input type = "hidden" name = "sessionid" value = "12345">

This entry means that, when the form is submitted, the specified name and value are automatically included in the GET or the POST data. Each time the web browser sends the request back, the session_id value can be used to keep the track of different web browsers.

This can be an effective way of keeping track of the session but clicking on a regular (<A HREF...>) hypertext link does not result in a form submission, so hidden form fields also cannot support general session tracking.

URL Rewriting

You can append some extra data at the end of each URL. This data identifies the session; the server can associate that session identifier with the data it has stored about that session.

For example, with <http://tutorialspoint.com/file.htm;sessionid=12345>, the session identifier is attached as sessionid = 12345 which can be accessed at the web server to identify the client.

URL rewriting is a better way to maintain sessions and works for the browsers when they don't support cookies. The drawback here is that you will have to generate every URL dynamically to assign a session ID though page is a simple static HTML page.

The session Object

Apart from the above mentioned options, JSP makes use of the servlet provided HttpSession Interface. This interface provides a way to identify a user across.

- a one page request or
- visit to a website or
- store information about that user

By default, JSPs have session tracking enabled and a new HttpSession object is instantiated for each new client automatically. Disabling session tracking requires explicitly turning it off by setting the page directive session attribute to false as follows <% @ page session = "false" %>

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

Here is a summary of important methods available through the session object –

.No.	Method & Description
1	<code>public Object getAttribute(String name)</code> This method returns the object bound with the specified name in this session, or null if no object is bound under the name.
2	<code>public Enumeration getAttributeNames()</code> This method returns an Enumeration of String objects containing the names of all the objects bound to this session.
3	<code>public long getCreationTime()</code> This method returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.
4	<code>public String getId()</code> This method returns a string containing the unique identifier assigned to this session.
5	<code>public long getLastAccessedTime()</code> This method returns the last time the client sent a request associated with the this session, as the number of milliseconds since midnight January 1, 1970 GMT.
6	<code>public int getMaxInactiveInterval()</code> This method returns the maximum time interval, in seconds, that the servlet container will keep this session open between client accesses.
7	<code>public void invalidate()</code> This method invalidates this session and unbinds any objects bound to it.
8	<code>public boolean isNew()</code> This method returns true if the client does not yet know about the session or if the client chooses not to join the session.
9	<code>public void removeAttribute(String name)</code> This method removes the object bound with the specified name from this session.
10	<code>public void setAttribute(String name, Object value)</code> This method binds an object to this session, using the name specified.
11	<code>public void setMaxInactiveInterval(int interval)</code> This method specifies the time, in seconds, between client requests before the servlet container will invalidate this session.

Session Tracking Example

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

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

```

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

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

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

// Check if this is new comer on your Webpage.
if (session.isNew() ){
    title = "Welcome to my website";
    session.setAttribute(userIDKey, userID);
    session.setAttribute(visitCountKey, visitCount);
}
visitCount = (Integer)session.getAttribute(visitCountKey);
visitCount = visitCount + 1;
userID = (String)session.getAttribute(userIDKey);
session.setAttribute(visitCountKey, visitCount);
%>
<html>
<head>
<title>Session Tracking</title>
</head>

<body>
<center>
<h1>Session Tracking</h1>
</center>

<table border = "1" align = "center">
<tr bgcolor = "#949494">
<th>Session info</th>
<th>Value</th>
</tr>
<tr>
<td>id</td>
<td><% out.print( session.getId()); %></td>
</tr>
<tr>
<td>Creation Time</td>
<td><% out.print(createTime); %></td>
</tr>
<tr>
<td>Time of Last Access</td>
<td><% out.print(lastAccessTime); %></td>
</tr>

```



```

<tr>
  <td>User ID</td>
  <td><% out.print(userID); %></td>
</tr>
<tr>
  <td>Number of visits</td>
  <td><% out.print(visitCount); %></td>
</tr>
</table>

```

```

</body>
</html>

```

Now put the above code in main.jsp and try to access <http://localhost:8080/main.jsp>. Once you run the URL, you will receive the following result –

Welcome to my website

Session Information

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

Now try to run the same JSP for the second time, you will receive the following result.

Welcome Back to my website

Session Information

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

Deleting Session Data

When you are done with a user's session data, you have several options –

Remove a particular attribute – You can call the *public void removeAttribute(String name)* method to delete the value associated with the a particular key.

Delete the whole session – You can call the *public void invalidate()* method to discard an entire session.

Setting Session timeout – You can call the *public void setMaxInactiveInterval(int interval)* method to set the timeout for a session individually.

Log the user out – The servers that support servlets 2.4, you can call *logout* to log the client out of the Web server and invalidate all sessions belonging to all the users.

web.xml Configuration – If you are using Tomcat, apart from the above mentioned methods, you can configure the session time out in web.xml file as follows.

```
<session-config>
  <session-timeout>15</session-timeout>
</session-config>
```

The timeout is expressed as minutes, and overrides the default timeout which is 30 minutes in Tomcat.

The *getMaxInactiveInterval()* method in a servlet returns the timeout period for that session in seconds. So if your session is configured in web.xml for 15 minutes, *getMaxInactiveInterval()* returns 900.

JSP Cookies Handling

Cookies are text files stored on the client computer and they are kept for various information tracking purposes. JSP transparently supports HTTP cookies using underlying servlet technology.

There are three steps involved in identifying and returning users –

Server script sends a set of cookies to the browser. For example, name, age, or identification number, etc.

Browser stores this information on the local machine for future use.

When the next time the browser sends any request to the web server then it sends those cookies information to the server and server uses that information to identify the user or may be for some other purpose as well.

This chapter will teach you how to set or reset cookies, how to access them and how to delete them using JSP programs.

The Anatomy of a Cookie

Cookies are usually set in an HTTP header (although JavaScript can also set a cookie directly on a browser). A JSP that sets a cookie might send headers that look something like this –

HTTP/1.1 200 OK

Date: Fri, 04 Feb 2000 21:03:38 GMT

Server: Apache/1.3.9 (UNIX) PHP/4.0b3

Set-Cookie: name = xyz; expires = Friday, 04-Feb-07 22:03:38 GMT;
path = /; domain = tutorialspoint.com

Connection: close

Content-Type: text/html

As you can see, the Set-Cookie header contains a name value pair, a GMT date, a path and a domain. The name and value will be URL encoded. The expires field is an instruction to the browser to "forget" the cookie after the given time and date.

If the browser is configured to store cookies, it will then keep this information until the expiry date. If the user points the browser at any page that matches the path and domain of the cookie, it will resend the cookie to the server. The browser's headers might look something like this –

GET / HTTP/1.0

Connection: Keep-Alive

User-Agent: Mozilla/4.6 (X11; I; Linux 2.2.6-15apmac ppc)

Host: zink.demon.co.uk:1126

Accept: image/gif, */*
 Accept-Encoding: gzip
 Accept-Language: en
 Accept-Charset: iso-8859-1,*,utf-8
 Cookie: name = xyz

A JSP script will then have access to the cookies through the request method *request.getCookies()* which returns an array of *Cookie* objects.

Servlet Cookies Methods

Following table lists out the useful methods associated with the *Cookie* object which you can use while manipulating cookies in JSP –

S.No.	Method & Description
1	public void setDomain(String pattern) This method sets the domain to which the cookie applies; for example, tutorialspoint.com.
2	public String getDomain() This method gets the domain to which the cookie applies; for example, tutorialspoint.com.
3	public void setMaxAge(int expiry) This method sets how much time (in seconds) should elapse before the cookie expires. If you don't set this, the cookie will last only for the current session.
4	public int getMaxAge() This method returns the maximum age of the cookie, specified in seconds, By default, -1 indicating the cookie will persist until the browser shutdown.
5	public String getName() This method returns the name of the cookie. The name cannot be changed after the creation.
6	public void setValue(String newValue) This method sets the value associated with the cookie.
7	public String getValue() This method gets the value associated with the cookie.
8	public void setPath(String uri) This method sets the path to which this cookie applies. If you don't specify a path, the cookie is returned for all URLs in the same directory as the current page as well as all subdirectories.
9	public String getPath() This method gets the path to which this cookie applies.
10	public void setSecure(boolean flag) This method sets the boolean value indicating whether the cookie should only be sent over encrypted (i.e, SSL) connections.

11	<pre>public void setComment(String purpose)</pre> <p>This method specifies a comment that describes a cookie's purpose. The comment is useful if the browser presents the cookie to the user.</p>
12	<pre>public String getComment()</pre> <p>This method returns the comment describing the purpose of this cookie, or null if the cookie has no comment.</p>

Setting Cookies with JSP

Setting cookies with JSP involves three steps –

Step 1: Creating a Cookie object

You call the Cookie constructor with a cookie name and a cookie value, both of which are strings.

```
Cookie cookie = new Cookie("key", "value");
```

Keep in mind, neither the name nor the value should contain white space or any of the following characters –

```
cookie.setMaxAge(60*60*24);
```

Step 3: Sending the Cookie into the HTTP response headers

You use response.addCookie to add cookies in the HTTP response header as follows

```
response.addCookie(cookie);
```

Example

set the cookies for the first and the last name.

```
<%
// Create cookies for first and last names.
Cookie firstName = new Cookie("first_name", request.getParameter("first_name"));
Cookie lastName = new Cookie("last_name", request.getParameter("last_name"));
```

```
// Set expiry date after 24 Hrs for both the cookies.
```

```
firstName.setMaxAge(60*60*24);
```

```
lastName.setMaxAge(60*60*24);
```

```
// Add both the cookies in the response header.
```

```
response.addCookie( firstName );
```

```
response.addCookie( lastName );
```

```
%>
```

```
<html>
```

```
<head>
```

```
<title>Setting Cookies</title>
```

```
</head>
```

```
<body>
```

```
<center>
```

```
<h1>Setting Cookies</h1>
```

```
</center>
```

```
<ul>
```

```
<li><p><b>First Name:</b>
```

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

```
</p></li>
```

```
<li><p><b>Last Name:</b>
```

```

        <%= request.getParameter("last_name")%>
    </p></li>
</ul>

```

```

</body>
</html>

```

Let us put the above code in main.jsp file and use it in the following HTML page

```

<html>
<body>

    <form action = "main.jsp" method = "GET">
        First Name: <input type = "text" name = "first_name">
        <br />
        Last Name: <input type = "text" name = "last_name" />
        <input type = "submit" value = "Submit" />
    </form>

</body>
</html>

```

Keep the above HTML content in a file hello.jsp and put hello.jsp and main.jsp in <Tomcat-installation-directory>/webapps/ROOT directory. When you will access <http://localhost:8080/hello.jsp>, here is the actual output of the above form.

First Name:

Last Name:

Try to enter the First Name and the Last Name and then click the submit button. This will display the first name and the last name on your screen and will also set two cookies firstName and lastName. These cookies will be passed back to the server when the next time you click the Submit button.

In the next section, we will explain how you can access these cookies back in your web application.

Reading Cookies with JSP

To read cookies, you need to create an array of *javax.servlet.http.Cookie* objects by calling the `getCookies()` method of *HttpServletRequest*. Then cycle through the array, and use `getName()` and `getValue()` methods to access each cookie and associated value.

Example

Let us now read cookies that were set in the previous example –

```

<html>
<head>
    <title>Reading Cookies</title>
</head>

<body>
    <center>
        <h1>Reading Cookies</h1>
    </center>
    <%
        Cookie cookie = null;
        Cookie[] cookies = null;

```

```
// Get an array of Cookies associated with the this domain
cookies = request.getCookies();

if( cookies != null ) {
    out.println("<h2> Found Cookies Name and Value</h2>");

    for (int i = 0; i < cookies.length; i++) {
        cookie = cookies[i];
        out.print("Name : " + cookie.getName( ) + ", ");
        out.print("Value: " + cookie.getValue( )+" <br/>");
    }
} else {
    out.println("<h2>No cookies founds</h2>");
}
%>
</body>
```

```
</html>
```

Let us now put the above code in main.jsp file and try to access it. If you set the first_name cookie as "John" and the last_name cookie as "Player" then running *http://localhost:8080/main.jsp* will display the following result –

Found Cookies Name and Value

Name : first_name, Value: John

Name : last_name, Value: Player

Delete Cookies with JSP

To delete cookies is very simple. If you want to delete a cookie, then you simply need to follow these three steps –

Read an already existing cookie and store it in Cookie object.

Set cookie age as zero using the `setMaxAge()` method to delete an existing cookie.

Add this cookie back into the response header.

Example

Following example will show you how to delete an existing cookie named "first_name" and when you run main.jsp JSP next time, it will return null value for first_name.

```
<html>
<head>
    <title>Reading Cookies</title>
</head>

<body>
    <center>
        <h1>Reading Cookies</h1>
    </center>
    <%
        Cookie cookie = null;
        Cookie[] cookies = null;

        // Get an array of Cookies associated with the this domain
        cookies = request.getCookies();
```

```

if( cookies != null ) {
    out.println("<h2> Found Cookies Name and Value</h2>");

    for (int i = 0; i < cookies.length; i++) {
        cookie = cookies[i];

        if((cookie.getName( )).compareTo("first_name") == 0 ) {
            cookie.setMaxAge(0);
            response.addCookie(cookie);
            out.print("Deleted cookie: " +
                cookie.getName( ) + "<br/>");
        }
        out.print("Name : " + cookie.getName( ) + ", ");
        out.print("Value: " + cookie.getValue( )+" <br/>");
    }
} else {
    out.println(
        "<h2>No cookies founds</h2>");
}
%>
</body>

```

</html>

Let us now put the above code in the main.jsp file and try to access it. It will display the following result –

Cookies Name and Value

Deleted cookie : first_name

Name : first_name, Value: John

Name : last_name, Value: Player

Now run *http://localhost:8080/main.jsp* once again and it should display only one cookie as follows

Found Cookies Name and Value

Name : last_name, Value: Player

You can delete your cookies in the Internet Explorer manually. Start at the Tools menu and select the Internet Options. To delete all cookies, click the Delete Cookies button.

3.8 Database Access and JSP Standard Tag Libraries

To start with basic concept, let us create a table and create a few records in that table as follows

Create Table

To create the **Employees** table in the EMP database, use the following steps –

Step 1

Open a **Command Prompt** and change to the installation directory as follows –

```
C:\>
```

```
C:\>cd Program Files\MySQL\bin
```

```
C:\Program Files\MySQL\bin>
```

Step 2

Login to the database as follows –

```
C:\Program Files\MySQL\bin>mysql -u root -p
```

```
Enter password: *****
```

```
mysql>
```

Step 3

Create the **Employee** table in the **TEST** database as follows – –

```
mysql> use TEST;
mysql> create table Employees
(
    id int not null,
    age int not null,
    first varchar (255),
    last varchar (255)
);
Query OK, 0 rows affected (0.08 sec)
```

```
mysql>
```

Create Data Records

Let us now create a few records in the **Employee** table as follows – –

```
mysql> INSERT INTO Employees VALUES (100, 18, 'Zara', 'Ali');
Query OK, 1 row affected (0.05 sec)
```

```
mysql> INSERT INTO Employees VALUES (101, 25, 'Mahnaz', 'Fatma');
Query OK, 1 row affected (0.00 sec)
```

```
mysql> INSERT INTO Employees VALUES (102, 30, 'Zaid', 'Khan');
Query OK, 1 row affected (0.00 sec)
```

```
mysql> INSERT INTO Employees VALUES (103, 28, 'Sumit', 'Mittal');
Query OK, 1 row affected (0.00 sec)
```

```
mysql>
```

SELECT Operation

Following example shows how we can execute the **SQL SELECT** statement using JSTL in JSP programming –

```
<%@ page import = "java.io.*,java.util.*,java.sql.*"%>
<%@ page import = "javax.servlet.http.*,javax.servlet.*" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix = "c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix = "sql"%>

<html>
<head>
    <title>SELECT Operation</title>
</head>

<body>
    <sql:setDataSource var = "snapshot" driver = "com.mysql.jdbc.Driver"
        url = "jdbc:mysql://localhost/TEST"
        user = "root" password = "pass123"/>

    <sql:query dataSource = "${snapshot}" var = "result">
        SELECT * from Employees;
    </sql:query>
```



```

<table border = "1" width = "100%">
  <tr>
    <th>Emp ID</th>
    <th>First Name</th>
    <th>Last Name</th>
    <th>Age</th>
  </tr>

  <c:forEach var = "row" items = "${result.rows}">
    <tr>
      <td><c:out value = "${row.id}"/></td>
      <td><c:out value = "${row.first}"/></td>
      <td><c:out value = "${row.last}"/></td>
      <td><c:out value = "${row.age}"/></td>
    </tr>
  </c:forEach>
</table>

```

```

</body>
</html>

```

INSERT Operation

Following example shows how we can execute the SQL INSERT statement using JSTL in JSP programming –

```

<% @ page import = "java.io.*,java.util.*,java.sql.*"%>
<% @ page import = "javax.servlet.http.*,javax.servlet.*" %>
<% @ taglib uri="http://java.sun.com/jsp/jstl/core" prefix = "c"%>
<% @ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix = "sql"%>

<html>
  <head>
    <title>JINSERT Operation</title>
  </head>

  <body>
    <sql:setDataSource var = "snapshot" driver = "com.mysql.jdbc.Driver"
      url = "jdbc:mysql://localhost/TEST"
      user = "root" password = "pass123"/>
    <sql:update dataSource = "${snapshot}" var = "result">
      INSERT INTO Employees VALUES (104, 2, 'Nuha', 'Ali');
    </sql:update>

    <sql:query dataSource = "${snapshot}" var = "result">
      SELECT * from Employees;
    </sql:query>

    <table border = "1" width = "100%">
      <tr>
        <th>Emp ID</th>
        <th>First Name</th>

```

```

        <th>Last Name</th>
        <th>Age</th>
    </tr>

    <c:forEach var = "row" items = "${result.rows}">
        <tr>
            <td><c:out value = "${row.id}"/></td>
            <td><c:out value = "${row.first}"/></td>
            <td><c:out value = "${row.last}"/></td>
            <td><c:out value = "${row.age}"/></td>
        </tr>
    </c:forEach>
</table>

</body>
</html>

```

DELETE Operation

Following example shows how we can execute the **SQL DELETE** statement using JTSL in JSP programming –

```

<% @ page import = "java.io.*,java.util.*,java.sql.*"%>
<% @ page import = "javax.servlet.http.*,javax.servlet.*" %>
<% @ taglib uri="http://java.sun.com/jsp/jstl/core" prefix = "c"%>
<% @ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix = "sql"%>

<html>
<head>
    <title>DELETE Operation</title>
</head>

<body>
    <sql:setDataSource var = "snapshot" driver = "com.mysql.jdbc.Driver"
        url = "jdbc:mysql://localhost/TEST"
        user = "root" password = "pass123"/>

    <c:set var = "empId" value = "103"/>

    <sql:update dataSource = "${snapshot}" var = "count">
        DELETE FROM Employees WHERE Id = ?
        <sql:param value = "${empId}" />
    </sql:update>

    <sql:query dataSource = "${snapshot}" var = "result">
        SELECT * from Employees;
    </sql:query>

    <table border = "1" width = "100%">
        <tr>
            <th>Emp ID</th>
            <th>First Name</th>

```

```

        <th>Last Name</th>
        <th>Age</th>
    </tr>

    <c:forEach var = "row" items = "${result.rows}">
        <tr>
            <td><c:out value = "${row.id}"/></td>
            <td><c:out value = "${row.first}"/></td>
            <td><c:out value = "${row.last}"/></td>
            <td><c:out value = "${row.age}"/></td>
        </tr>
    </c:forEach>
</table>

```

```

</body>
</html>

```

UPDATE Operation

Following example shows how we can execute the **SQL UPDATE** statement using JTSL in JSP programming –

```

<% @ page import = "java.io.*,java.util.*,java.sql.*"%>
<% @ page import = "javax.servlet.http.*,javax.servlet.*" %>
<% @ taglib uri = "http://java.sun.com/jsp/jstl/core" prefix = "c"%>
<% @ taglib uri = "http://java.sun.com/jsp/jstl/sql" prefix = "sql"%>

<html>
<head>
    <title>DELETE Operation</title>
</head>

<body>
    <sql:setDataSource var = "snapshot" driver = "com.mysql.jdbc.Driver"
        url = "jdbc:mysql://localhost/TEST"
        user = "root" password = "pass123"/>

    <c:set var = "empId" value = "102"/>

    <sql:update dataSource = "${snapshot}" var = "count">
        UPDATE Employees SET WHERE last = 'Ali'
        <sql:param value = "${empId}" />
    </sql:update>

    <sql:query dataSource = "${snapshot}" var = "result">
        SELECT * from Employees;
    </sql:query>

    <table border = "1" width = "100%">
        <tr>
            <th>Emp ID</th>
            <th>First Name</th>
            <th>Last Name</th>

```

```

        <th>Age</th>
    </tr>

    <c:forEach var = "row" items = "${result.rows}">
        <tr>
            <td><c:out value = "${row.id}" /></td>
            <td><c:out value = "${row.first}" /></td>
            <td><c:out value = "${row.last}" /></td>
            <td><c:out value = "${row.age}" /></td>
        </tr>
    </c:forEach>
</table>

</body>
</html>

```

JSP Standard Tag Libraries

The JavaServer Pages Standard Tag Library (JSTL) is a collection of useful JSP tags which encapsulates the core functionality common to many JSP applications.

JSTL has support for common, structural tasks such as iteration and conditionals, tags for manipulating XML documents, internationalization tags, and SQL tags. It also provides a framework for integrating the existing custom tags with the JSTL tags.

Install JSTL Library

To begin working with JSP tags you need to first install the JSTL library. If you are using the Apache Tomcat container, then follow these two steps –

Step 1 – Download the binary distribution from [Apache Standard Taglib](#) and unpack the compressed file.

Step 2 – To use the Standard Taglib from its **Jakarta Taglibs distribution**, simply copy the JAR files in the distribution's 'lib' directory to your application's **webapps\ROOT\WEB-INF\lib** directory.

To use any of the libraries, you must include a <taglib> directive at the top of each JSP that uses the library.

Classification of The JSTL Tags

The JSTL tags can be classified, according to their functions, into the following JSTL tag library groups that can be used when creating a JSP page –

Core Tags

Formatting tags

SQL tags

XML tags

JSTL Functions

Core Tags

The core group of tags are the most commonly used JSTL tags. Following is the syntax to include the JSTL Core library in your JSP –

```
<%@ taglib prefix = "c" uri = "http://java.sun.com/jsp/jstl/core" %>
```

Following table lists out the core JSTL Tags –

S.No.	Tag & Description
1	<u><c:out></u> Like <%= ... >, but for expressions.

2	<u><c:set ></u> Sets the result of an expression evaluation in a 'scope'
3	<u><c:remove ></u> Removes a scoped variable (from a particular scope, if specified).
4	<u><c:catch></u> Catches any Throwable that occurs in its body and optionally exposes it.
5	<u><c:if></u> Simple conditional tag which evaluates its body if the supplied condition is true.
6	<u><c:choose></u> Simple conditional tag that establishes a context for mutually exclusive conditional operations, marked by <when> and <otherwise>.
7	<u><c:when></u> Subtag of <choose> that includes its body if its condition evaluates to 'true'.
8	<u><c:otherwise ></u> Subtag of <choose> that follows the <when> tags and runs only if all of the prior conditions evaluated to 'false'.
9	<u><c:import></u> Retrieves an absolute or relative URL and exposes its contents to either the page, a String in 'var', or a Reader in 'varReader'.
10	<u><c:forEach ></u> The basic iteration tag, accepting many different collection types and supporting subsetting and other functionality .
11	<u><c:forEachTokens></u> Iterates over tokens, separated by the supplied delimiters.
12	<u><c:param></u> Adds a parameter to a containing 'import' tag's URL.
13	<u><c:redirect ></u> Redirects to a new URL.
14	<u><c:url></u> Creates a URL with optional query parameters

Formatting Tags

The JSTL formatting tags are used to format and display text, the date, the time, and numbers for internationalized Websites. Following is the syntax to include Formatting library in your JSP –

```
<%@ taglib prefix = "fmt" uri = "http://java.sun.com/jsp/jstl/fmt" %>
```

Following table lists out the Formatting JSTL Tags –

S.No.	Tag & Description
-------	-------------------

1	<u><fmt:formatNumber></u> To render numerical value with specific precision or format.
2	<u><fmt:parseNumber></u> Parses the string representation of a number, currency, or percentage.
3	<u><fmt:formatDate></u> Formats a date and/or time using the supplied styles and pattern.
4	<u><fmt:parseDate></u> Parses the string representation of a date and/or time
5	<u><fmt:bundle></u> Loads a resource bundle to be used by its tag body.
6	<u><fmt:setLocale></u> Stores the given locale in the locale configuration variable.
7	<u><fmt:setBundle></u> Loads a resource bundle and stores it in the named scoped variable or the bundle configuration variable.
8	<u><fmt:timeZone></u> Specifies the time zone for any time formatting or parsing actions nested in its body.
9	<u><fmt:setTimeZone></u> Stores the given time zone in the time zone configuration variable
10	<u><fmt:message></u> Displays an internationalized message.
11	<u><fmt:requestEncoding></u> Sets the request character encoding

SQL Tags

The JSTL SQL tag library provides tags for interacting with relational databases (RDBMSs) such as **Oracle**, **mySQL**, or **Microsoft SQL Server**.

Following is the syntax to include JSTL SQL library in your JSP –

```
<%@ taglib prefix = "sql" uri = "http://java.sun.com/jsp/jstl/sql" %>
```

Following table lists out the SQL JSTL Tags –

S.No.	Tag & Description
1	<u><sql:setDataSource></u> Creates a simple DataSource suitable only for prototyping
2	<u><sql:query></u> Executes the SQL query defined in its body or through the sql attribute.
3	<u><sql:update></u>

	Executes the SQL update defined in its body or through the sql attribute.
4	<sql:param> Sets a parameter in an SQL statement to the specified value.
5	<sql:dateParam> Sets a parameter in an SQL statement to the specified java.util.Date value.
6	<sql:transaction > Provides nested database action elements with a shared Connection, set up to execute all statements as one transaction.

XML tags

The JSTL XML tags provide a JSP-centric way of creating and manipulating the XML documents. Following is the syntax to include the JSTL XML library in your JSP.

The JSTL XML tag library has custom tags for interacting with the XML data. This includes parsing the XML, transforming the XML data, and the flow control based on the XPath expressions.

<% @ taglib prefix = "x"

uri = "http://java.sun.com/jsp/jstl/xml" %>

Before you proceed with the examples, you will need to copy the following two XML and XPath related libraries into your <Tomcat Installation Directory>\lib –

XercesImpl.jar – Download it from <https://www.apache.org/dist/xerces/j/>

xalan.jar – Download it from <https://xml.apache.org/xalan-j/index.html>

Following is the list of XML JSTL Tags –

S.No.	Tag & Description
1	<x:out> Like <%= ... >, but for XPath expressions.
2	<x:parse> Used to parse the XML data specified either via an attribute or in the tag body.
3	<x:set > Sets a variable to the value of an XPath expression.
4	<x:if > Evaluates a test XPath expression and if it is true, it processes its body. If the test condition is false, the body is ignored.
5	<x:forEach> To loop over nodes in an XML document.
6	<x:choose> Simple conditional tag that establishes a context for mutually exclusive conditional operations, marked by <when> and <otherwise> tags.
7	<x:when > Subtag of <choose> that includes its body if its expression evaluates to 'true'.

8	<x:otherwise > Subtag of <choose> that follows the <when> tags and runs only if all of the prior conditions evaluates to 'false'.
9	<x:transform > Applies an XSL transformation on a XML document
10	<x:param > Used along with the transform tag to set a parameter in the XSLT stylesheet

JSTL Functions

JSTL includes a number of standard functions, most of which are common string manipulation functions. Following is the syntax to include JSTL Functions library in your JSP

–

```
<% @ taglib prefix = "fn"
```

```
uri = "http://java.sun.com/jsp/jstl/functions" %>
```

Following table lists out the various JSTL Functions –

S.No.	Function & Description
1	fn:contains() Tests if an input string contains the specified substring.
2	fn:containsIgnoreCase() Tests if an input string contains the specified substring in a case insensitive way.
3	fn:endsWith() Tests if an input string ends with the specified suffix.
4	fn:escapeXml() Escapes characters that can be interpreted as XML markup.
5	fn:indexOf() Returns the index withing a string of the first occurrence of a specified substring.
6	fn:join() Joins all elements of an array into a string.
7	fn:length() Returns the number of items in a collection, or the number of characters in a string.
8	fn:replace() Returns a string resulting from replacing in an input string all occurrences with a given string.
9	fn:split() Splits a string into an array of substrings.
10	fn:startsWith() Tests if an input string starts with the specified prefix.
11	fn:substring() Returns a subset of a string.
12	fn:substringAfter() Returns a subset of a string following a specific

	substring.
13	fn:substringBefore() Returns a subset of a string before a specific substring.
14	fn:toLowerCase() Converts all of the characters of a string to lower case.
15	fn:toUpperCase() Converts all of the characters of a string to upper case.
16	fn:trim() Removes white spaces from both ends of a string.

3.9 JSP Custom Tag, JSP Expression Language, and JSP Exception Handling

A custom tag is a user-defined JSP language element. When a JSP page containing a custom tag is translated into a servlet, the tag is converted to operations on an object called a tag handler. The Web container then invokes those operations when the JSP page's servlet is executed.

JSP tag extensions lets you create new tags that you can insert directly into a JavaServer Page. The JSP 2.0 specification introduced the Simple Tag Handlers for writing these custom tags.

To write a custom tag, you can simply extend **SimpleTagSupport** class and override the **doTag()** method, where you can place your code to generate content for the tag.

Create "Hello" Tag

Consider you want to define a custom tag named `<ex:Hello>` and you want to use it in the following fashion without a body –

```
<ex:Hello />
```

To create a custom JSP tag, you must first create a Java class that acts as a tag handler. Let us now create the **HelloTag** class as follows –

```
package com.tutorialspoint;
```

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

```
public class HelloTag extends SimpleTagSupport {
    public void doTag() throws JspException, IOException {
        JspWriter out = getJspContext().getOut();
        out.println("Hello Custom Tag!");
    }
}
```

The above code has simple coding where the **doTag()** method takes the current **JspContext** object using the **getJspContext()** method and uses it to send **"Hello Custom Tag!"** to the current **JspWriter** object

Let us compile the above class and copy it in a directory available in the environment variable CLASSPATH. Finally, create the following tag library file: **<Tomcat-Installation-Directory>webapps\ROOT\WEB-INF\custom.tld**.

```
<taglib>
  <tlib-version>1.0</tlib-version>
```

```
<jsp-version>2.0</jsp-version>
<short-name>Example TLD</short-name>
```

```
<tag>
  <name>Hello</name>
  <tag-class>com.tutorialspoint.HelloTag</tag-class>
  <body-content>empty</body-content>
</tag>
</taglib>
```

Let us now use the above defined custom tag **Hello** in our JSP program as follows –

```
<% @ taglib prefix = "ex" uri = "WEB-INF/custom.tld"%>
```

```
<html>
  <head>
    <title>A sample custom tag</title>
  </head>

  <body>
    <ex:Hello/>
  </body>
</html>
```

Call the above JSP and this should produce the following result –
Hello Custom Tag!

Accessing the Tag Body

You can include a message in the body of the tag as you have seen with standard tags. Consider you want to define a custom tag named **<ex:Hello>** and you want to use it in the following fashion with a body –

```
<ex:Hello>
  This is message body
</ex:Hello>
```

Let us make the following changes in the above tag code to process the body of the tag –
package com.tutorialspoint;

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

public class HelloTag extends SimpleTagSupport {
  StringWriter sw = new StringWriter();
  public void doTag()

  throws JspException, IOException {
    getJspBody().invoke(sw);
    getJspContext().getOut().println(sw.toString());
  }
}
```

Here, the output resulting from the invocation is first captured into a **StringWriter** before being written to the JspWriter associated with the tag. We need to change TLD file as follows –

```

<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>2.0</jsp-version>
  <short-name>Example TLD with Body</short-name>

```

```

  <tag>
    <name>Hello</name>
    <tag-class>com.tutorialspoint.HelloTag</tag-class>
    <body-content>scriptless</body-content>
  </tag>
</taglib>

```

Let us now call the above tag with proper body as follows –

```
<% @ taglib prefix = "ex" uri = "WEB-INF/custom.tld"%>
```

```

<html>
  <head>
    <title>A sample custom tag</title>
  </head>

```

```

  <body>
    <ex:Hello>
      This is message body
    </ex:Hello>
  </body>
</html>

```

You will receive the following result –

This is message body

Custom Tag Attributes

You can use various attributes along with your custom tags. To accept an attribute value, a custom tag class needs to implement the **setter** methods, identical to the JavaBean setter methods as shown below –

```
package com.tutorialspoint;
```

```

import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
import java.io.*;

```

```

public class HelloTag extends SimpleTagSupport {
  private String message;

```

```

  public void setMessage(String msg) {
    this.message = msg;
  }

```

```

  StringWriter sw = new StringWriter();
  public void doTag()

```

```

    throws JspException, IOException {
    if (message != null) {
      /* Use message from attribute */
      JspWriter out = getJspContext().getOut();

```

```

        out.println( message );
    } else {
        /* use message from the body */
        getJspBody().invoke(sw);
        getJspContext().getOut().println(sw.toString());
    }
}
}

```

The attribute's name is "**message**", so the setter method is **setMessage()**. Let us now add this attribute in the TLD file using the **<attribute>** element as follows –

```

<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>2.0</jsp-version>
  <short-name>Example TLD with Body</short-name>

  <tag>
    <name>Hello</name>
    <tag-class>com.tutorialspoint.HelloTag</tag-class>
    <body-content>scriptless</body-content>

    <attribute>
      <name>message</name>
    </attribute>

  </tag>
</taglib>

```

Let us follow JSP with message attribute as follows –

```

<% @ taglib prefix = "ex" uri = "WEB-INF/custom.tld"%>

<html>
  <head>
    <title>A sample custom tag</title>
  </head>

  <body>
    <ex:Hello message = "This is custom tag" />
  </body>
</html>

```

This will produce following result

This is custom tag

Consider including the following properties for an attribute –

S.No.	Property & Purpose
1	Name The name element defines the name of an attribute. Each attribute name must be unique for a particular tag.
2	Required This specifies if this attribute is required or is an optional one. It would be false for optional.

3	Rtexprvalue Declares if a runtime expression value for a tag attribute is valid
4	Type Defines the Java class-type of this attribute. By default it is assumed as String
5	Description Informational description can be provided.
6	Fragment Declares if this attribute value should be treated as a JspFragment.

Following is the example to specify properties related to an attribute –

.....

```
<attribute>
  <name>attribute_name</name>
  <required>>false</required>
  <type>java.util.Date</type>
  <fragment>>false</fragment>
</attribute>
```

.....

If you are using two attributes, then you can modify your TLD as follows –

....

```
<attribute>
  <name>attribute_name1</name>
  <required>>false</required>
  <type>java.util.Boolean</type>
  <fragment>>false</fragment>
</attribute>
```

```
<attribute>
  <name>attribute_name2</name>
  <required>>true</required>
  <type>java.util.Date</type>
</attribute>
```

.....

JSP Expression Language and JSP Exception Handling

access to the header values makes it possible to easily access application data stored in JavaBeans components. JSP EL allows you to create expressions both (a) arithmetic and (b) logical. Within a JSP EL expression, you can use integers, floating point numbers, strings, the built-in constants true and false for boolean values, and null.

Simple Syntax

Typically, when you specify an attribute value in a JSP tag, you simply use a string. For example – `<jsp:setProperty name = "box" property = "perimeter" value = "100"/>`

JSP EL allows you to specify an expression for any of these attribute values. A simple syntax for JSP EL is as follows –

`${expr}`

Here expr specifies the expression itself. The most common operators in JSP EL are . and []. These two operators allow you to access various attributes of Java Beans and built-in JSP objects.

For example, the above syntax `<jsp:setProperty>` tag can be written with an expression like –

`<jsp:setProperty name = "box" property = "perimeter"
value = "${2*box.width+2*box.height}"/>`

When the JSP compiler sees the `${ }` form in an attribute, it generates code to evaluate the expression and substitutes the value of expression.

You can also use the JSP EL expressions within template text for a tag. For example, the `<jsp:text>` tag simply inserts its content within the body of a JSP. The following `<jsp:text>` declaration inserts `<h1>Hello JSP!</h1>` into the JSP output

```
<jsp:text>
  <h1>Hello JSP!</h1>
</jsp:text>
```

You can now include a JSP EL expression in the body of a `<jsp:text>` tag (or any other tag) with the same `${ }` syntax you use for attributes. For example –

```
<jsp:text>
  Box Perimeter is: ${2*box.width + 2*box.height}
</jsp:text>
```

EL expressions can use parentheses to group subexpressions. For example, `${(1 + 2) * 3}` equals 9, but `${1 + (2 * 3)}` equals 7.

To deactivate the evaluation of EL expressions, we specify the `isELIgnored` attribute of the page directive as below –

```
<%@ page isELIgnored = "true|false" %>
```

The valid values of this attribute are true and false. If it is true, EL expressions are ignored when they appear in static text or tag attributes. If it is false, EL expressions are evaluated by the container.

Basic Operators in EL

JSP Expression Language (EL) supports most of the arithmetic and logical operators supported by Java. Following table lists out the most frequently used operators –

S.No.	Operator & Description
1	. Access a bean property or Map entry
2	[] Access an array or List element
3	() Group a subexpression to change the evaluation order

4	+ Addition
5	- Subtraction or negation of a value
6	* Multiplication
7	/ or div Division
8	% or mod Modulo (remainder)
9	== or eq Test for equality
10	!= or ne Test for inequality
11	< or lt Test for less than
12	> or gt Test for greater than
13	<= or le Test for less than or equal
14	>= or ge Test for greater than or equal
15	&& or and Test for logical AND
16	or or Test for logical OR
17	! or not Unary Boolean complement
18	Empty Test for empty variable values

Functions in JSP EL

JSP EL allows you to use functions in expressions as well. These functions must be defined in the custom tag libraries. A function usage has the following syntax

```
${ns:func(param1, param2, ...)}
```

Where ns is the namespace of the function, func is the name of the function and param1 is the first parameter value. For example, the function fn:length, which is part of the JSTL library. This function can be used as follows to get the length of a string.

```
${fn:length("Get my length")}
```

o use a function from any tag library (standard or custom), you must install that library on your server and must include the library in your JSP using the <taglib> directive as explained in the JSTL chapter.

JSP EL Implicit Objects

The JSP expression language supports the following implicit objects –

S.No	Implicit object & Description
1	pageScope Scoped variables from page scope
2	requestScope Scoped variables from request scope

3	sessionScope	Scoped variables from session scope
4	applicationScope	Scoped variables from application scope
5	Param	Request parameters as strings
6	paramValues	Request parameters as collections of strings
7	Header	HTTP request headers as strings
8	headerValues	HTTP request headers as collections of strings
9	initParam	Context-initialization parameters
10	Cookie	Cookie values
11	pageContext	The JSP PageContext object for the current page

You can use these objects in an expression as if they were variables. The examples that follow will help you understand the concepts –

The pageContext Object

The pageContext object gives you access to the pageContext JSP object. Through the pageContext object, you can access the request object. For example, to access the incoming query string for a request, you can use the following expression –

```
${pageContext.request.queryString}
```

The Scope Objects

The pageScope, requestScope, sessionScope, and applicationScope variables provide access to variables stored at each scope level.

For example, if you need to explicitly access the box variable in the application scope, you can access it through the applicationScope variable as applicationScope.box.

The param and paramValues Objects

The param and paramValues objects give you access to the parameter values normally available through the request.getParameter and request.getParameterValues methods.

For example, to access a parameter named order, use the expression `${param.order}` or `${param["order"]}`.

Following is the example to access a request parameter named username –

```
<% @ page import = "java.io.*,java.util.*" %>
<%String title = "Accessing Request Param";%>
```

```
<html>
<head>
<title><% out.print(title); %></title>
</head>

<body>
<center>
<h1><% out.print(title); %></h1>
</center>

<div align = "center">
<p>${param["username"]}</p>
```



```

    </div>
  </body>
</html>

```

The param object returns single string values, whereas the paramValues object returns string arrays.

header and headerValues Objects

The header and headerValues objects give you access to the header values normally available through the request.getHeader and the request.getHeaders methods.

For example, to access a header named user-agent, use the expression `${header.user-agent}` or `${header["user-agent"]}`.

Following is the example to access a header parameter named user-agent –

```

<% @ page import = "java.io.*,java.util.*" %>
<%String title = "User Agent Example";%>

```

```

<html>
  <head>
    <title><% out.print(title); %></title>
  </head>

  <body>
    <center>
      <h1><% out.print(title); %></h1>
    </center>

    <div align = "center">
      <p>${header["user-agent"]}</p>
    </div>
  </body>
</html>

```

The output will somewhat be like the following

User Agent Example

Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; WOW64; Trident/4.0;
SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729;
Media Center PC 6.0; HPNTDF; .NET4.0C; InfoPath.2)

JSP Exception Handling

When you are writing a JSP code, you might make coding errors which can occur at any part of the code. There may occur the following type of errors in your JSP code –

Checked exceptions

A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.

Runtime exceptions

A runtime exception is an exception that probably could have been avoided by the programmer. As opposed to the checked exceptions, runtime exceptions are ignored at the time of compilation.

Errors

These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything

about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

We will further discuss ways to handle run time exception/error occurring in your JSP code.

Using Exception Object

The exception object is an instance of a subclass of Throwable (e.g., java.lang.NullPointerException) and is only available in error pages. Following table lists out the important methods available in the Throwable class.

S.No.	Methods & Description
1	public String getMessage() Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
2	public Throwable getCause() Returns the cause of the exception as represented by a Throwable object.
3	public String toString() Returns the name of the class concatenated with the result of getMessage().
4	public void printStackTrace() Prints the result of toString() along with the stack trace to System.err, the error output stream.
5	public StackTraceElement [] getStackTrace() Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.
6	public Throwable fillInStackTrace() Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

JSP gives you an option to specify Error Page for each JSP. Whenever the page throws an exception, the JSP container automatically invokes the error page.

Following is an example to specify an error page for a main.jsp. To set up an error page, use the `<%@ page errorPage = "xxx" %>` directive.

```
<%@ page errorPage = "ShowError.jsp" %>
```

```
<html>
<head>
  <title>Error Handling Example</title>
</head>

<body>
  <%
    // Throw an exception to invoke the error page
    int x = 1;

    if (x == 1) {
      throw new RuntimeException("Error condition!!!");
    }
  %>
</body>
</html>
```

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

We will now write one Error Handling JSP ShowError.jsp, which is given below. Notice that the error-handling page includes the directive `<%@ page isErrorPage = "true" %>`. This directive causes the JSP compiler to generate the exception instance variable.

```
<%@ page isErrorPage = "true" %>
```

```
<html>  
  <head>  
    <title>Show Error Page</title>  
  </head>  
  
  <body>  
    <h1>Opps...</h1>  
    <p>Sorry, an error occurred.</p>  
    <p>Here is the exception stack trace: </p>  
    <pre><% exception.printStackTrace(response.getWriter()); %></pre>  
  </body>  
</html>
```

Access the main.jsp, you will receive an output somewhat like the following –
java.lang.RuntimeException: Error condition!!!

.....

Opps...
Sorry, an error occurred.

Here is the exception stack trace:

Using JSTL Tags for Error Page

You can make use of JSTL tags to write an error page ShowError.jsp. This page has almost same logic as in the above example, with better structure and more information `<%@ taglib prefix = "c" uri = "http://java.sun.com/jsp/jstl/core" %>`

```
<%@page isErrorPage = "true" %>
```

```
<html>  
  <head>  
    <title>Show Error Page</title>  
  </head>  
  
  <body>  
    <h1>Opps...</h1>  
    <table width = "100%" border = "1">  
      <tr valign = "top">  
        <td width = "40%"><b>Error:</b></td>  
        <td>${pageContext.exception}</td>  
      </tr>  
  
      <tr valign = "top">
```

```

        <td><b>URI:</b></td>
        <td>${pageContext.errorData.requestURI}</td>
    </tr>

    <tr valign = "top">
        <td><b>Status code:</b></td>
        <td>${pageContext.errorData.statusCode}</td>
    </tr>

    <tr valign = "top">
        <td><b>Stack trace:</b></td>
        <td>
            <c:forEach var = "trace"
                items = "${pageContext.exception.stackTrace}">
                <p>${trace}</p>
            </c:forEach>
        </td>
    </tr>
</table>

</body>
</html>

```

Access the main.jsp, the following will be generated –

Error:	java.lang.RuntimeException: Error condition!!!
URI:	/main.jsp
Status code:	500
Stack trace:	org.apache.jsp.main_jsp._jspService(main_jsp.java:65)) org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:68) javax.servlet.http.HttpServlet.service(HttpServlet.java:722) org.apache.jasper.servlet.JspServlet.service(JspServlet.java:265) javax.servlet.http.HttpServlet.service(HttpServlet.java:722)

Using Try...Catch Block

If you want to handle errors within the same page and want to take some action instead of firing an error page, you can make use of the try....catch block.

Following is a simple example which shows how to use the try...catch block. Let us put the following code in main.jsp –

```

<html>
<head>
    <title>Try...Catch Example</title>
</head>

```

```

<body>
<%
    try {
        int i = 1;
        i = i / 0;
        out.println("The answer is " + i);
    }
    catch (Exception e) {
        out.println("An exception occurred: " + e.getMessage());
    }
%>
</body>
</html>

```

Access the main.jsp, it should generate an output somewhat like the following –
An exception occurred: / by zero

3.10 Assignment-3

Short Answer Questions

Knowledge/Remembering Level Questions

1. List out the two tags usually contained by a JSP.
2. Mention the three processes involved in JSP Compilation Process.
3. Which Phase of the JSP lifecycle represents all interactions with requests?
4. What is MVC?
5. Where you would develop your JSP programs, test them and finally run them?
6. Which command is used in UNIX to start Tomcat Server?
7. Write the command to stop Tomcat on the Windows machine
8. Which is used to provide instructions to the container?
9. Which is used to include a file during the translation phase?
10. Which declares that your JSP page uses a set of custom tags?
11. Can you recall which finds or instantiates a JavaBean?
12. What are the two attributes that are common to all Action elements?
13. Which attribute identifies the lifecycle of the Action element?
14. Which is used to insert Java components into a JSP page?
15. What action can be used to write the template text in JSP pages and documents?
16. Can you recall which object is an instance of javax.servlet.http.HttpSession?
17. Which is text files stored on the client computer for various information tracking purposes?
18. Which JSTL function is used to escapes characters that can be interpreted as XML markup?
19. Which makes easily access application data stored in JavaBeans components?
20. What are the two objects give you access to the header values?

Long Answer Questions

Understanding Level Questions

1. Explain the need and importance of JSP.
2. With a neat diagram explain JSP Lifecycle.
3. Write a short note on MVC Architecture.
4. Explain in detail any Two JSP Directives

5. Discuss on JSP Actions.
6. Explain JSP Standard Tag Libraries.

Application Level Questions

7. Differentiate Servlets and JSP and also demonstrate the advantages of JSP.
8. How would you explain the concept of implicit objects in connection with JSP?
9. Apply the concept of Session Tracking to find out the creation time and the last-accessed time for a session
10. Implement the cookies for the first and the last name.
11. With your own program implement the concept of Exception handling in JSP

Skill Level Question

12. Create a program for database access using JSP

UNIT-IV

ANNOTATIONS AND JAVA BEANS

4.1 Creating Packages, and Interfaces with examples

Packages: One of the main features of OOP is its ability to reuse the code already created. One way of achieving this is by extending the classes and implementing the interfaces. This is limited to reusing the classes within a program. If we need to use classes from other programs we have to use packages. Packages are Java's way of grouping a variety of classes and/or interfaces together. The grouping is usually done according to functionality. In fact, packages act as "containers" for classes. By organizing our classes into packages we achieve the following benefits:

1. The classes contained in the packages of other programs can be easily reused.
2. In packages, classes can be unique compared with classes in other packages. That is, two classes in two different packages can have the same name. They may be referred by their fully qualified name, comprising the package name and the class name.
3. Packages provide a way to "hide" classes thus preventing other programs or packages from accessing classes that are meant for internal use only.
4. Packages also provide a way for separating "design" from "coding". First we can design classes and decide their relationships, and then we can implement the Java code needed for the methods. It is possible to change the implementation of any method without affecting the rest of the design.

For most applications, we will need to use two different sets of classes, one for the internal representation of our program's data, and the other for external presentation purposes. We may have to build our own classes for handling our data and use existing class libraries for designing user interfaces.

Defining packages: All classes in Java belong to some package. When no package statement is specified, the default (or global) package is used. Furthermore, the default package has no name, which makes the default package transparent. This is why you haven't had to worry about packages before now. While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define one or more packages for your code.

To create a package, put a package command at the top of a Java source file. The classes declared within that file will then belong to the specified package. Since a package defines a namespace, the names of the classes that you put into the file become part of that package's namespace.

This is the general form of the package statement:

```
package pkg;
```

Here, pkg is the name of the package. For example, the following statement creates a package called Mypack

```
package mypack;
```

Java uses the file system to manage packages, with each package stored in its own directory. For example, the .class files for any classes you declare to be part of mypack must be stored in a directory called mypack. Like the rest of Java, package names are case sensitive. This means that the directory in which a package is stored must be precisely the same as the package name. Lowercase is often used for package names. More than one file can include the same package statement. The package statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files. You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here

```
package pack1.pack2.pack3...packN;
```

Of course, you must create directories that support the package hierarchy that you create.

Finding packages and CLASSPATH: Java run-time system knows where to look for packages has three parts. First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found. Second, you can specify a directory path or paths by setting the CLASSPATH environmental variable. Third, you can use the class path option with java and javac to specify the path to your classes. For example, assuming the following package specification:

```
package mypack;
```

In order for a program to find mypack, one of three things must be true: The program can be executed from a directory immediately above mypack, or CLASSPATH must be set to include the path to mypack, or the classpath option must specify the path to mypack when the program is run via java. To avoid problems, it is best to keep all .java and .class files associated with a package in that package's directory. Also, compile each file from the directory above the package directory.

The listing below shows a package named package1 containing a single class ClassA.

```
package package1;
public class ClassA
{
    public void displayA()
    {
        System.out.println("Class A");
    }
}
```

This source file should be named ClassA.java and stored in the subdirectory package1 as stated earlier.

Now compile this java file. The resultant ClassA.class will be stored in the same subdirectory.

Now consider the listing shown below:

```
import package1.ClassA;
class PackageTest1
{
    public static void main (String args[ ] )
    {
        ClassA objectA = new ClassA( ) ;
        objectA.displayA( )
    }
}
```

This listing shows a simple program that imports the class ClassA from the package package1. The source file should be saved as PackageTest1.java and then compiled. The source file and the compiled file would be saved in the directory of which package 1 was a subdirectory. Now we can run the program and obtain the results.

During the compilation of PackageTest1.java the compiler checks for the file ClassA.class in the package1 directory for information it needs, but it does not actually include the code from ClassA.class in the file PackageTest1.class. When the PackageTest1 program is run, Java looks

for the file `PackageTest1.class` and loads it using something called *class loader*. Now the interpreter knows that it also needs the code in the file `ClassA.class` and loads it as well.

Packages and Member Access

The visibility of an element is determined by its access specification, `private`, `public`, `protected`, or default and the package in which it resides. Thus, the visibility of an element is determined by its visibility within a class and its visibility within a package. This multilayered approach to access control supports a rich assortment of access privileges. Table 8.1 summarizes the various access levels.

<div> <div>Access modifier →</div> <div>Access Location ↓</div> </div>	Public	Protected	default (friendly)	private
Same class	Yes	Yes	Yes	Yes
Subclass in same package	Yes	Yes	Yes	No
Other classes in same package	Yes	Yes	Yes	No
Subclass in same packages	Yes	Yes	No	No
Non-subclasses in other packages	Yes	No	No	No

If a member of a class has no explicit access modifier, then it is visible within its package but not outside its package. Therefore, you will use the default access specification for elements that you want to keep private to a package but public within that package. Members explicitly declared `public` are visible everywhere, including different classes and different packages. There is no restriction on their use or access. A `private` member is accessible only to the other members of its class. A `private` member is unaffected by its membership in a package. A member specified as `protected` is accessible within its package and to all subclasses, including subclasses in other packages.

Above table applies only to members of classes. A top-level class has only two possible access levels: default and `public`. When a class is declared as `public`, it is accessible by any other code. If a class has default access, it can be accessed only by other code within its same package. Also, a class that is declared `public` must reside in a file by the same name.

When two classes were in the same package, so there one class can use another class because the default access privilege grants all members of the same package access. For example if `Book` were in one package and `BookDemo` were in another, the situation would be different. In this case, access to `Book` would be denied. To make `Book` available to other packages, you must declare class name, constructor and methods as `public`.

```
package package2;
public class ClassB
{
    public int m = 10
    public void displayB( )
```

```

    {
        System.out.println("Class B");
        System.out.println("m = " + m);
    }
}

```

The source file and the compiled file of this package are located in the subdirectory package2. Using import you can bring one or more members of a package into view. This allows you to use those members directly, without explicit package qualification.

```

import package name. class name;
or
import packagename.*;

```

These are known as *import statements* and must appear at the top of the file, before any class declarations. The first statement allows the specified class in the specified package to be imported.

```

// demonstrate import.
package Bookpackext;
import backpack.*;
// Use the Book class from backpack.
class UseBook {
    public static void main(String args [ ]) {
        ExtBook books[ ] = new ExtBook [5];
        book [0 ]= new ExtBook ( "Java: A Beginner's Guide", "Schildt", 2011, "McGraw-
            Hill");
        book [1]= new ExtBook ( "Java: The Complete Reference", "Schildt", 2011,
            "McGraw-Hill");
        book [2]= new ExtBook ( "TheArt of java", "Schildt and Holmes", 2003,
            "Osborne/McGraw-Hill");
        book [3]= new ExtBook ( "Red Storm Rising", "Clancy", 1986, "Putnam" );
        book [4]= new ExtBook ( "On the Road", "kerouac", 1955, "Viking" );
        for(int i=0; i<books.length; i++) book[i]. show( );
    }
}

```

Java defines a large number of standard classes that are available to all programs. This class library is often referred to as the Java API (Application Programming Interface). The Java API is stored in packages. At the top of the package hierarchy is java. Descending from java are several subpackages, including these;

Package Name	Contents
java.lang	Language support classes. These are classes that Java compiler itself uses and therefore they are automatically imported. They include classes for primitive types, strings, math functions, threads and exceptions.
java.util	Language utility classes such as vectors, hash tables, random numbers, date, etc.
java.io	Input/output support classes. They provide facilities for the input and output of data.
java.awt	Set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on.

java.net	Classes for networking. They include classes for communicating with local computers as well as with internet servers.
java.applet	Classes for creating and implementing applets.

Interfaces

Java does not support multiple inheritance. That is, classes in Java cannot have more than one superclass. For instance, a definition like

```
class A extends B extends C
{
}
}
```

A large number of real-life applications require the use of multiple inheritance whereby we inherit methods and properties from several distinct classes. Java provides an alternate approach known as *interfaces* to support the concept of multiple inheritance. Although a Java class cannot be a subclass of more than one superclass, it can *implement* more than one interface, thereby enabling us to create classes that build upon other classes without the problems created by multiple inheritance.

Defining interfaces: An interface is basically a kind of class. Like classes, interfaces contains methods and variables but with a major difference. The difference is that interfaces define only abstract methods and final fields. This means that interfaces do not specify any code to implement these methods and data fields contain only constant

Therefore, it is the responsibility of the class that implements an interface to define the code for implementation of these methods. The syntax for defining an interface is very similar to that for defining a class. The general form of an interface definition is

```
interface InterfaceName
{
    variables declaration;
    methods declaration;
}
```

Here, interface is the key word and *InterfaceName* is any valid Java variable (just like class names). Variables are declared as follows

```
static final type VariableName = Value;
```

Note that all variables are declared as constants. Methods declaration will contain only a list of methods without any body statements. Example

```
return-type methodName (parameter_list);
```

Here is an example of an interface definition that contains two variables and one method

```
interface Item
{
    static final int code = 1001;
    static final String name = "Fan";
    void display ( ) ;
}
```

Note that the code for the method is not included in the interface and the method declaration simply ends with a semicolon.

The class that implements this interface must define the code for the method. Another example of an interface is:

```
interface Area
{
    final static float pi = 3.142F;
    float compute (float X, float y);
    void show ( );
}
```

Implementing interfaces

Interfaces are used as "superclasses" whose properties are inherited by classes. It is therefore necessary to create a class that inherits the given interface. This is done as follows

```
class classname implements Interfacename
{
    body of classname
}
```

Here the class *classname* "implements" the interface *interfacename*. A more general form of implementation may look like this:

```
class classname extends superclass implements interfaced,interface2, ... ...
{
    body of classname
}
```

This shows that a class can extend another class while implementing interfaces. When a class implements more than one interface, they are separated by a comma.

In this program, first we create an interface Area and implement the same in two different classes, Rectangle and Circle. We create an instance of each class using the new operator. Then we declare an object of type Area, the interface class. Now, we assign the reference to the Rectangle object rect to area. When we call the compute method of area, the compute method of Rectangle class is invoked. We repeat the same thing with the Circle object.

```
//InterfaceTest.java
interface Area                //Interface defined
{
    final static float pi = 3.14F;
    float compute (float x, float y);
}
class Rectangle implements Area
{
    // Interface implemented
    public float compute (float x, float y)
    {
        return (x*y);
    }
}
class Circle implements Area
```

```

{
    // Another implementation
    public float compute (float x, float y)
    {
        return (pi*x*x);
    }
}
class InterfaceTest
{
    public static void main (String args[ ])
    {
        Rectangle rect = new Rectangle( );
        Circle cir = new Circle( );
        Area area;    //Interface object
        area = rect;
        System.out.println("Area of Rectangle = "+ area.compute(10,20));
        area = cir;
        System.out.println("Area of Circle = " + area.compute(10,10));
    }
}

```

4.2 Creating JAR files and Annotations and New java

Creating a JAR File

The basic format of the command for creating a JAR file is:

```
jar cf jar-file input-file(s)
```

The options and arguments used in this command are:

The c option indicates that you want to create a JAR file.

The f option indicates that you want the output to go to a file rather than to stdout.

jar-file is the name that you want the resulting JAR file to have. You can use any filename for a JAR file. By convention, JAR filenames are given a .jar extension, though this is not required.

The input-file(s) argument is a space-separated list of one or more files that you want to include in your JAR file. The input-file(s) argument can contain the wildcard * symbol. If any of the "input-files" are directories, the contents of those directories are added to the JAR archive recursively.

The c and f options can appear in either order, but there must not be any space between them.

This command will generate a compressed JAR file and place it in the current directory. The command will also generate a default manifest file for the JAR archive.

Note:

The metadata in the JAR file, such as the entry names, comments, and contents of the manifest, must be encoded in UTF8. You can add any of these additional options to the cf options of the basic command:

jar command options	
Option	Description
v	Produces <i>verbose</i> output on stdout while the JAR file is being built. The verbose output tells you the name of each file as it's added to the JAR file.
0 (zero)	Indicates that you don't want the JAR file to be compressed.
M	Indicates that the default manifest file should not be produced.

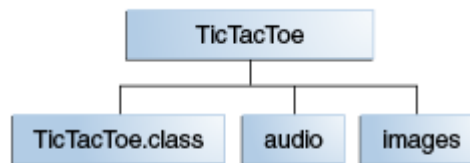
m	Used to include manifest information from an existing manifest file. The format for using this option is: <code>jar cmf jar-file existing-manifest input-file(s)</code> See Modifying a Manifest File for more information about this option.
	Warning: The manifest must end with a new line or carriage return. The last line will not be parsed properly if it does not end with a new line or carriage return.
-C	To change directories during execution of the command. See below for an example.

Note:

When you create a JAR file, the time of creation is stored in the JAR file. Therefore, even if the contents of the JAR file do not change, when you create a JAR file multiple times, the resulting files are not exactly identical. You should be aware of this when you are using JAR files in a build environment. It is recommended that you use versioning information in the manifest file, rather than creation time, to control versions of a JAR file. See the Setting Package Version Information section.

An Example

Let us look at an example. A simple TicTacToe applet. You can see the source code of this applet by downloading the JDK Demos and Samples bundle from Java SE Downloads. This demo contains class files, audio files, and images having this structure:

**TicTacToe folder Hierarchy**

The audio and images subdirectories contain sound files and GIF images used by the applet. You can obtain all these files from jar/examples directory when you download the entire Tutorial online. To package this demo into a single JAR file named TicTacToe.jar, you would run this command from inside the TicTacToe directory:

```
jar cvf TicTacToe.jar TicTacToe.class audio images
```

The audio and images arguments represent directories, so the Jar tool will recursively place them and their contents in the JAR file. The generated JAR file TicTacToe.jar will be placed in the current directory. Because the command used the v option for verbose output, you would see something similar to this output when you run the command:

```

adding: TicTacToe.class (in=3825) (out=2222) (deflated 41%)
adding: audio/ (in=0) (out=0) (stored 0%)
adding: audio/beep.au (in=4032) (out=3572) (deflated 11%)
adding: audio/ding.au (in=2566) (out=2055) (deflated 19%)
adding: audio/return.au (in=6558) (out=4401) (deflated 32%)
adding: audio/yahoo1.au (in=7834) (out=6985) (deflated 10%)
adding: audio/yahoo2.au (in=7463) (out=4607) (deflated 38%)
adding: images/ (in=0) (out=0) (stored 0%)
adding: images/cross.gif (in=157) (out=160) (deflated -1%)
adding: images/not.gif (in=158) (out=161) (deflated -1%)
  
```

You can see from this output that the JAR file TicTacToe.jar is compressed. The Jar tool compresses files by default. You can turn off the compression feature by using the 0 (zero) option, so that the command would look like:

```
jar cvf0 TicTacToe.jar TicTacToe.class audio images
```

You might want to avoid compression, for example, to increase the speed with which a JAR file could be loaded by a browser. Uncompressed JAR files can generally be loaded more quickly than compressed files because the need to decompress the files during loading is eliminated. However, there is a tradeoff in that download time over a network may be longer for larger, uncompressed files.

The Jar tool will accept arguments that use the wildcard * symbol. As long as there weren't any unwanted files in the TicTacToe directory, you could have used this alternative command to construct the JAR file:

```
jar cvf TicTacToe.jar *
```

Though the verbose output doesn't indicate it, the Jar tool automatically adds a manifest file to the JAR archive with path name META-INF/MANIFEST.MF. See the Working with Manifest Files: The Basics section for information about manifest files.

In the above example, the files in the archive retained their relative path names and directory structure. The Jar tool provides the -C option that you can use to create a JAR file in which the relative paths of the archived files are not preserved. It's modeled after TAR's -C option.

As an example, suppose you wanted to put audio files and gif images used by the TicTacToe demo into a JAR file, and that you wanted all the files to be on the top level, with no directory hierarchy. You could accomplish that by issuing this command from the parent directory of the images and audio directories:

```
jar cf ImageAudio.jar -C images . -C audio .
```

The -C images part of this command directs the Jar tool to go to the images directory, and the . following -C images directs the Jar tool to archive all the contents of that directory. The -C audio . part of the command then does the same with the audio directory. The resulting JAR file would have this table of contents:

META-INF/MANIFEST.MF

cross.gif

not.gif

beep.au

ding.au

return.au

yahoo1.au

yahoo2.au

By contrast, suppose that you used a command that did not employ the -C option:

```
jar cf ImageAudio.jar images audio
```

The resulting JAR file would have this table of contents:

META-INF/MANIFEST.MF

images/cross.gif

images/not.gif

audio/beep.au

audio/ding.au

audio/return.au

audio/yahoo1.au

audio/yahoo2.au

Java Annotations

Java Annotation is a tag that represents the metadata i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.

Annotations in Java are used to provide additional information, so it is an alternative option for XML and Java marker interfaces.

First, we will learn some built-in annotations then we will move on creating and using custom annotations.

Built-In Java Annotations

There are several built-in annotations in Java. Some annotations are applied to Java code and some to other annotations.

Built-In Java Annotations used in Java code

@Override

@SuppressWarnings

@Deprecated

Built-In Java Annotations used in other annotations

@Target

@Retention

@Inherited

@Documented

Understanding Built-In Annotations

@Override

@Override annotation assures that the subclass method is overriding the parent class method. If it is not so, compile time error occurs.

Sometimes, we do the silly mistake such as spelling mistakes etc. So, it is better to mark @Override annotation that provides assurance that method is overridden.

```
class Animal{
void eatSomething(){System.out.println("eating something");}
}
class Dog extends Animal{
@Override
void eatsomething(){System.out.println("eating foods");} //should be eatSomething
}
```

```
Class TestAnnotation1{
public static void main(String args[]){
Animal a=new Dog();
a.eatSomething();
}}
```

Output:Compile Time Error

@SuppressWarnings

@SuppressWarnings annotation: is used to suppress warnings issued by the compiler.

```
import java.util.*;
class TestAnnotation2{
@Override
public static void main(String args[]){
ArrayList list=new ArrayList();
list.add("sonoo");
list.add("vimal");
list.add("ratan");
for(Object obj:list)
```



```
System.out.println(obj);
}}
```

Now no warning at compile time.

If you remove the `@SuppressWarnings("unchecked")` annotation, it will show warning at compile time because we are using non-generic collection.

`@Deprecated`

`@Deprecated` annotation marks that this method is deprecated so compiler prints warning. It informs user that it may be removed in the future versions. So, it is better not to use such methods.

```
class A{
void m(){System.out.println("hello m");}
    @Deprecated
void n(){System.out.println("hello n");}
}
class TestAnnotation3{
public static void main(String args[]){
    A a=new A();
a.n();
}}
```

At Compile Time:

Note: Test.java uses or overrides a deprecated API.

Note: Recompile with `-Xlint:deprecation` for details.

At Runtime:

hello n

Java Custom Annotations

Java Custom annotations or Java User-defined annotations are easy to create and use. The `@interface` element is used to declare an annotation. For example:

```
@interface MyAnnotation{ }
```

Here, `MyAnnotation` is the custom annotation name.

Points to remember for java custom annotation signature

There are few points that should be remembered by the programmer.

Method should not have any throws clauses

Method should return one of the following: primitive data types, String, Class, enum or array of these data types.

Method should not have any parameter.

We should attach `@` just before interface keyword to define annotation.

It may assign a default value to the method.

Types of Annotation

There are three types of annotations.

- Marker Annotation
- Single-Value Annotation
- Multi-Value Annotation

1) Marker Annotation

An annotation that has no method, is called marker annotation. For example:

```
@interface MyAnnotation{ }
```

The `@Override` and `@Deprecated` are marker annotations.

2) Single-Value Annotation

An annotation that has one method, is called single-value annotation. For example:

```
@interface MyAnnotation{
int value();
}
```

```
}
```

We can provide the default value also. For example:

```
@interface MyAnnotation{
int value() default 0;
}
```

How to apply Single-Value Annotation

Let's see the code to apply the single value annotation.

```
@MyAnnotation(value=10)
```

The value can be anything.

3) Multi-Value Annotation

An annotation that has more than one method, is called Multi-Value annotation. For example:

```
@interface MyAnnotation{
int value1();
String value2();
String value3();
}
}
```

We can provide the default value also. For example:

```
@interface MyAnnotation{
int value1() default 1;
String value2() default "";
String value3() default "xyz";
}
}
```

How to apply Multi-Value Annotation

Let's see the code to apply the multi-value annotation.

```
@MyAnnotation(value1=10,value2="Arun Kumar",value3="Ghaziabad")
```

Built-in Annotations used in custom annotations in java

```
@Target
```

```
@Retention
```

```
@Inherited
```

```
@Documented
```

```
@Target
```

@Target tag is used to specify at which type, the annotation is used.

The java.lang.annotation.ElementType enum declares many constants to specify the type of element where annotation is to be applied such as TYPE, METHOD, FIELD etc. Let's see the constants of ElementType enum:

Element Types	Where the annotation can be applied
TYPE	class, interface or enumeration
FIELD	fields
METHOD	methods
CONSTRUCTOR	constructors
LOCAL_VARIABLE	local variables
ANNOTATION_TYPE	annotation type
PARAMETER	parameter

Example to specify annoation for a class

```
@Target(ElementType.TYPE)
```

```
@interface MyAnnotation{
```

```
int value1();
```

```
String value2();
```

```
}
```

Example to specify annotation for a class, methods or fields

```
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.METHOD})
```

```
@interface MyAnnotation{
```

```
int value1();
```

```
String value2();
```

```
}
```

```
@Retention
```

@Retention annotation is used to specify to what level annotation will be available.

RetentionPolicy

Availability

RetentionPolicy.SOURCE

refers to the source code, discarded during compilation.

It

will not be available in the compiled class.

RetentionPolicy.CLASS

refers to the .class file, available to java compiler but not

to

JVM . It is included in the class file.

RetentionPolicy.RUNTIME

refers to the runtime, available to java compiler and

JVM .

Example to specify the RetentionPolicy

```
@Retention(RetentionPolicy.RUNTIME)
```

```
@Target(ElementType.TYPE)
```

```
@interface MyAnnotation{
```

```
int value1();
```

```
String value2();
```

```
}
```

Example of custom annotation: creating, applying and accessing annotation

Let's see the simple example of creating, applying and accessing annotation.

File: Test.java

```
//Creating annotation
```

```
import java.lang.annotation.*;
```

```
import java.lang.reflect.*;
```

```
@Retention(RetentionPolicy.RUNTIME)
```

```
@Target(ElementType.METHOD)
```

```
@interface MyAnnotation{
```

```
int value();
```

```
}
```

```
//Applying annotation
```

```
class Hello{
```

```
@MyAnnotation(value=10)
```

```
public void sayHello(){System.out.println("hello annotation");}
```

```
}
```

```
//Accessing annotation
```

```
class TestCustomAnnotation1{
```

```
public static void main(String args[])throws Exception{
```

```
Hello h=new Hello();
```

```
Method m=h.getClass().getMethod("sayHello");
```

```
MyAnnotation manno=m.getAnnotation(MyAnnotation.class);
```

```
System.out.println("value is: "+manno.value());
}}
```

Output: value is: 10

download this example

How built-in annotations are used in real scenario?

In real scenario, java programmer only need to apply annotation. He/She doesn't need to create and access annotation. Creating and Accessing annotation is performed by the implementation provider. On behalf of the annotation, java compiler or JVM performs some additional operations.

@Inherited

By default, annotations are not inherited to subclasses. The **@Inherited** annotation marks the annotation to be inherited to subclasses.

@Inherited

```
@interface ForEveryone { }//Now it will be available to subclass also
```

```
@interface ForEveryone { }
```

```
class Superclass{ }
```

```
class Subclass extends Superclass{ }
```

@Documented

The **@Documented** Marks the annotation for inclusion in the documentation.

4.3 Lang Sub Package, Built-in Annotations

java.lang package which defines the core classes and interfaces that are indispensable to the Java platform and the Java programming language. It also covers more specialized subpackages:

java.lang.annotation: Defines the Annotation interface that all annotation types extend, and also defines meta-annotation types and related enumerated types. Added in Java 5.0.

java.lang.instrument: Provides support for Java-based "agents" that can instrument a Java program by transforming class files as they are loaded. Added in Java 5.0.

java.lang.management: Defines "management bean" interfaces for remote monitoring and management of a running Java interpreter.

java.lang.ref: Defines "reference" classes that are used to refer to objects without preventing the garbage collector from reclaiming those objects.

java.lang.reflect: Allows Java programs to examine the members of arbitrary classes, invoking methods, and querying and setting the value of fields.

Java Built-in Annotations

Java Built-in Annotations ship with core Java. First, there are several that inform compilation:

@Override

@SuppressWarnings

@Deprecated

@SafeVarargs

@FunctionalInterface

These annotations generate or suppress compiler warnings and errors. Applying them consistently is often a good practice since adding them can prevent future programmer error.

The **@Override** annotation is used to indicate that a method overrides or replaces the behavior of an

@SuppressWarnings indicates we want to ignore certain warnings from a part of the code.

The **@SafeVarargs** annotation also acts on a type of warning related to using varargs.

The **@Deprecated** annotation can be used to mark an API as not intended for use anymore.

For all these, you can find more detailed information in the articles linked.

@FunctionalInterface

Java 8 allows us to write code in a more functional way.

Single Abstract Method interfaces are a big part of this. If we intend a SAM interface to be used by lambdas, we can optionally mark it as such with @FunctionalInterface:

@FunctionalInterface

```
public interface Adder {
    int add(int a, int b);
}
```

Like @Override with methods, @FunctionalInterface declares our intentions with Adder.

Now, whether we use @FunctionalInterface or not, we can still use Adder in the same way:

```
Adder adder = (a,b) -> a + b;
```

```
int result = adder.add(4,5);
```

But, if we add a second method to Adder, then the compiler will complain:

@FunctionalInterface

```
public interface Adder {
    // compiler complains that the interface is not a SAM
    int add(int a, int b);
    int div(int a, int b);
}
```

Now, this would've compiled without the @FunctionalInterface annotation.

Like @Override, this annotation protects us against future programmer error. Even though it's legal to have more than one method on an interface, it isn't when that interface is being used as a lambda target. Without this annotation, the compiler would break in the dozens of places where Adder was used as a lambda. Now, it just breaks in Adder itself.

4.4 Java Beans, Introspection, Customizers, Creating a Java Bean

A JavaBean is a Java class that should follow the following conventions:

- It should have a no-arg constructor.
- It should be Serializable.
- It should provide methods to set and get the values of the properties, known as getter and setter methods.

Why use JavaBean?

According to Java white paper, it is a reusable software component. A bean encapsulates many objects into one object so that we can access this object from multiple places. Moreover, it provides easy maintenance.

Simple example of JavaBean class

```
//Employee.java
```

```
package mypack;
public class Employee implements java.io.Serializable{
    private int id;
    private String name;
    public Employee(){ }
    public void setId(int id){this.id=id;}
    public int getId(){return id;}
    public void setName(String name){this.name=name;}
    public String getName(){return name;}
}
```

How to access the JavaBean class?

To access the JavaBean class, we should use getter and setter methods.

```
package mypack;
```

```
public class Test{
public static void main(String args[]){
Employee e=new Employee();//object is created
e.setName("Arjun");//setting value to the object
System.out.println(e.getName());
}}
```

JavaBean Properties

A JavaBean property is a named feature that can be accessed by the user of the object. The feature can be of any Java data type, containing the classes that you define.

A JavaBean property may be read, write, read-only, or write-only. JavaBean features are accessed through two methods in the JavaBean's implementation class:

1. getPropertyname ()

For example, if the property name is firstName, the method name would be getFirstName() to read that property. This method is called the accessor.

2. setPropertyname ()

For example, if the property name is firstName, the method name would be setFirstName() to write that property. This method is called the mutator.

Advantages of JavaBean

The following are the advantages of JavaBean:/p>

- The JavaBean properties and methods can be exposed to another application.
- It provides an easiness to reuse the software components.

Disadvantages of JavaBean

- The following are the disadvantages of JavaBean:
- JavaBeans are mutable. So, it can't take advantages of immutable objects.
- Creating the setter and getter method for each property separately may lead to the boilerplate code.

Introspection

The introspection tool can be used to learn about the properties and operations provided by your class. BeanUtils package is depending on JavaBeans specification that determines the available properties for a particular bean class. The introspection mechanism can be customized from version 1.9.0 onwards and enables an application to alter or extend the default discovery of bean properties. You can achieve this by using the BeanIntrospector interface. By implementing this interface, we are able to process a specific target class and create its equivalent. PropertyDescriptor objects. By default, DefaultBeanIntrospector objects are used by BeanUtils for detecting properties that are matching with the JavaBeans specification.

You can extend the default discovery mechanism by using the PropertyUtils.addBeanIntrospector(BeanIntrospector) method of PropertyUtils. This custom BeanIntrospector can be called in the time of introspection of a class and adds the detected properties to the final result.

Introspection is an automatic process in which the bean's design patterns are analyzed to extract the bean's properties, events, and methods. Introspection has some great advantages as under:

- Portability.
- Re-usability.

The introspection API consists of the following classes from the java. Bean package:

- BeanDescriptor

- BeanInfo
- FeatureDescriptor
- EventSetDescriptor
- MethodDescriptor
- PropertyDescriptor
- IndexedPropertyDescriptor
- Introspector
- SimpleBeanInfo

Both reflection and introspection are important aspects of Java programming and should be used only when absolutely necessary. The following code snippet shows a combination of reflection and introspection:

Sample code showing introspection

```
package com.home.reflect;
import java.lang.reflect.Method;
public class DemoIntroSpect {
    public static void main(String[] args) {
        try {
            Class clazz = Class.forName("com.home.VO.EMPVO");
            Method [] methods = clazz.getMethods();
            for (Method method : methods) {
                System.out.println("Method Name = " +method.getName());
            }
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

Customization:

Customization provides the feature to visually modify the java beans properties as per the requirement of the situation. Once the beans are designed so that reflection and introspection can happen, we should test our code to check whether these features actually happen or not. E.g. let us consider a clock bean. The user should have the power to customize it to digital or analog based on some property. Some visual builders e.g. the netbeans IDE use property sheets in order to customize the beans.

A **JavaBean** is a specially constructed Java class written in the Java and coded according to the JavaBeans API specifications. Following are the unique characteristics that distinguish a JavaBean from other Java classes –

- It provides a default, no-argument constructor.
- It should be serializable and that which can implement the Serializable interface.
- It may have a number of properties which can be read or written.
- It may have a number of "getter" and "setter" methods for the properties.

JavaBeans Properties

A **JavaBean** property is a named attribute that can be accessed by the user of the object. The attribute can be of any Java data type, including the classes that you define. A **JavaBean** property may be read, write, read only, or write only. **JavaBean** properties are accessed through two methods in the **JavaBean**'s implementation class –

S.No.	Method & Description
1	getPropertyName()

For example, if property name is firstName, your method name would be getFirstName() to read that property. This method is called accessor.

2 setPropertyName()

For example, if property name is firstName, your method name would be setFirstName() to write that property. This method is called mutator.

A read-only attribute will have only a getPropertyName() method, and a write-only attribute will have only a setPropertyName() method.

JavaBeans Example

Consider a student class with few properties –

```
package com.tutorialspoint;
public class StudentsBean implements java.io.Serializable {
    private String firstName = null;
    private String lastName = null;
    private int age = 0;

    public StudentsBean() {
    }
    public String getFirstName(){
        return firstName;
    }
    public String getLastName(){
        return lastName;
    }
    public int getAge(){
        return age;
    }
    public void setFirstName(String firstName){
        this.firstName = firstName;
    }
    public void setLastName(String lastName){
        this.lastName = lastName;
    }
    public void setAge(Integer age){
        this.age = age;
    }
}
```

Accessing JavaBeans

The useBean action declares a JavaBean for use in a JSP. Once declared, the bean becomes a scripting variable that can be accessed by both scripting elements and other custom tags used in the JSP. The full syntax for the useBean tag is as follows –

```
<jsp:useBean id = "bean's name" scope = "bean's scope" typeSpec/>
```

Here values for the scope attribute can be a page, request, session or application based on your requirement. The value of the id attribute may be any value as long as it is a unique name among other useBean declarations in the same JSP.

Following example shows how to use the useBean action –

```
<html>
<head>
    <title>useBean Example</title>
</head>
```



```

<body>
  <jsp:useBean id = "date" class = "java.util.Date" />
  <p>The date/time is <%= date %>
</body>
</html>

```

You will receive the following result –

The date/time is Thu Sep 30 11:18:11 GST 2010

Accessing JavaBeans Properties

Along with <jsp:useBean...> action, you can use the <jsp:getProperty/> action to access the get methods and the <jsp:setProperty/> action to access the set methods. Here is the full syntax –

```

<jsp:useBean id = "id" class = "bean's class" scope = "bean's scope">
  <jsp:setProperty name = "bean's id" property = "property name"
    value = "value"/>
  <jsp:getProperty name = "bean's id" property = "property name"/>
  .....
</jsp:useBean>

```

The name attribute references the id of a JavaBean previously introduced to the JSP by the useBean action. The property attribute is the name of the get or the set methods that should be invoked.

Following example shows how to access the data using the above syntax –

```

<html>
  <head>
    <title>get and set properties Example</title>
  </head>
  <body>
    <jsp:useBean id = "students" class = "com.tutorialspoint.StudentsBean">
      <jsp:setProperty name = "students" property = "firstName" value = "Zara"/>
      <jsp:setProperty name = "students" property = "lastName" value = "Ali"/>
      <jsp:setProperty name = "students" property = "age" value = "10"/>
    </jsp:useBean>
    <p>Student First Name:
      <jsp:getProperty name = "students" property = "firstName"/>
    </p>
    <p>Student Last Name:
      <jsp:getProperty name = "students" property = "lastName"/>
    </p>
    <p>Student Age:
      <jsp:getProperty name = "students" property = "age"/>
    </p>
  </body>
</html>

```

Let us make the StudentsBean.class available in CLASSPATH. Access the above JSP. the following result will be displayed –

Student First Name: Zara

Student Last Name: Ali

Student Age: 10

4.5 Bean Manifest File, Creating a Bean JAR file

Example: This bean will just draw itself in red, and when you click it, it will display a count of the number of time it has been clicked. We can place this bean in the JDK's demo directory, so we create a directory named bean and store the class files for this bean in that directory.

In this bean class we can use Canvas to draw the bean itself. And add a mouse listener to the canvas to record mouse clicks and set the size of the canvas

```
package sunw.demo.bean;
import java.awt.*;
import java.awt.event.*;
public class bean extends Canvas {
    int count;
    public bean()
    {
        addMouseListener(new MouseAdapter()
        {
            public void mousePressed(MouseEvent me)
            {
                clicked();
            }
        });
        count = 0;
        setSize(200,100);
    }
    public void clicked()
    {
        count++;
        repaint();
    }
    public void paint(Graphics g)
    {
        g.setColor(Color.RED);
        g.fillRect(0,0,20,30);
        g.drawString("Click Count= "+count,50,50);
    }
}
```

Now this bean class has to put it into a JAR file and label it in that JAR file as a bean

Compile the bean.java file ; then bean.class file will be generated.

After creating the bean, create the manifest file for the bean class

Creating a Bean Manifest File

We use a manifest to indicate which classes are beans.

To indicate that a class in a JAR file is a Java Bean, you have to set its Java-Bean attribute to True.

bean.mft file

Name: sunw/demo/bean/bean.class

Java-Bean: True

Attributes:

Name: name of the bean class (full package and class name)

Java-Bean: true – indicate that a class in a JAR file is a Java Bean

Creating a Bean jar file:

To use a bean, you have to store the class file(s) and manifest file in a JAR file.

We can create a JAR file for this bean, i.e. bean.jar in the demo\jars directory located within the beans directory, where the beanbox will look for it.

Make sure you are in the demo directory and use the jar tool like
 jar cfm ../jars\bean.jar bean.mft sunw\demo\bean\bean.class
 jar utility at command line

Options

c – creating new JAR file

f – indicates archive filename

m – specify the manifest file

Bean is compressed and saved in the format of jar file which contains manifest file, class files, gif files, and other information of customization files

Using the New Bean:

After developing a new JavaBean and installing it in the demo\jars directory, we can open the beanbox to see this bean listed in the toolbox. When you draw the bean from the toolbox, the bean appears in the BeanBox window.

4.6 How to create new Bean, Adding controls to Beans, and setting bean properties

Creating a Simple bean consists of following steps.

Step 1: Put this source code into a file named "SimpleBean.java"

```
import java.awt.*;
import java.io.Serializable;
public class SimpleBean extends Canvas
    implements Serializable{

//Constructor sets inherited properties
    public SimpleBean(){
        setSize(60,40);
        setBackground(Color.red);
    }

}
```

What will the above code look like when displayed in the bean box?

Step 2: Compile the file:

```
javac SimpleBean.java
```

Step 3: Create a manifest file, named "manifest.tmp":

Name: SimpleBean.class

Java-Bean: True

Step 4: Create the JAR file, named "SimpleBean.jar":

```
jar cfm SimpleBean.jar manifest.tmp SimpleBean.class
```

Then verify that the content is correct by the command "jar tf SimpleBean.jar".

Step 5:

Start the Bean Box.

CD to c:\Program Files\BDK1.1\beanbox\.

Then type "run".

Load JAR into Bean Box by selecting "LoadJar..." under the File menu.

Step 6:

After the file selection dialog box is closed, change focus to the "ToolBox" window. You'll see "SimpleBean" appear at the bottom of the toolbox window.

Select SimpleBean.jar.

Cursor will change to a plus. In the middle BeanBox window, you can now click to drop in what will appear to be a colored rectangle.

Your screen should look like this:

Step 7:

Try changing the red box's color with the Properties windows.

Step 8:

Chose "Events" under the "Edit" menu in the middle window to see what events SimpleBean can send. These events are inherited from java.awt.Canvas.

As an exercise to try in class right now, let's add a colored rectangle inside of the red square. We'll add a property named "Color" and give it a setter and a getter method to change the interior rectangle from its default color of green. The needed code is shown below, so you just have to go through the steps shown above to make it show up in the Bean Box!

```
import java.awt.*;
import java.io.Serializable;

public class MediumBean extends Canvas
    implements Serializable {

    private Color color = Color.green;

    //property getter method
    public Color getColor(){
        return color;
    }

    //Property setter method. Sets new inside color and repaints.
    public void setColor(Color newColor){
        color = newColor;
        repaint();
    }

    public void paint(Graphics g) {
        g.setColor(color);
        g.fillRect(20, 5, 20, 30);
    }

    //Constructor sets inherited properties
    public MediumBean(){
        setSize(60,40);
        setBackground(Color.red);
    }
}
```

Adding Controls to a Bean:

We can add a button to a bean and display the number of times it has been clicked. First base your bean on a class that has a container, such as the Panel class. This button.java file creates the panel, sets it size, and adds a button to it.

```
package sunw.demo.button;
import java.awt.*;
import java.awt.event.*;
public class button extends Panel implements ActionListener
```

```

{
    int count;
    Button button1;
    public button()
    {
        count=0;
        setSize(200,100);
        button1=new Button("Click me");
        button1.addActionListener(this);
        add(button1);
    }
    public void actionPerformed(ActionEvent e)
    {
        count++;
        repaint();
    }
    public void paint(Graphics g)
    {
        g.drawString("Click count= "+count,50,50);
    }
}

```

Setting Bean Properties

A bean property is a named attribute of a bean that can affect its behavior or appearance.

Examples of bean properties include color, label, font, font size, and display size.

Bean's properties (i.e appearance and behaviour) can be changed at design time.

Bean properties are private values and can be accessed through getter and setter methods

The names of these methods follow specific rules called design patterns.

These design pattern-based method names allow builder tools such as the NetBeans GUI Builder, to provide the following features:

- A builder tool can:
- discover a bean's properties
- determine the properties' read/write attribute
- locate an appropriate "property editor" for each type
- display the properties (in a sheet)
- alter the properties at design-time

We should inform to the Java framework about the properties of your beans by implementing the BeanInfo interface. Most beans don't implement the BeanInfo interface directly. Instead, they extend the SimpleBeanInfo class, which implements BeanInfo interface. To actually describe a property, we can use the PropertyDescriptor class, which in turn is derived from the FeatureDescriptor class. Let us take an example that implements a property in a Java bean. We will add a property named filled to the click-counting operation. This property is a boolean property that, when True, makes sure the bean will be filled in with color. To keep track of the new filled property, we will add a private boolean variable of that name to the bean2 class. We initialize this property to False when the bean is created. When we implement a property, java will look for two methods: getProperty and setProperty, where PropertyName is the name of the property. The get method returns the current value of the property, and the set method takes an argument of that type

```

package sunw.demo.bean2;
import java.awt.*;

```

```

import java.awt.event.*;
public class bean2 extends Canvas {
    private boolean filled ;
    int count;
    public bean2()
    {
        addMouseListener(new MouseAdapter()
        {
            public void mousePressed(MouseEvent me)
            {
                clicked();
            }
        });
        count = 0;
        filled = false;
        setSize(200,100);
    }
    public void clicked()
    {
        count++;
        repaint();
    }

    public boolean getFilled()
    {
        return filled;
    }
    public void setFilled(boolean flag)
    {
        this.filled = flag;
        repaint();
    }
    public void paint(Graphics g)
    {
        if (filled)
        {
            g.setColor(Color.RED);
            g.fillRect(20,5,20,30);
        }
        g.setColor(Color.WHITE);
        g.drawString("Click Count= "+count,50,50);
    }
}

```

Implementing a BeanInfo class for the property:

Using BeanInfo class you can expose the features of your bean to java framework

In BeanInfo class, the bean developer has to give the description about each property by using this PropertyDescriptor class.

The bean info class should extend the SimpleBeanInfo class which implements BeanInfo interface

Now we have to create a new class, bean2BeanInfo which will return information about this new bean property

In this BeanInfo class we should implement the getPropertyDescriptors methods, which returns an array of PropertyDescriptor objects.

Each PropertyDescriptor object holds the name of a property and point to the class that supports that property.

```
package sunw.demo.bean2;
import java.beans.*;
public class bean2BeanInfo extends SimpleBeanInfo
{
    public PropertyDescriptor[] getPropertyDescriptors()
    {
        try
        {
            PropertyDescriptor filled = new PropertyDescriptor("filled",
                                                                bean2.class);

            PropertyDescriptor pd[] = { filled };
            return pd;
        }
        catch(Exception e) { }
        return null;
    }
}
```

After compiling this new class(bean2BeanInfo.java), we place bean2BeanInfo.class file in the directory sunw\demo\bean2 along with the classes created when we compiled bean2.java

Create a new manifest file that includes the bean2BeanInfo class

bean2.mft file:

Name: sunw/demo/bean2/bean2BeanInfo.class

Name: sunw/demo/bean2/bean2.class

Java-bean : True

Place this new manifest file in the demo directory.

Create a new bean2.jar file and install it:

bean2.jar file:

C:\...\demo > jar cfm ... \jars\bean2.jar bean2.mft sunw\demo\bean2*.class

Now when we run the beanbox and add a new bean2 bean to the beanbox, the new filled property will appear in the properties window. Setting filled to True causes the bean to be filled with color.

4.7 Design Patterns for Properties, Simple Properties, and Design Pattern for Events

Design Patterns for Properties:The Beans can perform various functions, such as it can generate events and send them to other objects.

These can be identified by the following design patterns, where T is the type of the event:

```
public void addTListener(TListener eventListener);
public void removeTListener(TListener eventListener);
```

To add or remove a listener for the specified event, these methods are used.

For example, an event interface type is TemperatureListener, a Bean that monitors temperature might supply the following methods:

```
public void addTemperatureListener(TemperatureListener tl)
{ ... }
public void removeTemperatureListener(TemperatureListener tl)
{ ... }
```

Simple Properties

A simple property represents a single value and can be defined with a pair of get/set methods.

It can be identified by the following design patterns, where N is the name of the property and T is its type.

```
public T getN()
```

```
public void setN(T arg)
```

- A read/write property has both the getN() and setN() methods.
- A read-only property has only a getN() method,
- A write-only property has only a setN() method.

The following are 3 read/write simple properties along with their getter() and setter() methods:

```
private double depth,height,width;
public double getDepth() { return depth; }
public void setDepth(double dp) { depth = dp; }
public double getHeight() { return height; }
public void setHeight(double ht) { height = ht; }
public double getWidth() { return width; }
public void setWidth(double wd) { width = wd; }
```

Adding a Color Property to SimpleBean

```
public class SimpleBean extends Canvas implements Serializable
```

```
{
private Color color = Color.green; // Property name is clr
public SimpleBean() // default constructor
{
setSize(60,40);
setBackground(Color.red);
}
public Color getColor() // get property
{
return color;
}
```

```
public void setColor(Color c) // set property
{
color = c;
repaint();
}
public void paint(Graphics g){
g.setColor(color);
g.fillRect(20,5,20,30);
}
}
```

When you execute the bean, you will get following results:

SimpleBean will be displayed with a green centered rectangle.

The Properties window contains a new clr property

Indexed Properties:

- Indexed properties consist of multiple values.
- Property element get/set methods take an integer index as a parameter.
- The property may also support getting and setting the entire array at once.
- Multiple values are passed to and retrieved from the setter and getter methods respectively in case of the indexed properties.
- It can be identified by the following design pattern, N is the name of the property and T is its type

```
public void setN(int index, T value);
public T getN(int index);
```



```

public T[] getN();
public void setN(T values[]);
private double data[]; // data is an indexed property
public double getData(int index) //get one element of array
{
return data[index];
}
public void setData(int index, double x) //set one element of array
{
data[index]=x;
}
public double [] getData() // get entire array
{
return data;
}
public void setData(double[] x) // set entire array
{
data=x; }

```

Design Pattern for Events

The Beans can perform various functions, such as it can generate events and send them to other objects.

- These can be identified by the following design patterns, where T is the type of the event:

```

public void addTListener(TListener eventListener);
public void removeTListener(TListener eventListener);

```

- To add or remove a listener for the specified event, these methods are used.

- For example, an event interface type is TemperatureListener, a Bean that monitors temperature might supply the following methods:

```

public void addTemperatureListener(TemperatureListener tl)
{ ... }
public void removeTemperatureListener(TemperatureListener tl)
{ ... }

```

4.8 Creating Bound Properties, and giving a Bean Methods

Bound properties generate an event when their values change.

This event is of type PropertyChangeEvent and is sent to all registered event listeners of this type.

To make a property a bound property, use the setBound method like

```
PropertyName.setBound(true)
```

For example filled.setBound(true);

When bound property changes, an event of type PropertyChangeEvent, is generated and a notification is sent to interested listeners.

There is a standard listener class for this kind of event. Listeners need to implement this interface PropertyChangeListener

It has one method:

```
public void propertyChange(PropertyChangeEvent)
```

A class that handles this event must implement the PropertyChangeListener interface

Implement Bound Property in a Bean

Declare and instantiate a PropertyChangeSupport object that provides the bulk of bound property's functionality,

```
private PropertyChangeSupport changes = new PropertyChangeSupport(this);
```

Implement registration and unregistration methods . The BeanBox will call these methods when a connection is made.

```
public void addPropertyChangeListener(PropertyChangeListener p )
{
changes.addPropertyChangeListener(p);
}
public void removePropertyChangeListener( PropertyChangeListener p)
{
changes.removePropertyChangeListener(p);
}
```

Send change event to listeners when property is changed. i.e each bound property must invoke the firePropertyChange() method from its set() method:

PropertyChangeSupport object handles the notification of all registered target.

The method firePropertyChange() must provide the property name, as well as the old and new values

```
public void setX(int new)
{
int old = x;
x = new;
changes.firePropertyChange("x", old, new);
}
```

The listener (target object) must provide a propertyChange() method to receive the property-related notifications:

```
public void propertyChange(PropertyChangeEvent e) {
// ...
}
```

Giving a Bean Methods

We can declare a method in a JavaBean, which can be called by other beans.

Any public bean method is accessible from other beans.

We can describe the methods of JavaBeans to the Java framework by using the MethodDescriptor class.

Constructor Summary

MethodDescriptor(Method method)

Constructs a MethodDescriptor from a Method.

MethodDescriptor(Method method, ParameterDescriptor[] parameterDescriptors)

Constructs a MethodDescriptor from a Method providing descriptive information for each of the method's parameters.

Method Summary:

Method getMethod() - Gets the method that this MethodDescriptor encapsualtes.

ParameterDescriptor[] getParameterDescriptors() - Returns the ParameterDescriptor for each of the parameters of this MethodDescriptor's methods.

This Bean3 will count the number of times it has been clicked and will also support a method named increment that, when invoked, will increment the click count.

```
public class bean3 extends Canvas {
int count;
public bean3()
{ addMouseListener(new MouseAdapter()
{ public void mousePressed(MouseEvent me)
{
clicked();
}
```

```

}
});
count = 0;
setSize(200,100);
}
public void clicked()
{
count++;
repaint();
}
public void increment() /*public bean method is
accessible from other beans */
{
count++;
repaint();
}
public void paint(Graphics g)
{
g.setColor(Color.RED);
g.fillRect(0,0,20,30);
g.drawString("Click Count= "+count,50,50);
}

```

After creating bean3 and adding it to the beanbox, we can connect other beans to the increment methods. We can connect a button to that method, and each time the button is clicked, the click count in the bean3 is incremented and displayed. •In order to make available the methods defined in one bean to another bean, we have to create a BeanInfo class and implement the method called getMethodDescriptors()

```

public MethodDescriptor[] getMethodDescriptors()
{
MethodDescriptor m1 = new MethodDescriptor("increment");
MethodDescriptor md[] = {m1 };
return md;
}

```

Constructors of the PropertyDescriptor Class:

PropertyDescriptor(String propertyName, Class beanClass)

Constructs a property descriptor for a property

PropertyDescriptor(String propertyName, Class beanClass, String getterName, String setterName)

This constructor takes the name of a simple property, and method names for reading and writing the property.

PropertyDescriptor(String propertyName, Method getter, Method setter)

This constructor takes the name of a simple property, and method objects for reading and writing the property

Methods of the PropertyDescriptor Class:

Class getPropertyEditorClass() - Returns any explicit PropertyEditor class that has been registered for this property

Class getPropertyType() - Returns the Class object for the property

Method getReadMethod() – Returns the method that should be used to read the property value

Method `getWriteMethod()` – Returns the method that should be used to write the property value
 boolean `isBound()` - Returns true if this is a bound property
 boolean `isConstrained()` - Returns true if this is a constrained property
 void `setBound(boolean bound)` - Sets the bound property of this object. Updates to bound properties cause a `PropertyChange` event to be fired when the property is changed.
 void `setConstrained(boolean constrained)` – Sets the constrained property of this object.
 void `setReadMethod(Method getter)` – Sets the method that is used to read the property value
 void `setWriteMethod(Method setter)` – Sets the method that is used to write the property value
 void `setPropertyEditorClass(Class propertyEditorClass)` – Returns the class for the desired `PropertyEditor`. Property editors are found by using the property editor manager

4.9 Bean an Icon, and creating a Bean Info Class

You can add your own icons to your beans by adding a `getIcon` method to the `BeanInfo` class.

- Implement this method and handle all possibilities like monochrome or color icons of either 16X16 or 32X32

```
public Image getIcon(int iconkind)
{
    if (iconkind == BeanInfo.ICON_MONO_16X16 || iconkind ==
        BeanInfo.ICON_COLOR_16X16)
    {
        Image image = loadImage("Icon16.gif");
        return image;
    }
    if (iconkind == BeanInfo.ICON_MONO_32X32 || iconkind ==
        BeanInfo.ICON_COLOR_32X32)
    {
        Image image = loadImage("Icon32.gif");
        return image;
    }
    return null;
}
```

Using the BeanInfo Interface

- The role of `BeanInfo` interface is to enable you to explicitly control what information is available.
- To create a `BeanInfo` class, you should extend `SimpleBeanInfo` class which implements `BeanInfo` interface
- The `BeanInfo` interface defines several methods. Like
`PropertyDescriptor[] getPropertyDescriptors()`
`EventSetDescriptor[] getEventSetDescriptors()`
`MethodDescriptor[] getMethodDescriptors()`

Here, these array of objects provide information about the properties, events, and methods of a Bean. By implementing these methods, a developer can designate exactly what is presented to a user. At the time of creating a class that implements `BeanInfo`, you have to remember that you must call that class `bnameBeanInfo`, where `bname` is the name of the Bean. For example, if the Bean is called as `MyBean` then the information class must be called `MyBeanBeanInfo`

Feature Descriptors

`BeanInfo` classes contain descriptors which describe the target Bean's features.

- FeatureDescriptor- It is the base class for the other descriptor classes. It declares the aspects that are common to all descriptor types.
- PropertyDescriptor – It describes the target Bean's properties
- IndexedPropertyDescriptor – It describes the target Bean's indexed properties
- EventSetDescriptor- It describe the events the target Bean fires
- MethodDescriptor- It describes the target Bean's methods.
- ParameterDescriptor – It describes the method parameters

Creating a BeanInfo Class

To create a BeanInfo class, first name your BeanInfo class. According to naming convention, you must add "BeanInfo" to the target class name. If the target class name is ExplicitButton then its bean information class name should be ExplicitButtonBeanInfo. By extending SimpleBeanInfo class you can override only those methods which returns the properties, methods and events

```
public class ExplicitButtonBeanInfo extends SimpleBeanInfo
{
    public PropertyDescriptor[] getPropertyDescriptors()
    {
        PropertyDescriptor background = new PropertyDescriptor("background",
        "ExplicitButton.class");
        PropertyDescriptor foreground = new PropertyDescriptor("foreground",
        "ExplicitButton.class");
        PropertyDescriptor font = new PropertyDescriptor("font", "ExplicitButton.class");
        PropertyDescriptor label= new PropertyDescriptor("label",
        "ExplicitButton.class");
        PropertyDescriptor pd[] = { background,foreground,font,label};
        return pd;
    }
}
```

4.10 Persistence and JavaBeans API

Persistence

The ability to store state of a component in persistent storage is called persistence. A bean needs to implement the interface java.io.Serializable to support the persistence. The Serializable interface does not prescribe the implementation of any methods, but is an approval, that the bean may be saved in persistent storage as in a file or database. In doing so, the bean can be restored after an application was shut down or transferred across networks. To make fields in a Bean class persistent, simply define the class as implementing java.io.Serializable.

```
public class Button implements Serializable
{ .....//.....}
```

You can prevent selected fields from being saved by marking them transient or static; since transient and static variables are not saved. Alternatively, a component can be stored in a customized manner (e.g. in xml format) by implementing the Externalizable interface.

The Java Beans API (java.beans package)

The JavaBeans API contains classes and interfaces that enable a Java developer to work with beans in a Java program. The classes, interfaces and methods of the JavaBeans API are provided in the java.beans package. With the JavaBeans API you can create reusable, platform-independent components. Using JavaBeans-compliant application builder tools, you

can combine these components into applets, applications, or composite components. (Note: Sun NetBeans, BDK, Visual Café, JBuilder, Visual Age are the bean builder tools)

List of Interfaces in java.beans package

BeanInfo: Any class that implements this BeanInfo interface provides explicit information about the methods, properties, events, etc, of their bean.

Customizer: A customizer class provides a complete custom GUI for customizing a target Java Bean.

ExceptionListener: An ExceptionListener is notified of internal exceptions.

PropertyChangeListener: A "PropertyChange" event gets fired whenever a bean changes a "bound" property.

PropertyEditor: This interface provides the methods that is used to edit the property

VetoableChangeListener: A VetoableChange event gets fired whenever a bean changes a "constrained" property.

Visibility: This interface determines that bean needs GUI or not and whether GUI is available.

AppletInitializer: This interface is used to initialize the Applet with java Bean

DesignMode: This interface is used to define the notation of design time as a mode in which JavaBean instances should function

List of Classes in java.beans package

BeanDescriptor: A BeanDescriptor provides the complete information about the bean

Beans: This class provides some general purpose beans basic methods.

EventHandler: The EventHandler class is used by developers to make connections between user interface bean(source) and an application logic bean(target)

EventSetDescriptor: An EventSetDescriptor describes a group of events that a given Java bean fires.

FeatureDescriptor: It is the baseclass for PropertyDescriptor, EventSetDescriptor, and MethodDescriptor, etc.

IndexedPropertyChangeEvent: It gets delivered whenever a component of JavaBeans changes a bound indexed property.

IndexedPropertyDescriptor: It describes a property that acts like an array having index read or write method of the array.

Introspector: It class provides a standard way for tools to learn about the properties, events, and methods supported by a target Java Bean.

MethodDescriptor: A MethodDescriptor describes a particular method that a Java Bean supports for external access from other components.

ParameterDescriptor: This class allows bean implementors to provide additional information on each of their parameters

PropertyChangeEvent: A "PropertyChange" event gets delivered whenever a bean changes a "bound" or "constrained" property.

PropertyChangeSupport: This is a utility class that can be used by beans that support bound properties.

PropertyDescriptor: A PropertyDescriptor describes one property that a Java Bean exports via a pair of accessor methods.

PropertyEditorManager: The PropertyEditorManager can be used to locate a property editor for any given type name.

PropertyEditorSupport: This is a support class which helps to build property editors.

SimpleBeanInfo: This is a support class to make it easier for people to provide BeanInfo classes.

VetoableChangeSupport: This is a utility class that can be used by beans that support constrained properties.

4.11 Assignment-4

Short Answer Questions
Knowledge/Remembering Level Questions

1. What are the two ways in which we can reuse the code already created?
2. To create a package what we have to do in source file?
3. Which specifies the visibility of a data field, method or class?
4. What are the two elements of an interface?
5. Which is the compressed version of the normal file?
6. List out any two types of Annotations.
7. Which defines the core classes and interfaces that is indispensable to the Java platform and the Java programming language?
8. What are the two methods used to access the JavaBean class?
9. Which tool can be used to learn about the properties and operations provided by a class?
10. Could you mention which provides the feature to visually modify the java beans properties as per the requirement of the situation?
11. Which is a named attribute of a bean that can affect its behavior or appearance?
12. Could you mention how you can expose the features of your bean to java framework?
13. What are the two types of properties for a bean?
14. Which type of bean property consists of multiple values?
15. Which bean property generates an event when their values change?

Long Answer Questions
Understanding Level Questions

13. Explain the need and importance of a Package.
14. Explain how to create a package with an example
15. Write a short note on Interfaces.
16. Explain how to create a JAR File.
17. Discuss on Java Annotations.
18. Explain How to create a Java Bean with an example.

Application Level Questions

19. Differentiate Class and Interface. Also create your own program by applying the concept of package.
20. How would you explain the concept Adding controls to Beans, and setting bean properties?
21. How would you explain Simple Properties and Design Pattern for Events by applying java bean concept?
22. Implement the concept of Creating Bound Properties in Java Beans with your own example.
23. With your own program implement the concept of creating Bound Properties and Creating Bean Info class.

Skill Level Question

24. Create your own program to demonstrate java bean properties and also to access java bean and its properties.

UNIT-V

INTRODUCTION TO SPRING FRAMEWORK

Introduction to Spring

In its early days, spring was created as an alternative to heavier enterprise Java technologies, especially EJB. Spring offered a lighter and leaner programming model as compared to Enterprise Java Beans (EJB). It empowered plain old Java objects (POJOs) with powers previously only available using EJB and other enterprise Java specifications. Over time, EJB and the Java 2 Enterprise Edition (J2EE) evolved. EJB started offering a simple POJO-oriented programming model of its own. Now EJB employs ideas such as dependency injection (DI) and aspect-oriented programming (AOP), arguably inspired by the success of spring. Although J2EE (now known as JEE) was able to catch up with Spring, Spring never stopped moving forward. Spring has continued to progress in areas where, even now, JEE is just starting to explore or isn't innovating at all. Mobile development, social API integration, NoSQL databases, cloud computing, and big data are just a few areas where Spring has been and is innovating. And the future continues to look bright for Spring.

5.1 Springing into Action - Simplifying Java development

Spring is an open source framework, originally created by Rod Johnson and described in his book *Expert One-on-One: J2EE Design and Development*. Spring was created to address the complexity of enterprise application development and makes it possible to use plain-vanilla JavaBeans to achieve things that were previously only possible with EJB. But Spring's usefulness isn't limited to server-side development. Any Java application can benefit from Spring in terms of simplicity, testability, and loose coupling. A bean by any other name... Although Spring uses the words *bean* and *JavaBean* liberally when referring to application components, this doesn't mean a Spring component must follow the JavaBeans specification to the letter. A Spring component can be any type of POJO. Spring does many things. But at the root of almost everything Spring provides are a few foundational ideas, all focused on Spring's fundamental mission: *Spring simplifies Java development*. A lot of frameworks claim to simplify something or other. But Spring aims to simplify the broad subject of Java development. This begs for more explanation. How does Spring simplify Java development?

- To back up its attack on Java complexity, Spring employs four key strategies:
- Lightweight and minimally invasive development with POJOs
- Loose coupling through DI and interface orientation
- Declarative programming through aspects and common conventions
- Eliminating boilerplate code with aspects and templates

Almost everything Spring does can be traced back to one or more of these four strategies. Let's start with seeing how Spring remains minimally invasive by encouraging POJO-oriented development.

Unleashing the power of POJOs

If you've been doing Java development for long, you've probably seen (and may have even worked with) frameworks that lock you in by forcing you to extend one of their classes or implement one of their interfaces. The easy-target example of such an invasive programming model was EJB 2-era stateless session beans. But even though early EJBs were such an easy target, invasive programming could easily be found in earlier versions of Struts, WebWork, Tapestry, and countless other Java specifications and frameworks. Spring avoids (as much as possible) littering your application code with its API. Spring almost never forces you to implement a Spring-specific interface or extend a

Spring-specific class. Instead, the classes in a Spring-based application often have no indication that they're being used by Spring. At worst, a class may be annotated with one of Spring's annotations, but it's otherwise a POJO.

To illustrate, consider the HelloWorldBean class shown in the following listing.

```
package com.habuma.spring;
public class HelloWorldBean {
    public String sayHello() {
        return "Hello World";
    }
}
```

As you can see, this is a simple, garden-variety Java class—a POJO. Nothing special about it indicates that it's a Spring component. Spring's non-invasive programming model means this class could function equally well in a Spring application as it could in a non-Spring application.

Despite their simple form, POJOs can be powerful. One of the ways Spring empowers POJOs is by assembling them using DI. Let's see how DI can help keep application objects decoupled from each other.

Injecting dependencies

The phrase dependency injection may sound intimidating, conjuring up notions of a complex programming technique or design pattern. But as it turns out, DI isn't nearly as complex as it sounds. By applying DI in your projects, you'll find that your code will become significantly simpler, easier to understand, and easier to test.

How DI works

Any nontrivial application (pretty much anything more complex than a Hello World example) is made up of two or more classes that collaborate with each other to perform some business logic. Traditionally, each object is responsible for obtaining its own references to the objects it collaborates with (its dependencies). This can lead to highly coupled and hard-to-test code. Coupling is a two-headed beast. On the one hand, tightly coupled code is difficult to test, difficult to reuse, and difficult to understand, and it typically exhibits “whack-a-mole” bug behavior (fixing one bug results in the creation of one or more new bugs). On the other hand, a certain amount of coupling is necessary—completely uncoupled code doesn't do anything. In order to do anything useful, classes need to know about each other somehow. Coupling is necessary but should be carefully managed. With DI, objects are given their dependencies at creation time by some third party that coordinates each object in the system. Objects aren't expected to create or obtain their dependencies. The dependencies are injected into the objects that need them. The act of creating associations between application components is commonly referred to as *wiring*. In Spring, there are many ways to wire components together, but a common approach has always been via XML. In a Spring application, an *application context* loads bean definitions and wires them together. The Spring application context is fully responsible for the creation of and wiring of the objects that make up the application. Spring comes with several implementations of its application context, each primarily differing only in how it loads its configuration..

Applying aspect

Although DI makes it possible to tie software components together loosely, aspect-oriented programming (AOP) enables you to capture functionality that's used throughout your application in reusable components. AOP is often defined as a technique that promotes separation of concerns in a software system. Systems are composed of several components, each responsible for a specific piece of functionality. But often these components also carry

additional responsibilities beyond their core functionality. System services such as logging, transaction management, and security often find their way into components whose core responsibilities is something else. These system services are commonly referred to as *cross-cutting concerns* because they tend to cut across multiple components in a system. By spreading these concerns across multiple components, you introduce two levels of complexity to your code:

The code that implements the system-wide concerns is duplicated across multiple components. This means that if you need to change how those concerns work, you'll need to visit multiple components. Even if you've abstracted the concern to a separate module so that the impact to your components is a single method call, that method call is duplicated in multiple places.

Your components are littered with code that isn't aligned with their core functionality. A method that adds an entry to an address book should only be concerned with how to add the address and not with whether it's secure or transactional.

AOP makes it possible to modularize these services and then apply them declaratively to the components they should affect. This results in components that are more cohesive and that focus on their own specific concerns, completely ignorant of any system Services that may be involved. In short, aspects ensure that POJOs remain plain.

Eliminating boilerplate code with templates

The *boilerplate code*—the code that you often have to write over and over again to accomplish common and otherwise simple tasks. Unfortunately, there are a lot of places where Java APIs involve a bunch of boilerplate code. A common example of boilerplate code can be seen when working with JDBC to query data from a database. If you've ever worked with JDBC, you've probably written something similar to the following.

```
public Employee getEmployeeById(long id) {
    Connection conn = null;
    PreparedStatement stmt = null;
    ResultSet rs = null;
    try {
        conn = dataSource.getConnection();
        stmt = conn.prepareStatement(
            "select id, firstname, lastname, salary from " + "employee where id=?");
        stmt.setLong(1, id);
        rs = stmt.executeQuery();
        Employee employee = null;
        if (rs.next()) {
            employee = new Employee();
            employee.setId(rs.getLong("id"));
            employee.setFirstName(rs.getString("firstname"));
            employee.setLastName(rs.getString("lastname"));
            employee.setSalary(rs.getBigDecimal("salary"));
        }
        return employee;
    } catch (SQLException e) {
    } finally {
        if(rs != null) {
            try {
                rs.close();
            } catch (SQLException e) {}
        }
    }
}
```

```

    }
    if(stmt != null) {
    try {
    stmt.close();
    } catch(SQLException e) {}
    }
    if(conn != null) {
    try {
    conn.close();
    } catch(SQLException e) {}
    }
    }
    return null;
    }

```

As you can see, this JDBC code queries the database for an employee's name and salary. But I'll bet you had to look hard to see that. That's because the small bit of code that's specific to querying for an employee is buried in a heap of JDBC ceremony. You first have to create a connection, then create a statement, and finally query for the results. And, to appease JDBC's anger, you must catch `SQLException`, a checked exception, even though there's not a lot you can do if it's thrown. Finally, after all is said and done, you have to clean up the mess, closing down the connection, statement, and result set. This could also stir JDBC's anger, so you must catch `SQLException` here as well. What's most notable about listing 1.12 is that much of it is the exact same code you'd write for pretty much any JDBC operation. Little of it has anything to do with querying for an employee, and much of it is JDBC boilerplate. JDBC is not alone in the boilerplate code business. Many activities require similar boilerplate code. JMS, JNDI, and the consumption of REST services often involve a lot of commonly repeated code. Spring seeks to eliminate boilerplate code by encapsulating it in templates. Spring's `JdbcTemplate` makes it possible to perform database operations without all the ceremony required by traditional JDBC.

For example, using Spring's `SimpleJdbcTemplate` (a specialization of `JdbcTemplate` that takes advantage of Java 5 features), the `getEmployeeById()` method can be rewritten so that its focus is on the task of retrieving employee data and not catering to the demands of the JDBC API. The following shows what such an updated `getEmployeeById()` method might look like.

```

public Employee getEmployeeById(long id) {
    return jdbcTemplate.queryForObject(
        "select id, firstname, lastname, salary " +
        "from employee where id=?",
        new RowMapper<Employee>() {
            public Employee mapRow(ResultSet rs,
                int rowNum) throws SQLException {
                Employee employee = new Employee();
                employee.setId(rs.getLong("id"));
                employee.setFirstName(rs.getString("firstname"));
                employee.setLastName(rs.getString("lastname"));
                employee.setSalary(rs.getBigDecimal("salary"));
                return employee;
            }
        },

```

```
id);
}
```

As you can see, this new version of `getEmployeeById()` is much simpler and acutely focused on selecting an employee from the database. The template's `queryForObject()` method is given the SQL query, a `RowMapper` (for mapping result set data to a domain object), and zero or more query parameters. What you don't see in `getEmployeeById()` is any of the JDBC boilerplate from before. Everything is handled inside the template.

5.2 Containing your Beans

In a Spring-based application, your application objects live in the Spring *container*. As illustrated in figure 5.1, the container creates the objects, wires them together, configures them, and manages their complete lifecycle from cradle to grave (or new to `finalize()`, as the case may be).

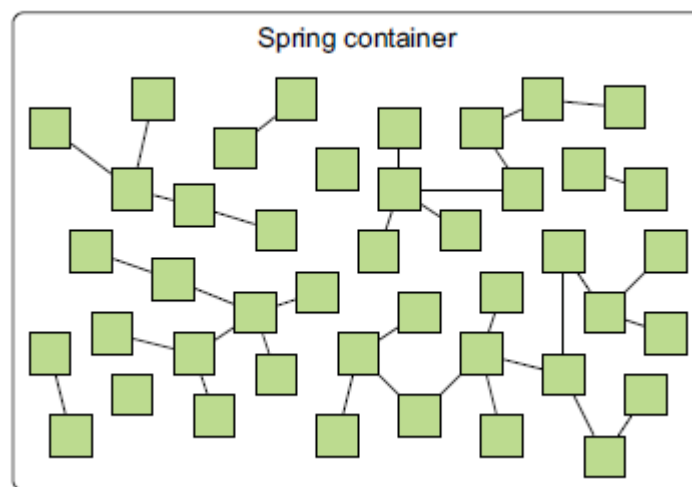


Figure 5.1: In a Spring application, objects are created, are wired together, and live in the Spring container.

First, though, it's important to get to know the container where your objects will be hanging out. Understanding the container will help you grasp how your objects will be managed. The container is at the core of the Spring Framework. Spring's container uses DI to manage the components that make up an application. This includes creating associations between collaborating components. As such, these objects are cleaner and easier to understand, they support reuse, and they're easy to unit test. There's no single Spring container. Spring comes with several container implementations that can be categorized into two distinct types. *Bean factories* (defined by the `org.springframework.beans.factory.BeanFactory` interface) are the simplest of containers, providing basic support for DI. *Application contexts* (defined by the `org.springframework.context.ApplicationContext` interface) build on the notion of a bean factory by providing application-framework services, such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners. Although it's possible to work with Spring using either bean factories or application contexts, bean factories are often too low-level for most applications. Therefore, application contexts are preferred over bean factories.

Working with an application context

Spring comes with several flavors of application context. Here are a few that you'll most likely encounter:

- AnnotationConfigApplicationContext—Loads a Spring application context from one or more Java-based configuration classes
- AnnotationConfigWebApplicationContext—Loads a Spring web application context from one or more Java-based configuration classes
- ClassPathXmlApplicationContext—Loads a context definition from one or more XML files located in the classpath, treating context-definition files as classpath resources
- FileSystemXmlApplicationContext—Loads a context definition from one or more XML files in the filesystem
- XmlWebApplicationContext—Loads context definitions from one or more XML files contained in a web application

let's load the application context from the filesystem using `FileSystemXmlApplicationContext` or from the classpath using `ClassPathXmlApplicationContext`.

Loading an application context from the filesystem or from the classpath is similar to how you load beans into a bean factory. For example, here's how you'd load a `File-`

`SystemXmlApplicationContext`: `ApplicationContext context = new`

`FileSystemXmlApplicationContext("c:/knight.xml");`

Similarly, you can load an application context from the application's classpath using

`ClassPathXmlApplicationContext`:

`ApplicationContext context = new`

`ClassPathXmlApplicationContext("knight.xml");`

The difference between using `FileSystemXmlApplicationContext` and `ClassPathXml-`

`ApplicationContext` is that `FileSystemXmlApplicationContext` looks for `knight.xml` in a specific location within the filesystem, whereas `ClassPathXmlApplicationContext` looks for `knight.xml` anywhere in the classpath (including JAR files).

Alternatively, if you'd rather load your application context from a Java configuration, you can use `AnnotationConfigApplicationContext`:

`ApplicationContext context = new AnnotationConfigApplicationContext(`

`com.springinaction.knights.config.KnightConfig.class);`

Instead of specifying an XML file from which to load the Spring application context,

`AnnotationConfigApplicationContext` has been given a configuration class from which to load beans. With an application context in hand, you can retrieve beans from the Spring container by calling the context's `getBean()` method.

A bean's life

In a traditional Java application, the lifecycle of a bean is simple. Java's `new` keyword is used to instantiate the bean, and it's ready to use. Once the bean is no longer in use, it's eligible for garbage collection and eventually goes to the big bit bucket in the sky. In contrast, the lifecycle of a bean in a Spring container is more elaborate. It's important to understand the lifecycle of a Spring bean, because you may want to take advantage of some of the opportunities that Spring offers to customize how a bean is created. Figure 5.2 shows the startup lifecycle of a typical bean as it's loaded into a Spring application context.

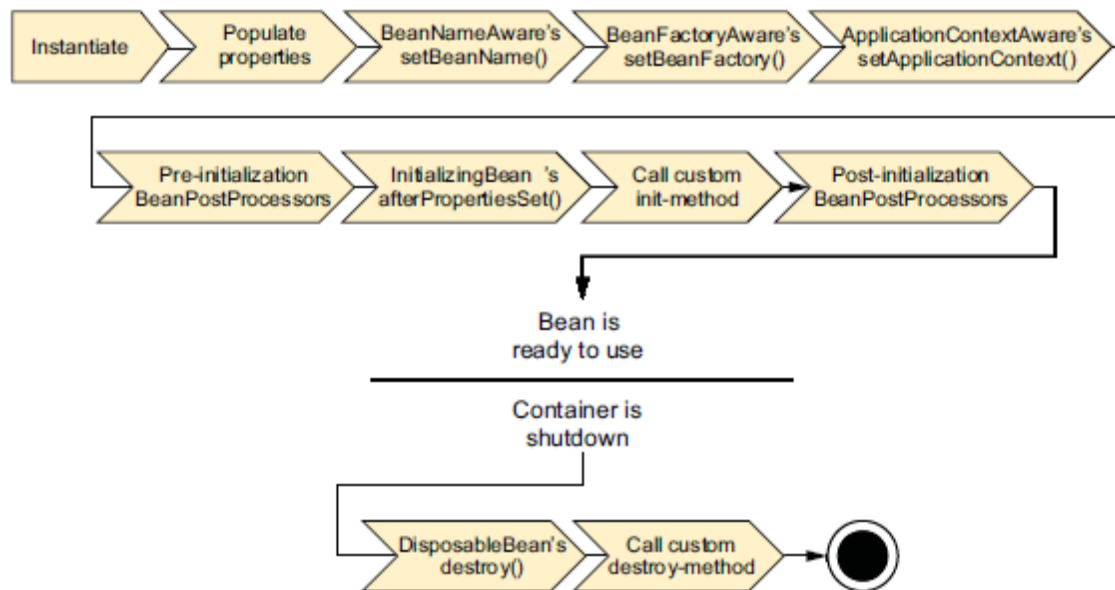


Figure 5.2: A bean goes through several steps between creation and destruction in the Spring container. Each step is an opportunity to customize how the bean is managed in Spring.

As you can see, a bean factory performs several setup steps before a bean is ready to use. Let's break down figure 5.2 in more detail:

1. Spring instantiates the bean.
2. Spring injects values and bean references into the bean's properties.
3. If the bean implements `BeanNameAware`, Spring passes the bean's ID to the `setName()` method.
4. If the bean implements `BeanFactoryAware`, Spring calls the `setBeanFactory()` method, passing in the bean factory itself.
5. If the bean implements `ApplicationContextAware`, Spring calls the `setApplicationContext()` method, passing in a reference to the enclosing application context.
6. If the bean implements the `BeanPostProcessor` interface, Spring calls its `postProcessBeforeInitialization()` method.
7. If the bean implements the `InitializingBean` interface, Spring calls its `afterPropertiesSet()` method. Similarly, if the bean was declared with an `initMethod`, then the specified initialization method is called.
8. If the bean implements `BeanPostProcessor`, Spring calls its `postProcessAfterInitialization()` method.
9. At this point, the bean is ready to be used by the application and remains in the application context until the application context is destroyed.
10. If the bean implements the `DisposableBean` interface, Spring calls its `destroy()` method. Likewise, if the bean was declared with a `destroyMethod`, the specified method is called.

Now you know how to create and load a Spring container. But an empty container isn't much good by itself; it doesn't contain anything unless you put something in it. To achieve the benefits of Spring DI, you must wire your application objects into the Spring container.

5.3 Surveying the Spring landscape

Spring Framework is focused on simplifying enterprise Java development through DI, AOP, and boilerplate reduction. Even if that were all Spring did, it'd be worth using. But there's more to Spring than meets the eye. Within the Spring Framework proper, you'll find several ways that Spring can ease Java development. But beyond the Spring Framework is a greater ecosystem of projects that build on the core framework, extending Spring into areas such as web services, REST, mobile, and NoSQL. Let's first break down the core Spring Framework to see what it brings to the table. Then we'll expand our sights to review the other members of the greater Spring portfolio.

Spring modules

When you download the Spring distribution and dig into its `libs` folder, you'll find several JAR files. As of Spring 4.0, there are 20 distinct modules in the Spring Framework distribution, with three JAR files for each module (the binary class library, the source JAR file, and a JavaDoc JAR file). The complete list of library JAR files. These modules can be arranged into six categories of functionality, as illustrated in figure 5.3. Taken as a whole, these modules give you everything you need to develop enterprise-ready applications. But you don't have to base your application fully on the Spring Framework. You're free to choose the modules that suit your application and look to other options when Spring doesn't fit the bill. Spring even offers integration points with several other frameworks and libraries so that you don't have to write them yourself. Let's look at each of Spring's modules, one at a time, to see how each fits in the overall Spring picture

CORE SPRING CONTAINER

The centerpiece of the Spring Framework is a container that manages how the beans in a Spring-enabled application are created, configured, and managed. In this module is the Spring bean factory, which is the portion of Spring that provides DI. Building on the bean factory, you'll find several implementations of Spring's application context, each of which provides a different way to configure Spring. In addition to the bean factory and application context, this module also supplies many enterprise services such as email, JNDI access, EJB integration, and scheduling. All of Spring's modules are built on top of the core container. You'll implicitly use these classes when you configure your application.

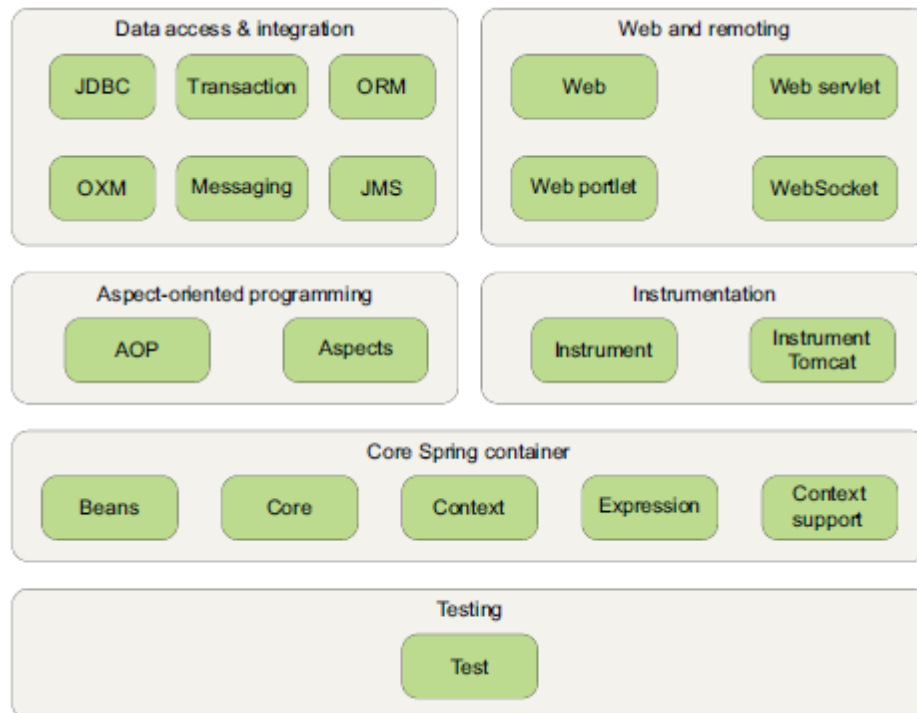


Figure 5.3: The Spring Framework is made up of six well-defined module categories.

SPRING'S AOP MODULE

Spring provides rich support for aspect-oriented programming in its AOP module. This module serves as the basis for developing your own aspects for your Spring enabled application. Like DI, AOP supports loose coupling of application objects. But with AOP, application-wide concerns (such as transactions and security) are decoupled from the objects to which they're applied.

DATA ACCESS AND INTEGRATION

Working with JDBC often results in a lot of boilerplate code that gets a connection, creates a statement, processes a result set, and then closes the connection. Spring's JDBC and *data-access objects* (DAO) module abstracts away the boilerplate code so that you can keep your database code clean and simple, and prevents problems that result from a failure to close database resources. This module also builds a layer of meaningful exceptions on top of the error messages given by several database servers. No more trying to decipher cryptic and proprietary SQL error messages! For those who prefer using an *object-relational mapping* (ORM) tool over straight JDBC, Spring provides the ORM module. Spring's ORM support builds on the DAO support, providing a convenient way to build DAOs for several ORM solutions. Spring doesn't attempt to implement its own ORM solution but does provide hooks into several popular ORM frameworks, including Hibernate, Java Persistence API, Java Data Objects, and iBATIS SQL Maps. Spring's transaction management supports each of these ORM frameworks as well as JDBC. This module also includes a Spring abstraction over the Java Message Service (JMS) for asynchronous integration with other applications through messaging. And, as of Spring 3.0, this module includes the object-to-XML mapping features that were originally part of the Spring Web Services project. In addition, this module uses Spring's AOP module to provide transaction management services for objects in a Spring application.

WEB AND REMOTING

The *Model-View-Controller* (MVC) paradigm is a commonly accepted approach to building web applications such that the user interface is separate from the application logic. Java has no shortage of MVC frameworks, with Apache Struts, JSF, WebWork, and Tapestry being among the most popular MVC choices. Even though Spring integrates with several popular MVC frameworks, its web and remoting module comes with a capable MVC framework that promotes Spring's loosely coupled techniques in the web layer of an application. In addition to user-facing web applications, this module also provides several remoting options for building applications that interact with other applications. Spring's remoting capabilities include *Remote Method Invocation* (RMI), Hessian, Burlap, JAX-WS, and Spring's own HTTP invoker. Spring also offers first-class support for exposing and consuming REST APIs.

INSTRUMENTATION

Spring's instrumentation module includes support for adding agents to the JVM. Specifically, it provides a weaving agent for Tomcat that transforms class files as they're loaded by the classloader. If that sounds like a lot to understand, don't worry too much about it. The instrumentation provided by this module has a narrow set of use case.

TESTING

Recognizing the importance of developer-written tests, Spring provides a module dedicated to testing Spring applications. In this module you'll find a collection of mock object implementations for writing unit tests against code that works with JNDI, servlets, and portlets. For integration-level testing, this module provides support for loading a collection of beans in a Spring application context and working with the beans in that context. Throughout this book, many of the examples will be driven by tests, utilizing the testing facilities offered by Spring.

The Spring Portfolio

When it comes to Spring, there's more than meets the eye. In fact, there's more than what comes in the Spring Framework download. If you stop at just the core Spring Framework, you'll miss out on a wealth of potential afforded by the larger Spring portfolio. The whole Spring portfolio includes several frameworks and libraries that build on the core Spring Framework and on each other. All together, the entire Spring portfolio brings the Spring programming model to almost every facet of Java development. It would take several volumes to cover everything the Spring portfolio has to offer, and much of it is outside the scope of this book. But we'll look at some of the elements of the Spring portfolio; here's a taste of what lies beyond the core Spring Framework.

SPRING WEB FLOW

Spring Web Flow builds on Spring's core MVC framework to provide support for building conversational, flow-based web applications that guide users toward a goal (think wizards or shopping carts).

SPRING WEB SERVICES

Although the core Spring Framework provides for declaratively publishing Spring beans as web services, those services are based on an arguably architecturally inferior contract-last model. The contract for the service is determined from the bean's interface. Spring Web Services offers a contract-first web services model where service implementations are written to satisfy the service contract.

SPRING SECURITY

Security is a critical aspect of many applications. Implemented using Spring AOP, Spring Security offers a declarative security mechanism for Spring-based applications.

SPRING INTEGRATION

Many enterprise applications must interact with other enterprise applications. Spring Integration offers implementations of several common integration patterns in Spring's declarative style.

SPRING BATCH

When it's necessary to perform bulk operations on data, nothing beats batch processing. If you're going to be developing a batch application, you can use Spring's robust, POJO-oriented development model to do it using Spring Batch.

SPRING DATA

Spring Data makes it easy to work with all kinds of databases in Spring. Although the relational database has been ubiquitous in enterprise applications for many years, modern applications are recognizing that not all data is best served by columns and rows in a table. A new breed of databases, commonly referred to as *NoSQL databases*² offer new ways of working with data that are more fitting than the traditional relational database. Whether you're using a document database like MongoDB, a graph database such as Neo4j, or even a traditional relational database, Spring Data offers a simplified programming model for persistence. This includes, for many database types, an automatic repository mechanism that creates repository implementations for you.

SPRING SOCIAL

Social networking is a rising trend on the internet, and more and more applications are being outfitted with integration into social networking sites such as Facebook and Twitter. Spring Social, a social networking extension to Spring. But Spring Social is about more than just tweets and friends. Despite its name, Spring Social is less about the word *social* and more about the word *connect*. It helps you connect your Spring application with REST APIs, including many that may not have any social purpose to them.

SPRING MOBILE

Mobile applications are another significant area of software development. Smartphones and tablet devices are taking over as the preferred client for many users. Spring Mobile is a new extension to Spring MVC to support development of mobile web applications.

SPRING FOR ANDROID

Related to Spring Mobile is the Spring Android project. This project aims to bring some of the simplicity afforded by the Spring Framework to development of native applications for Android-based devices. Initially, this project is offering a version of Spring's RestTemplate that can be used in an Android application. It also works with Spring Social to enable native Android apps to connect with REST APIs.

SPRING BOOT

Spring greatly simplifies many programming tasks, reducing or even eliminating much of the boilerplate code you might normally be required to write without it. Spring Boot is an exciting new project that takes an opinionated view of developing with Spring to simplify Spring itself. Spring Boot heavily employs automatic configuration techniques that can eliminate most (and in many cases, all) Spring configuration. It also provides several starter projects to help reduce the size of your Spring project build files, whether you're using Maven or Gradle.

5.5 Wiring Beans – Declaring Beans, and Injecting with JavaConfig

Although automatic Spring configuration with component scanning and automatic wiring is preferable in many cases, there are times when automatic configuration isn't an option and you must configure Spring explicitly. For instance, let's say that you want to wire components from some third-party library into your application.

Because you don't have the source code for that library, there's no opportunity to annotate its classes with `@Component` and `@Autowired`. Therefore, automatic configuration isn't an option.

In that case, you must turn to explicit configuration. You have two choices for explicit configuration: Java and XML. In this section, we'll look at how to use Java-Config. We'll then follow up in the next section on Spring's XML configuration. JavaConfig is the preferred option for explicit configuration because it's more powerful, type-safe, and refactor-friendly. That's because it's just Java code, like any other Java code in your application. At the same time, it's important to recognize that JavaConfig code isn't just any other Java code. It's conceptually set apart from the business logic and domain code in your application. Even though it's expressed in the same language as those components, JavaConfig is configuration code. This means it shouldn't contain any business logic, nor should JavaConfig invade any code where business logic resides. In fact, although it's not required, JavaConfig is often set apart in a separate package from the rest of an application's logic so there's no confusion as to its purpose. Let's see how to explicitly configure Spring with JavaConfig.

Creating a configuration class

Consider an example `CDPlayerConfig`

```
package soundsystem;
import org.springframework.context.annotation.Configuration;
@Configuration
public class CDPlayerConfig {
}
```

The key to creating a JavaConfig class is to annotate it with `@Configuration`. The `@Configuration` annotation identifies this as a configuration class, and it's expected to contain details on beans that are to be created in the Spring application context. So far, you've relied on component scanning to discover the beans that Spring should create. Although there's no reason you can't use component scanning and explicit configuration together, we're focusing on explicit configuration in this section. If you were to run `CDPlayerTest` now, the test would fail with a `BeanCreationException`. The test expects to be injected with `CDPlayer` and `CompactDisc`, but those beans are never created because they're never discovered by component scanning. To make the test happy again, you could put `@ComponentScan` back in. Keeping the focus on explicit configuration, however, let's see how you can wire the `CDPlayer` and `CompactDisc` beans in JavaConfig.

Declaring a simple bean

To declare a bean in JavaConfig, you write a method that creates an instance of the desired type and annotate it with `@Bean`. For example, the following method declares the `CompactDisc` bean:

```
@Bean
public CompactDisc sgtPeppers() {
    return new SgtPeppers();
}
```

The `@Bean` annotation tells Spring that this method will return an object that should be registered as a bean in the Spring application context. The body of the method contains logic that ultimately results in the creation of the bean instance. By default, the bean will be given an ID that is the same as the `@Bean`-annotated method's name. In this case, the bean will be named `compactDisc`. If you'd rather it have a different name, you can either rename the method or prescribe a different name with the `name` attribute:

```
@Bean(name="lonelyHeartsClubBand")
public CompactDisc sgtPeppers() {
    return new SgtPeppers();
}
```

No matter how you name the bean, this bean declaration is about as simple as they come. The body of the method returns a new instance of `SgtPeppers`. But because it's expressed in Java, it has every capability afforded it by the Java language to do almost anything to arrive at the `CompactDisc` that is returned. Unleashing your imagination a bit, you might do something crazy like randomly selecting a `CompactDisc` from a selection of choices:

```
@Bean
public CompactDisc randomBeatlesCD() {
    int choice = (int) Math.floor(Math.random() * 4);
    if (choice == 0) {
        return new SgtPeppers();
    } else if (choice == 1) {
        return new WhiteAlbum();
    } else if (choice == 2) {
        return new HardDaysNight();
    } else {
        return new Revolver();
    }
}
```

Injecting with JavaConfig

The `CompactDisc` bean you declared was simple and had no dependencies of its own. But now you must declare the `CDPlayer` bean, which depends on a `CompactDisc`. How can you wire that up in JavaConfig?

The simplest way to wire up beans in JavaConfig is to refer to the referenced bean's method. For example, here's how you might declare the `CDPlayer` bean:

```
@Bean
public CDPlayer cdPlayer() {
    return new CDPlayer(sgtPeppers());
}
```

The `cdPlayer()` method, like the `sgtPeppers()` method, is annotated with `@Bean` to indicate that it will produce an instance of a bean to be registered in the Spring application context. The ID of the bean will be `cdPlayer`, the same as the method's name. The body of the `cdPlayer()` method differs subtly from that of the `sgtPeppers()` method. Rather than construct

an instance via its default method, the CDPlayer instance is created by calling its constructor that takes a CompactDisc. It appears that the CompactDisc is provided by calling sgtPeppers, but that's not exactly true. Because the sgtPeppers() method is annotated with @Bean, Spring will intercept any calls to it and ensure that the bean produced by that method is returned rather than allowing it to be invoked again.

For example, suppose you were to introduce another CDPlayer bean that is just like the first:

```
@Bean
public CDPlayer cdPlayer() {
    return new CDPlayer(sgtPeppers());
}
@Bean
public CDPlayer anotherCDPlayer() {
    return new CDPlayer(sgtPeppers());
}
```

If the call to sgtPeppers() was treated like any other call to a Java method, then each CDPlayer would be given its own instance of SgtPeppers. That would make sense if we were talking about real CD players and compact discs. If you have two CD players, there's no physical way for a single compact disc to simultaneously be inserted into two CD players.

In software, however, there's no reason you couldn't inject the same instance of SgtPeppers into as many other beans as you want. By default, all beans in Spring are singletons, and there's no reason you need to create a duplicate instance for the second CDPlayer bean. So Spring intercepts the call to sgtPeppers() and makes sure that what is returned is the Spring bean that was created when Spring itself called sgtPeppers() to create the CompactDisc bean. Therefore, both CDPlayer beans will be given the same instance of SgtPeppers.

I can see how referring to a bean by calling its method can be confusing. There's another way that might be easier to digest:

```
@Bean
public CDPlayer cdPlayer(CompactDisc compactDisc) {
    return new CDPlayer(compactDisc);
}
```

Here, the cdPlayer() method asks for a CompactDisc as a parameter. When Spring calls cdPlayer() to create the CDPlayer bean, it autowires a CompactDisc into the configuration method. Then the body of the method can use it however it sees fit. With this technique, the cdPlayer() method can still inject the CompactDisc into the CDPlayer's constructor without explicitly referring to the CompactDisc's @Bean method.

This approach to referring to other beans is usually the best choice because it doesn't depend on the CompactDisc bean being declared in the same configuration class. In fact, there's nothing that says the CompactDisc bean even needs to be declared in JavaConfig; it could have been discovered by component scanning or declared in XML. You could break up your configuration into a healthy mix of configuration classes, XML files, and automatically scanned and wired beans. No matter how the CompactDisc was created, Spring will be happy to hand it to this configuration method to create the CDPlayer bean. In any event, it's important to recognize that although you're performing DI via the CDPlayer's constructor, there's no reason you couldn't apply other styles of DI here. For example, if you wanted to inject a CompactDisc via a setter method, it might look like this:

```
@Bean
public CDPlayer cdPlayer(CompactDisc compactDisc) {
    CDPlayer cdPlayer = new CDPlayer(compactDisc);
```

```
cdPlayer.setCompactDisc(compactDisc);
return cdPlayer;
}
```

Once again, it bears repeating that the body of an @Bean method can utilize whatever Java is necessary to produce the bean instance. Constructor and setter injection just happen to be two simple examples of what you can do in an @Bean-annotated method. The possibilities are limited only by the capabilities of the Java language.

5.6 Using Spring's Java-based configuration

Java-based configuration option enables you to write most of your Spring configuration without XML but with the help of few Java-based annotations.

@Configuration & @Bean Annotations

Annotating a class with the @Configuration indicates that the class can be used by the Spring IoC container as a source of bean definitions. The @Bean annotation tells Spring that a method annotated with @Bean will return an object that should be registered as a bean in the Spring application context. The simplest possible @Configuration class would be as follows.

```
package com.tutorialspoint;
import org.springframework.context.annotation.*;
```

```
@Configuration
public class HelloWorldConfig {
    @Bean
    public HelloWorld helloWorld(){
        return new HelloWorld();
    }
}
```

The above code will be equivalent to the following XML configuration

```
<beans>
    <bean id = "helloWorld" class = "com.tutorialspoint.HelloWorld" />
</beans>
```

Here, the method name is annotated with @Bean works as bean ID and it creates and returns the actual bean. Your configuration class can have a declaration for more than one @Bean. Once your configuration classes are defined, you can load and provide them to Spring container using *AnnotationConfigApplicationContext* as follows –

```
public static void main(String[] args) {
    ApplicationContext ctx = new
    AnnotationConfigApplicationContext(HelloWorldConfig.class);

    HelloWorld helloWorld = ctx.getBean(HelloWorld.class);
    helloWorld.setMessage("Hello World!");
    helloWorld.getMessage();
}
```

You can load various configuration classes as follows –

```
public static void main(String[] args) {
    AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
```



```
ctx.register(AppConfig.class, OtherConfig.class);
ctx.register(AdditionalConfig.class);
ctx.refresh();
```

```
MyService myService = ctx.getBean(MyService.class);
myService.doStuff();
}
```

Example

Let us have a working Eclipse IDE in place and take the following steps to create a Spring application –

- Create a project with a name SpringExample and create a package com.tutorialspoint under the src folder in the created project.
- Add required Spring libraries using Add External JARs option as explained in the Spring Hello World Example chapter.
- Because you are using Java-based annotations, so you also need to add CGLIB.jar from your Java installation directory and ASM.jar library which can be downloaded from asm.ow2.org.
- Create Java classes HelloWorldConfig, HelloWorld and MainApp under the com.tutorialspoint package.
- The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below.

Here is the content of **HelloWorldConfig.java** file
package com.tutorialspoint;
import org.springframework.context.annotation.*;

```
@Configuration
public class HelloWorldConfig {
    @Bean
    public HelloWorld helloWorld(){
        return new HelloWorld();
    }
}
```

Here is the content of **HelloWorld.java** file
package com.tutorialspoint;

```
public class HelloWorld {
    private String message;

    public void setMessage(String message){
        this.message = message;
    }
    public void getMessage(){
        System.out.println("Your Message : " + message);
    }
}
```

Following is the content of the **MainApp.java** file
package com.tutorialspoint;

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.*;

public class MainApp {
    public static void main(String[] args) {
        ApplicationContext ctx =
            new AnnotationConfigApplicationContext(HelloWorldConfig.class);

        HelloWorld helloWorld = ctx.getBean(HelloWorld.class);
        helloWorld.setMessage("Hello World!");
        helloWorld.getMessage();
    }
}
```

Once you are done creating all the source files and adding the required additional libraries, let us run the application. You should note that there is no configuration file required. If everything is fine with your application, it will print the following message –
Your Message : Hello World!

Injecting Bean Dependencies

When @Beans have dependencies on one another, expressing that the dependency is as simple as having one bean method calling another as follows –

```
package com.tutorialspoint;
import org.springframework.context.annotation.*;
```

```
@Configuration
public class AppConfig {
    @Bean
    public Foo foo() {
        return new Foo(bar());
    }
    @Bean
    public Bar bar() {
        return new Bar();
    }
}
```

Here, the foo bean receives a reference to bar via the constructor injection. Now let us look at another working example.

Example

Let us have a working Eclipse IDE in place and take the following steps to create a Spring application

- Create a project with a name SpringExample and create a package com.tutorialspoint under the src folder in the created project.
- Add required Spring libraries using Add External JARs option as explained in the Spring Hello World Example chapter.
- Because you are using Java-based annotations, so you also need to add CGLIB.jar from your Java installation directory and ASM.jar library which can be downloaded from asm.ow2.org.
- Create Java classes TextEditorConfig, TextEditor, SpellChecker and MainApp under the com.tutorialspoint package.

- The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below.

Here is the content of **TextEditorConfig.java** file

```
package com.tutorialspoint;
import org.springframework.context.annotation.*;
```

```
@Configuration
public class TextEditorConfig {
    @Bean
    public TextEditor textEditor(){
        return new TextEditor( spellChecker() );
    }

    @Bean
    public SpellChecker spellChecker(){
        return new SpellChecker( );
    }
}
```

Here is the content of **TextEditor.java** file

```
package com.tutorialspoint;

public class TextEditor {
    private SpellChecker spellChecker;

    public TextEditor(SpellChecker spellChecker){
        System.out.println("Inside TextEditor constructor." );
        this.spellChecker = spellChecker;
    }
    public void spellCheck(){
        spellChecker.checkSpelling();
    }
}
```

Following is the content of another dependent class file **SpellChecker.java**

```
package com.tutorialspoint;

public class SpellChecker {
    public SpellChecker(){
        System.out.println("Inside SpellChecker constructor." );
    }
    public void checkSpelling(){
        System.out.println("Inside checkSpelling." );
    }
}
```

Following is the content of the **MainApp.java** file

```
package com.tutorialspoint;

import org.springframework.context.ApplicationContext;
```

```
import org.springframework.context.annotation.*;

public class MainApp {
    public static void main(String[] args) {
        ApplicationContext ctx =
            new AnnotationConfigApplicationContext(TextEditorConfig.class);

        TextEditor te = ctx.getBean(TextEditor.class);
        te.spellCheck();
    }
}
```

Once you are done creating all the source files and adding the required additional libraries, let us run the application. You should note that there is no configuration file required. If everything is fine with your application, it will print the following message –
Inside SpellChecker constructor.

Inside TextEditor constructor.

Inside checkSpelling.

The @Import Annotation

The **@Import** annotation allows for loading @Bean definitions from another configuration class. Consider a ConfigA class as follows –

@Configuration

```
public class ConfigA {
    @Bean
    public A a() {
        return new A();
    }
}
```

You can import above Bean declaration in another Bean Declaration as follows –

@Configuration

@Import(ConfigA.class)

```
public class ConfigB {
    @Bean
    public B b() {
        return new B();
    }
}
```

Now, rather than needing to specify both ConfigA.class and ConfigB.class when instantiating the context, only ConfigB needs to be supplied as follows –

```
public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(ConfigB.class);

    // now both beans A and B will be available...
    A a = ctx.getBean(A.class);
    B b = ctx.getBean(B.class);
}
```

Lifecycle Callbacks

The @Bean annotation supports specifying arbitrary initialization and destruction callback methods, much like Spring XML's init-method and destroy-method attributes on the bean element –

```

public class Foo {
    public void init() {
        // initialization logic
    }
    public void cleanup() {
        // destruction logic
    }
}
@Configuration
public class AppConfig {
    @Bean(initMethod = "init", destroyMethod = "cleanup" )
    public Foo foo() {
        return new Foo();
    }
}

```

Specifying Bean Scope

The default scope is singleton, but you can override this with the @Scope annotation as follows –

```

@Configuration
public class AppConfig {
    @Bean
    @Scope("prototype")
    public Foo foo() {
        return new Foo();
    }
}

```

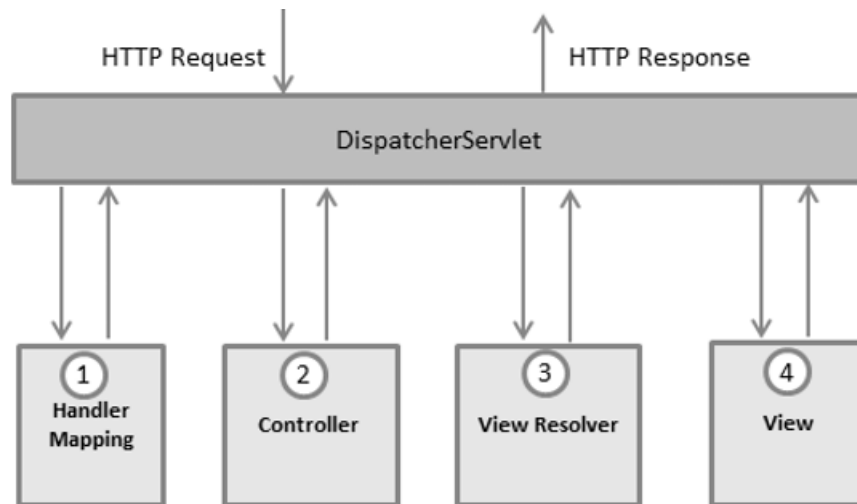
5.7 Building web applications with Spring Model View Controller (MVC)- Getting started with Spring MVC, and Writing a basic controller

The Spring Web MVC framework provides Model-View-Controller (MVC) architecture and ready components that can be used to develop flexible and loosely coupled web applications. The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

- The Model encapsulates the application data and in general they will consist of POJO.
- The View is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.
- The Controller is responsible for processing user requests and building an appropriate model and passes it to the view for rendering.

The DispatcherServlet

The Spring Web model-view-controller (MVC) framework is designed around a *DispatcherServlet* that handles all the HTTP requests and responses. The request processing workflow of the Spring Web MVC *DispatcherServlet* is illustrated in the following diagram –



Following is the sequence of events corresponding to an incoming HTTP request to *DispatcherServlet* –

- After receiving an HTTP request, *DispatcherServlet* consults the *HandlerMapping* to call the appropriate *Controller*.
- The *Controller* takes the request and calls the appropriate service methods based on used GET or POST method. The service method will set model data based on defined business logic and returns view name to the *DispatcherServlet*.
- The *DispatcherServlet* will take help from *ViewResolver* to pickup the defined view for the request.
- Once view is finalized, The *DispatcherServlet* passes the model data to the view which is finally rendered on the browser.

All the above-mentioned components, i.e. *HandlerMapping*, *Controller*, and *ViewResolver* are parts of *WebApplicationContext* which is an extension of the *plainApplicationContext* with some extra features necessary for web applications.

Required Configuration

You need to map requests that you want the *DispatcherServlet* to handle, by using a URL mapping in the web.xml file. The following is an example to show declaration and mapping for HelloWeb *DispatcherServlet* example –

```

<web-app id = "WebApp_ID" 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">
  <display-name>Spring MVC Application</display-name>
  <servlet>
    <servlet-name>HelloWeb</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>HelloWeb</servlet-name>
    <url-pattern>*.jsp</url-pattern>
  </servlet-mapping>
</web-app>
  
```

```
</servlet-mapping>
</web-app>
```

The **web.xml** file will be kept in the WebContent/WEB-INF directory of your web application. Upon initialization of **HelloWeb** DispatcherServlet, the framework will try to load the application context from a file named **[servlet-name]-servlet.xml** located in the application's WebContent/WEB-INF directory. In this case, our file will be **HelloWeb-servlet.xml**. Next, `<servlet-mapping>` tag indicates what URLs will be handled by which DispatcherServlet. Here all the HTTP requests ending with **.jsp** will be handled by the **HelloWeb** DispatcherServlet.

If you do not want to go with default filename as `[servlet-name]-servlet.xml` and default location as `WebContent/WEB-INF`, you can customize this file name and location by adding the servlet listener `ContextLoaderListener` in your web.xml file as follows –

```
<web-app...>
  <!-- DispatcherServlet definition goes here -->
  ...
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/HelloWeb-servlet.xml</param-value>
  </context-param>
  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>
</web-app>
```

Now, let us check the required configuration for `HelloWeb-servlet.xml` file, placed in your web application's WebContent/WEB-INF directory –

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd">
  <context:component-scan base-package="com.tutorialspoint" />
  <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/jsp/" />
    <property name="suffix" value=".jsp" />
  </bean>
</beans>
```

Following are the important points about **HelloWeb-servlet.xml** file –

The `[servlet-name]-servlet.xml` file will be used to create the beans defined, overriding the definitions of any beans defined with the same name in the global scope.

The `<context:component-scan...>` tag will be used to activate Spring MVC annotation scanning capability which allows to make use of annotations like `@Controller` and `@RequestMapping` etc.

The *InternalResourceViewResolver* will have rules defined to resolve the view names. As per the above defined rule, a logical view named **hello** is delegated to a view implementation located at */WEB-INF/jsp/hello.jsp*.

The following section will show you how to create your actual components, i.e., Controller, Model, and View.

Defining a Controller

The *DispatcherServlet* delegates the request to the controllers to execute the functionality specific to it. The **@Controller** annotation indicates that a particular class serves the role of a controller. The **@RequestMapping** annotation is used to map a URL to either an entire class or a particular handler method.

@Controller

@RequestMapping("/hello")

```
public class HelloController {
    @RequestMapping(method = RequestMethod.GET)
    public String printHello(ModelMap model) {
        model.addAttribute("message", "Hello Spring MVC Framework!");
        return "hello";
    }
}
```

The **@Controller** annotation defines the class as a Spring MVC controller. Here, the first usage of **@RequestMapping** indicates that all handling methods on this controller are relative to the **/hello** path. Next annotation **@RequestMapping(method = RequestMethod.GET)** is used to declare the *printHello()* method as the controller's default service method to handle HTTP GET request. You can define another method to handle any POST request at the same URL.

You can write the above controller in another form where you can add additional attributes in *@RequestMapping* as follows –

@Controller

```
public class HelloController {
    @RequestMapping(value = "/hello", method = RequestMethod.GET)
    public String printHello(ModelMap model) {
        model.addAttribute("message", "Hello Spring MVC Framework!");
        return "hello";
    }
}
```

The value attribute indicates the URL to which the handler method is mapped and the method attribute defines the service method to handle HTTP GET request. The following important points are to be noted about the controller defined above –

- You will define required business logic inside a service method. You can call another method inside this method as per requirement.
- Based on the business logic defined, you will create a model within this method. You can use setter different model attributes and these attributes will be accessed by the view to present the final result. This example creates a model with its attribute "message".
- A defined service method can return a String, which contains the name of the view to be used to render the model. This example returns "hello" as logical view name.

Creating JSP Views

Spring MVC supports many types of views for different presentation technologies. These include - JSPs, HTML, PDF, Excel worksheets, XML, Velocity templates, XSLT, JSON, Atom and RSS feeds, JasperReports, etc. But most commonly we use JSP templates written with JSTL.

Let us write a simple **hello** view in /WEB-INF/hello/hello.jsp –

```
<html>
<head>
<title>Hello Spring MVC</title>
</head>

<body>
<h2>${message}</h2>
</body>
</html>
```

Here **\${message}** is the attribute which we have set up inside the Controller. You can have multiple attributes to be displayed inside your view.

Spring Web MVC Framework Examples

Based on the above concepts, let us check few important examples which will help you in building your Spring Web Applications –

Sr. No.	Example & Description
1	<u>Spring MVC Hello World Example</u> This example will explain how to write a simple Spring Web Hello World application.
2	<u>Spring MVC Form Handling Example</u> This example will explain how to write a Spring Web application using HTML forms to submit the data to the controller and display a processed result.
3	<u>Spring Page Redirection Example</u> Learn how to use page redirection functionality in Spring MVC Framework.
4	<u>Spring Static Pages Example</u> Learn how to access static pages along with dynamic pages in Spring MVC Framework.
5	<u>Spring Exception Handling Example</u> Learn how to handle exceptions in Spring MVC Framework.

5.8 Handling controller input, Processing forms

The following example shows how to write a simple web-based application, which makes use of HTML forms using Spring Web MVC framework. To start with, let us have a working Eclipse IDE in place and take the following steps to develop a Dynamic Form-based Web Application using Spring Web Framework –

Create a Dynamic Web Project with a name HelloWorld and create a package com.tutorialspoint under the src folder in the created project.

Drag and drop below mentioned Spring and other libraries into the folder WebContent/WEB-INF/lib.

Create a Java classes Student and StudentController under the com.tutorialspoint package.

Create Spring configuration files Web.xml and HelloWeb-servlet.xml under the WebContent/WEB-INF folder.

Create a sub-folder with a name jsp under the WebContent/WEB-INF folder. Create a view files student.jsp and result.jsp under this sub-folder.

The final step is to create the content of all the source and configuration files and export the application as explained below.

Here is the content of **Student.java** file

package com.tutorialspoint;

```
public class Student {
    private Integer age;
    private String name;
    private Integer id;

    public void setAge(Integer age) {
        this.age = age;
    }
    public Integer getAge() {
        return age;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public Integer getId() {
        return id;
    }
}
```

Following is the content of **StudentController.java** file

package com.tutorialspoint;

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.ui.ModelMap;
```

@Controller

```
public class StudentController {
    @RequestMapping(value = "/student", method = RequestMethod.GET)
    public ModelAndView student() {
```



```

        return new ModelAndView("student", "command", new Student());
    }
    @RequestMapping(value = "/addStudent", method = RequestMethod.POST)
    public String addStudent(@ModelAttribute("SpringWeb")Student student,

    ModelMap model) {
        model.addAttribute("name", student.getName());
        model.addAttribute("age", student.getAge());
        model.addAttribute("id", student.getId());

        return "result";
    }
}

```

Here the first service method **student()**, we have passed a blank **Student** object in the ModelAndView object with the name "command" because the Spring framework expects an object with the name "command" if you are using <form:form> tags in your JSP file. So, when **student()** method is called, it returns **student.jsp** view.

The second service method **addStudent()** will be called against a POST method on the **HelloWeb/addStudent** URL. You will prepare your model object based on the submitted information. Finally a "result" view will be returned from the service method, which will result in rendering result.jsp

Following is the content of Spring Web configuration file **web.xml**

```

<web-app id = "WebApp_ID" 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">

    <display-name>Spring MVC Form Handling</display-name>

    <servlet>
        <servlet-name>HelloWeb</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>HelloWeb</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

</web-app>

```

Following is the content of another Spring Web configuration file **HelloWeb-servlet.xml**

```

<beans xmlns = "http://www.springframework.org/schema/beans"
    xmlns:context = "http://www.springframework.org/schema/context"
    xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

```

```

xsi:schemaLocation = "http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">

```

```

<context:component-scan base-package = "com.tutorialspoint" />

```

```

<bean class = "org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name = "prefix" value = "/WEB-INF/jsp/" />
    <property name = "suffix" value = ".jsp" />
</bean>

```

```

</beans>

```

Following is the content of Spring view file **student.jsp**

```

<% @taglib uri = "http://www.springframework.org/tags/form" prefix = "form"%>

```

```

<html>

```

```

    <head>

```

```

        <title>Spring MVC Form Handling</title>

```

```

    </head>

```

```

    <body>

```

```

        <h2>Student Information</h2>

```

```

        <form:form method = "POST" action = "/HelloWeb/addStudent">

```

```

            <table>

```

```

                <tr>

```

```

                    <td><form:label path = "name">Name</form:label></td>

```

```

                    <td><form:input path = "name" /></td>

```

```

                </tr>

```

```

                <tr>

```

```

                    <td><form:label path = "age">Age</form:label></td>

```

```

                    <td><form:input path = "age" /></td>

```

```

                </tr>

```

```

                <tr>

```

```

                    <td><form:label path = "id">id</form:label></td>

```

```

                    <td><form:input path = "id" /></td>

```

```

                </tr>

```

```

                <tr>

```

```

                    <td colspan = "2">

```

```

                        <input type = "submit" value = "Submit"/>

```

```

                    </td>

```

```

                </tr>

```

```

            </table>

```

```

        </form:form>

```

```

    </body>

```

```

</html>

```

Following is the content of Spring view file **result.jsp**

```

<% @page contentType = "text/html; charset = UTF-8" language = "java" %>

```

```

<% @page isELIgnored = "false" %>

```

```

<% @taglib uri = "http://www.springframework.org/tags/form" prefix = "form"%>

```

```

<html>
<head>
  <title>Spring MVC Form Handling</title>
</head>

<body>
  <h2>Submitted Student Information</h2>
  <table>
    <tr>
      <td>Name</td>
      <td>${name}</td>
    </tr>
    <tr>
      <td>Age</td>
      <td>${age}</td>
    </tr>
    <tr>
      <td>ID</td>
      <td>${id}</td>
    </tr>
  </table>
</body>

</html>

```

Finally, following is the list of Spring and other libraries to be included in your web application. You simply drag these files and drop them in **WebContent/WEB-INF/lib** folder.

- commons-logging-x.y.z.jar
- org.springframework.asm-x.y.z.jar
- org.springframework.beans-x.y.z.jar
- org.springframework.context-x.y.z.jar
- org.springframework.core-x.y.z.jar
- org.springframework.expression-x.y.z.jar
- org.springframework.web.servlet-x.y.z.jar
- org.springframework.web-x.y.z.jar
- spring-web.jar

Once you are done with creating source and configuration files, export your application. Right click on your application and use the **Export > WAR File** option and save your **SpringWeb.war** file in Tomcat's *webapps* folder.

Now start your Tomcat server and make sure you are able to access other web pages from webapps folder using a standard browser.

Now try a URL <http://localhost:8080/SpringWeb/student>.

5.9 Spring-MVC File Upload

The following example shows how to use File Upload Control in forms using the Spring Web MVC framework. To start with, let us have a working Eclipse IDE in place and adhere to the following steps to develop a Dynamic Form based Web Application using the Spring Web Framework.

- Create a project with a name **HelloWeb** under a package **com.tutorialspoint** as explained in the Spring MVC - Hello World chapter.

- Create Java classes FileModel, FileUploadController under the com.tutorialspoint package.
- Create view files fileUpload.jsp, success.jsp under the jsp sub-folder.
- Create a folder temp under the WebContent sub-folder.
- Download Apache Commons FileUpload library commons-fileupload.jar and Apache Commons IO library commons-io.jar. Put them in your CLASSPATH.
- The final step is to create the content of the source and configuration files and export the application as explained below.

FileModel.java

```
package com.tutorialspoint;
```

```
import org.springframework.web.multipart.MultipartFile;
```

```
public class FileModel {  
    private MultipartFile file;
```

```
    public MultipartFile getFile() {  
        return file;  
    }  
}
```

```
    public void setFile(MultipartFile file) {  
        this.file = file;  
    }  
}
```

FileUploadController.java

```
package com.tutorialspoint;
```

```
import java.io.File;
```

```
import java.io.IOException;
```

```
import javax.servlet.ServletContext;
```

```
import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.stereotype.Controller;
```

```
import org.springframework.ui.ModelMap;
```

```
import org.springframework.util.FileCopyUtils;
```

```
import org.springframework.validation.BindingResult;
```

```
import org.springframework.validation.annotation.Validated;
```

```
import org.springframework.web.bind.annotation.RequestMapping;
```

```
import org.springframework.web.bind.annotation.RequestMethod;
```

```
import org.springframework.web.multipart.MultipartFile;
```

```
import org.springframework.web.servlet.ModelAndView;
```

```
@Controller
```

```
public class FileUploadController {
```

```
    @Autowired
```

```
    ServletContext context;
```

```

@RequestMapping(value = "/fileUploadPage", method = RequestMethod.GET)
public ModelAndView fileUploadPage() {
    FileModel file = new FileModel();
    ModelAndView modelAndView = new ModelAndView("fileUpload", "command", file);
    return modelAndView;
}

@RequestMapping(value="/fileUploadPage", method = RequestMethod.POST)
public String fileUpload(@Validated FileModel file, BindingResult result, ModelMap
model) throws IOException {
    if (result.hasErrors()) {
        System.out.println("validation errors");
        return "fileUploadPage";
    } else {
        System.out.println("Fetching file");
        MultipartFile multipartFile = file.getFile();
        String uploadPath = context.getRealPath("") + File.separator + "temp" + File.separator;
        //Now do something with file...
        FileCopyUtils.copy(file.getFile().getBytes(),                                new
File(uploadPath+file.getFile().getOriginalFilename()));
        String fileName = multipartFile.getOriginalFilename();
        model.addAttribute("fileName", fileName);
        return "success";
    }
}
}

```

HelloWeb-servlet.xml

```

<beans xmlns = "http://www.springframework.org/schema/beans"
xmlns:context = "http://www.springframework.org/schema/context"
xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation = "
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:component-scan base-package = "com.tutorialspoint" />

<bean class = "org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name = "prefix" value = "/WEB-INF/jsp/" />
    <property name = "suffix" value = ".jsp" />
</bean>

<bean id = "multipartResolver"
    class = "org.springframework.web.multipart.commons.CommonsMultipartResolver" />
</beans>

```

Here, for the first service method fileUploadPage(), we have passed a blank FileModel object in the ModelAndView object with name "command", because the spring framework expects

an object with name "command", if you are using <form:form> tags in your JSP file. So, when fileUploadPage() method is called, it returns fileUpload.jsp view.

The second service method fileUpload() will be called against a POST method on the HelloWeb/fileUploadPage URL. You will prepare the file to be uploaded based on the submitted information. Finally, a "success" view will be returned from the service method, which will result in rendering success.jsp.

fileUpload.jsp

```
<%@ page contentType="text/html; charset = UTF-8" %>
<%@ taglib prefix = "form" uri = "http://www.springframework.org/tags/form"%>
<html>
  <head>
    <title>File Upload Example</title>
  </head>

  <body>
    <form:form method = "POST" modelAttribute = "fileUpload"
      enctype = "multipart/form-data">
      Please select a file to upload :
      <input type = "file" name = "file" />
      <input type = "submit" value = "upload" />
    </form:form>
  </body>
</html>
```

Here, we are using modelAttribute attribute with value="fileUpload" to map the file Upload control with the server model.

success.jsp

```
<%@ page contentType = "text/html; charset = UTF-8" %>
<html>
  <head>
    <title>File Upload Example</title>
  </head>
  <body>
    FileName :
    <b> ${ fileName} </b> - Uploaded Successfully.
  </body>
</html>
```

Once you are done with creating source and configuration files, export your application. Right click on your application, use Export → WAR File option and save the HelloWeb.war file in the Tomcat's webapps folder.

Now, start your Tomcat server and make sure you are able to access other webpages from the webapps folder using a standard browser.

Try a URL– <http://localhost:8080/HelloWeb/fileUploadPage> to get output

5.10 Assignment-5

Short Answer Questions

Knowledge/Remembering Level Questions

16. Expand the word POJOs
17. Who simplifies the java development?

18. Which is often defined as a technique that promotes separation of concerns in a software system?
19. In a Spring-based application, where java application objects lives?
20. Which loads a Spring application context from one or more Java-based configuration classes?
21. Could you mention where to spring injects values and bean references?
22. Which is the centerpiece of the Spring Framework?
23. Which is a commonly accepted approach to building web applications such that the user interface is separate from the application logic?
24. List out one document database?
25. Which is a rising trend on the internet, and more and more applications are being outfitted with integration?
26. What is the simplest way to wire up beans in JavaConfig?
27. How to indicate that the class can be used by the Spring IOC container as a source of bean definitions?
28. Expand MVC?
29. Which is designed around the Spring web MVC framework?
30. Which delegates the request to the controllers to execute the functionality specific to it?

Long Answer Questions

Understanding Level Questions

25. Explain how injecting dependencies and making use of aspects helps for simplifying java programs?
26. Explain eliminating boilerplate code with templates?
27. Explain Bean's life.
28. Could you briefly explain the concept of Spring portfolio
29. Write a short note on creating a configuration class and declaring a simple bean.
30. Briefly explain spring web MVC framework with DispatcherServlet.

Application Level Questions

31. Interpret in your own the concept of spring into actions.
32. How would you explain surveying spring landscape with spring modules?
33. How would you explain using spring Java-based configuration?
34. Implement the concept of processing forms in spring.

Skill Level Question

35. Create your own program to demonstrate Spring MVC File Upload.

EXAM: IV Semester MCA

19MCA43

SUBJECT: JAVA & J2EE**CHAPTERS:** 5**MAXIMUM MARKS:** 50**PAPER:****CLASS:** MCA**TIME:** 2 Hours**WEIGHTAGE TO OBJECTIVES TABLE**

SL.NO	OBJECTIVES	MARKS	% MARKS
1.	Knowledge (Remembering)	09	18
2.	Understanding	10	20
3.	Application	17	34
4.	Skill	14	28
Total		50	100

BLUE PRINT

Content Areas (UNITS)	Remembering			Understanding			Application			Skill			Total			
	VS A	L A	Essay	VS A	LA	Essay	VS A	LA	Essay	VS A	L A	Essay	VS A	L A	Essay	Total
1	1(1)				1(7)								1	7		8
2	1(1)				1(7)								1	7		8
3	1(1)							1(7)					1	7		8
4	1(1)							1(7)					1	7		8
5	1(1)							1(7)				1(10)	1	7	10	18
Sub-Total	5			14			21			10			50			

Please Note: Questions are entered in numbers & marks are represented by numbers in brackets.

UNIT-1 (JDBC)**Very Short Answer Questions (1 Mark Questions)****Knowledge/Remembering Level Questions**

16. What is full form of JDBC?
17. Which manages a list of database drivers in JDBC?
18. How would you abbreviate JDBC-ODBC?
19. Which method is used to register the driver class in Java Database Connectivity?
20. The getConnection() method of which class is used to establish connection with the database?
21. Which method of Statement interface is used to execute queries to the database?
22. Can you list the method which is used to close the connection in database connection programs?
23. Can you recall the method of a class act as an interface between user and driver?
24. Which interface is a sub interface of Statement?
25. Mention the connection URL for the MySQL database.
26. Which is the session between java application and database?
27. List out the interface used to call the stored procedures and functions.
28. Which converts the Java data type to the appropriate JDBC type?
29. Which allows grouping related SQL statements into a batch?
30. Can you mention the method used to set the commit status and it is by default true.

Long Answer Questions (7 Marks Question)**Understanding Level Questions**

11. Can you explain the types of JDBC drivers with diagram?
12. Explain Database connectivity steps with syntax and example.
13. Describe Different types of JDBC Drivers.
14. Explain Database connectivity steps using JDBC
15. Explain Batch Processing with its importance.

Application Level Questions

16. Demonstrate the CallableStatement using a program.
17. Interpret the prepared statements with an example.
18. Implement 'INSERT' stored procedure that receives id and name as the parameter and inserts it into the table.
19. Demonstrate different types of JDBC datatypes.

Skill Level Questions

20. Create a JDBC and MYSL program to demonstrate basic database Transactions.

UNIT-II (Servlets)**Short Answer Questions (1 Mark Questions)****Knowledge/Remembering Level Questions**

19. List out the two package used for servlet creation.
20. Which type of java programs run on a Web or Application server?
21. What is the method used by the servlet to initialize something?
22. Which method is used by the servlet to terminate its function?
23. Which method is the main method to perform the actual task in servlets?
24. What are the two most frequently used methods within each service request.
25. Which is an open source web server for testing servlets and JSP technology?
26. Can you recall a file required to compile a Servlet?
27. Which is an XML document that is used by Web Container to run Servlets?

28. To start Tomcat Server for the first time what you need to set in the Environment variable?
29. Which method returns an array containing all of the Cookie objects the client sent along with its request?
30. Which method returns an Enumeration containing the names of the attributes available to this request?
31. Which header instructs the browser whether to use persistent in HTTP connections or not.
32. By which method we can come to know how long (in seconds) the cookie should be valid.
33. Which is used to append some extra data on the end of each URL that identifies the session
34. Which Interface provides a way to identify a user across more than one page request or visit to a Web site.
35. What method is used to return the time when this session was created.
36. What are the two important packages that encapsulate all the important classes and interface?

Long Answer Questions (7 mark question)

Understanding Level Questions

1. Can you explain the advantages of servlets?
2. Discuss the Life cycle of Servlet with an example.
3. Explain the Servlet API.
4. Explain the usage of Cookies with an example.

Application Level Questions

5. Demonstrate how tomcat is used for servlet development.
6. How would you apply the concept of handling HTTP request and Response using your own example?
7. Implement database access using servlet?
8. Differentiate Session tracking and HTTP Session object
9. Implement HTTP client Request with an example.

Skill Level Question (10 Marks Question)

10. Create a program for database access using servlet

UNIT-III (JSP)

Short Answer Questions

Knowledge/Remembering Level Questions

21. List out the two tags usually contained by a JSP.
22. Mention the three processes involved in JSP Compilation Process.
23. Which Phase of the JSP lifecycle represents all interactions with requests?
24. What is MVC?
25. Where you would develop your JSP programs, test them and finally run them?
26. Which command is used in UNIX to start Tomcat Server?
27. Write the command to stop Tomcat on the Windows machine
28. Which is used to provide instructions to the container?
29. Which is used to include a file during the translation phase?
30. Which declares that your JSP page uses a set of custom tags?
31. Can you recall which finds or instantiates a JavaBean?
32. What are the two attributes that are common to all Action elements?
33. Which attribute identifies the lifecycle of the Action element?
34. Which is used to insert Java components into a JSP page?

35. What action can be used to write the template text in JSP pages and documents?
36. Can you recall which object is an instance of javax.servlet.http.HttpSession?
37. Which is text files stored on the client computer for various information tracking purposes?
38. Which JSTL function is used to escapes characters that can be interpreted as XML markup?
39. Which makes easily access application data stored in JavaBeans components?
40. What are the two objects give you access to the header values?

Long Answer Questions

Understanding Level Questions

36. Explain the need and importance of JSP.
37. With a neat diagram explain JSP Lifecycle.
38. Write a short note on MVC Architecture.
39. Explain in detail any Two JSP Directives
40. Discuss on JSP Actions.
41. Explain JSP Standard Tag Libraries.

Application Level Questions

42. Differentiate Servlets and JSP and also demonstrate the advantages of JSP.
43. How would you explain the concept of implicit objects in connection with JSP?
44. Apply the concept of Session Tracking to find out the creation time and the last-accessed time for a session
45. Implement the cookies for the first and the last name.
46. With your own program implement the concept of Exception handling in JSP

Skill Level Question

47. Create a program for database access using JSP

UNIT-IV (ANNOTATIONS AND JAVA BEANS)

Short Answer Questions

Knowledge/Remembering Level Questions

31. What are the two ways in which we can reuse the code already created?
32. To create a package what we have to do in source file?
33. Which specifies the visibility of a data field, method or class?
34. What are the two elements of an interface?
35. Which is the compressed version of the normal file?
36. List out any two types of Annotations.
37. Which defines the core classes and interfaces that is indispensable to the Java platform and the Java programming language?
38. What are the two methods used to access the JavaBean class?
39. Which tool can be used to learn about the properties and operations provided by a class?

40. Could you mention which provides the feature to visually modify the java beans properties as per the requirement of the situation?
41. Which is a named attribute of a bean that can affect its behavior or appearance?
42. Could you mention how you can expose the features of your bean to java framework?
43. What are the two types of properties for a bean?
44. Which type of bean property consists of multiple values?
45. Which bean property generates an event when their values change?

Long Answer Questions**Understanding Level Questions**

1. Explain the need and importance of a Package.
2. Explain how to create a package with an example
3. Write a short note on Interfaces.
4. Explain how to create a JAR File.
5. Discuss on Java Annotations.
6. Explain How to create a Java Bean with an example.

Application Level Questions

7. Differentiate Class and Interface. Also create your own program by applying the concept of package.
8. How would you explain the concept Adding controls to Beans, and setting bean properties?
9. How would you explain Simple Properties and Design Pattern for Events by applying java bean concept?
10. Implement the concept of Creating Bound Properties in Java Beans with your own example.
11. With your own program implement the concept of creating Bound Properties and Creating Bean Info class.

Skill Level Question

12. Create your own program to demonstrate java bean properties and also to access java bean and its properties.

UNIT-V (INTRODUCTION TO SPRING FRAMEWORK)**Short Answer Questions****Knowledge/Remembering Level Questions**

1. Expand the word POJOs
2. Who simplifies the java development?
3. Which is often defined as a technique that promotes separation of concerns in a software system?
4. In a Spring-based application, where java application objects lives?
5. Which loads a Spring application context from one or more Java-based configuration classes?
6. Could you mention where to spring injects values and bean references?
7. Which is the centerpiece of the Spring Framework?
8. Which is a commonly accepted approach to building web applications such that the user interface is separate from the application logic?
9. List out one document database?
10. Which is a rising trend on the internet, and more and more applications are being outfitted with integration?
11. What is the simplest way to wire up beans in JavaConfig?
12. How to indicate that the class can be used by the Spring IOC container as a source of bean definitions?
13. Expand MVC?
14. Which is designed around the Spring web MVC framework?

15. Which delegates the request to the controllers to execute the functionality specific to it?

Long Answer Questions

Understanding Level Questions

1. Explain how injecting dependencies and making use of aspects helps for simplifying java programs?
2. Explain eliminating boilerplate code with templates?
3. Explain Bean's life.
4. Could you briefly explain the concept of Spring portfolio
5. Write a short note on creating a configuration class and declaring a simple bean.
6. Briefly explain spring web MVC framework with DispatcherServlet.

Application Level Questions

7. Interpret in your own the concept of spring into actions.
8. How would you explain surveying spring landscape with spring modules?
9. How would you explain using spring Java-based configuration?
10. Implement the concept of processing forms in spring.

Skill Level Question

11. Create your own program to demonstrate Spring MVC File Upload.

USN:

19MC43

SRINIVAS UNIVERSITY
IV SEMISTER M.C.A DEGREE EXAMINATION MAY-2018
JAVA AND J2EE

Time: 2 Hours

Max.Marks: 50

SECTION-A
(Compulsory)

(5x1=5)

Note: Answer ALL questions. Each question carries ONE mark.

1. a) Which tool is used to create HTML format documentation?
- b) Which method is used to schedule thread for running?
- c) Which method is used for moving file pointer?
- d) Which method is used to register the driver class?
- e) What are the two objects useful when dealing with RMI?

SECTION-B

(5x7=35)

Note: Answer any 5 questions. Each question carries SEVEN marks.

2. Explain the structure of java program and main function
3. Explain the various ways of creating a threads in java
4. Discuss the steps involved in developing and executing an applet
5. Explain the architecture of RMI with the help of diagram
6. Explain the Life cycle of Servlet with an example
7. What is random access file? With example explain how to handle random access file

SECTION-C
(Compulsory)

(1x10=10)

8. With a simple program explain implementation of RMI program with database.