

Java RMI - Introduction

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

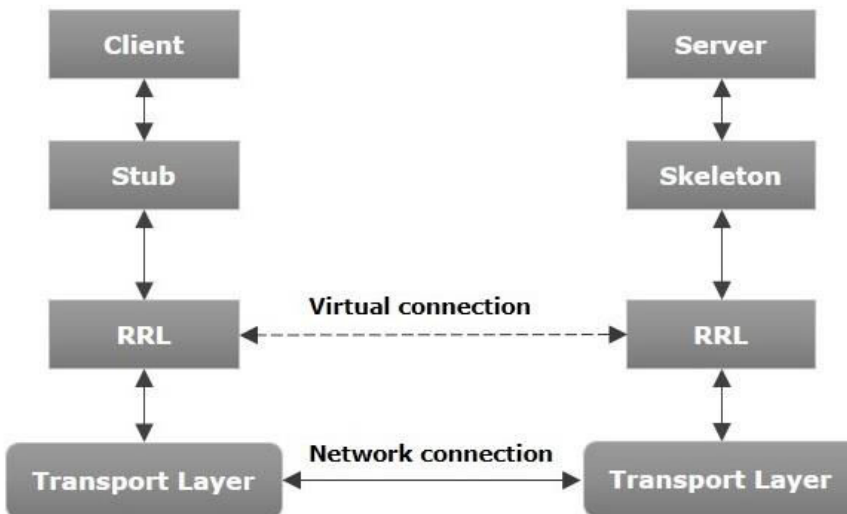
RMI is used to build distributed applications; it provides remote communication between Java programs. It is provided in the package **java.rmi**.

Architecture of an RMI Application

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

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

The following diagram shows the architecture of an RMI application.



Let us now discuss the components of this architecture.

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

Working of an RMI Application

The following points summarize how an RMI application works –

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

Marshalling and Unmarshalling

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

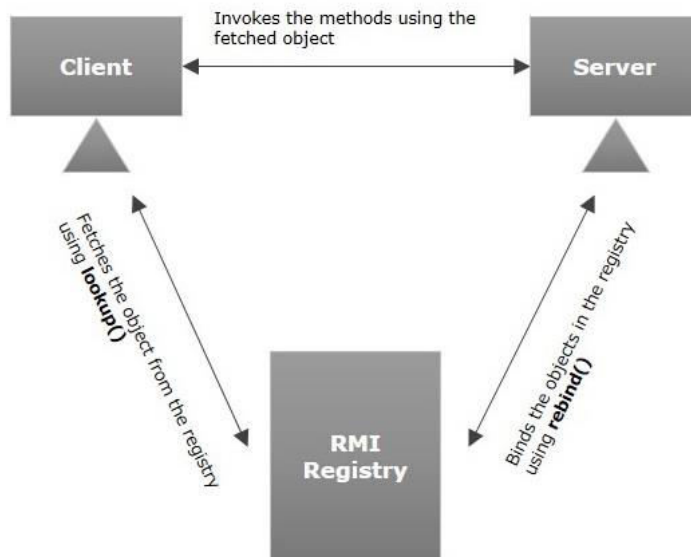
At the server side, the packed parameters are unbundled and then the required method is invoked. This process is known as **unmarshalling**.

RMI Registry

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

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

The following illustration explains the entire process –



Goals of RMI

Following are the goals of RMI –

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

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

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

Defining the Remote Interface

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

To create a remote interface –

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

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

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

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

Developing the Implementation Class (Remote Object)

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

To develop an implementation class –

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

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

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

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

Developing the Server Program

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

To develop a server program –

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

Following is an example of an RMI server program.

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

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

            // Exporting the object of implementation class
            // (here we are exporting the remote object to the stub)
        }
    }
}
```

```

    Hello stub = (Hello) UnicastRemoteObject.exportObject(obj,
0);

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

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

```

Developing the Client Program

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

To develop a client program –

- Create a client class from where your intended to invoke the remote object.
- Get the RMI registry using the **getRegistry()** method of the **LocateRegistry** class which belongs to the package **java.rmi.registry**.
- Fetch the object from the registry using the method **lookup()** of the class **Registry** which belongs to the package **java.rmi.registry**.
To this method, you need to pass a string value representing the bind name as a parameter. This will return you the remote object.
- The lookup() returns an object of type remote, down cast it to the type Hello.
- Finally invoke the required method using the obtained remote object.

Following is an example of an RMI client program.

```

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

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

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

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

```

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

Compiling the Application

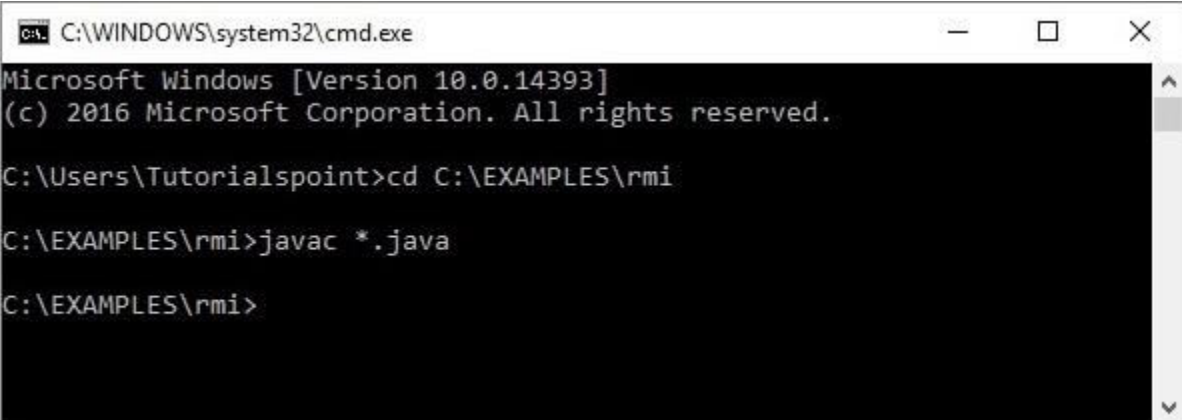
To compile the application –

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

Or,

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

```
Javac *.java
```



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

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

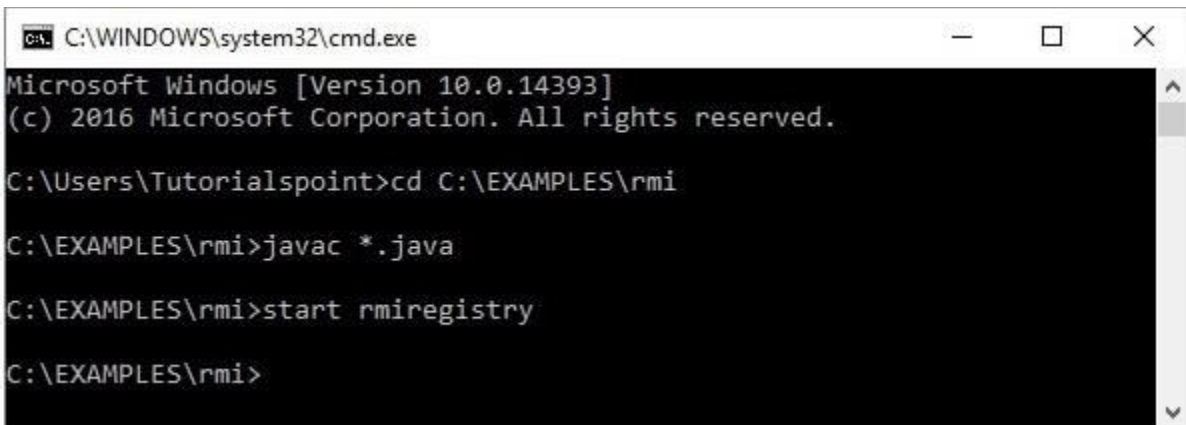
C:\EXAMPLES\rmi>javac *.java

C:\EXAMPLES\rmi>
```

Executing the Application

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

```
start rmiregistry
```



```

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

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

C:\EXAMPLES\rmi>javac *.java

C:\EXAMPLES\rmi>start rmiregistry

C:\EXAMPLES\rmi>

```

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



```

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

```

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

Java Server



```

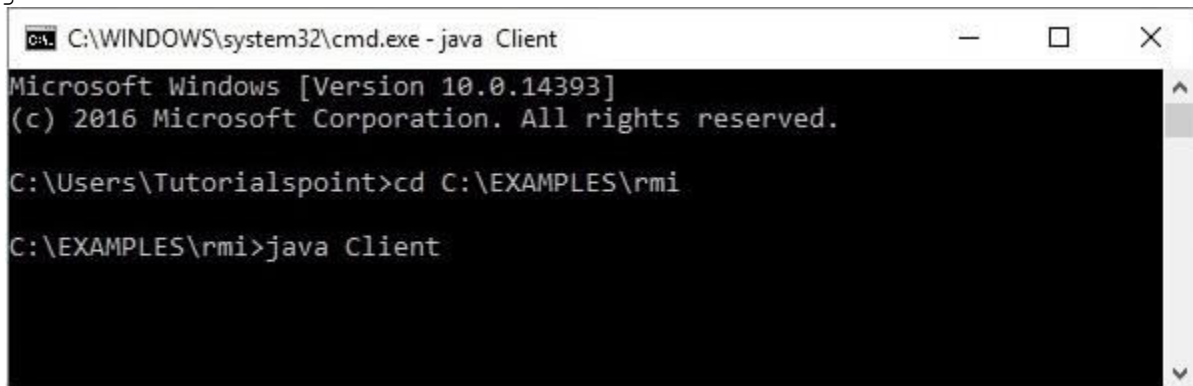
C:\WINDOWS\system32\cmd.exe - java Server

C:\EXAMPLES\rmi>java Server
Server ready
_

```

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

java Client



```

C:\WINDOWS\system32\cmd.exe - java Client

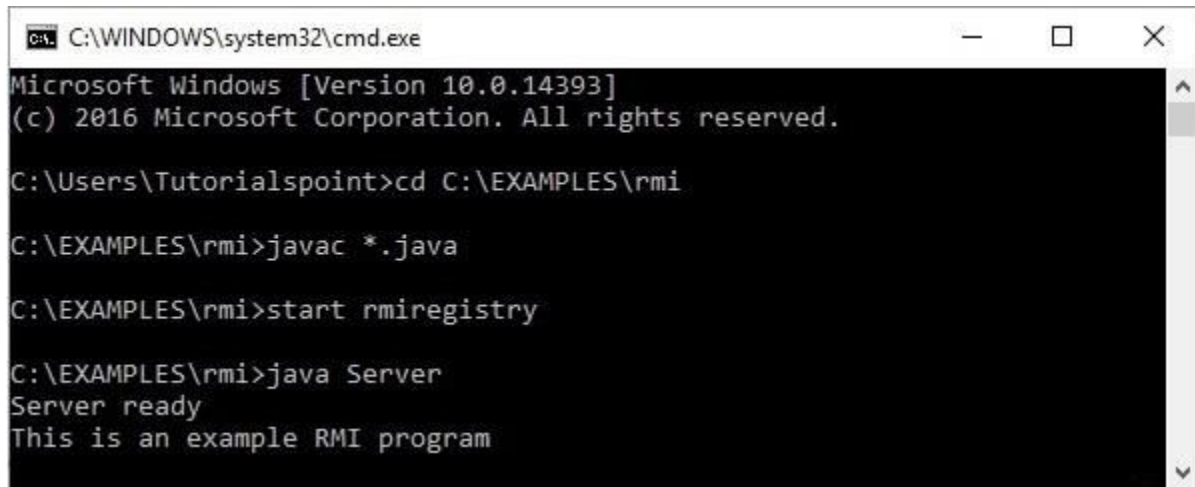
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

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

C:\EXAMPLES\rmi>java Client

```

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



```

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

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

C:\EXAMPLES\rmi>javac *.java

C:\EXAMPLES\rmi>start rmiregistry

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

```

RMI applications

1. When the client makes a call to the remote object, it is received by the stub which eventually passes this request to the RRL.

When the client-side RRL receives the request, it invokes a method called **invoke()** of the object **remoteRef**. It passes the request to the RRL on the server side.

The RRL on the server side passes the request to the Skeleton (proxy on the server) which finally invokes the required object on the server.

The result is passed all the way back to the client.

Marshalling and Unmarshalling

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

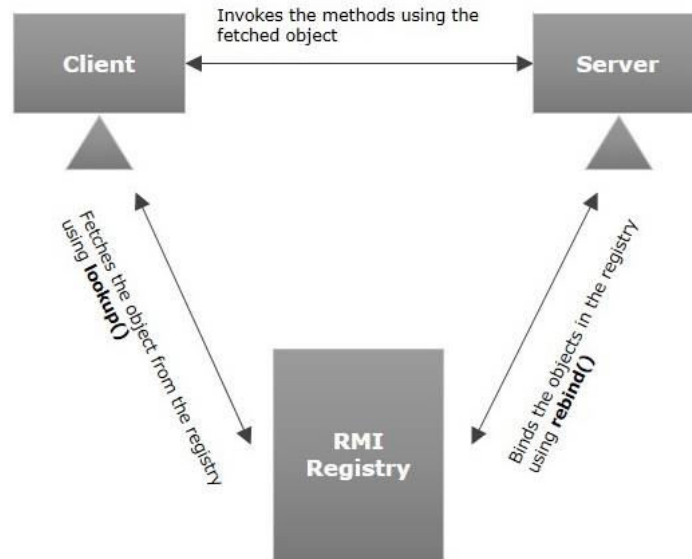
At the server side, the packed parameters are unbundled and then the required method is invoked. This process is known as **unmarshalling**.

RMI Registry

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

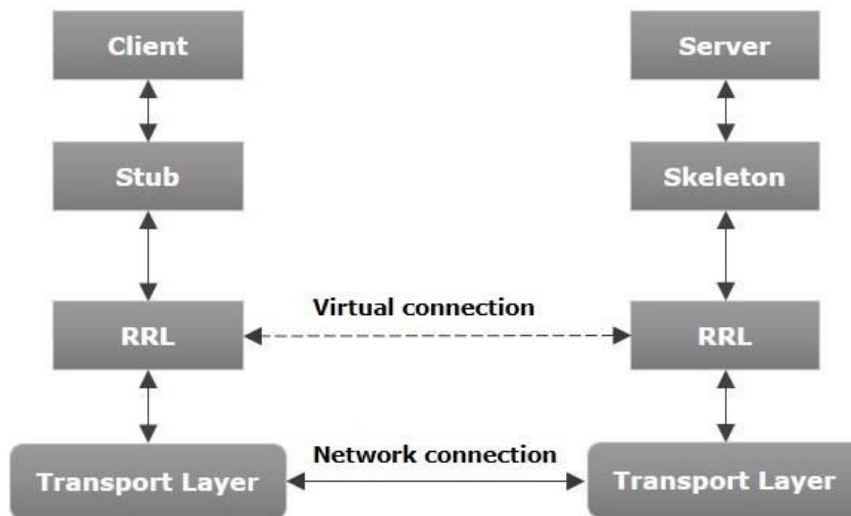
To invoke a remote object, the client needs a reference of that object. At that time, the client fetches the object from the registry using its bind name (using **lookup()** method).

The following illustration explains the entire process –



Components of RMI

1. Transport Layer – This layer connects the client and the server. It manages the existing connection and also sets up new connections.
 Stub – A stub is a representation (proxy) of the remote object at client. It resides in the client system; it acts as a gateway for the client program.
 Skeleton – This is the object which resides on the server side. stub communicates with this skeleton to pass request to the remote object.
 RRL(Remote Reference Layer) – It is the layer which manages the references made by the client to the remote object.



Implementation class for remote object of RMI

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

To develop an implementation class –

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

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

```
// Implementing the remote interface

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

Unit V JSP - JavaBeans

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	<p>getPropertyName()</p> <p>For example, if property name is <i>firstName</i>, your method name would be getFirstName() to read that property. This method is called accessor.</p>
2	<p>setPropertyName()</p> <p>For example, if property name is <i>firstName</i>, your method name would be setFirstName() to write that property. This method is called mutator.</p>

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
```

JSP - Custom Tags

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!

JSP - Expression Language (EL)

JSP Expression Language (EL) 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")}
```

To 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>
```