



## P.S.R. ENGINEERING COLLEGE

(An Autonomous Institution, Affiliated to Anna University,  
Chennai)

Sevalpatti (P.O), Sivakasi - 626140.



### COURSE MATERIAL

<b>Subject Name</b>	: ADVANCED JAVA PROGRAMMING
<b>Subject Code</b>	: 191CSEB
<b>YEAR/SEM</b>	IV/VII
<b>Faculty Name</b>	: Amutha J AP/CSE

**191CSEB**

**ADVANCED JAVA PROGRAMMING**

**L T P C**  
**3 0 0 3**

**Programme:** B.E. Computer Science and Engineering **Sem** : - **Category:** PE

**Prerequisites:** 191CS43 – Object Oriented Programming

**Aim:** To design and develop enterprise strength distributed and multitier applications using Java Technology.

**Course Outcomes:** The Students will be able to

**CO1:** Gain the basic concepts of Java..

**CO2:** Learn advanced Java programming concepts like RMI, Collections etc.

**CO3:** Develop network programs in Java.

**CO4:** Relate the concepts needed for distributed and multi-tier applications.

**CO5:** Solve the real world problems using concepts like JDBC, JNDI.

**CO6:** Explore the features of various platforms and frameworks like hibernate, Java Server Face used in web applications development.

**COLLECTIONS & NETWORKING** 9

**Collections:** Collection Interfaces, Concrete Collections, The Collections Framework.

**Networking:** Internet Addressing – Inet Address - Factory Methods - Instance Methods - TCP/IP Client Sockets - URL - URL Connection - TCP/IP Server Sockets.

**DISTRIBUTED APPLICATIONS** 9

Introduction to J2EE - Enterprise Java Bean: Preparing a Class to be a JavaBean, Creating a JavaBean, JavaBean Properties - Types of beans - Stateful Session bean, Stateless Session bean, and Entity bean. **CORBA:** Technical/Architectural Overview-RMI-IIOP.

**JAVA DATABASE CONNECTIVITY** 9

Java Database Connectivity (JDBC): Merging Data from Multiple Tables: Joining, Manipulating - Databases with JDBC, Transaction Processing, Retrieve the employee details from the Server through JDBC Driver. Java Naming and Directory Interface - Naming concepts - directory concepts - JNDI Interface - Example.

**SERVER SIDE PROGRAMMING** 9

Servlets - Introduction to servlets - Servlets life cycle – Java Server Pages (JSP): Introduction, Java Server Pages Overview, First Java Server Page Example, Implicit Objects, Scripting, Standard Actions, Directives, Custom Tag Libraries

**RECENT JAVA TOOLS** 9

**Spring Boot** - Deploying a Spring-Boot application running with Java8 - Hibernate: Introduction to Hibernate 3.0 - Hibernate Architecture - First Hibernate Application. Java Server Faces - Installing application - writing - deploying and testing application - Request Process life cycle - Basic JSF Tags - Expression Language.

**Total Periods: 45**

**Text Books:**

1. Uttam K. Roy, “Advanced Java Programming”, Oxford University press, 2015.

**References:**

1. Elliott Rusty Harold, “Java Network Programming”, O’Reilly Publishers, 4/e, 2013.
2. Ed Roman, “Mastering Enterprise Java Beans”, John Wiley & Sons Inc., 3/e, 2004.
3. S. Malhotra and S. Choudhary, “Programming in Java”, Oxford University Press. 2/e, 2014.

## UNIT-I

### **COLLECTIONS & NETWORKING**

**Collections:** Collection Interfaces, Concrete Collections, The Collections Framework.

**Networking:** Internet Addressing – Inet Address - Factory Methods - Instance Methods - TCP/IP Client Sockets - URL - URL Connection - TCP/IP Server Sockets.

- The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.
- Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.
- Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes ( ArrayList, Vector, LinkedList, PriorityQueue , HashSet, LinkedHashSet, TreeSet).

#### **What is Collection in Java**

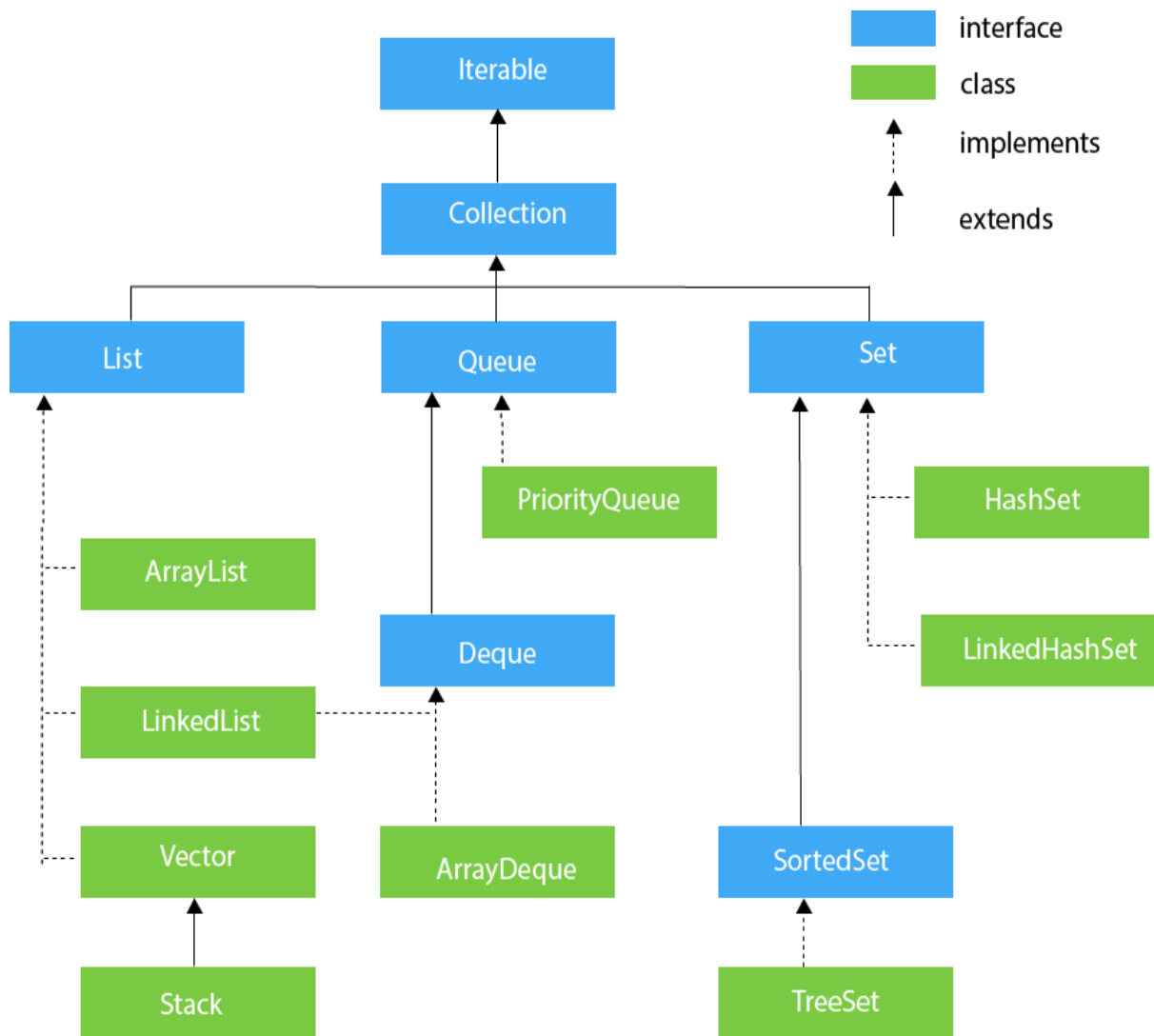
- A Collection represents a single unit of objects, i.e., a group.

#### **What is a framework in Java**

- It provides readymade architecture.
- It represents a set of classes and interfaces.
- It is optional.

#### **What is Collection framework**

- The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:
- Interfaces and its implementations, i.e., classes, Algorithm



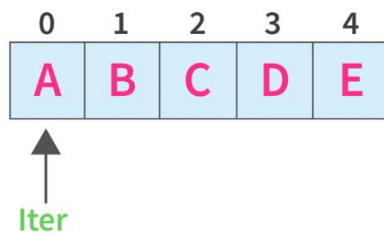
### Iterable interface:

The Iterable interface is present in java.lang.Iterable package. It was introduced in JDK 1.5. It allows users to iterate through elements individually. Using Iterable, elements of collections like arrays, sets, queues, maps, etc. can be traversed easily.

The Collection framework extends the Iterable interface. Thus, all the classes implementing the collections framework also implement the Iterable interface, and objects of these classes can use Java's Iterable feature.

By using Iterator, we can access each item in the collection, one item at a time. Here, we have an array of 5 elements, iterator named iter of this array returns each element of the array one by one starting from 0<sup>th</sup> index. Iterate through elements sequentially from a collection. It returns each element of the collection one after the other, beginning from the front and moving

forward. There are three ways in which elements can be iterated in Java: the enhanced for loop, the `forEach()` method, and the `iterator()` method. The `Collection` interface extends the `Iterable` interface thus, all the classes implementing the `Collection` interface are iterable.



### Methods of Iterator

The `Iterator` interface provides 4 methods that can be used to perform various operations on elements of collections.

`hasNext()` - returns true if there exists an element in the collection

`next()` - returns the next element of the collection

`remove()` - removes the last element returned by the `next()`

`forEachRemaining()` - performs the specified action for each remaining element of the collection

package testing;

```
import java.util.ArrayList;
```

```
import java.util.Iterator;
```

```
public class arraylist_main {
```

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

```
        // TODO Auto-generated method stub
```

```
        ArrayList<Integer> numbers = new ArrayList<>();
```

```
        numbers.add(1);
```

```
        numbers.add(3);
```

```
        numbers.add(2);
```

```
        System.out.println("ArrayList: " + numbers);
```

```
        // Creating an instance of Iterator
```

```

Iterator<Integer> iterate = numbers.iterator();

// Using the next() method
int number = iterate.next();

System.out.println("Accessed Element: " + number);

// Using the remove() method
iterate.remove();

System.out.println("Removed Element: " + number);

System.out.print("Updated ArrayList: "+numbers);

    }

}

```

**Output:**

```

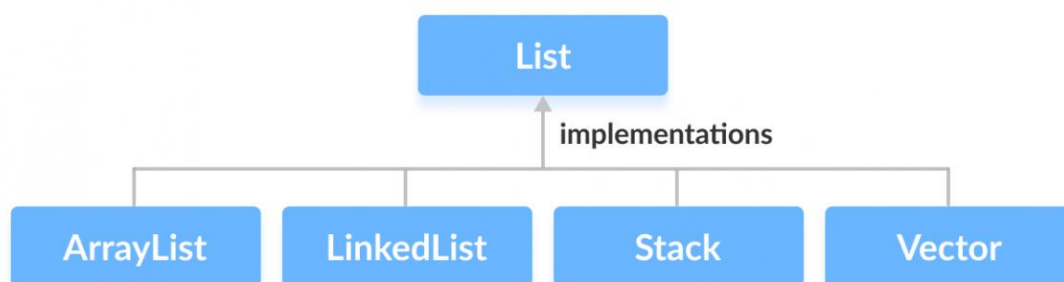
ArrayList: [1, 3, 2]
Accessed Element: 1
Removed Element: 1
Updated ArrayList: [3, 2]

```

**Classes that Implement List**

Since List is an interface, we cannot create objects from it.

In order to use the functionalities of the List interface, we can use these classes:



These classes are defined in the Collections framework and implement the List interface.

How to use List?

In Java, we must import java.util.List package in order to use List.

```
// ArrayList implementation of List
```

```
List<String> list1 = new ArrayList<>();
```

```
// LinkedList implementation of List
```

```
List<String> list2 = new LinkedList<>();
```

Here, we have created objects list1 and list2 of classes ArrayList and LinkedList. These objects can use the functionalities of the List interface.

### Methods of List

The List interface includes all the methods of the Collection interface. Its because Collection is a super interface of List.

Some of the commonly used methods of the Collection interface that's also available in the List interface are:

Methods	Description
<a href="#">add()</a>	adds an element to a list
<a href="#">addAll()</a>	adds all elements of one list to another
<a href="#">get()</a>	helps to randomly access elements from lists
<a href="#">iterator()</a>	returns <a href="#">iterator</a> object that can be used to sequentially access elements of lists
<a href="#">set()</a>	changes elements of lists
<a href="#">remove()</a>	removes an element from the list
<a href="#">removeAll()</a>	removes all the elements from the list
<a href="#">clear()</a>	removes all the elements from the list (more efficient than removeAll())
<a href="#">size()</a>	returns the length of lists
<a href="#">toArray()</a>	converts a list into an array
<a href="#">contains()</a>	returns true if a list contains specific element

## Implementation of the List Interface

### 1. Implementing the ArrayList Class

```
package testing;

import java.util.ArrayList;
import java.util.List;

public class arraylist1 {

    public static void main(String[] args)
    {

        List<Integer> numbers=new ArrayList<>();
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        numbers.add(4);

        System.out.println("Arraylist:"+numbers);
        System.out.println("Arraylist:"+numbers.get(2));
        System.out.println("Arraylist:"+numbers.remove(1));
        System.out.println("Arraylist:"+numbers);

    } }
```

Output:

Arraylist:[1, 2, 3, 4]

Arraylist:3

Arraylist:2

Arraylist:[1, 3, 4]

### 2. Implementing the LinkedList Class

```
package testing;

import java.util.LinkedList;
import java.util.List;

public class list_Linkedlist {

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



```

List<Integer> n=new LinkedList<>();
n.add(1);
n.add(2);
n.add(3);
System.out.println("List"+n);
System.out.println("Access Element"+n.get(1));
//System.out.println("List"+n.indexOf(2));
System.out.println("List"+n.removeFirst());
System.out.println("List"+n);
System.out.println("Last element List"+n.getLast());
//System.out.println("List"+n);
System.out.println("Insert the index 1 position "+n.set(1,4));
System.out.println("List"+n);
n.add(7);
System.out.println("Add the new element"+n);
}
}

```

OUTPUT:

List[1, 2, 3]

Access Element2

List1

List[2, 3]

Last element List3

Insert the index 1 position 3

List[2, 4]

Add the new element[2, 4, 7]

## Vector

Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.

```
package testing;
import java.util.Iterator;
import java.util.Vector;
public class List_vector {
public static void main(String[] args) {
    // TODO Auto-generated method stub
    Vector<String> v=new Vector<>();
    v.add("kumar");
    v.add("ram");
    v.add("sri");
    Iterator<String> itr=v.iterator();
    while(itr.hasNext())
    {
        System.out.println(itr.next());
    }
}
}
```

#### **OUTPUT:**

kumar

ram

sri

#### **Stack**

The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

Consider the following example.

```
package testing;
import java.util.Iterator;
```

```

import java.util.Stack;
public class List_stack {
    public static void main(String[] args) {
        Stack<String> s=new Stack<>();
        s.push("sri");
        s.push("ram");
        s.push("anu");
        System.out.println("stack "+s);
        s.pop();
        System.out.println("stack "+s);
        Iterator<String> i=s.iterator();
        System.out.println("Iterator point the Stack ");
        while(i.hasNext())
        {
            System.out.println(i.next());
        }
    }
}

```

### **Output:**

stack [sri, ram, anu]

stack [sri, ram]

Iterator point the Stack

sri

ram

### **Queue Interface**

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

Queue interface can be instantiated as:

```
Queue<String> q1 = new PriorityQueue();
```

```
Queue<String> q2 = new ArrayDeque();
```

There are various classes that implement the Queue interface, some of them are given below.

### PriorityQueue

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

Consider the following example.

```
package testing;

import java.util.Iterator;
import java.util.PriorityQueue;

public class Collection_PriorityQueue {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        PriorityQueue<String> q1=new PriorityQueue<>();
        q1.add("sivakasi");
        q1.add("chennai");
        q1.add("madurai");
        System.out.println("Head element in queue\n"+q1.element());
        System.out.println("Peek element in queue\n"+q1.peek());
        System.out.println("Display the list of queue\n");
        for(String str:q1)
        {
            System.out.println(str);
        }

        //q1.addAll(q1);
        Iterator i=q1.iterator();
        System.out.println("Priority Queue List items\n");
        while(i.hasNext())
        {
```

```
        System.out.println(i.next());
    }
    System.out.println("Pop operation after element");
    q1.remove();
    q1.poll();
    Iterator i1=q1.iterator();
    while(i1.hasNext())
    {
        System.out.println(i1.next());
    }
}
}
```

## **OUTPUT**

Head element in queue

chennai

Peek element in queue

chennai

Display the list of queue

chennai

sivakasi

madurai

Priority Queue List items

chennai

sivakasi

madurai

Pop operation after element

Sivakasi

## **Deque Interface**

Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

Deque can be instantiated as:

1. Deque d = **new** ArrayDeque();

## **ArrayDeque**

ArrayDeque class implements the Deque interface. It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends.

ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

Consider the following example.

```
package testing;
import java.util.ArrayDeque;
import java.util.Deque
public class Collection_Arrayqueue {
    public static void main(String[] args) {
        Deque<String> de=new ArrayDeque<String>();
        de.add("jeni");
        de.add("aarthi");
        de.add("sri");
        System.out.println("Dequeue Element list");
        for(String str:de)
        {
            System.out.println(str);
        }
        System.out.println("Dequeue remove the first element and poll element:");
        de.removeFirst();
        de.pollLast();
        for(String str1:de)
        {
            System.out.println(str1);
        }
    }
}
```

## OUTPUT:

### Dequeue Element list

jeni

aarathi

sri

Dequeue remove the first element and poll element:

Aarathi

## Set Interface

Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

1. **Set<data-type> s1 = new HashSet<data-type>();**
2. **Set<data-type> s2 = new LinkedHashSet<data-type>();**
3. **Set<data-type> s3 = new TreeSet<data-type>();**

## HashSet

HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items.

Consider the following example

```
package testing;
import java.util.HashSet;
import java.util.Iterator;
public class Set_hashset {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        HashSet<String> set=new HashSet<>();
        set.add("sivakasi");
        set.add("chennai");
```

```
        set.add("sivakasi");
        set.add("theni");
        set.add("madurai");
        System.out.println("Display the city name:");
        for(String str:set)
        {
            System.out.println(str);
        }
        System.out.println("Iterator class based display the city details");
        Iterator i=set.iterator();
        while(i.hasNext())
        {
            System.out.println(i.next());
        }
    }
}
```

### **OUTPUT:**

Display the city name:

madurai

sivakasi

chennai

theni

Iterator class based display the city details

madurai

sivakasi

chennai

theni

### **LinkedHashSet**



LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

Consider the following example.

```
import java.util.*;

public class TestJavaCollection8 {

public static void main(String args[]){

LinkedHashSet<String> set=new LinkedHashSet<String>();

set.add("Ravi");

set.add("Vijay");

set.add("Ravi");

set.add("Ajay");

Iterator<String> itr=set.iterator();

while(itr.hasNext()){

System.out.println(itr.next());

}

}

}
```

### **Output:**

Ravi

Vijay

Ajay

### **SortedSet Interface**

SortedSet is the alternate of Set interface that provides a total ordering on its elements. The elements of the SortedSet are arranged in the increasing (ascending) order. The SortedSet provides the additional methods that inhibit the natural ordering of the elements.

The SortedSet can be instantiated as:

```
SortedSet<data-type> set = new TreeSet();
```

### **TreeSet**

Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.

Consider the following example:

```
import java.util.*;

public class TestJavaCollection9 {

    public static void main(String args[]) {

        //Creating and adding elements

        TreeSet<String> set=new TreeSet<String>();

        set.add("Ravi");

        set.add("Vijay");

        set.add("Ravi");

        set.add("Ajay");

        //traversing elements

        Iterator<String> itr=set.iterator();

        while(itr.hasNext()){

            System.out.println(itr.next());

        }

    }

}
```

Output:

Ajay

Ravi

Vijay

### **Java InetAddress class**

**Java InetAddress** class represents an IP address. The `java.net.InetAddress` class provides methods to get the IP of any host name *for example* `www.javatpoint.com`, `www.google.com`, `www.facebook.com`, etc.

An IP address is represented by 32-bit or 128-bit unsigned number. An instance of `InetAddress` represents the IP address with its corresponding host name. There are two types of addresses:

Unicast and Multicast. The Unicast is an identifier for a single interface whereas Multicast is an identifier for a set of interfaces.

Moreover, InetAddress has a cache mechanism to store successful and unsuccessful host name resolutions.

## IP Address

- An IP address helps to identify a specific resource on the network using a numerical representation.
- Most networks combine IP with TCP (Transmission Control Protocol). It builds a virtual bridge among the destination and the source.

There are two versions of IP address:

### 1. IPv4

IPv4 is the primary Internet protocol. It is the first version of IP deployed for production in the ARPANET in 1983. It is a widely used IP version to differentiate devices on network using an addressing scheme. A 32-bit addressing scheme is used to store  $2^{32}$  addresses that is more than 4 million addresses.

#### Features of IPv4:

- It is a connectionless protocol.
- It utilizes less memory and the addresses can be remembered easily with the class based addressing scheme.
- It also offers video conferencing and libraries.

### 2. IPv6

IPv6 is the latest version of Internet protocol. It aims at fulfilling the need of more internet addresses. It provides solutions for the problems present in IPv4. It provides 128-bit address space that can be used to form a network of 340 undecillion unique IP addresses. IPv6 is also identified with a name IPng (Internet Protocol next generation).

#### Features of IPv6:

- It has a stateful and stateless both configurations.
- It provides support for quality of service (QoS).
- It has a hierarchical addressing and routing infrastructure.

## TCP/IP Protocol

- TCP/IP is a communication protocol model used connect devices over a network via internet.
- TCP/IP helps in the process of addressing, transmitting, routing and receiving the data packets over the internet.
- The two main protocols used in this communication model are:
  1. TCP i.e. Transmission Control Protocol. TCP provides the way to create a communication channel across the network. It also helps in transmission of packets at sender end as well as receiver end.
  2. IP i.e. Internet Protocol. IP provides the address to the nodes connected on the internet. It uses a gateway computer to check whether the IP address is correct and the message is forwarded correctly or not.

### Java InetAddress Class Methods

Method	Description
public static InetAddress getByName(String host) throws UnknownHostException	It returns the instance of InetAddress containing LocalHost IP and name.
public static InetAddress getLocalHost() throws UnknownHostException	It returns the instance of InetAddress containing local host name and address.
public String getHostName()	It returns the host name of the IP address.
public String getHostAddress()	It returns the IP address in string format.

### Example of Java InetAddress Class

Let's see a simple example of InetAddress class to get ip address of [www.javatpoint.com](http://www.javatpoint.com) website.

#### InetDemo.java

```
package testing;
import java.net.InetAddress;
public class inetaddress_1 {
    public static void main(String arg[])
    {
```

```
        try
        {
            InetAddress ip=InetAddress.getByName("www.tneaonline.org");
            System.out.println("Host Name: "+ip.getHostName());
            System.out.println("IP Address: "+ip.getHostAddress());
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
}
```

Output:

Host Name: www.tneaonline.org

IP Address: 3.109.215.89

## **Java DatagramSocket and DatagramPacket**

Java DatagramSocket and DatagramPacket classes are used for connection-less socket programming using the UDP instead of TCP.

### **Datagram**

Datagrams are collection of information sent from one device to another device via the established network. When the datagram is sent to the targeted device, there is no assurance that it will reach to the target device safely and completely. It may get damaged or lost in between. Likewise, the receiving device also never know if the datagram received is damaged or not. The UDP protocol is used to implement the datagrams in Java.

### **Java DatagramSocket class**

**Java DatagramSocket** class represents a connection-less socket for sending and receiving datagram packets. It is a mechanism used for transmitting datagram packets over network.`

A datagram is basically an information but there is no guarantee of its content, arrival or arrival time.

## Commonly used Constructors of DatagramSocket class

Method	Description
void bind(SocketAddress addr)	It binds the DatagramSocket to a specific address and port.
void close()	It closes the datagram socket.
void connect(InetAddress address, int port)	It connects the socket to a remote address for the socket.
void disconnect()	It disconnects the socket.
boolean getBroadcast()	It tests if SO_BROADCAST is enabled.
DatagramChannel getChannel()	It returns the unique DatagramChannel object associated with the datagram socket.
InetAddress getInetAddress()	It returns the address to where the socket is connected.
InetAddress getLocalAddress()	It gets the local address to which the socket is connected.
int getLocalPort()	It returns the port number on the local host to which the socket is bound.
SocketAddress getLocalSocketAddress()	It returns the address of the endpoint the socket is bound to.
int getPort()	It returns the port number to which the socket is connected.
int getReceiverBufferSize()	It gets the value of the SO_RCVBUF option for this DatagramSocket that is the buffer size used by the platform for input on the DatagramSocket.
boolean isClosed()	It returns the status of socket i.e. closed or not.

boolean isConnected()	It returns the connection state of the socket.
void send(DatagramPacket p)	It sends the datagram packet from the socket.
void receive(DatagramPacket p)	It receives the datagram packet from the socket.

- **DatagramSocket() throws SocketEeption:** it creates a datagram socket and binds it with the available Port Number on the localhost machine.
- **DatagramSocket(int port) throws SocketEeption:** it creates a datagram socket and binds it with the given Port Number.
- **DatagramSocket(int port, InetAddress address) throws SocketEeption:** it creates a datagram socket and binds it with the specified port number and host address.

### Java DatagramSocket Class

### Java DatagramPacket Class

**Java DatagramPacket** is a message that can be sent or received. It is a data container. If you send multiple packet, it may arrive in any order. Additionally, packet delivery is not guaranteed.

### Commonly used Constructors of DatagramPacket class

- **DatagramPacket(byte[] barr, int length):** it creates a datagram packet. This constructor is used to receive the packets.
- **DatagramPacket(byte[] barr, int length, InetAddress address, int port):** it creates a datagram packet. This constructor is used to send the packets.

### Java DatagramPacket Class Methods

Method	Description
1) InetAddress getAddress()	It returns the IP address of the machine to which the datagram is being sent or from which the datagram was received.
2) byte[] getData()	It returns the data buffer.

3) int getLength()	It returns the length of the data to be sent or the length of the data received.
4) int getOffset()	It returns the offset of the data to be sent or the offset of the data received.
5) int getPort()	It returns the port number on the remote host to which the datagram is being sent or from which the datagram was received.
6) SocketAddress getSocketAddress()	It gets the SocketAddress (IP address + port number) of the remote host that the packet is being sent to or is coming from.
7) void setAddress(InetAddress iaddr)	It sets the IP address of the machine to which the datagram is being sent.
8) void setData(byte[] buff)	It sets the data buffer for the packet.
9) void setLength(int length)	It sets the length of the packet.
10) void setPort(int iport)	It sets the port number on the remote host to which the datagram is being sent.
11) void setSocketAddress(SocketAddress addr)	It sets the SocketAddress (IP address + port number) of the remote host to which the datagram is being sent.

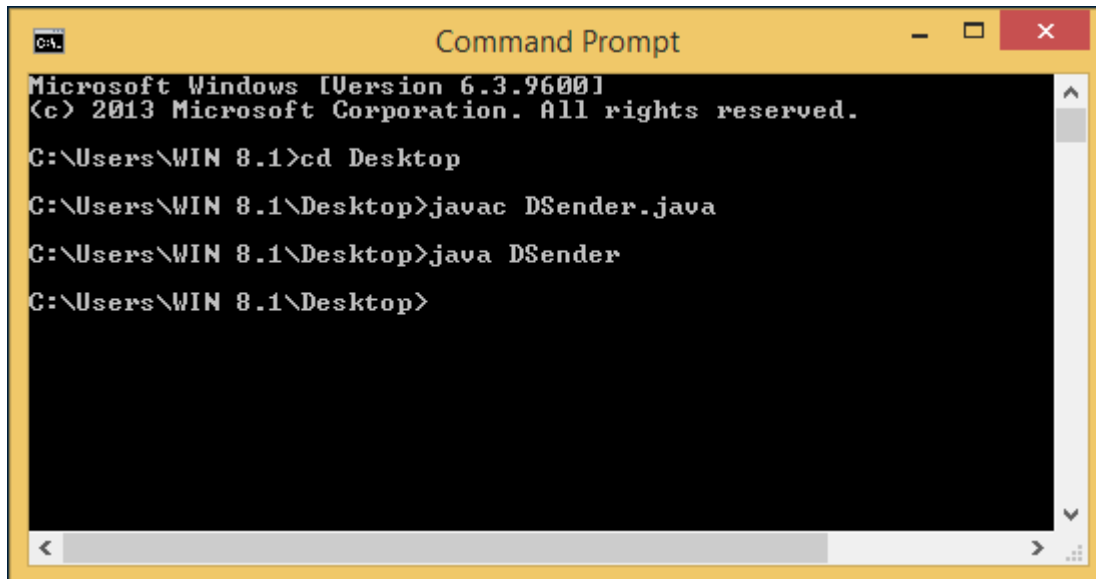
### Example of Sending DatagramPacket by DatagramSocket

1. //DSender.java
2. **import** java.net.\*;
3. **public class** DSender{
4.   **public static void** main(String[] args) **throws** Exception {
5.     DatagramSocket ds = **new** DatagramSocket();
6.     String str = "Welcome java";
7.     InetAddress ip = InetAddress.getByName("127.0.0.1");
- 8.



9. DatagramPacket dp = **new** DatagramPacket(str.getBytes(), str.length(), ip, 3000);
10. ds.send(dp);
11. ds.close();
12. }
13. }

**Output:**



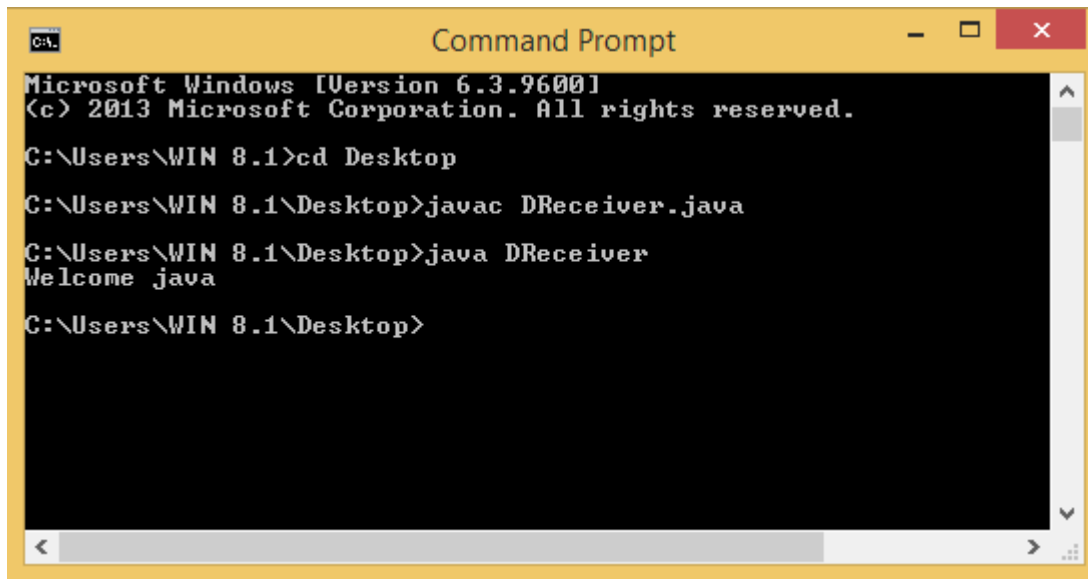
```
C:\>
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\WIN 8.1>cd Desktop
C:\Users\WIN 8.1\Desktop>javac DSender.java
C:\Users\WIN 8.1\Desktop>java DSender
C:\Users\WIN 8.1\Desktop>
```

**Example of Receiving DatagramPacket by DatagramSocket**

1. //DReceiver.java
2. **import** java.net.\*;
3. **public class** DReceiver{
4. **public static void** main(String[] args) **throws** Exception {
5. DatagramSocket ds = **new** DatagramSocket(3000);
6. **byte**[] buf = **new byte**[1024];
7. DatagramPacket dp = **new** DatagramPacket(buf, 1024);
8. ds.receive(dp);
9. String str = **new** String(dp.getData(), 0, dp.getLength());
10. System.out.println(str);
11. ds.close();
12. }
13. }

**Output:**



```
CA. Command Prompt
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.
C:\Users\WIN 8.1>cd Desktop
C:\Users\WIN 8.1\Desktop>javac DReceiver.java
C:\Users\WIN 8.1\Desktop>java DReceiver
Welcome java
C:\Users\WIN 8.1\Desktop>
```

## Java Socket Programming

Java Socket programming is used for communication between the applications running on different JRE.

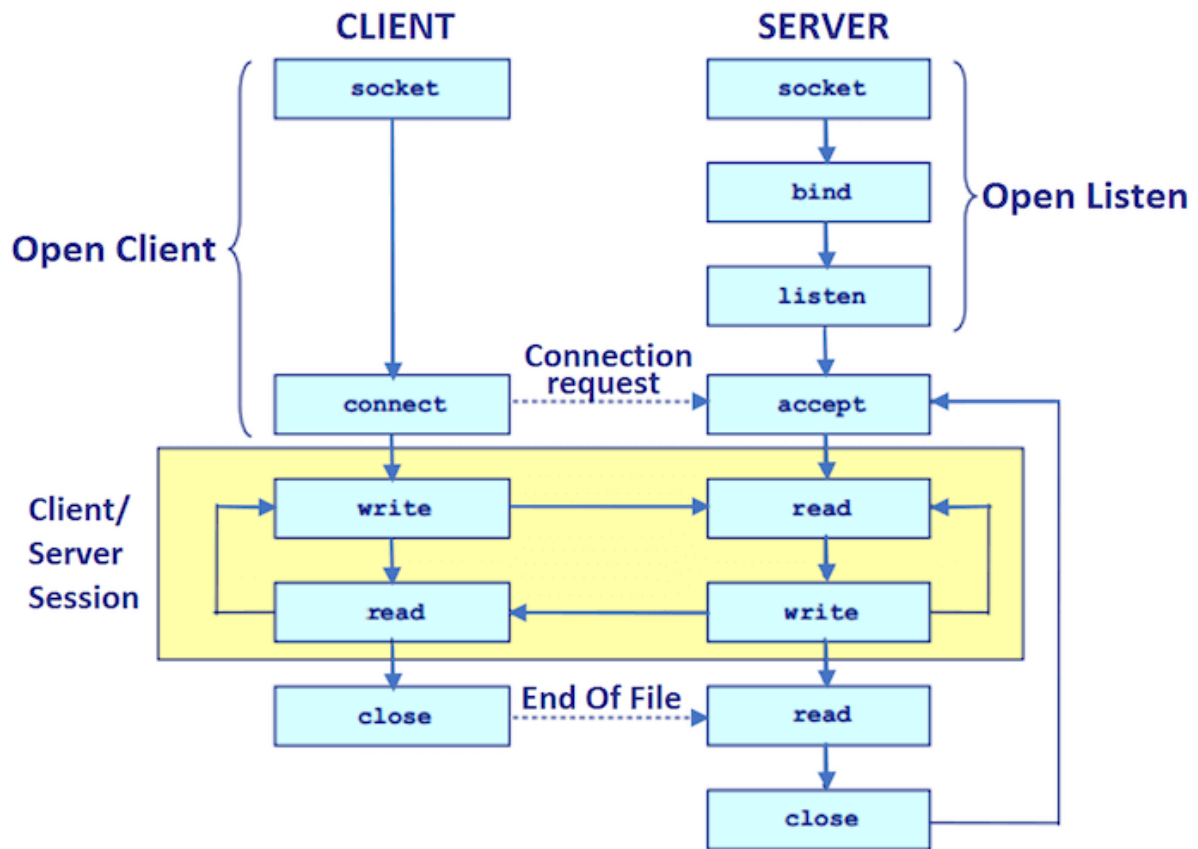
Java Socket programming can be connection-oriented or connection-less.

Socket and ServerSocket classes are used for connection-oriented socket programming and DatagramSocket and DatagramPacket classes are used for connection-less socket programming.

The client in socket programming must know two information:

1. IP Address of Server, and
2. Port number.

Here, we are going to make one-way client and server communication. In this application, client sends a message to the server, server reads the message and prints it. Here, two classes are being used: Socket and ServerSocket. The Socket class is used to communicate client and server. Through this class, we can read and write message. The ServerSocket class is used at server-side. The accept() method of ServerSocket class blocks the console until the client is connected. After the successful connection of client, it returns the instance of Socket at server-side.



## SOCKET API

### Socket class

A socket is simply an endpoint for communications between the machines. The Socket class can be used to create a socket.

### Important methods

Method	Description
1) public InputStream getInputStream()	returns the InputStream attached with this socket.
2) public OutputStream getOutputStream()	returns the OutputStream attached with this socket.
3) public synchronized void close()	closes this socket

### ServerSocket class

The ServerSocket class can be used to create a server socket. This object is used to establish communication with the clients.

### Important methods

Method	Description
1) public Socket accept()	returns the socket and establish a connection between server and client.
2) public synchronized void close()	closes the server socket.

### Example of Java Socket Programming

#### Creating Server:

To create the server application, we need to create the instance of ServerSocket class. Here, we are using 6666 port number for the communication between the client and server. You may also choose any other port number. The accept() method waits for the client. If clients connects with the given port number, it returns an instance of Socket.

1. ServerSocket ss=**new** ServerSocket(6666);
2. Socket s=ss.accept();//establishes connection and waits for the client

#### Creating Client:

To create the client application, we need to create the instance of Socket class. Here, we need to pass the IP address or hostname of the Server and a port number. Here, we are using "localhost" because our server is running on same system.

1. Socket s=**new** Socket("localhost",6666);

Let's see a simple of Java socket programming where client sends a text and server receives and prints it.

*File: MyServer.java*

1. **import** java.io.\*;
2. **import** java.net.\*;
3. **public class** MyServer {
4. **public static void** main(String[] args){
5. **try**{
6. ServerSocket ss=**new** ServerSocket(6666);
7. Socket s=ss.accept();//establishes connection
8. DataInputStream dis=**new** DataInputStream(s.getInputStream());
9. String str=(String)dis.readUTF();
10. System.out.println("message= "+str);
11. ss.close();
12. }**catch**(Exception e){System.out.println(e);}
13. }
14. }

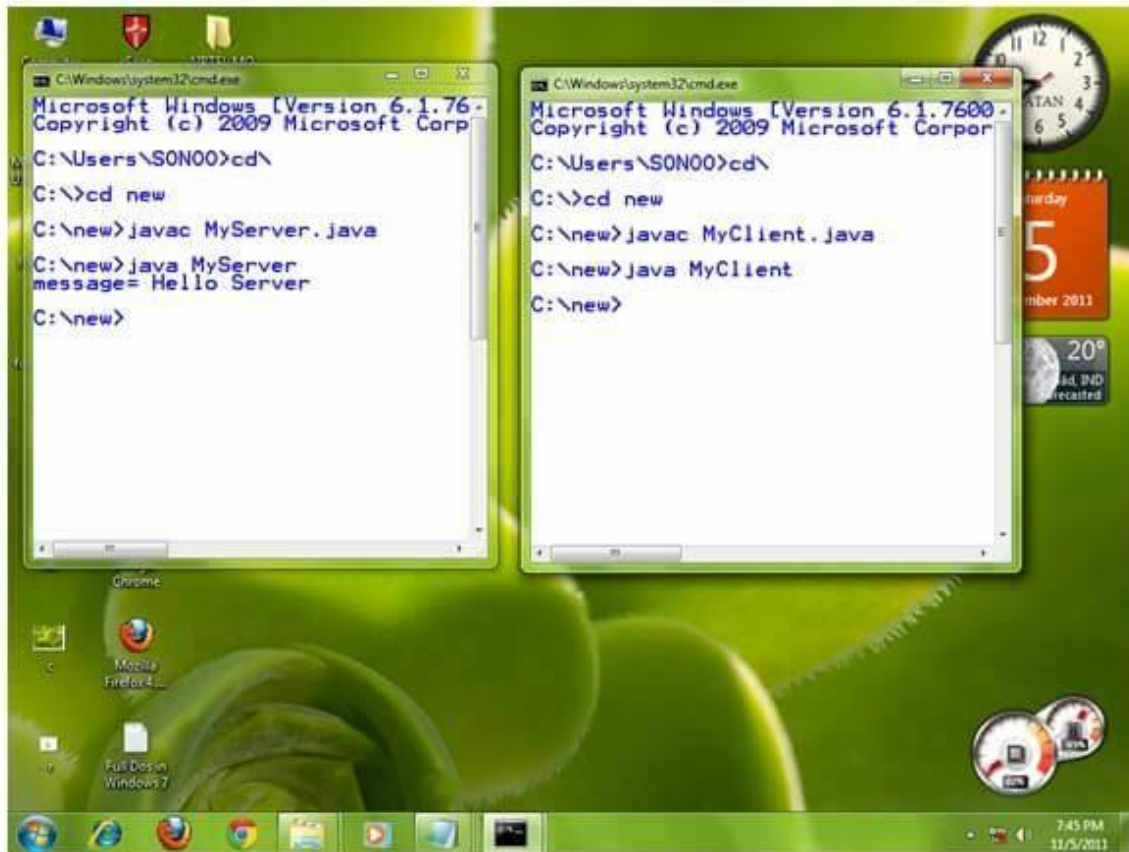
*File: MyClient.java*

1. **import** java.io.\*;
2. **import** java.net.\*;
3. **public class** MyClient {
4. **public static void** main(String[] args) {
5. **try**{
6. Socket s=**new** Socket("localhost",6666);
7. DataOutputStream dout=**new** DataOutputStream(s.getOutputStream());
8. dout.writeUTF("Hello Server");
9. dout.flush();
10. dout.close();
11. s.close();
12. }**catch**(Exception e){System.out.println(e);}
13. }
14. }

[download this example](#)

To execute this program open two command prompts and execute each program at each command prompt as displayed in the below figure.

After running the client application, a message will be displayed on the server console.



## Java URL

The **Java URL** class represents an URL. URL is an acronym for Uniform Resource Locator. It points to a resource on the World Wide Web. For example:

1. <https://www.javatpoint.com/java-tutorial>  

Protocol                      Host Name                      File

A URL contains many information:

1. **Protocol:** In this case, http is the protocol.
  2. **Server name or IP Address:** In this case, www.javatpoint.com is the server name.
  3. **Port Number:** It is an optional attribute. If we write <http://www.javatpoint.com:80/sonoojaiswal/>, 80 is the port number. If port number is not mentioned in the URL, it returns -1.
  4. **File Name or directory name:** In this case, index.jsp is the file name.
-

## Constructors of Java URL class

### **URL(String spec)**

Creates an instance of a URL from the String representation.

### **URL(String protocol, String host, int port, String file)**

Creates an instance of a URL from the given protocol, host, port number, and file.

### **URL(String protocol, String host, int port, String file, URLStreamHandler handler)**

Creates an instance of a URL from the given protocol, host, port number, file, and handler.

### **URL(String protocol, String host, String file)**

Creates an instance of a URL from the given protocol name, host name, and file name.

### **URL(URL context, String spec)**

Creates an instance of a URL by parsing the given spec within a specified context.

### **URL(URL context, String spec, URLStreamHandler handler)**

Creates an instance of a URL by parsing the given spec with the specified handler within a given context.

## Commonly used methods of Java URL class

The java.net.URL class provides many methods. The important methods of URL class are given below.

Method	Description
public String getProtocol()	it returns the protocol of the URL.
public String getHost()	it returns the host name of the URL.
public String getPort()	it returns the Port Number of the URL.
public String getFile()	it returns the file name of the URL.
public String getAuthority()	it returns the authority of the URL.
public String toString()	it returns the string representation of the URL.
public String getQuery()	it returns the query string of the URL.

<code>public String getDefaultPort()</code>	it returns the default port of the URL.
<code>public URLConnection openConnection()</code>	it returns the instance of URLConnection i.e. associated with this URL.
<code>public boolean equals(Object obj)</code>	it compares the URL with the given object.
<code>public Object getContent()</code>	it returns the content of the URL.
<code>public String getRef()</code>	it returns the anchor or reference of the URL.
<code>public URI toURI()</code>	it returns a URI of the URL.

### Example of Java URL class

```

package ip_concept;
import java.net.URL;
public class URLEDemo {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        try{
            URL url=new
URL("http://www.tneaonline.com/TNEARegistration/index");

            System.out.println("Protocol: "+url.getProtocol());
            System.out.println("Host Name: "+url.getHost());
            System.out.println("Port Number: "+url.getPort());
            System.out.println("File Name: "+url.getFile());

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

    }
}

```

Protocol: http



Host Name: www.tneaonline.com

Port Number: -1

File Name: /TNEARegistration/index

## Java URLConnection Class

The **Java URLConnection** class represents a communication link between the URL and the application. It can be used to read and write data to the specified resource referred by the URL.

### What is the URL?

- URL is an abbreviation for Uniform Resource Locator. An URL is a form of string that helps to find a resource on the World Wide Web (WWW).
- URL has two components:
  1. The protocol required to access the resource.
  2. The location of the resource.

### Features of URLConnection class

1. URLConnection is an abstract class. The two subclasses HttpURLConnection and JarURLConnection makes the connection between the client Java program and URL resource on the internet.
2. With the help of URLConnection class, a user can read and write to and from any resource referenced by an URL object.
3. Once a connection is established and the Java program has an URLConnection object, we can use it to read or write or get further information like content length, etc.

### Constructors

Constructor	Description
1) protected URLConnection(URL url)	It constructs a URL connection to the specified UR

### URLConnection Class Methods

Method	Description
--------	-------------

<code>void addRequestProperty(String key, String value)</code>	It adds a general request property specified by a key-value pair
<code>void connect()</code>	It opens a communications link to the resource referenced by this URL, if such a connection has not already been established.
<code>boolean getAllowUserInteraction()</code>	It returns the value of the <code>allowUserInteraction</code> field for the object.
<code>int getConnectionTimeout()</code>	It returns setting for connect timeout.
<code>Object getContent()</code>	It retrieves the contents of the URL connection.
<code>Object getContent(Class[] classes)</code>	It retrieves the contents of the URL connection.
<code>String getContentEncoding()</code>	It returns the value of the content-encoding header field.
<code>int getContentLength()</code>	It returns the value of the content-length header field.
<code>long getContentLengthLong()</code>	It returns the value of the content-length header field as long.
<code>String getContentType()</code>	It returns the value of the date header field.
<code>long getDate()</code>	It returns the value of the date header field.
<code>static boolean getDefaultAllowUserInteraction()</code>	It returns the default value of the <code>allowUserInteraction</code> field.
<code>boolean getDefaultUseCaches()</code>	It returns the default value of an <code>URLConnection</code> 's <code>useCaches</code> flag.

<code>boolean getDoInput()</code>	It returns the value of the <code>URLConnection</code> 's <code>doInput</code> flag.
<code>boolean getDoOutput()</code>	It returns the value of the <code>URLConnection</code> 's <code>doOutput</code> flag.
<code>long getExpiration()</code>	It returns the value of the expires header field.
<code>static FileNameMap getFilenameMap()</code>	It loads the filename map from a data file.
<code>String getHeaderField(int n)</code>	It returns the value of $n^{\text{th}}$ header field
<code>String getHeaderField(String name)</code>	It returns the value of the named header field.
<code>long getHeaderFieldDate(String name, long Default)</code>	It returns the value of the named field parsed as a number.
<code>int getHeaderFieldInt(String name, int Default)</code>	It returns the value of the named field parsed as a number.
<code>String getHeaderFieldKey(int n)</code>	It returns the key for the $n^{\text{th}}$ header field.
<code>long getHeaderFieldLong(String name, long Default)</code>	It returns the value of the named field parsed as a number.
<code>Map&lt;String, List&lt;String&gt;&gt; getHeaderFields()</code>	It returns the unmodifiable <code>Map</code> of the header field.
<code>long getIfModifiedSince()</code>	It returns the value of the object's <code>ifModifiedSince</code> field.
<code>InputStream getInputStream()</code>	It returns an input stream that reads from the open connection.
<code>long getLastModified()</code>	It returns the value of the last-modified header field.

OutputStream getOutputStream()	It returns an output stream that writes to the connection.
Permission getPermission()	It returns a permission object representing the permission necessary to make the connection represented by the object.
int getReadTimeout()	It returns setting for read timeout.
Map<String, List<String>> getRequestProperties()	It returns the value of the named general request property for the connection.
URL getURL()	It returns the value of the URLConnection's URL field.
boolean getUseCaches()	It returns the value of the URLConnection's useCaches field.
Static String guessContentTypeFromName(String fname)	It tries to determine the content type of an object, based on the specified <b>file</b> component of a URL.
static String guessContentTypeFromStream(InputStream is)	It tries to determine the type of an input stream based on the characters at the beginning of the input stream.
void setAllowUserInteraction(boolean allowuserinteraction)	It sets the value of the allowUserInteraction field of this URLConnection.
static void setContentHandlerFactory(ContentHandlerFactory fac)	It sets the ContentHandlerFactory of an application.
static void setDefaultAllowUserInteraction(boolean defaultallowuserinteraction)	It sets the default value of the allowUserInteraction field for all future URLConnection objects to the specified value.

<code>void setDefaultUseCaches(boolean defaultUseCaches)</code>	It sets the default value of the <code>useCaches</code> field to the specified value.
<code>void setDoInput(boolean doInput)</code>	It sets the value of the <code>doInput</code> field for this <code>URLConnection</code> to the specified value.
<code>void setDoOutput(boolean doOutput)</code>	It sets the value of the <code>doOutput</code> field for the <code>URLConnection</code> to the specified value.

### How to get the object of URLConnection Class

The `openConnection()` method of the `URL` class returns the object of `URLConnection` class.

#### Syntax:

1. **public** `URLConnection` `openConnection()` **throws** `IOException` { }

### Displaying Source Code of a Webpage by URLConnecton Class

The `URLConnection` class provides many methods. We can display all the data of a webpage by using the `getInputStream()` method. It returns all the data of the specified URL in the stream that can be read and displayed.

```
package ip_concept;

import java.io.InputStream;
import java.net.URL;
import java.net.URLConnection;

public class URLConnection_1 {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        try{
```

```

        URL url=new
URL("http://www.tneaonline.com/TNEARegistration/index");
        URLConnection urlcon=url.openConnection();
        InputStream stream=urlcon.getInputStream();
        int i;
        while((i=stream.read())!=-1){
        System.out.print((char)i);
        }
        }catch(Exception e){System.out.println(e);}
    }
}

```

Output(get the web page information)

```

<!DOCTYPE html>
<!--
Template Name: Metronic - Responsive Admin Dashboard Template build with Twitter
Bootstrap 3.3.4
Version: 4.0.2
Author: KeenThemes
Website: http://www.keenthemes.com/
Contact: support@keenthemes.com
Follow: www.twitter.com/keenthemes
Like: www.facebook.com/keenthemes
Purchase: http://themeforest.net/item/metronic-responsive-admin-dashboard-
template/4021469?ref=keenthemes
License: You must have a valid license purchased only from themeforest(the above link) in
order to legally use the theme for your project.
-->
<!--[if IE 8]> <html lang="en" class="ie8 no-js"> <![endif]-->

```

## What is a method in Java?

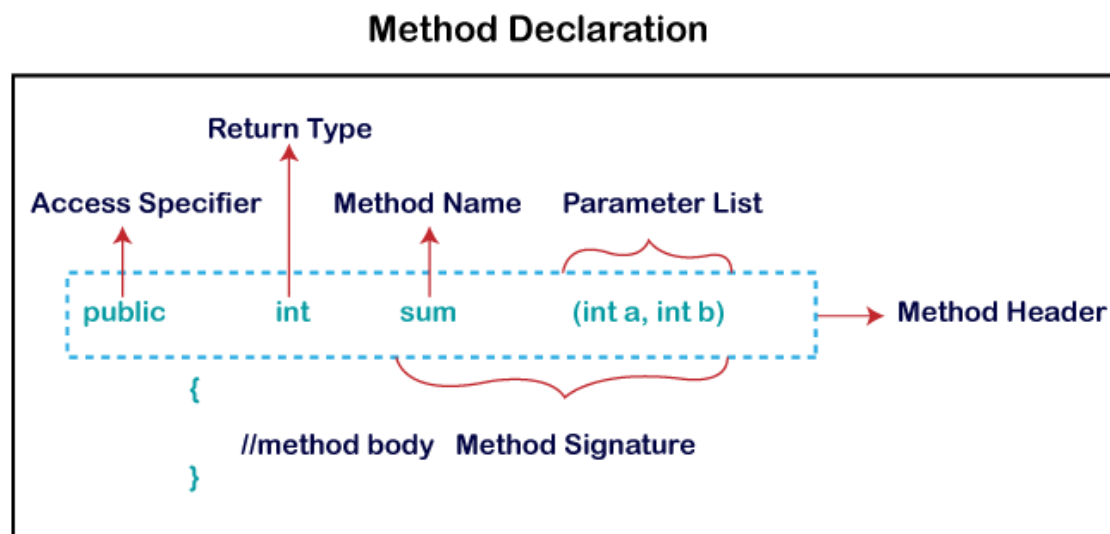
A **method** is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the **reusability** of code. We write a method once and use it many times. We do not require to write code again and again. It also

provides the **easy modification** and **readability** of code, just by adding or removing a chunk of code. The method is executed only when we call or invoke it.

The most important method in Java is the **main()** method. If you want to read more about the main() method, go through the link <https://www.javatpoint.com/java-main-method>.

## Method Declaration

The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments. It has six components that are known as **method header**, as we have shown in the following figure.



**Method Signature:** Every method has a method signature. It is a part of the method declaration. It includes the **method name** and **parameter list**.

**Access Specifier:** Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides **four** types of access specifier:

- **Public:** The method is accessible by all classes when we use public specifier in our application.
- **Private:** When we use a private access specifier, the method is accessible only in the classes in which it is defined.
- **Protected:** When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.
- **Default:** When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.

**Return Type:** Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword.

**Method Name:** It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. Suppose, if we are creating a method for

subtraction of two numbers, the method name must be **subtraction()**. A method is invoked by its name.

**Parameter List:** It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.

**Method Body:** It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

### Instance Method

The method of the class is known as an instance method. It is a non-static method defined in the class. Before calling or invoking the instance method, it is necessary to create an object of its class. Let's see an example of an instance method.

#### InstanceMethodExample.java

```
public class InstanceMethodExample
{
    public static void main(String [] args)
    {
        //Creating an object of the class
        InstanceMethodExample obj = new InstanceMethodExample();
        //invoking instance method
        System.out.println("The sum is: "+obj.add(12, 13));
    }
    int s;
    //user-defined method because we have not used static keyword
    public int add(int a, int b)
    {
        s = a+b;
        //returning the sum
        return s;
    }
}
```



```
}
```

Output:

The sum is: 25

There are two types of instance method:

Accessor Method

Mutator Method

**Accessor Method:** The method(s) that reads the instance variable(s) is known as the accessor method. We can easily identify it because the method is prefixed with the word get. It is also known as getters. It returns the value of the private field. It is used to get the value of the private field.

Example

```
public int getId()  
{  
    return Id;  
}
```

**Mutator Method:** The method(s) read the instance variable(s) and also modify the values. We can easily identify it because the method is prefixed with the word set. It is also known as setters or modifiers. It does not return anything. It accepts a parameter of the same data type that depends on the field. It is used to set the value of the private field.

Example

```
public void setRoll(int roll)  
{
```

```
this.roll = roll;  
}
```

Example of accessor and mutator method

Student.java

```
public class Student  
{  
    private int roll;  
    private String name;  
    public int getRoll() //accessor method  
    {  
        return roll;  
    }  
    public void setRoll(int roll) //mutator method  
    {  
        this.roll = roll;  
    }  
    public String getName()  
    {  
        return name;  
    }  
    public void setName(String name)  
    {  
        this.name = name;  
    }  
    public void display()  
    {  
        System.out.println("Roll no.: "+roll);  
    }  
}
```

```
System.out.println("Student name: "+name);  
}  
}
```

## Factory Method Pattern

A Factory Pattern or Factory Method Pattern says that just **define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate**. In other words, subclasses are responsible to create the instance of the class.

The Factory Method Pattern is also known as **Virtual Constructor**.

### *Advantage of Factory Design Pattern*

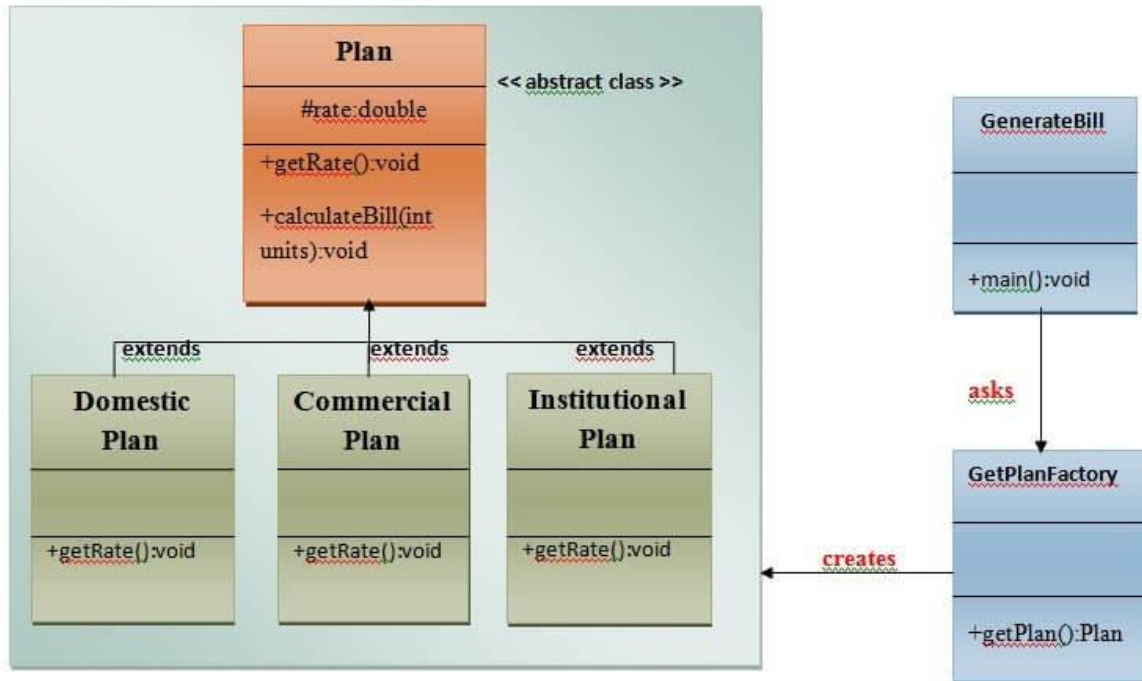
- Factory Method Pattern allows the sub-classes to choose the type of objects to create.
- It promotes the **loose-coupling** by eliminating the need to bind application-specific classes into the code. That means the code interacts solely with the resultant interface or abstract class, so that it will work with any classes that implement that interface or that extends that abstract class.

### *Usage of Factory Design Pattern*

- When a class doesn't know what sub-classes will be required to create
- When a class wants that its sub-classes specify the objects to be created.
- When the parent classes choose the creation of objects to its sub-classes.

### *UML for Factory Method Pattern*

- We are going to create a Plan abstract class and concrete classes that extends the Plan abstract class. A factory class GetPlanFactory is defined as a next step.
- GenerateBill class will use GetPlanFactory to get a Plan object. It will pass information (DOMESTICPLAN / COMMERCIALPLAN / INSTITUTIONALPLAN) to GetPalnFactory to get the type of object it needs.



## Calculate Electricity Bill : A Real World Example of Factory Method

### Step 1: Create a Plan abstract class.

1. import java.io.\*;
2. abstract class Plan{
3. protected double rate;
4. abstract void getRate();
- 5.
6. public void calculateBill(int units){
7. System.out.println(units\*rate);
8. }
9. }//end of Plan class.

### Step 2: Create the concrete classes that extends Plan abstract class.

1. class DomesticPlan extends Plan{
2. //@override
3. public void getRate(){
4. rate=3.50;
5. }
6. }//end of DomesticPlan class.
  
1. class CommercialPlan extends Plan{
2. //@override
3. public void getRate(){
4. rate=7.50;
5. }
6. }//end of CommercialPlan class.

```

1. class InstitutionalPlan extends Plan{
2.     //@override
3.     public void getRate(){
4.         rate=5.50;
5.     }
6. } //end of InstitutionalPlan class.

```

**Step 3:** Create a GetPlanFactory to generate object of concrete classes based on given information..

```

1. class GetPlanFactory{
2.
3.     //use getPlan method to get object of type Plan
4.     public Plan getPlan(String planType){
5.         if(planType == null){
6.             return null;
7.         }
8.         if(planType.equalsIgnoreCase("DOMESTICPLAN")) {
9.             return new DomesticPlan();
10.        }
11.        else if(planType.equalsIgnoreCase("COMMERCIALPLAN")){
12.            return new CommercialPlan();
13.        }
14.        else if(planType.equalsIgnoreCase("INSTITUTIONALPLAN")) {
15.            return new InstitutionalPlan();
16.        }
17.        return null;
18.    }
19. } //end of GetPlanFactory class.

```

**Step 4:** Generate Bill by using the GetPlanFactory to get the object of concrete classes by passing an information such as type of plan DOMESTICPLAN or COMMERCIALPLAN or INSTITUTIONALPLAN.

```

1. import java.io.*;
2. class GenerateBill{
3.     public static void main(String args[])throws IOException{
4.         GetPlanFactory planFactory = new GetPlanFactory();
5.
6.         System.out.print("Enter the name of plan for which the bill will be generated: ");
7.         BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
8.
9.         String planName=br.readLine();
10.        System.out.print("Enter the number of units for bill will be calculated: ");
11.        int units=Integer.parseInt(br.readLine());
12.
13.        Plan p = planFactory.getPlan(planName);
14.        //call getRate() method and calculateBill()method of DomesticPaln.
15.
16.        System.out.print("Bill amount for "+planName+" of "+units+" units is: ");

```

```
17.     p.getRate();
18.     p.calculateBill(units);
19.     }
20. }//end of GenerateBill class.
```

---

[download this Electricity bill Example](#)

---

### *Output*

```
Command Prompt
E:\All design patterns\Design patterns and their codes>javac GenerateBill.java

E:\All design patterns\Design patterns and their codes>java GenerateBill
Enter the name of plan for which the bill will be generated: commercialplan
Enter the number of units for bill will be calculated: 500
Bill amount for commercialplan of 500 units is: 3750.0

E:\All design patterns\Design patterns and their codes>java GenerateBill
Enter the name of plan for which the bill will be generated: domesticPLAN
Enter the number of units for bill will be calculated: 500
Bill amount for domesticPLAN of 500 units is: 1750.0

E:\All design patterns\Design patterns and their codes>java GenerateBill
Enter the name of plan for which the bill will be generated: insTITUTIONALPlan
Enter the number of units for bill will be calculated: 500
Bill amount for insTITUTIONALPlan of 500 units is: 2750.0

E:\All design patterns\Design patterns and their codes>
```

## UNIT-II

### **Introduction**

**J2EE** stands for Java 2 Platform, Enterprise Edition. J2EE is the standard platform for developing applications in the enterprise and is designed for enterprise applications that run on servers. J2EE provides **APIs** that let developers create workflows and make use of resources such as databases or web services. J2EE consists of a set of APIs. Developers can use these APIs to build applications for business computing.

### Benefits of J2EE

Below is the list of benefits that J2EE provides:

1. **Portability:** If you build a J2EE application on a specific platform, it runs the same way on any other J2EE-compliant platform. This makes it easy to move applications from one environment to another. For example, moving an application from one computer to another or relocating an application from a test server to a production server.
2. **Reusability:** The components in J2EE are reused, so the average size of an application is much smaller than it would be if you had to write equivalent functionality from scratch for each program. For example, one component lets you read objects from a database. You can use that object-reading feature in any J2EE application. Since this functionality is already written and tested, you don't have to write it yourself every time you need it.
3. **Security:** Java technology lets programmers handle sensitive data far more securely than they can in C/C++ programs.
4. **Scalability:** J2EE lets developers build applications that run well on both small, single-processor computers and large, multi-processor systems.
5. **Reliability:** Many of the services (such as transaction management and monitoring) that applications need to be reliable are built into J2EE.

Java SE	Java EE
Java SE provide basic functionalities such as defining types and objects.	Java EE facilitates development of large scale applications.
SE is a normal Java specification	EE is built upon Java SE. It provides functionalities like web applications, and Servlets.
It has features like class libraries, deployment environments, etc.	Java EE is a structured application with a separate client, business, and Enterprise layers.
It is mostly used to develop APIs for Desktop Applications like antivirus software, game, etc.	It is mainly used for developing web applications.
Suitable for beginning Java developers.	Suitable for experienced Java developers who build enterprise-wide applications.

It does not provide user authentication.

It provides user authentication.

### **When use Enterprise Java Bean?**

1. **Application needs Remote Access.** In other words, it is distributed.
2. **Application needs to be scalable.** EJB applications supports load balancing, clustering and fail-over.
3. **Application needs encapsulated business logic.** EJB application is separated from presentation and persistent layer.

### **Types of Enterprise Java Bean**

There are 3 types of enterprise bean in java.

#### ***Session Bean***

Session bean contains business logic that can be invoked by local, remote or webservice client.

#### ***Message Driven Bean***

Like Session Bean, it contains the business logic but it is invoked by passing message.

#### ***Entity Bean***

It encapsulates the state that can be persisted in the database. It is deprecated. Now, it is replaced with JPA (Java Persistent API).

### **Disadvantages of EJB**

1. Requires application server
2. Requires only java client. For other language client, you need to go for webservice.
3. Complex to understand and develop ejb applications.

### **Session Bean**

Session bean encapsulates business logic only, it can be invoked by local, remote and webservice client.

It can be used for calculations, database access etc.

The life cycle of session bean is maintained by the application server (EJB Container).

### **Types of Session Bean**

There are 3 types of session bean.



1) **Stateless Session Bean:** It doesn't maintain state of a client between multiple method calls.

2) **Stateful Session Bean:** It maintains state of a client across multiple requests.

3) **Singleton Session Bean:** One instance per application, it is shared between clients and supports concurrent access.

### **Stateless Session Bean**

**Stateless Session bean** is a business object that represents business logic only. It doesn't have state (data).

In other words, *conversational state* between multiple method calls is not maintained by the container in case of stateless session bean.

The stateless bean objects are pooled by the EJB container to service the request on demand.

It can be accessed by one client at a time. In case of concurrent access, EJB container routes each request to different instance.

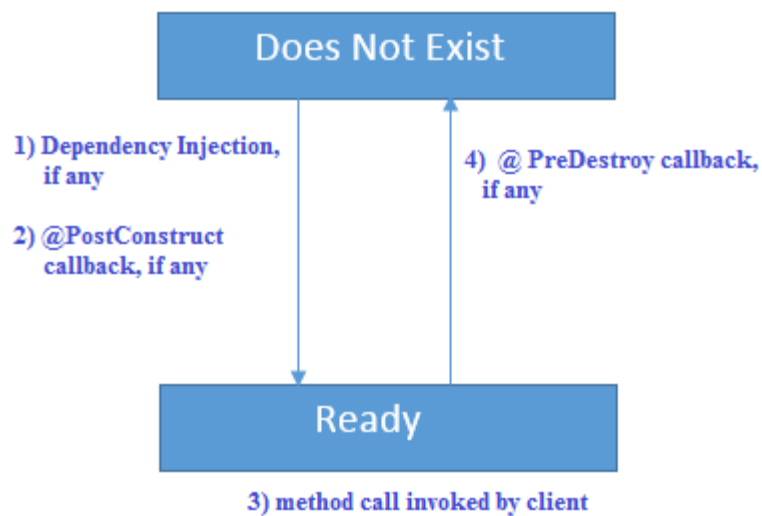
### **Annotations used in Stateless Session Bean**

There are 3 important annotations used in stateless session bean:

1. `@Stateless`
  2. `@PostConstruct`
  3. `@PreDestroy`
- 

### **Life cycle of Stateless Session Bean**

There is only two states of stateless session bean: does not exist and ready. It is explained by the figure given below.



javaTpoint.com

EJB Container creates and maintains a pool of session bean first. It injects the dependency if then calls the @PostConstruct method if any. Now actual business logic method is invoked by the client. Then, container calls @PreDestory method if any. Now bean is ready for garbage collection.

### Example of Stateless Session Bean

To develop stateless bean application, we are going to use **Eclipse IDE** and **glassfish 3** server.

To create EJB application, you need to create bean component and bean client.

#### 1) Create stateless bean component

To create the stateless bean component, you need to create a remote interface and a bean class.

File: AdderImplRemote.java

```
package com.javatpoint;
import javax.ejb.Remote;
```

```
@Remote
```

```
public interface AdderImplRemote {
    int add(int a,int b);
}
```

File: AdderImpl.java

```
package com.javatpoint;
import javax.ejb.Stateless;
```

```
@Stateless(mappedName="st1")
```

```
public class AdderImpl implements AdderImplRemote {
    public int add(int a,int b){
        return a+b;
    }
}
```

```
}  
}
```

## 2) Create stateless bean client

The stateless bean client may be local, remote or webservice client. Here, we are going to create remote client. It is console based application. Here, we are not using dependency injection. The dependency injection can be used with web based client only.

File: AdderImpl.java

```
package com.javatpoint;  
import javax.naming.Context;  
import javax.naming.InitialContext;  
  
public class Test {  
public static void main(String[] args) throws Exception {  
    Context context=new InitialContext();  
    AdderImplRemote remote=(AdderImplRemote)context.lookup("st1");  
    System.out.println(remote.add(32,32));  
}  
}
```

Output

Output: 64

## Stateful Session Bean

**Stateful Session bean** is a business object that represents business logic like stateless session bean. But, it maintains state (data).

In other words, *conversational state* between multiple method calls is maintained by the container in stateful session bean.

---

## Annotations used in Stateful Session Bean

There are 5 important annotations used in stateful session bean:

1. @Stateful
2. @PostConstruct
3. @PreDestroy
4. @PrePassivate
5. @PostActivate

## Example of Stateful Session Bean

To develop stateful session bean application, we are going to use **Eclipse IDE** and **glassfish 3** server.

As described in the previous example, you need to create bean component and bean client for creating session bean application.

### 1) Create stateful bean component

Let's create a remote interface and a bean class for developing stateful bean component.

File: *BankRemote.java*

1. **package** com.javatpoint;
2. **import** javax.ejb.Remote;
3. **@Remote**
4. **public interface** BankRemote {
5.     **boolean** withdraw(**int** amount);
6.     **void** deposit(**int** amount);
7.     **int** getBalance();
8. }

File: *Bank.java*

1. **package** com.javatpoint;
  2. **import** javax.ejb.Stateful;
  3. **@Stateful**(mappedName = "stateful123")
  4. **public class** Bank **implements** BankRemote {
  5.     **private int** amount=0;
  6.     **public boolean** withdraw(**int** amount){
  7.         **if**(amount<=**this**.amount){
  8.             **this**.amount-=amount;
  9.             **return true**;
  10.         }**else**{
  11.             **return false**;
  12.         }
  13.     }
  14.     **public void** deposit(**int** amount){
  15.         **this**.amount+=amount;
  16.     }
  17.     **public int** getBalance(){
  18.         **return** amount;
  19.     }
  20. }
-

## 2) Create stateful bean client

The stateful bean client may be local, remote or webservice client. Here, we are going to create web based client and not using dependency injection.

File: *index.jsp*

1. `<a href="OpenAccount">Open Account</a>`

File: *operation.jsp*

1. `<form action="operationprocess.jsp">`
2. Enter Amount:`<input type="text" name="amount"/><br>`
- 3.
4. Choose Operation:
5. Deposit`<input type="radio" name="operation" value="deposit"/>`
6. Withdraw`<input type="radio" name="operation" value="withdraw"/>`
7. Check balance`<input type="radio" name="operation" value="checkbalance"/>`
8. `<br>`
9. `<input type="submit" value="submit">`
10. `</form>`

File: *operationprocess.jsp*

1. `<%@ page import="com.javatpoint.*" %>`
2. `<%`
3. `BankRemote remote=(BankRemote)session.getAttribute("remote");`
4. `String operation=request.getParameter("operation");`
5. `String amount=request.getParameter("amount");`
- 6.
7. `if(operation!=null){`
- 8.
9. `if(operation.equals("deposit")){`
10. `remote.deposit(Integer.parseInt(amount));`
11. `out.print("Amount successfully deposited!");`
12. `}else`
13. `if(operation.equals("withdraw")){`
14. `boolean status=remote.withdraw(Integer.parseInt(amount));`
15. `if(status){`
16. `out.print("Amount successfully withdrawn!");`
17. `}else{`
18. `out.println("Enter less amount");`
19. `}`
20. `}else{`

```
21.         out.println("Current Amount: "+remote.getBalance());
22.     }
23. }
24. %>
25. <hr/>
26. <jsp:include page="operation.jsp"></jsp:include>
```

*File: OpenAccount.java*

```
1.  package com.javatpoint;
2.  import java.io.IOException;
3.  import javax.ejb.EJB;
4.  import javax.naming.InitialContext;
5.  import javax.servlet.ServletException;
6.  import javax.servlet.annotation.WebServlet;
7.  import javax.servlet.http.HttpServlet;
8.  import javax.servlet.http.HttpServletRequest;
9.  import javax.servlet.http.HttpServletResponse;
10. @WebServlet("/OpenAccount")
11. public class OpenAccount extends HttpServlet {
12.     //@EJB(mappedName="stateful123")
13.     //@BankRemote b;
14.     protected void doGet(HttpServletRequest request, HttpServletResponse response)
15.         throws ServletException, IOException {
16.     try{
17.         InitialContext context=new InitialContext();
18.         BankRemote b=(BankRemote)context.lookup("stateful123");
19.
20.         request.getSession().setAttribute("remote",b);
21.         request.getRequestDispatcher("/operation.jsp").forward(request, response);
22.
23.     }catch(Exception e){System.out.println(e);}
24. }
25. }
```

<https://examples.javacodegeeks.com/java-development/enterprise-java/ejb3/ejb-tutorial-beginners-example/>

## 1. Introduction

The Enterprise Java Beans (EJB) is a specification for deployable server-side components in Java. It is an agreement between components and application servers that enable any component to run in any application server. EJB components (called enterprise beans) are deployable, and can be imported and loaded into an application server, which hosts those components to develop secured, robust and scalable distributed applications.

To run EJB application, you need an application server (EJB Container) such as Jboss, Glassfish, Weblogic, Websphere etc. It performs:

1. Life Cycle Management
2. Security
3. Transaction Management
4. Load Balancing
5. Persistence Mechanism
6. Exception Handling
7. Object Pooling

EJB application is deployed on the server, so it is called server side component also. We'll be discussing EJB 3.0 in this tutorial.

## 2. Types of Enterprise Java Beans

EJB defines three different kinds of enterprise beans.

1. **Session beans:** Session bean contains business logic that can be invoked by local, remote or webservice client. The bean class typically contains business-process related logic, such as logic to compute prices, transfer funds between bank accounts, or perform order entry. Session bean are of 3 types:
  - **Stateless Session Bean:** A stateless session bean doesn't maintain state of a client between multiple method calls. When a client invokes the methods of a stateless bean, the bean's instance variables may contain a state specific to that client but only for the duration of the invocation. When the method is finished, the client-specific state should not be retained.
  - **Stateful Session Bean:** A stateful session bean maintain state of a client across multiple requests. In a stateful session bean, the instance variables represent the state of a unique client/bean session. This state is often called the conversational state as the client interacts with its bean,.
  - **Singleton Session Bean:** A singleton session bean is instantiated once per application and exists for the lifecycle of the application. Singleton session beans are designed for circumstances in which a single enterprise bean instance is shared across and concurrently accessed by clients.

2. **Entity beans:** Entity beans encapsulates the state that can be persisted in the database. User data can be saved to database via entity beans and later on can be retrieved from the database in the entity bean. The bean class contains data-related logic such as logic to reduce balance of bank account or modify customer details.
3. **Message-driven beans:** Message-driven beans are similar to session beans in their actions. It contains the business logic but it is invoked by passing message. The difference is that you can call message driven beans only by sending messages to those beans. These message-driven beans might call other enterprise beans as well.

#### Tip

You may skip project creation and jump directly to the [beginning of the example](#) below.

### 3. Create EJB Module

In example below, We'll create a ejb module project named BasicOperationsEJBModule using NetBeans.

Open NetBeans IDE, choose File > New Project.

In the New Project wizard, expand the **Java EE category** and select **EJB Module** as shown in the figure below. Then click Next.

Create EJB Module



You have to specify the **Project Name** and the **Project Location** in the appropriate text fields and then click Next.

Specify EJB Module Name

In the next window, add the Server and select the Java EE version and click Finish.

Select Server and finish

#### **4. Create a new Application Class Project**

In this section you will create an application class library project for EJB remote interface and entity class.

Open NetBeans IDE, choose File > New Project.

In the New Project wizard, expand the **Java category** and select **Java Class Library** as shown in the figure below. Then click Next.

Specify Client Project Name

You have to specify the **Project Name** and the **Project Location** in the appropriate text fields and then click Next.

Specifyn Client Name

ADVERTISEMENT



## 5. Create Session Bean

The EJB 3.1 specification introduces `@Stateless` annotation that enables you to easily create stateless session beans. A stateless session bean as per its name does not have any associated client state, but it may preserve its instance state. EJB Container normally creates a pool of few stateless bean's objects and use these objects to process client's request.

To create the stateless session bean, perform the following steps.

- Right-click the EJB module and choose New > Other to open the New File wizard.
- Select Session Bean in the Enterprise JavaBeans category. Click Next.

### Specify EJB Name

- Type **OperationsSessionBean** for the EJB Name.
- Type **com.javacodegeeks.example.ejb** for the Package name.
- Select Stateless.
- Click Finish.

### 5.1 Adding a Business Method

In this exercise you will create a simple business method in the session bean that returns a string.

1. Right-click in the editor of **OperationsSessionBean** and choose Insert Code

- Select Insert Code
2. Select Add Business Method.

Add Business Method

3. Type **add** in Method Name and float as return type and x, y as parameter names

Enter business method details

4. Similarly, create **subtract** method as shown in below figure



- Add business method
5. Create **multiply** business method

- Add business method
6. Create **divide** business method

Add business method

*OperationsSessionBean.java*

```
01 package com.javacodegeeks.example.ejb;
02
03 import javax.ejb.Stateless;
04
05 /**
06  *
07  * @author RadhaKrishna
08  */
09 @Stateless
10 public class OperationsSessionBean implements OperationsSessionBeanRemote {
11
12     // Add business logic below. (Right-click in editor and choose
13     // "Insert Code > Add Business Method")
14
15     @Override
16     public float add(float x, float y) {
17         return x + y;
18     }
19
20     @Override
21     public float subtract(float x, float y) {
22         return x - y;
```

```
23 }
24
25 @Override
26 public float multiply(float x, float y) {
27     return x * y;
28 }
29
30 @Override
31 public float divide(float x, float y) {
32     return x / y;
33 }
34}
```

## 6. Deploy the EJB Module

You can now build and deploy the EJB module. Right-click the `BasicOperationsEJBModule` module and choose Deploy. When you click Deploy, the IDE builds the ejb module and deploys the JAR archive to the server.

In the Services window, if you expand the Applications node of GlassFish Server you can see that `BasicOperationsEJBModule` was deployed.

## 7. Create a new Web Module to test EJB

Choose File > New Project.

In the New Project wizard, expand the **Java Web category** and select **Web Application** as shown in the figure below. Then click Next.

### Create Web Application

You have to specify the **Project Name** and the **Project Location** in the appropriate text fields and then click Next.

Specify Project Name

In the next window, add the J2EE Server and select the J2EE version and click Finish.

Select Server

## 8. Create JSP Files to test EJB

In this exercise you will create a JSP to test user operations and obtain result.

- Right-click the Web module and choose File > New File wizard.
- In the New File wizard, expand the **Web category** and select **JSP** as shown in the figure below.
- Then click Next.

Specify JSP Name

form.jsp

```
01<html>
02 <head>
03   <title>Calculator</title>
04 </head>
05
06 <body bgcolor="lightgreen">
07   <h1>Basic Operations</h1>
08   <hr>
09
10   <form action="Result.jsp" method="POST">
11     <p>Enter first value:
12       <input type="text" name="num1" size="25"></p>
13     <br>
14     <p>Enter second value:
15       <input type="text" name="num2" size="25"></p>
16     <br>
17
18     <b>Select your choice:</b><br>
19     <input type="radio" name="group1" value ="add">Addition<br>
20     <input type="radio" name="group1" value ="sub">Subtraction<br>
21     <input type="radio" name="group1" value ="multi">Multiplication<br>
22     <input type="radio" name="group1" value ="div">Division<br>
23     <p>
```



```
24     <input type="submit" value="Submit">
25     <input type="reset" value="Reset">
26     </p>
27 </form>
28 </body>
29</html>
30</form>
```

The result will be displayed in `Result.jsp`. Create jsp as per below.

- Right-click the Web module and choose File > New File wizard.
- In the New File wizard, expand the **Web category** and select **JSP** as shown in the figure below.
- Then click Next.

Specify JSP Name

*Result.jsp*

```
01<%@ page contentType="text/html; charset=UTF-8" %>
02<%@ page import="com.javacodegeeks.example.ejb.*, javax.naming.*"%>
03
04<%!
05 private OperationsSessionBeanRemote ops = null;
06 float result = 0;
07
08 public void jspInit() {
09     try {
10
11         InitialContext ic = new InitialContext();
```

```

12     ops = (OperationsSessionBeanRemote)ic.lookup(OperationsSessionBeanRemote.class.getName());
13
14
15     System.out.println("Loaded Calculator Bean");
16
17
18     } catch (Exception ex) {
19         System.out.println("Error:"
20             + ex.getMessage());
21     }
22 }
23
24 public void jspDestroy() {
25     ops = null;
26 }
27%>
28
29
30<%
31
32 try {
33     String s1 = request.getParameter("num1");
34     String s2 = request.getParameter("num2");
35     String s3 = request.getParameter("group1");
36
37     System.out.println(s3);
38
39     if (s1 != null && s2 != null) {
40         Float num1 = new Float(s1);
41         Float num2 = new Float(s2);
42
43         if (s3.equals("add")) {
44             result = ops.add(num1.floatValue(), num2.floatValue());
45         } else if (s3.equals("sub")) {
46             result = ops.subtract(num1.floatValue(), num2.floatValue());
47         } else if (s3.equals("multi")) {
48             result = ops.mutliply(num1.floatValue(), num2.floatValue());
49         } else {
50             result = ops.divide(num1.floatValue(), num2.floatValue());
51         }
52
53%>
54<p>
55 <b>The result is:</b> <%= result%>
56<p>
57
58 <%
59     }
60 } // end of try
61 catch (Exception e) {

```

```
62     e.printStackTrace();
63     //result = "Not valid";
64 }
65 %>
```

- Right click **BasicOperationsWebClient** project and select Properties
- In the menu select **Libraries** and click **Add Project** and add **BasicOperationsEJBModule** and **BasicOperationsEJBClient** projects
- Click OK.

Add Projects

## 9. Run the Project

You can now run the project. When you run the project, you want the browser to open to the page with the **form.jsp**. You do this by specifying the URL in the Properties dialog box for the web application. The URL is relative to the context path for the application. After you

enter the relative URL, you can build, deploy and run the application from the Projects window.

To set the **relative URL** and run the application, do the following:

- In the Projects window, right-click the BasicOperationsWebClient application node
- Select Properties in the pop-up menu.
- Select Run in the Categories pane.
- In the Relative URL textfield, type /form.jsp.
- Click OK.

Specify Relative URL

form.jsp

When you submit the request Result.jsp is called to display the result

Result.jsp

## **10. Download the NetBeans Project**

This was an example of Session bean in EJB.

### **Message Driven Bean**

A Messaging system allows and promotes the loose coupling of components

- Allows components to post messages for other components.
- Asynchronous rather than synchronous
- Also known as MOM (Message-Oriented Middleware).

Two basic models

- Point-to-point
- Publish/subscribe

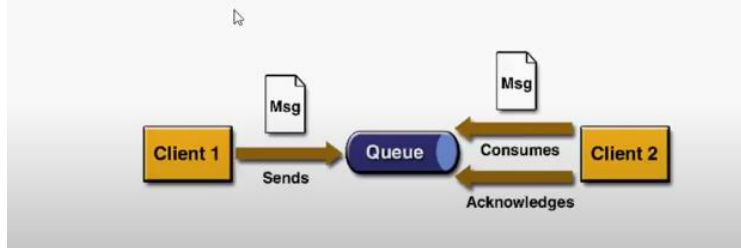
A java API that allows applications to:

- Create

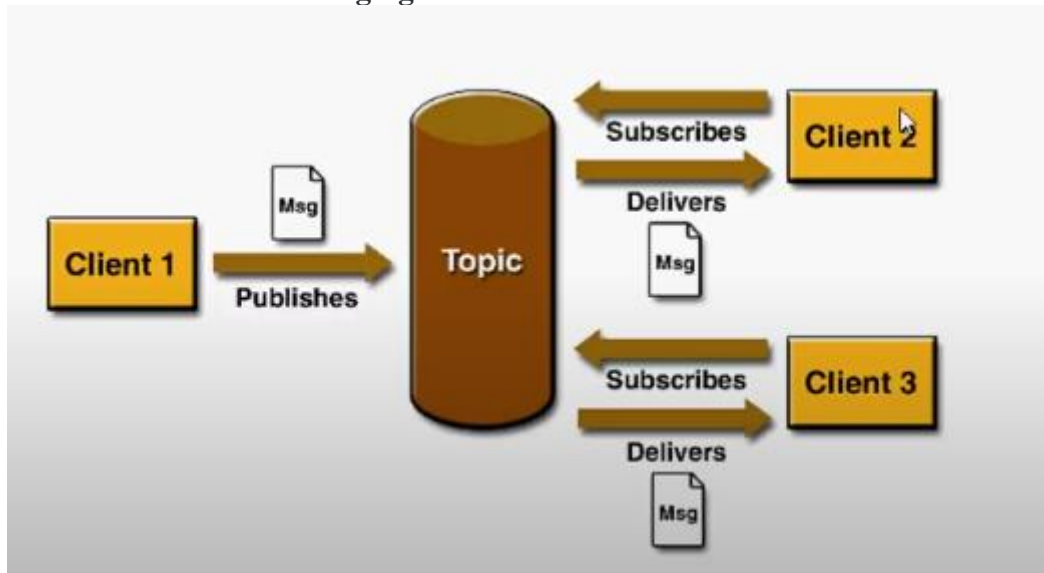
- Send
- Receive
- Read messages

In principle kind of like a news system, but doesn't involve e-mail.

## Point-to-Point Messaging Domain



## Publish /Subscribe Messaging Domain



## Message Consumption

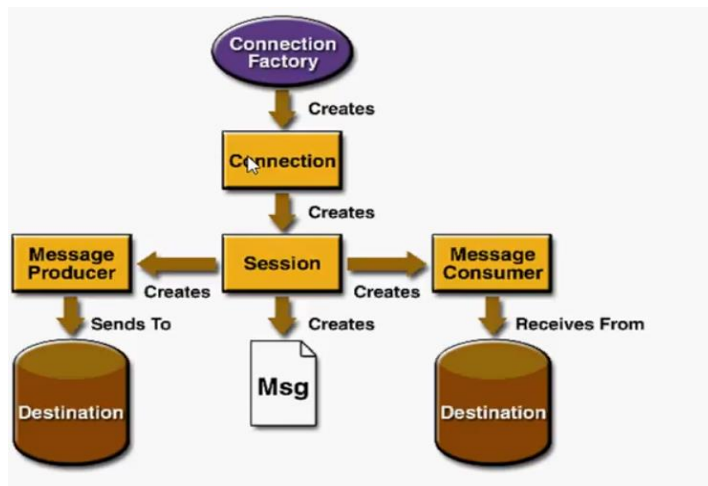
In JMS messages can be consumed in two ways:

### Synchronously

A subscriber or receiver explicitly fetches a messages from the destination using the “receive” methods

### Asynchronously

A client can register a message listener (like an event listener) with a consumer.



## RMI (Remote Method Invocation)

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

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

### Understanding stub and skeleton

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

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

#### stub

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

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

#### skeleton

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

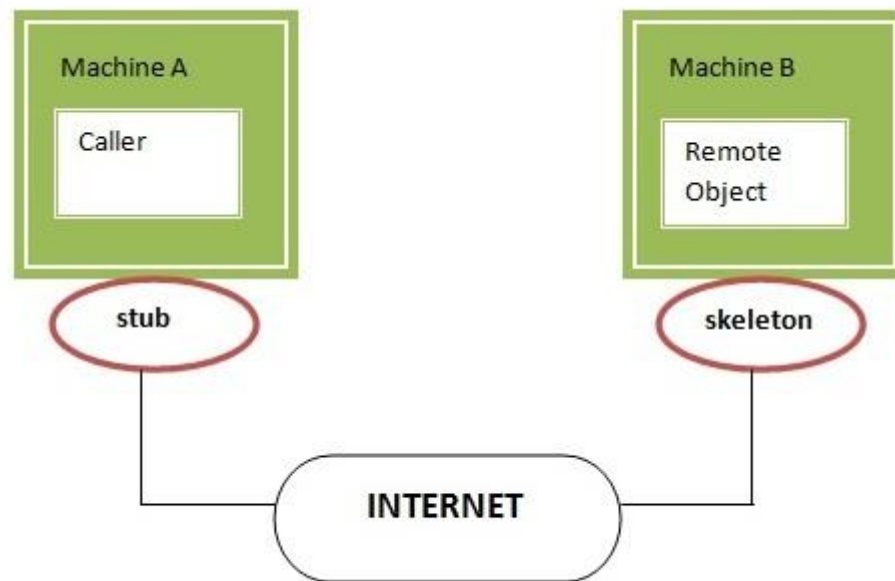
1. It reads the parameter for the remote method



2. It invokes the method on the actual remote object, and

3. It writes and transmits (marshals) the result to the caller.

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



skeletons.

---

### **Understanding requirements for the distributed applications**

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

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

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

---

### **Java RMI Example**

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

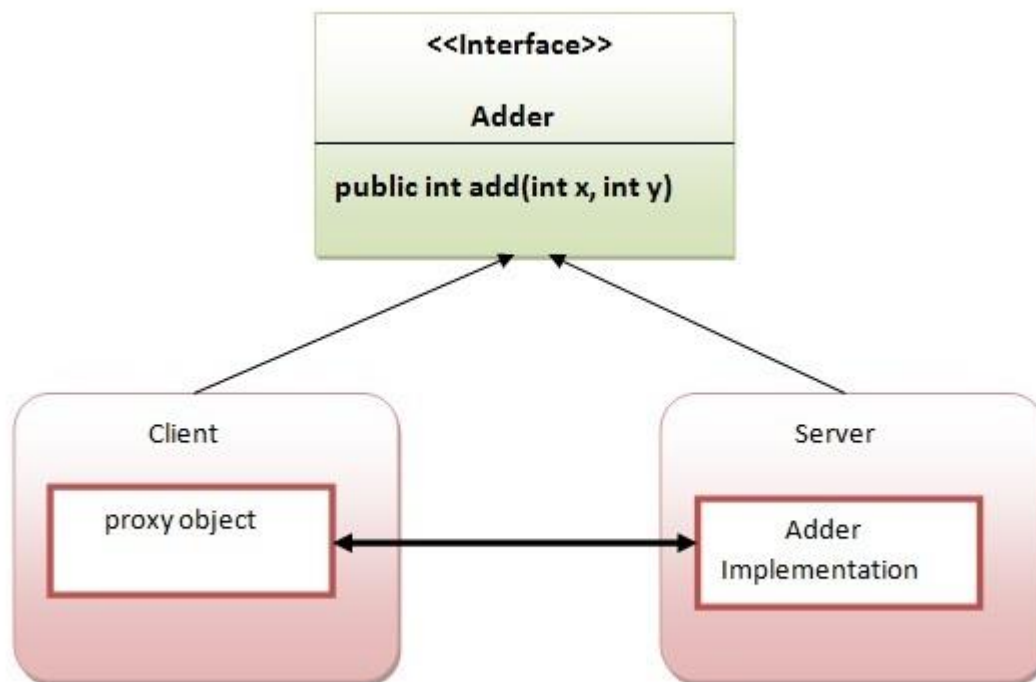
1. Create the remote interface
2. Provide the implementation of the remote interface
3. Compile the implementation class and create the stub and skeleton objects using the rmic tool
4. Start the registry service by rmiregistry tool
5. Create and start the remote application

## 6. Create and start the client application

---

### RMI Example

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



#### 1) create the remote interface

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

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

---

#### 2) Provide the implementation of the remote interface

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

ADVERTISEMENT

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

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

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

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

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

1. rmic AdderRemote

### 4) Start the registry service by the rmiregistry tool

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

1. rmiregistry 5000

### 5) Create and run the server application

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

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

```
public static java.lang.String[] list(java.lang.String) throws java.rmi.RemoteException,  
java.net.MalformedURLException;
```

It returns an array of remote objects

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

1. **import** java.rmi.\*;
2. **import** java.rmi.registry.\*;
3. **public class** MyServer{
4. **public static void** main(String args[]){
5. **try**{
6. Adder stub=**new** AdderRemote();
7. Naming.rebind("rmi://localhost:5000/sonoo",stub);
8. }**catch**(Exception e){System.out.println(e);}
9. }
10. }

---

#### 6) Create and run the client application

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

#### ADVERTISEMENT

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

---

[download this example of rmi](#)

---

1. For running **this** rmi example,
- 2.
- 1) compile all the java files

```
javac *.java
```

2)create stub and skeleton object by rmic tool

```
rmic AdderRemote
```

3)start rmi registry in one command prompt

```
rmiregistry 5000
```

4)start the server in another command prompt

```
java MyServer
```

5)start the client application in another command prompt

```
java MyClient
```

### Output of this RMI example

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

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

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

C:\Users\Sonoo>d:
D:\>cd "Sonoo Programs\core\rmi\1"
D:\Sonoo Programs\core\rmi\1>java MyServer
```

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

C:\Users\Sonoo>d:
D:\>cd "Sonoo Programs\core\rmi\1"
D:\Sonoo Programs\core\rmi\1>java MyClient
38
D:\Sonoo Programs\core\rmi\1>
```




## Meaningful example of RMI application with database

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

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

### 1) Create the table

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

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

### 2) Create Customer class and Remote interface

File: Customer.java

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

**Note: Customer class must be Serializable.**

File: Bank.java

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

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

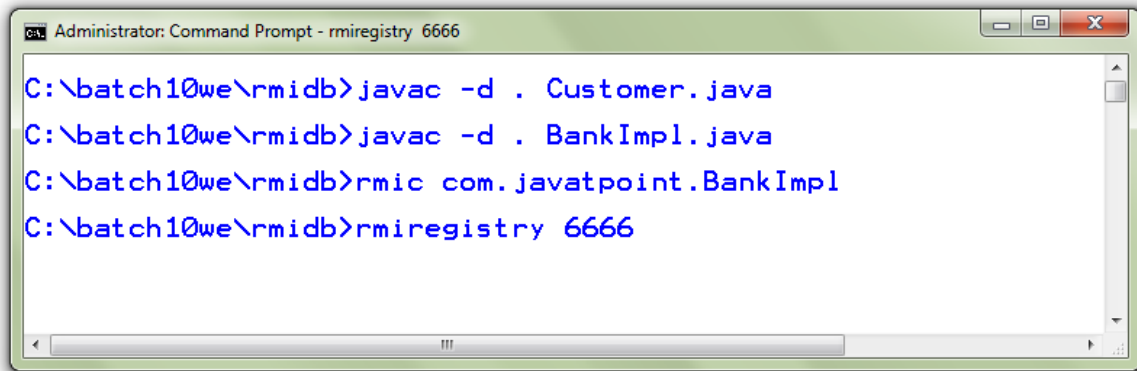
File: BankImpl.java

1. **package** com.javatpoint;
2. **import** java.rmi.\*;
3. **import** java.rmi.server.\*;
4. **import** java.sql.\*;
5. **import** java.util.\*;
6. **class** BankImpl **extends** UnicastRemoteObject **implements** Bank {
7. BankImpl()**throws** RemoteException{ }
- 8.
9. **public** List<Customer> getCustomers(){
10. List<Customer> list=**new** ArrayList<Customer>();
11. **try**{
12. Class.forName("oracle.jdbc.driver.OracleDriver");
13. Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe", "system", "oracle");
14. PreparedStatement ps=con.prepareStatement("select \* from customer400");
15. ResultSet rs=ps.executeQuery();
- 16.
17. **while**(rs.next()){
18. Customer c=**new** Customer();
19. c.setAcc\_no(rs.getInt(1));
20. c.setFirstname(rs.getString(2));
21. c.setLastname(rs.getString(3));
22. c.setEmail(rs.getString(4));
23. c.setAmount(rs.getFloat(5));
24. list.add(c);
25. }
- 26.
27. con.close();
28. }**catch**(Exception e){System.out.println(e);}
29. **return** list;
30. }**//end of getCustomers()**
31. }

---



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

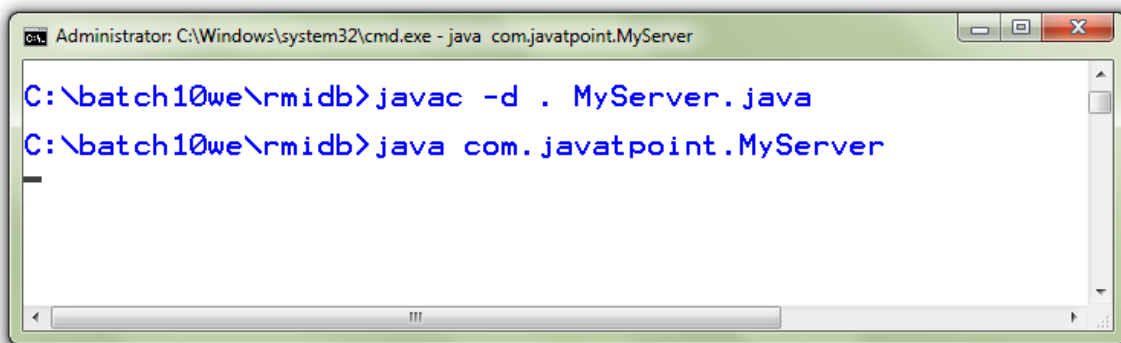


```
Administrator: Command Prompt - rmiregistry 6666
C:\batch10we\rmidb>javac -d . Customer.java
C:\batch10we\rmidb>javac -d . BankImpl.java
C:\batch10we\rmidb>rmic com.javatpoint.BankImpl
C:\batch10we\rmidb>rmiregistry 6666
```

#### 5) Create and run the Server

File: MyServer.java

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



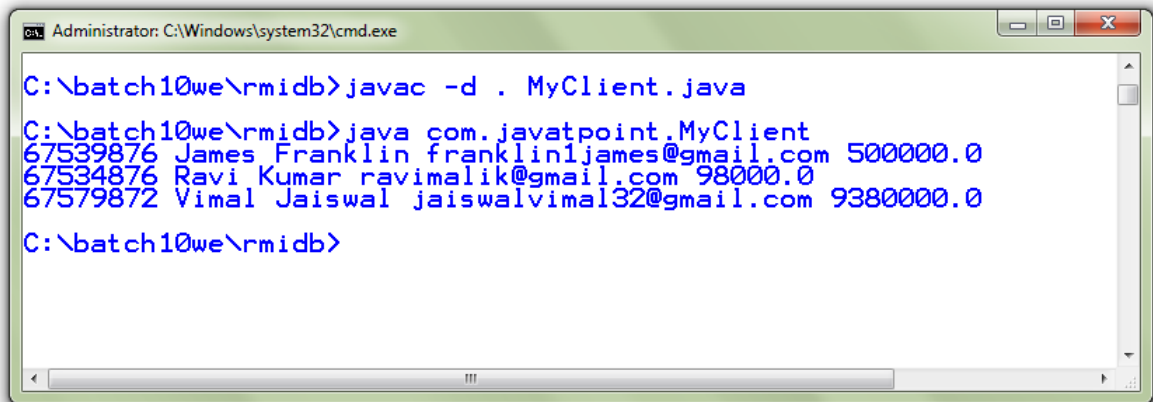
```
Administrator: C:\Windows\system32\cmd.exe - java com.javatpoint.MyServer
C:\batch10we\rmidb>javac -d . MyServer.java
C:\batch10we\rmidb>java com.javatpoint.MyServer
```

#### 6) Create and run the Client

File: MyClient.java

1. **package** com.javatpoint;
2. **import** java.util.\*;
3. **import** java.rmi.\*;
4. **public class** MyClient{
5. **public static void** main(String args[])**throws** Exception{
6. Bank b=(Bank)Naming.lookup("rmi://localhost:6666/javatpoint");

- 7.
8. List<Customer> list=b.getCustomers();
9. **for**(Customer c:list){
10. System.out.println(c.getAcc\_no()+" "+c.getFirstname()+" "+c.getLastname()
11. +" "+c.getEmail()+" "+c.getAmount());
12. }
- 13.
14. }}



```
Administrator: C:\Windows\system32\cmd.exe
C:\batch10we\rmidb>javac -d . MyClient.java
C:\batch10we\rmidb>java com.javatpoint.MyClient
67539876 James Franklin franklinjames@gmail.com 500000.0
67534876 Ravi Kumar ravimalik@gmail.com 98000.0
67579872 Vimal Jaiswal jaiswalvimal32@gmail.com 9380000.0
C:\batch10we\rmidb>
```

---

## RMI-IIOP

RMI-IIOP is another development framework that is used for developing applications based on the [Distributed Object Model](#). This framework enhances the standard RMI to work with the Internet Inter-ORB Protocol (IIOP). Since IIOP is the communication protocol of CORBA, you can use the RMI-IIOP to connect your Java remote objects to CORBA clients. It also works well for Java-to-Java objects communication over IIOP.

CORBA technology provides a language-independent approach to the communication of distributed objects. It uses entities, called Object Request Brokers (ORB), to transmit requests from the client to the server, and return the results from the request back to the client. These ORBs must support the Object by Value and Java to IDL mapping CORBA standards. By imposing those standards and introducing Interface Definition Language (IDL), CORBA provides interoperability between ORB implementations from different vendors.

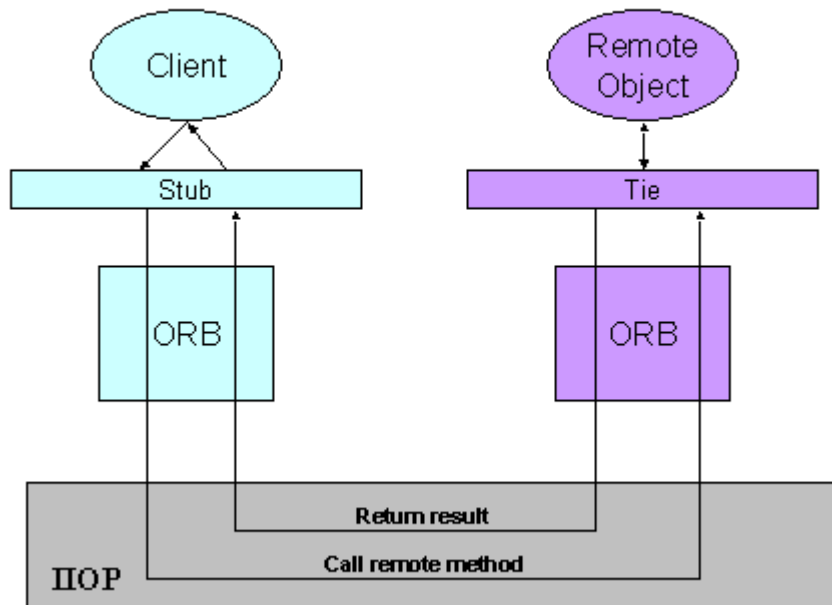
The RMI-IIOP framework actually brings together those interoperability features of CORBA and the ease-of-use features of RMI.

The RMI-IIOP functions are logically concentrated into the J2EE Engine's IIOP Provider Service.

### *RMI-IIOP Objects Communication*

The process of communication between the client and the server parts of an RMI-IIOP application is similar to that of RMI-P4 objects. The client obtains a reference to the server-side object and calls remote methods on it. The call is transmitted by the IIOP protocol. RMI-IIOP uses stubs and ties to facilitate remote communication between the remote objects.

#### **Remote Objects Communication in RMI-IIOP**



#### **Definition of RMI-IIOP Stubs and Ties**

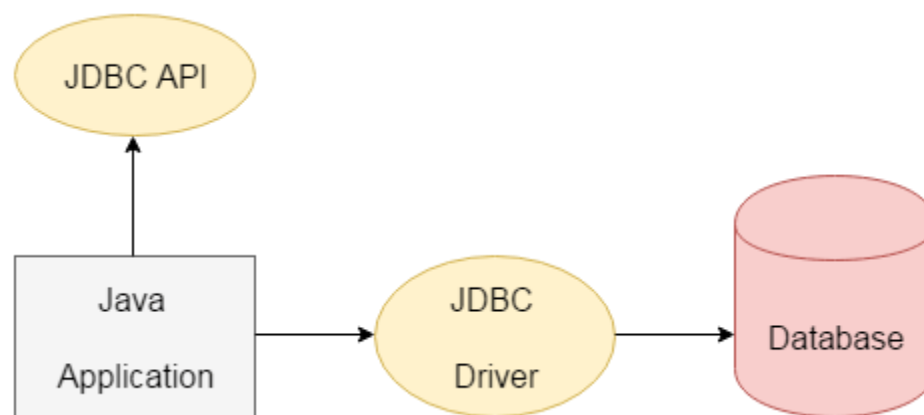
The stub is a class that extends `javax.rmi.CORBA.Stub` and implements the remote interface. A tie is a class that extends `org.omg.CORBA_2_3.portableObjectImpl` and implements `javax.rmi.CORBA.Tie`. The stub is the implementation of the interface that is passed to the client, and the tie is the server-side class. Both classes facilitate communication between the client and the server-side object.

JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database. There are four types of JDBC drivers:

- JDBC-ODBC Bridge Driver,
- Native Driver,
- Network Protocol Driver, and
- Thin Driver

We have discussed the above four drivers in the next chapter.

We can use JDBC API to access tabular data stored in any relational database. By the help of JDBC API, we can save, update, delete and fetch data from the database. It is like Open Database Connectivity (ODBC) provided by Microsoft.



The current version of JDBC is 4.3. It is the stable release since 21st September, 2017. It is based on the X/Open SQL Call Level Interface. The **java.sql** package contains classes and interfaces for JDBC API. A list of popular *interfaces* of JDBC API are given below:

- Driver interface
- Connection interface
- Statement interface
- PreparedStatement interface
- CallableStatement interface
- ResultSet interface
- ResultSetMetaData interface
- DatabaseMetaData interface
- RowSet interface

A list of popular *classes* of JDBC API are given below:

- DriverManager class
- Blob class
- Clob class
- Types class

### Why Should We Use JDBC

Before JDBC, ODBC API was the database API to connect and execute the 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).

We can use JDBC API to handle database using Java program and can perform the following activities:

1. Connect to the database
2. Execute queries and update statements to the database
3. Retrieve the result received from the database.

### ADVERTISEMENT

#### Do You Know

- How to connect Java application with Oracle and Mysql database using JDBC?
- What is the difference between Statement and PreparedStatement interface?
- How to print total numbers of tables and views of a database using JDBC?
- How to store and retrieve images from Oracle database using JDBC?
- How to store and retrieve files from Oracle database using JDBC?

#### What is API

API (Application programming interface) is a document that contains a 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.

#### JDBC Driver

1. JDBC Drivers
  1. JDBC-ODBC bridge driver
  2. Native-API driver
  3. Network Protocol driver
  4. Thin driver

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

1. JDBC-ODBC bridge driver
2. Native-API driver (partially java driver)
3. Network Protocol driver (fully java driver)
4. Thin driver (fully java driver)

### 1) JDBC-ODBC bridge driver

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

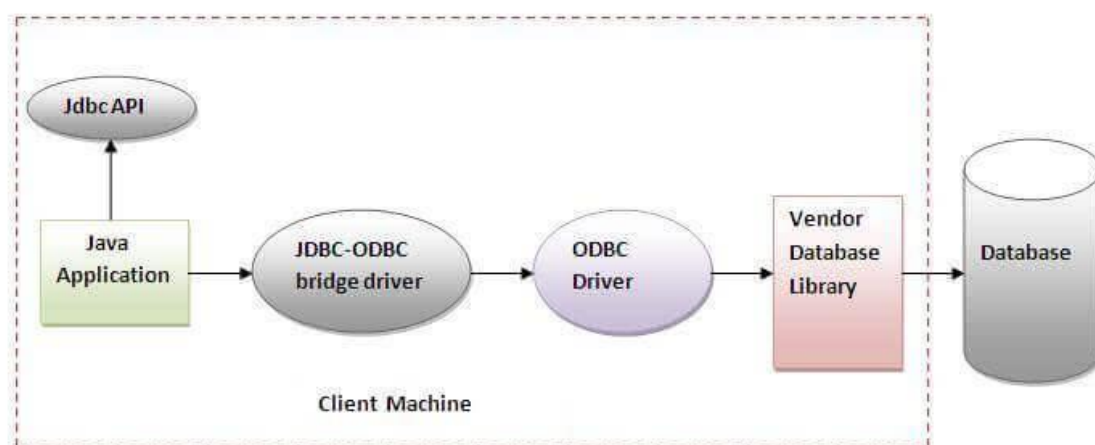


Figure- JDBC-ODBC Bridge Driver

***In Java 8, the JDBC-ODBC Bridge has been removed.***

Oracle does not support the JDBC-ODBC Bridge from Java 8. Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of the JDBC-ODBC Bridge.

**Advantages:**

ADVERTISEMENT

- 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.
-

## 2) Native-API driver

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

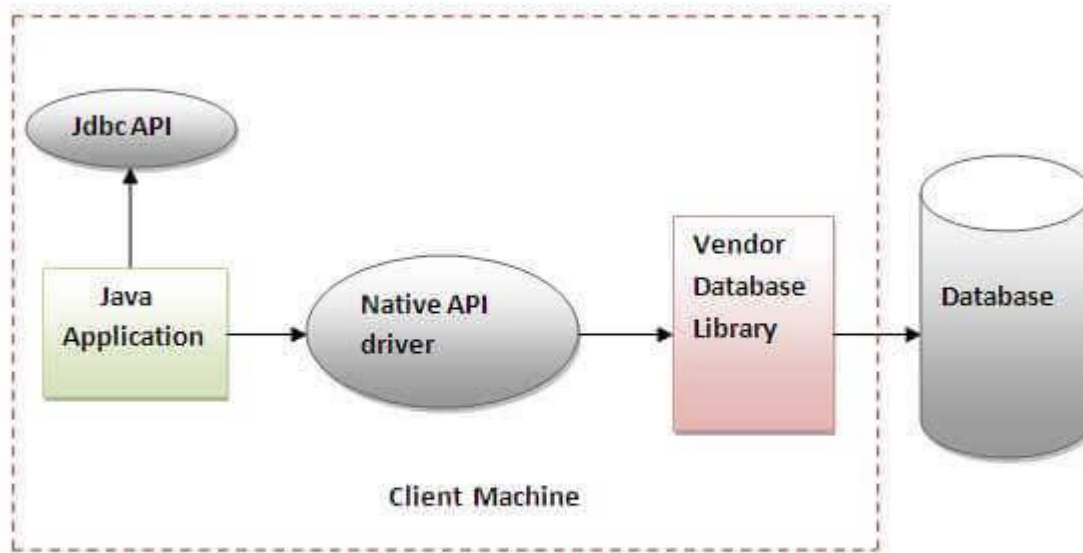


Figure- Native API Driver

### 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.

---

## 3) 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.

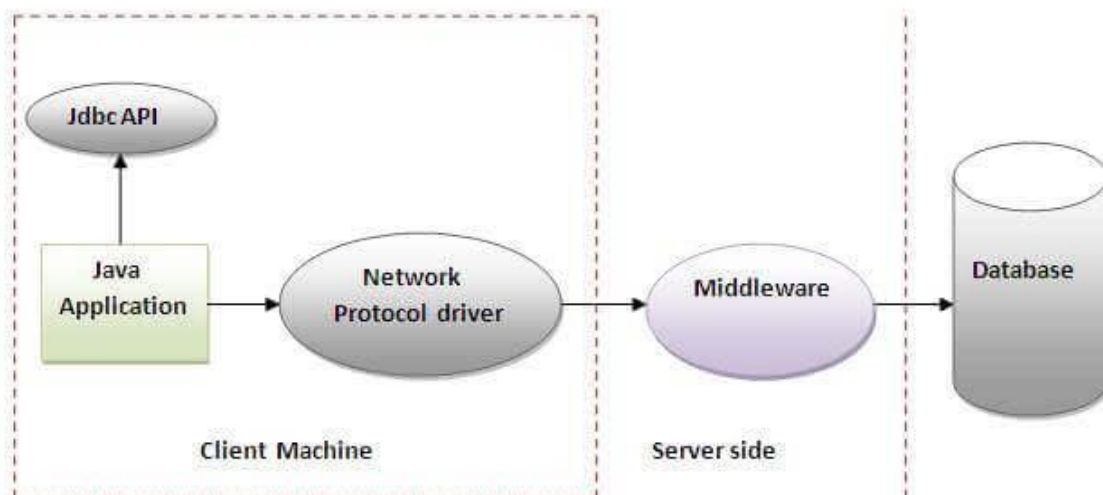


Figure- Network Protocol Driver

**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.

**4) 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.



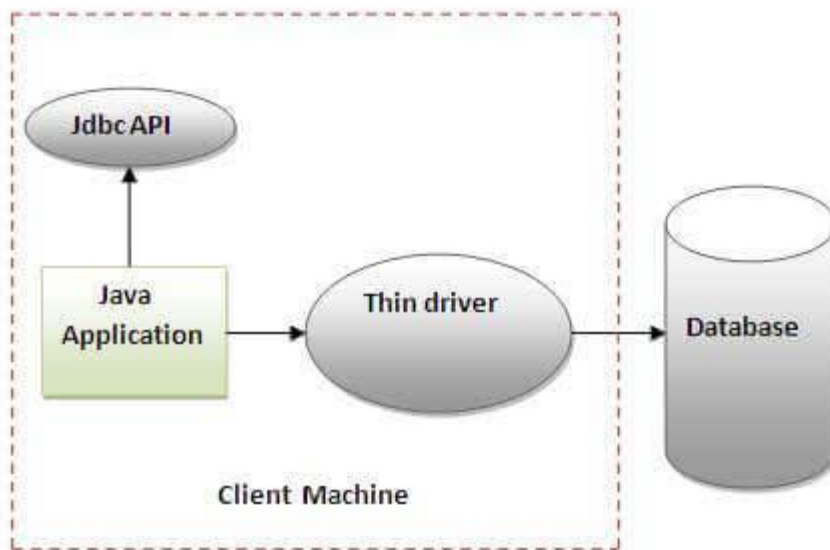


Figure- Thin Driver

#### Advantage:

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

#### Disadvantage:

- Drivers depend on the Database.

### Java Database Connectivity with MySQL

To connect Java application with the MySQL database, we need to follow 5 following steps.

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

1. **Driver class:** The driver class for the mysql database is **com.mysql.jdbc.Driver**.
2. **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, we need to replace the sonoo with our database name.
3. **Username:** The default username for the mysql database is **root**.

4. **Password:** It is the password 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.

1. create database sonoo;
  2. use sonoo;
  3. create table emp(id **int**(10),name varchar(40),age **int**(3));
- 

### Example to Connect Java Application with mysql database

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

1. **import** java.sql.\*;
2. **class** MysqlCon{
3. **public static void** main(String args[]){
4. **try**{
5. Class.forName("com.mysql.jdbc.Driver");
6. Connection con=DriverManager.getConnection(
7. "jdbc:mysql://localhost:3306/sonoo","root","root");
8. //here sonoo is database name, root is username and password
9. Statement stmt=con.createStatement();
10. ResultSet rs=stmt.executeQuery("select \* from emp");
11. **while**(rs.next())
12. System.out.println(rs.getInt(1)+" "+rs.getString(2)+" "+rs.getString(3));
13. con.close();
14. }**catch**(Exception e){ System.out.println(e);}
15. }
16. }

[download this example](#)

The above example will fetch all the records of emp table.

---

To connect java application with the mysql database, **mysqlconnector.jar** file is required to be loaded.

[download the jar file mysql-connector.jar](#)

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:

- o temporary
- o permanent

How to set the temporary classpath

open command prompt and write:

1. 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,;

Solution

Following example uses inner join sql command to combine data from two tables. To display the contents of the table getString() method of resultset is used.

```
import java.sql.*;

public class jdbcConn {
    public static void main(String[] args) throws Exception {
        Class.forName("org.apache.derby.jdbc.ClientDriver");
        Connection con = DriverManager.getConnection (
            "jdbc:derby://localhost:1527/testDb","username", "password");

        Statement stmt = con.createStatement();
        String query ="SELECT fname,lname,isbn from author inner join books on
author.AUTHORID = books.AUTHORID";
        ResultSet rs = stmt.executeQuery(query);
        System.out.println("Fname Lname ISBN");
    }
}
```

```
while (rs.next()) {
    String fname = rs.getString("fname");
    String lname = rs.getString("lname");
    int isbn = rs.getInt("isbn");
    System.out.println(fname + " " + lname+" " +isbn);
}
System.out.println();
System.out.println();
}
}
```

### Result

The above code sample will produce the following result. The result may vary.

```
Fname Lname ISBN
john grisham 123
jeffry archer 113
jeffry archer 112
jeffry archer 122
```

### JDBC Transaction Management

The sequence of actions (**SQL** statements) is treated as a single unit that is known as a transaction. Transaction Management is important for RDBMS-oriented applications to maintain data integrity and consistency.

While performing the transaction, we will use `getXXX` and `setXXX` methods to retrieve and set the data in the `ResultSet` object. `XXX` represents the data types of the columns. We will discuss the transaction and data types of JDBC in this tutorial.

#### Transaction Types

In JDBC every **SQL query** will be considered as a transaction. When we create a **Database** connection in JDBC, it will run in auto-commit mode (auto-commit value is `TRUE`). After the execution of the **SQL** statement, it will be committed automatically. Sometimes, we may want to commit the transaction after the execution of some more **SQL** statements. At that time, we need to set the auto-commit value to `False`. So that data won't be committed before executing all the queries. If we get an exception in the transaction, we can `rollback()` changes and make it like before. Transaction Management can be explained well – using ACID properties.

#### ACID means

- **A–Atomicity** -> If all queries are executed successfully, data will be committed, else won't.
- **C–Consistency** -> **DB** must be in a consistent state after any transaction.
- **I– Isolation** -> Transaction is isolated from other transactions.
- **D–Durability** -> If the transaction is committed once, it will remain always committed.

**There are three most important functions in Transaction Management. They are:**

- **Commit:** After the execution of the SQL statements, we want to make the changes permanent in the Database. We should call the commit() method. Normally, what is commit means it will make the changes permanently in the Database. We can't undo/ revoke the changes. But we can change the data in the Database.
- **Rollback:** Rollback undoes the changes till the last commit or mentioned savepoint. Sometimes we may want to undo the changes. **For example**, we have one nested query, one part has been executed successfully, and the other has thrown some exception. At that time, we want to undo the changes done by the first part, we should call Rollback() method to do that if an exception has occurred.
- **Savepoint:** Savepoint helps to create checkpoint in a transaction and it allows to perform a rollback to that particular savepoint. Any savepoint that has been created for a transaction will be automatically destroyed and become invalid once the transaction is committed or rolled back.

Till now we have seen what is commit, rollback, and savepoint and its operations. Below, we will see the methods of it and how to use it in the program.

#### Methods Of Transaction Management

**The connection interface provides 5 methods for transaction management. They are as follows:**

##### *#1) setAutoCommit() Method*

By default, the value of AutoCommit value is TRUE. After the execution of the SQL statement, it will be committed automatically. By using the setAutoCommit() method we can set the value to AutoCommit.

##### *#2) Commit() Method*

The commit method is used to commit the data. After the execution of the SQL statement, we can call the commit(). It will commit the changes which are made by the SQL statement.

**Syntax:** conn.commit();

##### *#3) Rollback() Method*

The rollback method is used to undo the changes till the last commit has happened. If we face any issue or exception in the execution flow of the SQL statements, we may roll back the transaction.

**Syntax:** conn.rollback();

##### *#4) setSavepoint() Method*

Savepoint gives you additional control over the transaction. When you set a savepoint in the transaction (a group of SQL statements), you can use the rollback() method to undo all the changes till the savepoint or after the savepoint(). setSavepoint() method is used to create a new savepoint.

##### *#5) releaseSavepoint() Method*

It is used to delete the created savepoint.

In the below program, you will get to know more about these methods and will also learn how to use it in the Java program.

1. **import** java.sql.\*;

```

2. class FetchRecords{
3. public static void main(String args[])throws Exception{
4. Class.forName("oracle.jdbc.driver.OracleDriver");
5. Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe
   ", "system", "oracle");
6. con.setAutoCommit(false);
7.
8. Statement stmt=con.createStatement();
9. stmt.executeUpdate("insert into user420 values(190,'abhi',40000)");
10. stmt.executeUpdate("insert into user420 values(191,'umesh',50000)");
11.
12. con.commit();
13. con.close();
14. }}

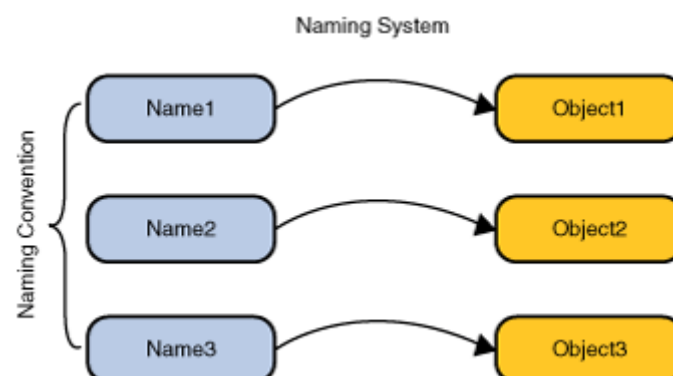
```

If you see the table emp400, you will see that 2 records has been added.

Lesson: Naming and Directory Concepts

### Naming Concepts

A fundamental facility in any computing system is the *naming service*--the means by which names are associated with objects and objects are found based on their names. When using almost any computer program or system, you are always naming one object or another. For example, when you use an electronic mail system, you must provide the name of the recipient. To access a file in the computer, you must supply its name. A naming service allows you to look up an object given its name.



A naming service's primary function is to map people friendly names to objects, such as addresses, identifiers, or objects typically used by computer programs.

For example, the [Internet Domain Name System \(DNS\)](#) maps machine names to IP Addresses:

www.example.com ==> 192.0.2.5

A file system maps a filename to a file reference that a program can use to access the contents of the file.

c:\bin\autoexec.bat ==> File Reference

These two examples also illustrate the wide range of scale at which naming services exist—from naming an object on the Internet to naming a file on the local file system.

## Names

To look up an object in a naming system, you supply it the *name* of the object. The naming system determines the syntax that the name must follow. This syntax is sometimes called the naming systems *naming convention*. A name is made up components. A name's representation consist of a component separator marking the components of the name.

Naming System	Component Separator	Names
UNIX file system	"/"	/usr/hello
DNS	"."	sales.Wiz.COM
LDAP	", " and "="	cn=Rosanna Lee, o=Sun, c=US

The UNIX file system's naming convention is that a file is named from its path relative to the root of the file system, with each component in the path separated from left to right using the forward slash character ("/"). The UNIX *pathname*, /usr/hello, for example, names a file hello in the file directory usr, which is located in the root of the file system.

DNS naming convention calls for components in the DNS name to be ordered from right to left and delimited by the dot character ("."). Thus the DNS name sales.Wiz.COM names a DNS entry with the name sales, relative to the DNS entry Wiz.COM. The DNS entry Wiz.COM, in turn, names an entry with the name Wiz in the COM entry.

The [Lightweight Directory Access Protocol \(LDAP\)](#) naming convention orders components from right to left, delimited by the comma character (","). Thus the LDAP name cn=Rosanna Lee, o=Sun, c=US names an LDAP entry cn=Rosanna Lee, relative to the entry o=Sun, which in turn, is relative to c=us. LDAP has the further rule that each component of the name must be a name/value pair with the name and value separated by an equals character ("=").

## Bindings

The association of a name with an object is called a *binding*. A file name is *bound* to a file.

The DNS contains bindings that map machine names to IP addresses. An LDAP name is bound to an LDAP entry.

## References and Addresses

Depending on the naming service, some objects cannot be stored directly by the naming service; that is, a copy of the object cannot be placed inside the naming service. Instead, they must be stored by reference; that is, a *pointer* or *reference* to the object is placed inside the naming service. A reference represents information about how to access an object. Typically, it is a compact representation that can be used to communicate with the object, while the object itself might contain more state information. Using the reference, you can contact the object and obtain more information about the object.

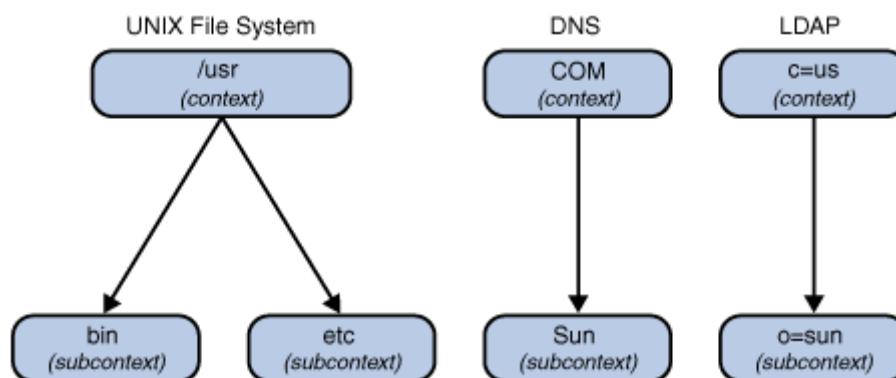
For example, an airplane object might contain a list of the airplane's passengers and crew, its flight plan, and fuel and instrument status, and its flight number and departure time. By contrast, an airplane object reference might contain only its flight number and departure time. The reference is a much more compact representation of information about the airplane object and can be used to obtain additional information. A file object, for example, is accessed using a *file reference*. A printer object, for example, might contain the state of the printer, such as its current queue and the amount of paper in the paper tray. A printer object reference, on the other hand, might contain only information on how to reach the printer, such as its print server name and printing protocol.

Although in general a reference can contain any arbitrary information, it is useful to refer to its contents as *addresses* (or communication end points): specific information about how to access the object.

For simplicity, this tutorial uses "object" to refer to both objects and object references when a distinction between the two is not required.

## Context

A *context* is a set of name-to-object bindings. Every context has an associated naming convention. A context always provides a lookup (*resolution*) operation that returns the object, it typically also provides operations such as those for binding names, unbinding names, and listing bound names. A name in one context object can be bound to another context object (called a *subcontext*) that has the same naming convention.





A file directory, such as /usr, in the UNIX file system represents a context. A file directory named relative to another file directory represents a subcontext (UNIX users refer to this as a *subdirectory*). That is, in a file directory /usr/bin, the directory bin is a subcontext of usr. A DNS domain, such as COM, represents a context. A DNS domain named relative to another DNS domain represents a subcontext. For the DNS domain Sun.COM, the DNS domain Sun is a subcontext of COM.

Finally, an LDAP entry, such as c=us, represents a context. An LDAP entry named relative to another LDAP entry represents a subcontext. For the LDAP entry o=sun,c=us, the entry o=sun is a subcontext of c=us.

## Naming Systems and Namespaces

A *naming system* is a connected set of contexts of the same type (they have the same naming convention) and provides a common set of operations.

A system that implements the DNS is a naming system. A system that communicates using the LDAP is a naming system.

A naming system provides a *naming service* to its customers for performing naming-related operations. A naming service is accessed through its own interface. The DNS offers a naming service that maps machine names to IP addresses. LDAP offers a naming service that maps LDAP names to LDAP entries. A file system offers a naming service that maps filenames to files and directories.

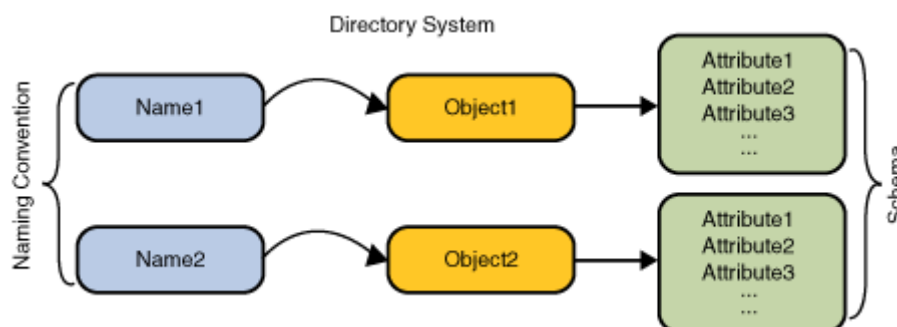
A *namespace* is the set of all possible names in a naming system. The UNIX file system has a namespace consisting of all of the names of files and directories in that file system. The DNS namespace contains names of DNS domains and entries. The LDAP namespace contains names of LDAP entries.

## Directory Concepts

Many naming services are extended with a *directory service*. A directory service associates names with objects and also associates such objects with *attributes*.

directory service = naming service + objects containing attributes

You not only can look up an object by its name but also get the object's attributes or *search* for the object based on its attributes.



An example is the telephone company's directory service. It maps a subscriber's name to his address and phone number. A computer's directory service is very much like a telephone company's directory service in that both can be used to store information such as telephone numbers and addresses. The computer's directory service is much more powerful, however, because it is available online and can be used to store a variety of information that can be utilized by users, programs, and even the computer itself and other computers.

A *directory object* represents an object in a computing environment. A directory object can be used, for example, to represent a printer, a person, a computer, or a network. A directory object contains *attributes* that describe the object that it represents.

### Attributes

A directory object can have *attributes*. For example, a printer might be represented by a directory object that has as attributes its speed, resolution, and color. A user might be represented by a directory object that has as attributes the user's e-mail address, various telephone numbers, postal mail address, and computer account information.

An attribute has an *attribute identifier* and a set of *attribute values*. An attribute identifier is a token that identifies an attribute independent of its values. For example, two different computer accounts might have a "mail" attribute; "mail" is the attribute identifier. An attribute value is the contents of the attribute. The email address, for example, might have:

Attribute Identifier : Attribute Value  
mail john.smith@example.com

### Directories and Directory Services

A *directory* is a connected set of directory objects. A *directory service* is a service that provides operations for creating, adding, removing, and modifying the attributes associated with objects in a directory. The service is accessed through its own interface.

Many examples of directory services are possible.

#### Network Information Service (NIS)

NIS is a directory service available on the UNIX operating system for storing system-related information, such as that relating to machines, networks, printers, and users.

#### [Oracle Directory Server](#)

The Oracle Directory Server is a general-purpose directory service based on the Internet standard [LDAP](#).

### Search Service

You can look up a directory object by supplying its name to the directory service. Alternatively, many directories, such as those based on the LDAP, support the notion of *searches*. When you search, you can supply not a name but a *query* consisting of a logical expression in which you specify the attributes that the object or objects must have. The query is called a *search filter*. This style of searching is sometimes called *reverse lookup* or *content-*

*based searching*. The directory service searches for and returns the objects that satisfy the search filter.

For example, you can query the directory service to find:

- all users that have the attribute "age" greater than 40 years.
- all machines whose IP address starts with "192.113.50".

### Combining Naming and Directory Services

Directories often arrange their objects in a hierarchy. For example, the LDAP arranges all directory objects in a tree, called a *directory information tree (DIT)*. Within the DIT, an organization object, for example, might contain group objects that might in turn contain person objects. When directory objects are arranged in this way, they play the role of naming contexts in addition to that of containers of attributes.

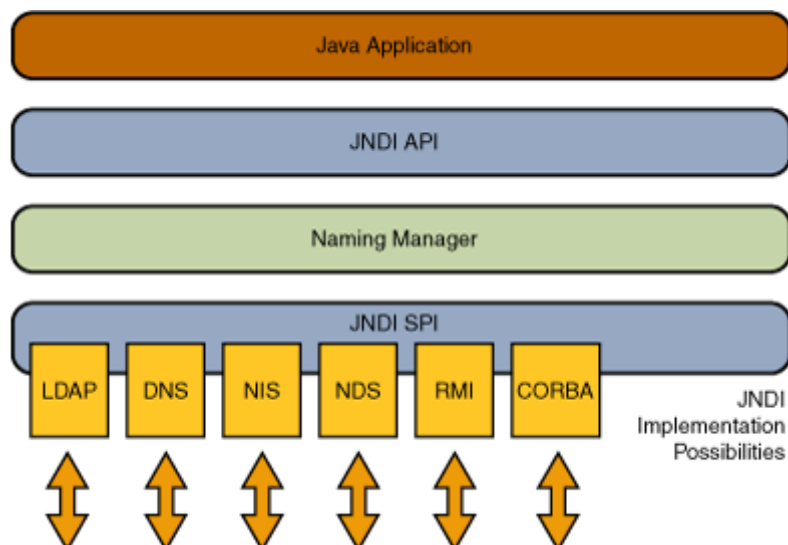
[« Previous](#) • [Trail](#) • [Next »](#)

Lesson: Overview of JNDI

The Java Naming and Directory Interface™ (JNDI) is an application programming interface (API) that provides [naming](#) and [directory](#) functionality to applications written using the Java™ programming language. It is defined to be independent of any specific directory service implementation. Thus a variety of directories -new, emerging, and already deployed can be accessed in a common way.

### Architecture

The JNDI architecture consists of an API and a service provider interface (SPI). Java applications use the JNDI API to access a variety of naming and directory services. The SPI enables a variety of naming and directory services to be plugged in transparently, thereby allowing the Java application using the JNDI API to access their services. See the following figure:



## Packaging

JNDI is included in the Java SE Platform. To use the JNDI, you must have the JNDI classes and one or more service providers. The JDK includes service providers for the following naming/directory services:

- Lightweight Directory Access Protocol (LDAP)
- Common Object Request Broker Architecture (CORBA) Common Object Services (COS) name service
- Java Remote Method Invocation (RMI) Registry
- Domain Name Service (DNS)

Other service providers can be downloaded from the [JNDI page](#) or obtained from other vendors.

The JNDI is divided into five packages:

- [javax.naming](#)
- [javax.naming.directory](#)
- [javax.naming.ldap](#)
- [javax.naming.event](#)
- [javax.naming.spi](#)

The next part of the lesson has a brief description of the JNDI packages.

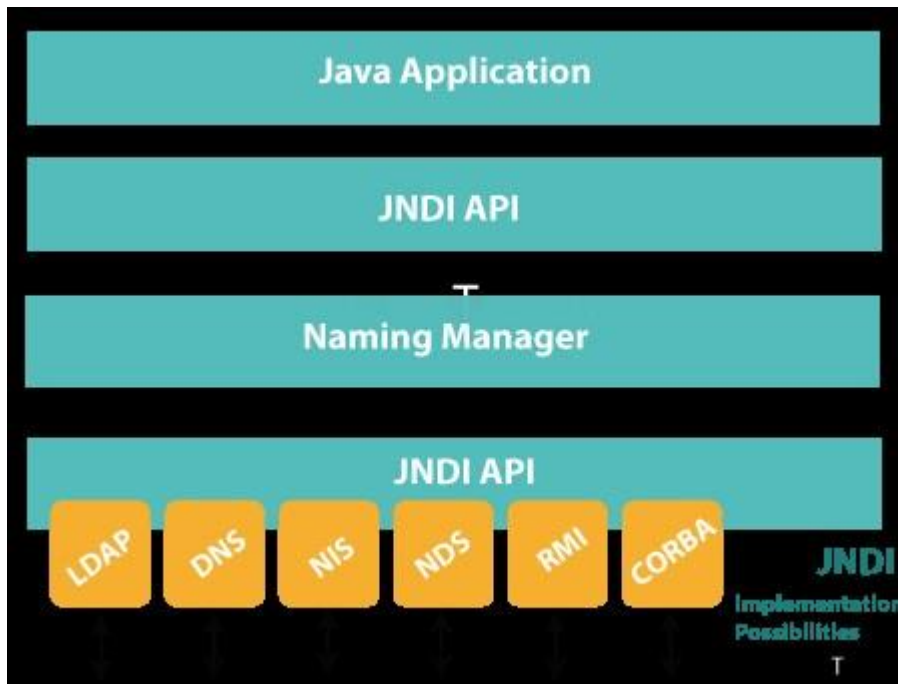
### What is JNDI in Java

The interface used by the Java programming language is by the name Java Naming and Directory Interface (JNDI). It is an API (application programming interface) that communicates with servers and uses naming conventions to get files from databases. A word or a single phrase might serve as the naming convention.

It can also be added to a socket in order to use servers that transfer flat files or data files in a project to perform socket programming. In browsers with a lot of directories, it can also be used on web pages. Using the Java programming language, JNDI gives Java users the ability to search Java objects.

### The architecture of JNDI in the Java Interface:

The Service Provider Interface (SPI) which is made up of an API and an interface known as JNDI.



JNDI, is visible in the architecture as a series of various directories. There is a connection between the Java program and the JNDI architecture, as seen in this diagram. Since the interface is used to connect to many directories, the levels make it apparent that the JNDI API is above it. The following lists a few of the directory services.

- Domain name service
- Lightweight Directory Access Protocol.
- Remote Method Invocation in Java

The JNDI SPI connects with the directories listed above to create a platform with the JNDI implementation choices.

### JNDI Packages:

In Java, JNDI SPI is specifically used by five packages. Some of the packages use the javax.naming language. There are classes and interfaces for name service access in the package known as javax.naming. Lookup, list Bindings, and Name are a few of the available functions.

Java.naming.directory is the second one. This package is a more sophisticated version of Java.naming directory that helps in obtaining the data as objects. The packages java. Naming. event and Java. naming. spi are two more examples.

**Additionally, JNDI is a key component of three of the newest Java technologies. They are as follows:**

- The Java Database Connectivity (JDBC) package
- The Java Messaging Service (JMS)

- Enterprise Java Beans (EJB)

In the Java programming language, there are two functions called `bind()` and `lookup()` that are used to name objects and look them up in directories, respectively.

```
Context.bind("name", object)
```

Any name can be given to the current object in the directory in this case by changing the name. In this instance of the `bind` function, the object's name has been set.

```
Object hello= Context.lookup("name")
```

The `hello` object in this function searches the directory for the item's name. Depending on the type of directory supported, different types of serialized or non-serialized data are also used.

#### Example of JNDI Interface in Java:

This program, which operates through a menu system, asks the user to input the principal amount before printing the simple interest, compound interest, and the difference between the simple and compound interest based on the user's preferences.

Additionally, the program ends if the user decides not to utilize it any further. The amount of time it takes for interest to start accruing is 7 years, and the rate of interest is fixed at 8.5 percent. All interest rates are calculated as a result.

The development of a menu-driven application that allows users to enter a principal amount and calculate simple interest, compound interest, and the absolute difference between the two.

#### Implementation:

**FileName:** JndiExample.java

```
1. import java.util.*;
2. import java.io.*;
3. public class JndiExample
4. {
5.     public static void main(String[] args) throws Exception
6.     {
7.         BufferedReader ob = new BufferedReader(new InputStreamReader(System.in));

8.         System.out.print("Enter the Principal Amount : ");
9.         float P = Float.parseFloat(ob.readLine());
10.        int ch = 0;
11.        do
12.        {
```

```

13.     ch = 0;
14.     // Resetting the selection made by the user
15.     // showing the menu of options
16.     System.out.println("----- M E N U -----");
17.     System.out.println("1 - To Find the Simple Interest amount");
18.     System.out.println("2 - To Find the Compound Interest amount");
19.     System.out.println("3 - To Determine the Simple Interest and Compound Inter
est Difference ");
20.     System.out.println("4 - To quit the program's operation");
21.     System.out.print("Enter the user's Choice : ");
22.     ch = Integer.parseInt(ob.readLine());
23.     System.out.println("");
24.     switch(ch)
25.     {
26.         case 1://for the case simple interest
27.             System.out.println("The Simple Interest is Rs."+simple(P));
28.             break;
29.         case 2://for the case compound interest
30.             System.out.println("The Compound Interest is Rs."+compound(P));
31.             break;
32.         case 3://for the case difference between simple and compound interests
33.             System.out.println("The Absolute Difference is Rs."+(compound(P)-
simple(P)));
34.             break;
35.         case 4:
36.             System.out.println("Program Terminated");
37.             break;
38.         default:
39.             System.out.println("Invalid Option");
40.     }
41.     System.out.println("\n");
42. }while(ch!=4);
43. }
44. public static float simple(float p)//to calculate the simple interest
45. {
46.     return (float)((p*8.5*7.0)/100.0); //returning the simple interest
47. }

```

```

48.     public static float compound(float p)//to calculate the compound interest
49.     {
50.         return (p*(float)(Math.pow((1.0+(8.5/100.0)),7.0)-
           1.0));//returning the compound interest
51.     }
52. }

```

### Output:

```

D:\JavaTPoint>javac JndiExample.java

D:\JavaTPoint>java JndiExample
Enter the Principal Amount : 1000
----- M E N U -----
1 - To Find the Simple Interest amount
2 - To Find the Compound Interest amount
3 - To Determine the Simple Interest and Compound Interest Difference
4 - To quit the program's operation
Enter the user's Choice : 1

The Simple Interest is Rs.595.0

----- M E N U -----
1 - To Find the Simple Interest amount
2 - To Find the Compound Interest amount
3 - To Determine the Simple Interest and Compound Interest Difference
4 - To quit the program's operation
Enter the user's Choice : 2

The Compound Interest is Rs.770.1423

----- M E N U -----
1 - To Find the Simple Interest amount
2 - To Find the Compound Interest amount
3 - To Determine the Simple Interest and Compound Interest Difference
4 - To quit the program's operation
Enter the user's Choice : 3

The Absolute Difference is Rs.175.14227

```

### Benefits of JNDI Interface in Java:

The advantages of a JNDI naming service include:

- You don't need to understand how the data from your application is stored in a directory service to create programs using JNDI APIs.



- As long as the resources are reachable from a JNDI directory, you can develop code to retrieve data on a single machine or across numerous systems.
- Any programming language that offers a JNDI API can be used to create programs utilizing JNDI.
- Standardizes the configuration of database connections.
- Enables the use of password encryption (passwords are currently stored as clear text strings). See *Encrypting Passwords in Tomcat* for more on encrypting your database passwords.
- Allows more fine-grained control over database connection pooling settings.

#### Limitations of JNDI Interface in Java:

JNDI has some restrictions and, regrettably, is not intended for high-performance environments:

- Data like configuration information does not fit into this paradigm since only specific sorts of data can be stored using the standard set mechanism offered by JNDI.
- The provision does not cover transactions.
- Although some implementations offer APIs to build up SSL connections to directory servers, there is no built-in security model.

## UNIT-IV

### SERVER SIDE PROGRAMMING

Servlets - Introduction to servlets - Servlets life cycle - Java Server Pages (JSP): Introduction, Java Server Pages Overview, First Java Server Page Example, Implicit Objects, Scripting, Standard Actions, Directives, Custom Tag Libraries

#### Servlets

**Servlet** technology is used to create a web application (resides at server side and generates a dynamic web page).

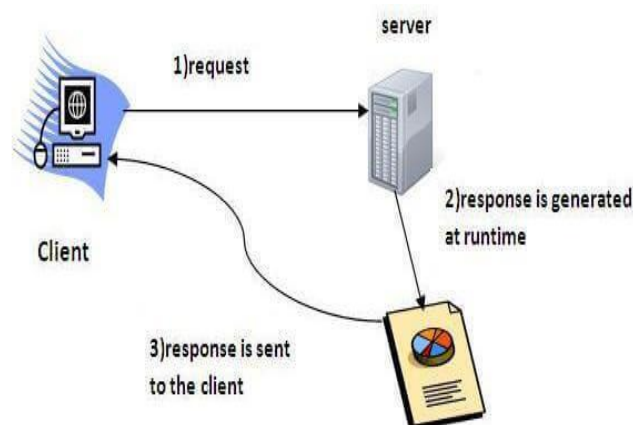
**Servlet** technology is robust and scalable because of java language. Before Servlet, CGI (Common Gateway Interface) scripting language was common as a server-side programming language. However, there were many disadvantages to this technology. We have discussed these disadvantages below.

There are many interfaces and classes in the Servlet API such as Servlet, GenericServlet, HttpServlet, ServletRequest, ServletResponse, etc.

#### What is a Servlet?

Servlet can be described in many ways, depending on the context.

- Servlet is a technology which is used to create a web application.
- Servlet is an API that provides many interfaces and classes including documentation.
- Servlet is an interface that must be implemented for creating any Servlet.
- Servlet is a class that extends the capabilities of the servers and responds to the incoming requests. It can respond to any requests.
- Servlet is a web component that is deployed on the server to create a dynamic web page.



#### What is a 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 elements such as HTML, CSS, and JavaScript. The web components typically execute in Web Server and respond to the HTTP request.

---

## CGI (Common Gateway Interface)

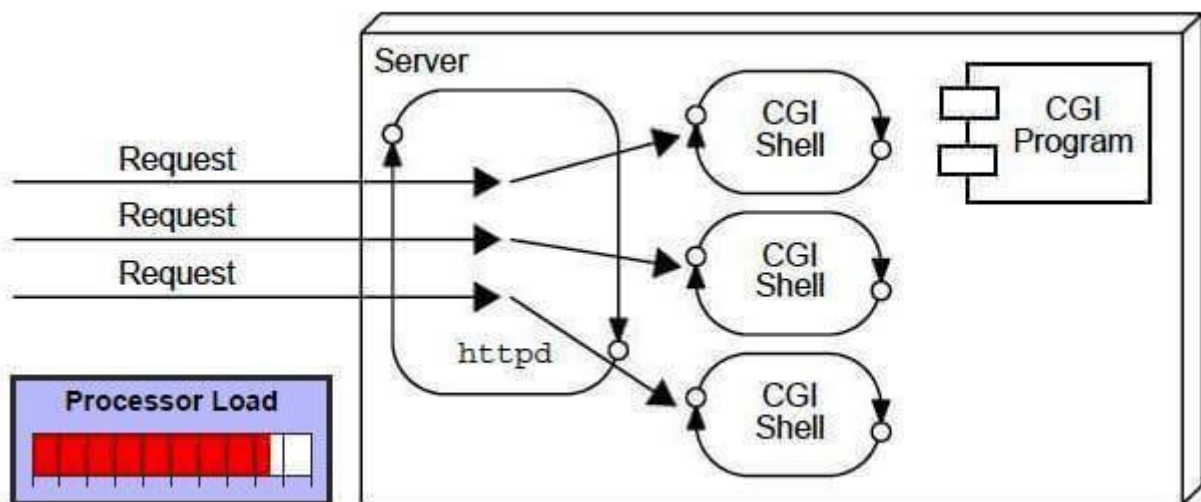
CGI technology enables the web server to call an external program and pass HTTP request information to the external program to process the request. For each request, it starts a new process **Servlets Packages**

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

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

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

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

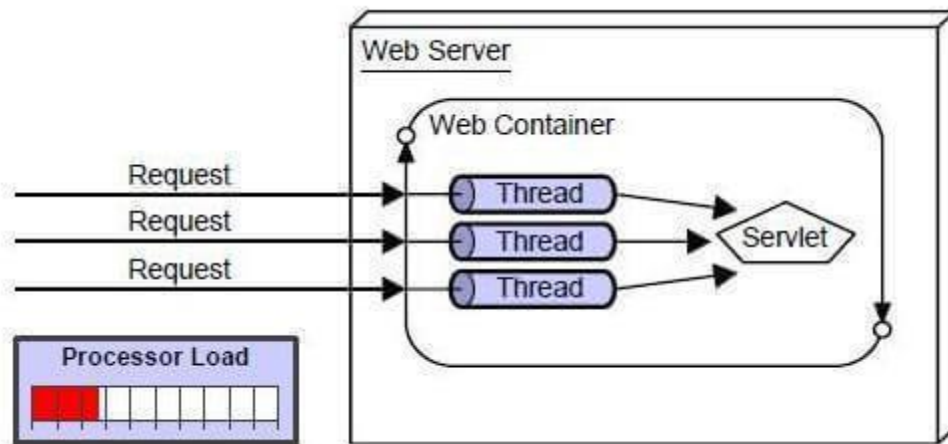


## Disadvantages of CGI

There are many problems in CGI technology:

1. If the number of clients increases, it takes more time for sending the response.
  2. For each request, it starts a process, and the web server is limited to start processes.
  3. It uses platform dependent language e.g. [C](#), [C++](#), [perl](#).
-

## Advantages of Servlet



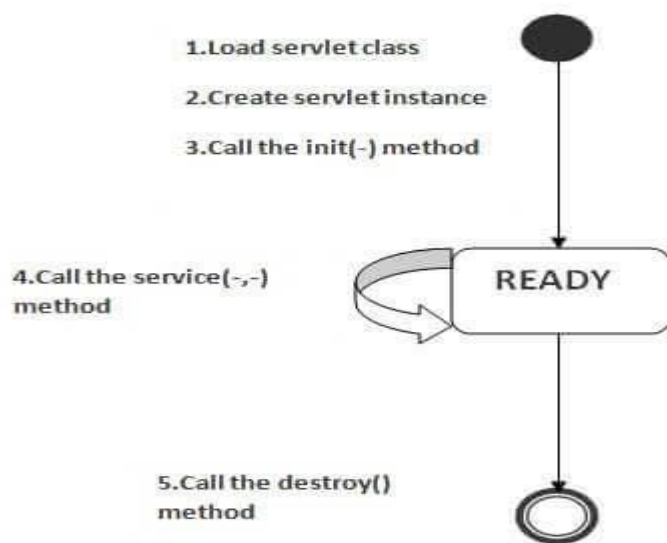
There are many advantages of Servlet over CGI. The web container creates threads for handling the multiple requests to the Servlet. Threads have many benefits over the Processes such as they share a common memory area, lightweight, cost of communication between the threads are low. The advantages of Servlet are as follows:

1. **Better performance:** because it creates a thread for each request, not process.
2. **Portability:** because it uses Java language.
3. **Robust:** [JVM](#) manages Servlets, so we don't need to worry about the memory leak, [garbage collection](#), etc.
4. **Secure:** because it uses java language.

## Life Cycle of a Servlet (Servlet Life Cycle)

The web container maintains the life cycle of a servlet instance. Let's see the life cycle of the servlet:

1. Servlet class is loaded.
2. Servlet instance is created.
3. init method is invoked.
4. service method is invoked.
5. destroy method is invoked.



As displayed in the above diagram, there are three states of a servlet: new, ready and end. The servlet is in new state if servlet instance is created. After invoking the `init()` method, Servlet comes in the ready state. In the ready state, servlet performs all the tasks. When the web container invokes the `destroy()` method, it shifts to the end state.

### 1) Servlet class is loaded

The classloader is responsible to load the servlet class. The servlet class is loaded when the first request for the servlet is received by the web container.

### 2) Servlet instance is created

The web container creates the instance of a servlet after loading the servlet class. The servlet instance is created only once in the servlet life cycle.

### 3) init method is invoked

The web container calls the `init` method only once after creating the servlet instance. The `init` method is used to initialize the servlet. It is the life cycle method of the `javax.servlet.Servlet` interface. Syntax of the `init` method is given below:

1. **public void** `init(ServletConfig config)` **throws** `ServletException`

### 4) service method is invoked

The web container calls the `service` method each time when request for the servlet is received. If servlet is not initialized, it follows the first three steps as described above then calls the `service` method. If servlet is initialized, it calls the `service` method. Notice that servlet is initialized only once. The syntax of the `service` method of the `Servlet` interface is given below:

1. **public void** `service(ServletRequest request, ServletResponse response)`
2. **throws** `ServletException, IOException`

### 5) destroy method is invoked

The web container calls the destroy method before removing the servlet instance from the service. It gives the servlet an opportunity to clean up any resource for example memory, thread etc. The syntax of the destroy method of the Servlet interface is given below:

1. **public void** destroy()

### Servlets Packages

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

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

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

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

like any other Java program, you need to compile a servlet by using the Java compiler **javac** and after compilation the servlet application, it would be deployed in a configured environment to test and run..

This development environment setup involves the following steps –

### Setting up Java Development Kit

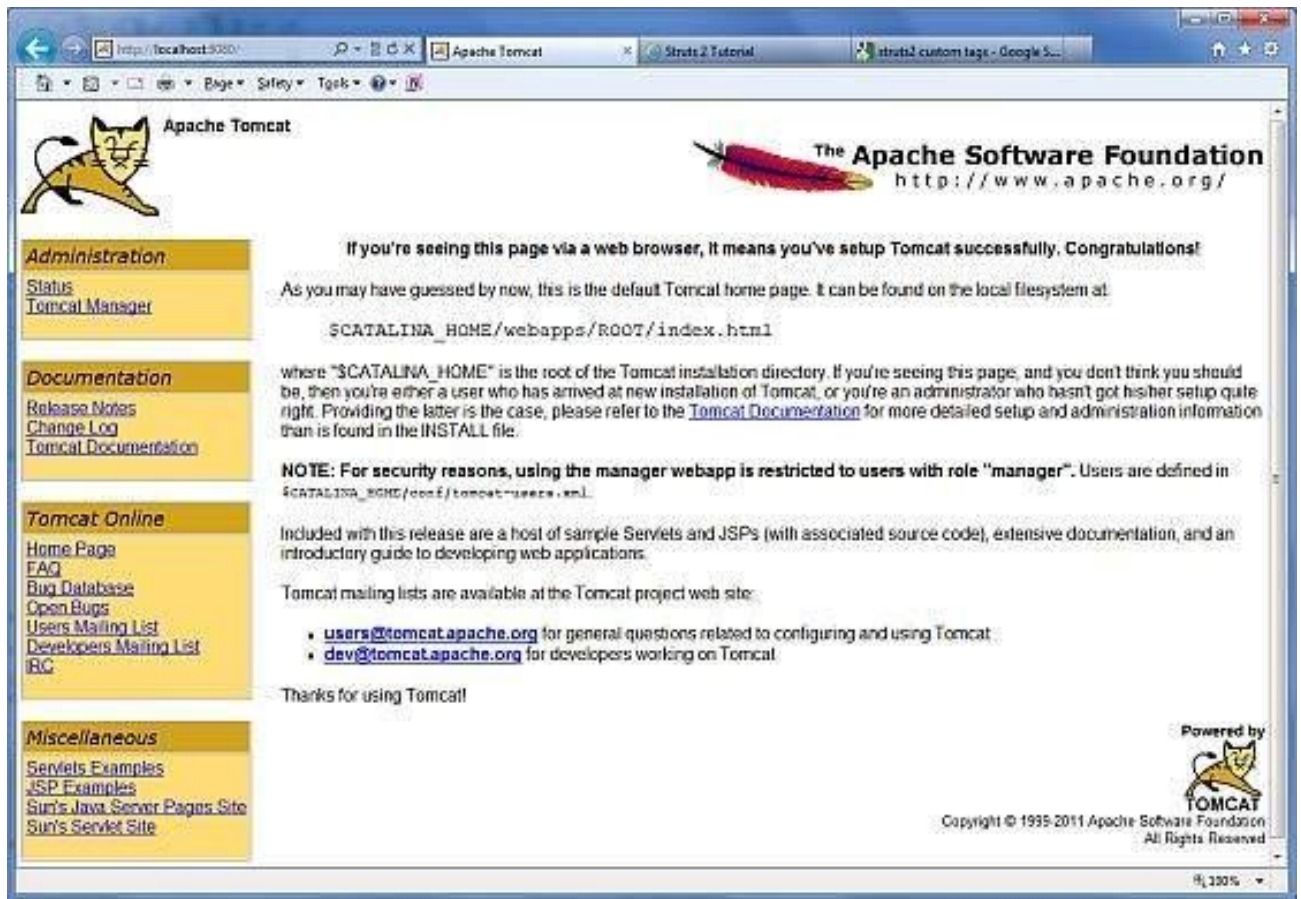
This step involves downloading an implementation of the Java Software Development Kit (SDK) and setting up PATH environment variable appropriately.

### Setting up Web Server – Tomcat

A number of Web Servers that support servlets are available in the market. Some web servers are freely downloadable and Tomcat is one of them.

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

After startup, the default web applications included with Tomcat will be available by visiting **http://localhost:8080/**. If everything is fine then it should display following result –



## Life Cycle of a Servlet (Servlet Life Cycle)

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

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

Now let us discuss the life cycle methods in detail.

### The **init()** Method

The init method is called only once. It is called only when the servlet is created, and not called for any user requests afterwards. So, it is used for one-time initializations, just as with the init method of applets.

The servlet is normally created when a user first invokes a URL corresponding to the servlet, but you can also specify that the servlet be loaded when the server is first started.

When a user invokes a servlet, a single instance of each servlet gets created, with each user request resulting in a new thread that is handed off to doGet or doPost as appropriate. The



init() method simply creates or loads some data that will be used throughout the life of the servlet.

The init method definition looks like this –

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

### The service() Method

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

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

Here is the signature of this method –

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

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

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

### 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.

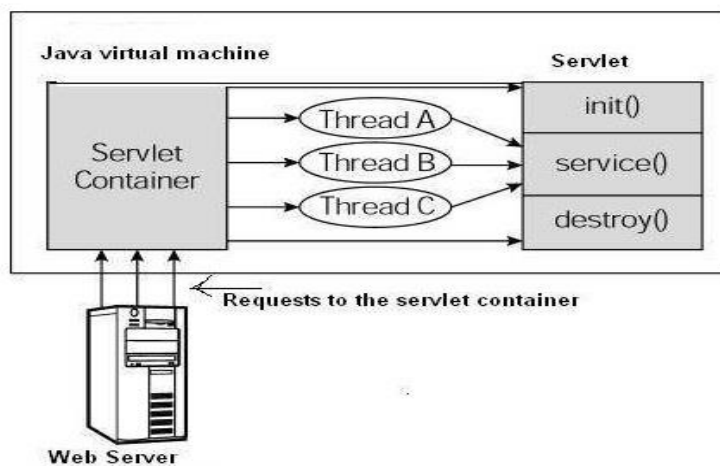
After the destroy() method is called, the servlet object is marked for garbage collection. The destroy method definition looks like this –

```
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.



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

## JSP Implicit Objects – Syntax and Examples

In dynamic web application development, client and server interactions are essential for sending and receiving information over the Internet. When the browser requests a webpage, a lot of information is sent to the webserver. Such information cannot be read directly because

such information is part of an HTTP header request. In this chapter, you will learn about the various request headers provided by JSP.

The JSP request can be defined as an implicit object is an instance of "HttpServletRequest" and is formed for all JSP requests through the web container. This JSP request gets request information like a parameter, remote address, header information, server port, server name, character encoding, content type, etc.

### JSP request Implicit Object

- A request object is an implicit object that is used to request an implicit object, which is to receive data on a JSP page, which has been submitted by the user on the previous JSP/HTML page.
- The request implicit object used in Java is an instance of a `javax.servlet.http.HttpServletRequest` interface where a client requests a page every time the JSP engine has to create a new object for characterizing that request.
- The container creates it for every request.
- It is used to request information such as parameters, header information, server names, cookies, and HTTP methods.
- It uses the `getParameter()` method to access the request parameter.

Here is an example of a JSP request implicit object where a user submits login information, and another JSP page receives it for processing:

#### Example (HTML file):

Copy Code<!DOCTYPE html>

```
<html>
  <head>
    <title>User login form</title>
  </head>
  <body>
    <form action="login.jsp">
      Please insert Username: <input type="text" name="u_name" /> <br />
      Please insert Password: <input type="text" name="passwd" /> <br />
      <input type="submit" value="Submit Details" />
    </form>
  </body>
</html>
```

#### Example (login.jsp):

```
Copy Code<%@ page import = " java.util.* " %>
<%
String username = request.getParameter("u_name");
String password = request.getParameter("passwd");
out.print("Name: "+username+" Password: " +passwd);
%>
```

The brief information about the implicit objects are given below:

### The out Implicit Object

- An out object is an implicit object for writing data to the buffer and sending output as a response to the client's browser.

- The out implicit object is an instance of a javax.servlet.jsp.jspWriter class.
- You will learn more about the various concepts of the out Object in subsequent chapters.

Example (HTML file):

```
Copy Code<!DOCTYPE html>
<html>
  <head>
    <title>Please insert a User name and a password</title>
  </head>
  <body>
    <% out.println("Today's date-time: "+java.util.Calendar.getInstance().getTime()); %>
  </body>
</html>
```

Output:

Today's date-time: Nov 01 12:10:05 IST 2020

The request Implicit Object

- A request object is an implicit object that is used to request an implicit object, which is to receive data on a JSP page, which has been submitted by the user on the previous JSP/HTML page.
- The request implicit object used in Java is an instance of a javax.servlet.http.HttpServletRequest interface where a client requests a page every time the JSP engine has to create a new object for characterizing that request.
- The container creates it for every request.
- It is used to request information such as parameters, header information, server names, cookies, and HTTP methods.
- It uses the getParameter() method to access the request parameter.

Here is an example of a JSP request implicit object where a user submits login information, and another JSP page receives it for processing:

Example (HTML file):

```
Copy Code<!DOCTYPE html>
<html>
  <head>
    <title>Please insert a User name and a password</title>
  </head>
  <body>
    <form action="login.jsp">
      Please insert Username: <input type="text" name="u_name" /> <br />
      Please insert Password: <input type="text" name="passwd" /> <br />
      <input type="submit" value="Submit Details" />
    </form>
  </body>
</html>
```

Example (login.jsp):

```
Copy Code<%@ page import = " java.util.* " %>
<%
```

```
String username = request.getParameter("u_name");
String password = request.getParameter("passwd");
out.print("Name: "+username+" Password: " +passwd);
%>
```

The lesson "[JSP Request](#)" contains more detailed information about "The request Implicit Object" and its methods.

### The response Implicit Object

- A response object is an implicit object implemented to modify or deal with the reply sent to the client (i.e., browser) after processing the request, such as redirect responding to another resource or an error sent to a client.
- The response implicit object is an instance of a `javax.servlet.http.HttpServletResponse` interface.
- The container creates it for every request.
- You will learn more about the various concepts of the request and response in subsequent chapters.

### The session Implicit Object

- A session object is the most commonly used implicit object implemented to store user data to make it available on other JSP pages until the user's session is active.
- The session implicit object is an instance of a `javax.servlet.http.HttpSession` interface.
- This session object has different session methods to manage data within the session scope.
- You will learn more about the use of the session in subsequent chapters.

### The application Implicit Object

An application object is another implicit object implemented to initialize application-wide parameters and maintain functional data throughout the JSP application.

### The exception Implicit Object

- An exception implicit object is implemented to handle exceptions to display error messages.
- The exception implicit object is an instance of the `java.lang.Throwable` class.
- It is only available for JSP pages, with the `isErrorPage` value set as "True". This means Exception objects can only be used in error pages.

### Example (HTML file):

```
Copy Code<!DOCTYPE html>
<html>
  <head>
    <title>Enter two Integers for Division</title>
  </head>
  <body>
    <form action="submit.jsp">
      Insert first Integer: <input type="text" name="numone" /><br />
      Insert second Integer: <input type="text" name="numtwo" /><br />
      <input type="submit" value="Get Results" />
    </form>
  </body>
</html>
```

```
</form>
</body>
</html>
```

Example (submit.jsp):

```
Copy Code<%@ page errorPage="exception.jsp" %>
```

```
<%
String num1 = request.getParameter("numone");
String num2 = request.getParameter("numtwo");
int var1 = Integer.parseInt(num1);
int var2 = Integer.parseInt(num2);
int r = var1 / var2;
out.print("Output is: " + r);
%>
```

Example (exception.jsp):

```
Copy Code<%@ page isErrorPage='true' %>
```

```
<%
out.print("Error Message : ");
out.print(exception.getMessage());
%>
```

### The page Implicit Object

A page object is an implicit object that is referenced to the current instance of the servlet. You can use it instead. Covering it specifically is hardly ever used and not a valuable implicit object while building a JSP application.

```
Copy Code<% String pageName = page.toString();
out.println("The current page is: " + pageName);%>
```

### The config Implicit Object

A config object is a configuration object of a servlet that is mainly used to access and receive configuration information such as servlet context, servlet name, configuration parameters, etc. It uses various methods used to fetch configuration information.

## JSP:

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

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

### Advantages of JSP over Servlet

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

### ***1) Extension to Servlet***

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

### ***2) Easy to maintain***

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

### ***3) Fast Development: No need to recompile and redeploy***

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

### ***4) Less code than Servlet***

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

## **What is JSP LifeCycle?**

JSP Life Cycle is defined as translation of JSP Page into servlet as a JSP Page needs to be converted into servlet first in order to process the service requests. The Life Cycle starts with the creation of JSP and ends with the disintegration of that.

## **Different Phases of JSP Life Cycle**

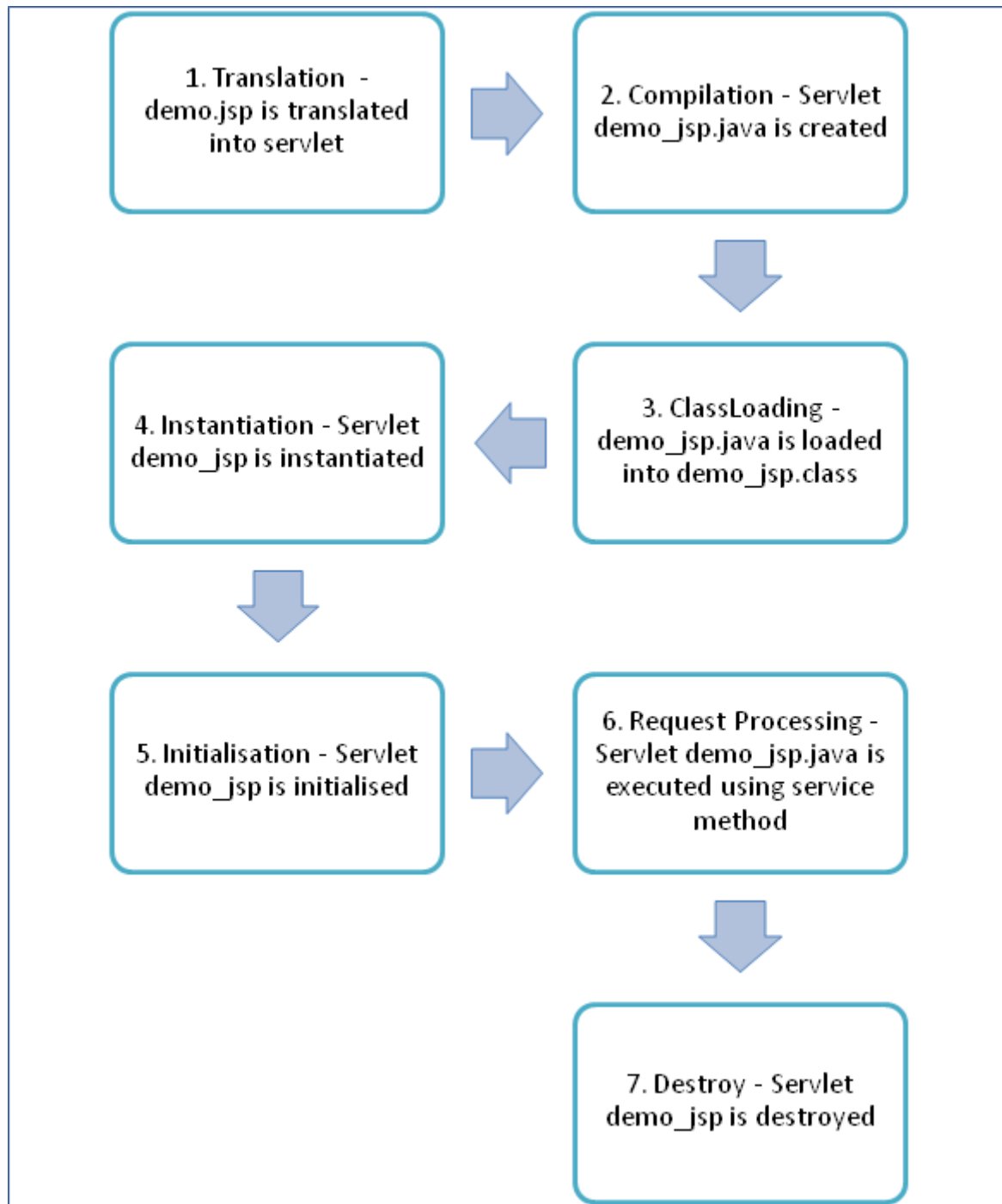
When the browser asks for a JSP, JSP engine first checks whether it needs to compile the page. If the JSP is last compiled or the recent modification is done in JSP, then the JSP engine compiles the page.

## **Compilation process of JSP page involves three steps:**

- Parsing of JSP
- Turning JSP into servlet
- Compiling the servlet

## **JSP Life Cycle Diagram**

JSP Lifecycle is depicted in the below diagram.



**The following steps explain the life cycle of JSP:**

1. Translation of JSP page
2. Compilation of JSP page(Compilation of JSP page into \_jsp.java)
3. Classloading (\_jsp.java is converted to class file \_jsp.class)
4. Instantiation(Object of generated servlet is created)
5. Initialisation(`_jspinit()` method is invoked by container)
6. Request Processing(`_jspervice()` method is invoked by the container)
7. Destroy (`_jspDestroy()` method invoked by the container)

## 1) Translation of the JSP Page:

A [Java](#) servlet file is generated from a JSP source file. This is the first step of JSP life cycle. In translation phase, container validates the syntactic correctness of JSP page and tag files.

- The JSP container interprets the standard directives and actions, and the custom actions referencing tag libraries (they are all part of JSP page and will be discussed in the later section) used in this JSP page.
- In the above pictorial description, demo.jsp is translated to demo\_jsp.java in the first step
- Let's take an example of "demo.jsp" as shown below:

### Demo.jsp

```
<html>
<head>
<title>Demo JSP</title>
</head>
<%
int demvar=0;%>
<body>
Count is:
<% Out.println(demovar++); %>
<body>
</html>
```

### Code Explanation for Demo.jsp

**Code Line 1:** html start tag

**Code Line 2:** Head tag

**Code Line 3 – 4:** Title Tag i.e. Demo JSP and closing head tag

**Code Line 5 – 6:** Scriptlet tag wherein initializing the variable demo

**Code Line 7 – 8:** In body tag, a text to be printed in the output (Count is: )

**Code Line 9:** Scriptlet tag where trying to print the variable demovar with incremented value

**Code Line 10 – 11:** Body and HTML tags closed

Demo JSP Page is converted into demo\_jsp servlet in the below code.



```

1  Public class demp_jsp extends HttpServlet{
2      Public void _jspervice(HttpServletRequest request, HttpServletResponse response)
3          Throws IOException, ServletException
4      {
5  PrintWriter out = response.getWriter();
6  response.setContentType("text/html");
7  out.write("<html><body>");
8  int demovar=0;
9  out.write("Count is:");
10 out.print(demovar++);
11 out.write("</body></html>");
12 }
13 }
14

```

### Code explanation for Demo\_jsp.java

**Code Line 1:** Servlet class demo\_jsp is extending parent class HttpServlet

**Code Line 2 – 3:** Overriding the service method of jsp i.e. \_jspervice which has HttpServletRequest and HttpServletResponse objects as its parameters

**Code Line 4:** Opening method

**Code Line 5:** Calling the method `getWriter()` of response object to get PrintWriter object (prints formatted representation of objects to text output stream)

**Code Line 6:** Calling `setContentType` method of response object to set the content type

**Code Line 7:** Using `write()` method of PrintWriter object trying to parse html

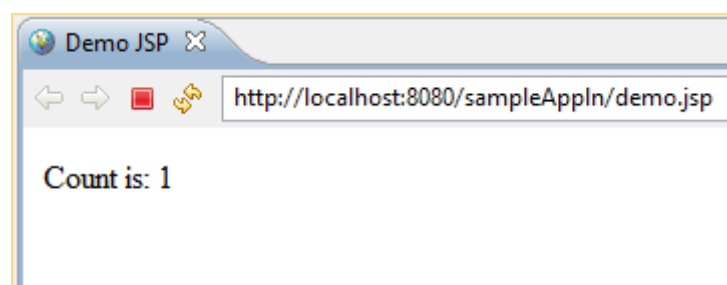
**Code Line 8:** Initializing demovar variable to 0

**Code Line 9:** Calling `write()` method of PrintWriter object to parse the text

**Code Line 10:** Calling `print()` method of PrintWriter object to increment the variable demovar from  $0+1=1$ . Hence, the output will be 1

**Code Line 11:** Using `write()` method of PrintWriter object trying to parse html

**Output:**



- Here you can see that in the screenshot the output is 1 because demvar is initialized to 0 and then incremented to  $0+1=1$

### In the above example,

- demo.jsp, is a JSP where one variable is initialized and incremented. This JSP is converted to the servlet (demo\_jsp.class ) wherein the JSP engine loads the JSP Page and converts to servlet content.
- When the conversion happens all template text is converted to `println()` statements and all [JSP elements](#) are converted to Java code.

This is how a simple JSP page is translated into a servlet class.

### 2) Compilation of the JSP Page

- The generated java servlet file is compiled into java servlet class
- The translation of java source page to its implementation class can happen at any time between the deployment of JSP page into the container and processing of the JSP page.
- In the above pictorial description demo\_jsp.java is compiled to a class file demo\_jsp.class

### 3) Classloading

- Servlet class that has been loaded from JSP source is now loaded into the container

### 4) Instantiation

- In this step the object i.e. the instance of the class is generated.
- The container manages one or more instances of this class in the response to requests and other events. Typically, a JSP container is built using a servlet container. A JSP container is an extension of servlet container as both the container support JSP and servlet.
- A JSPPage interface which is provided by container provides `init()` and `destroy()` methods.
- There is an interface HttpJSPPage which serves HTTP requests, and it also contains the service method.

### 5) Initialization

```
public void jspInit()
{
    //initializing the code
}
```

- `_jspinit()` method will initiate the servlet instance which was generated from JSP and will be invoked by the container in this phase.
- Once the instance gets created, `init` method will be invoked immediately after that
- It is only called once during a JSP life cycle, the method for initialization is declared as shown above

### 6) Request Processing

```
void _jspervice(HttpServletRequest request HttpServletResponse response)
```

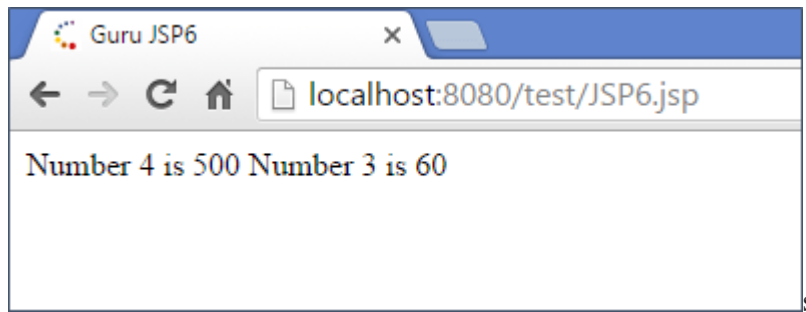
```
{
    //handling all request and responses
}
```

- `_jspervice()` method is invoked by the container for all the requests raised by the JSP page during its life cycle
- For this phase, it has to go through all the above phases and then only service method can be invoked.
- It passes request and response objects
- This method cannot be overridden
- The method is shown above: It is responsible for generating of all HTTP methods i.e GET, POST, etc.

## 7) Destroy

```
public void _jspdestroy()
{
    //all clean up code
}
```

- `_jspdestroy()` method is also invoked by the container
- This method is called when container decides it no longer needs the servlet instance to service requests.
- When the call to destroy method is made then, the servlet is ready for a garbage collection
- This is the end of the life cycle.
- We can override `jspdestroy()` method when we perform any cleanup such as releasing database connections or closing open files.
- `<%@ 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>Guru JSP6</title>`
- `</head>`
- `<body>`
- `<% int num1=10; int num2 = 50;`
- `int num3 = num1+num2;`
- `if(num3 != 0 || num3 > 0){`
- `int num4= num1*num2;`
- `out.println("Number 4 is " +num4);`
- `out.println("Number 3 is " +num3);`
- `}%>`
- `</body>`
- `</html>`



## JSP Scriptlet tag (Scripting elements)

In JSP, java code can be written inside the jsp page using the scriptlet tag. Let's see what are the scripting elements first.

### JSP Scripting elements

The scripting elements provides the ability to insert java code inside the jsp. There are three types of scripting elements:

- scriptlet tag
- expression tag
- declaration tag

### JSP scriptlet tag

A scriptlet tag is used to execute java source code in JSP. Syntax is as follows:

1. `<% java source code %>`

#### Example of JSP scriptlet tag

In this example, we are displaying a welcome message.

1. `<html>`
2. `<body>`
3. `<% out.print("welcome to jsp"); %>`
4. `</body>`
5. `</html>`

---

#### Example of JSP scriptlet tag that prints the user name

In this example, we have created two files index.html and welcome.jsp. The index.html file gets the username from the user and the welcome.jsp file prints the username with the welcome message.

*File: index.html*

1. `<html>`

2. `<body>`
3. `<form action="welcome.jsp">`
4. `<input type="text" name="uname">`
5. `<input type="submit" value="go"><br/>`
6. `</form>`
7. `</body>`
8. `</html>`

*File: welcome.jsp*

1. `<html>`
2. `<body>`
3. `<%`
4. `String name=request.getParameter("uname");`
5. `out.print("welcome "+name);`
6. `%>`
7. `</form>`
8. `</body>`
9. `</html>`

### JSP expression tag

The code placed within **JSP expression tag** is *written to the output stream of the response*. So you need not write `out.print()` to write data. It is mainly used to print the values of variable or method.

### Syntax of JSP expression tag

1. `<%= statement %>`

### Example of JSP expression tag

In this example of jsp expression tag, we are simply displaying a welcome message.

1. `<html>`
2. `<body>`
3. `<%= "welcome to jsp" %>`
4. `</body>`
5. `</html>`

### Example of JSP expression tag that prints current time

To display the current time, we have used the `getTime()` method of `Calendar` class. The `getTime()` is an instance method of `Calendar` class, so we have called it after getting the instance of `Calendar` class by the `getInstance()` method.

*index.jsp*

1. `<html>`
2. `<body>`
3. `Current Time: <%= java.util.Calendar.getInstance().getTime() %>`
4. `</body>`
5. `</html>`

### Example of JSP expression tag that prints the user name

In this example, we are printing the username using the expression tag. The index.html file gets the username and sends the request to the welcome.jsp file, which displays the username.

File: index.jsp

1. `<html>`
2. `<body>`
3. `<form action="welcome.jsp">`
4. `<input type="text" name="uname"><br/>`
5. `<input type="submit" value="go">`
6. `</form>`
7. `</body>`
8. `</html>`

File: welcome.jsp

1. `<html>`
2. `<body>`
3. `<%= "Welcome "+request.getParameter("uname") %>`
4. `</body>`
5. `</html>`

### JSP Declaration Tag

The **JSP declaration tag** is used to declare fields and methods.

The code written inside the jsp declaration tag is placed outside the service() method of auto generated servlet.

So it doesn't get memory at each request.

### Syntax of JSP declaration tag

The syntax of the declaration tag is as follows:

1. `<%! field or method declaration %>`

### Difference between JSP Scriptlet tag and Declaration tag

Jsp Scriptlet Tag	Jsp Declaration Tag
The jsp scriptlet tag can only declare variables not methods.	The jsp declaration tag can declare variables as well as methods.
The declaration of scriptlet tag is placed inside the _jspService() method.	The declaration of jsp declaration tag is placed outside the _jspService() method.

### Example of JSP declaration tag that declares field

In this example of JSP declaration tag, we are declaring the field and printing the value of the declared field using the jsp expression tag.

index.jsp

1. `<html>`
2. `<body>`

3. `<%! int data=50; %>`
4. `<%= "Value of the variable is:"+data %>`
5. `</body>`
6. `</html>`

### Example of JSP declaration tag that declares method

In this example of JSP declaration tag, we are defining the method which returns the cube of given number and calling this method from the jsp expression tag. But we can also use jsp scriptlet tag to call the declared method.

#### index.jsp

1. `<html>`
2. `<body>`
3. `<%!`
4. `int cube(int n){`
5. `return n*n*n*;`
6. `}`
7. `%>`
8. `<%= "Cube of 3 is:"+cube(3) %>`
9. `</body>`
10. `</html>`

### JSP Action Tags

JSP action tags are the special tags that can be used to provide instructions to the JSP container on how to manage the server-side actions. It can be enclosed with the `<jsp: .. >` tags and it can allow the developers to perform various tasks such as including other files, forwarding the requests, and manipulating the session attributes.

#### List of the Commonly used JSP Action Tags in JSP

Tag Name	Syntax	Description	Example
<b>Include Tag</b>	<code>&lt;jsp:include page="filename.jsp" /&gt;</code>	It can be used to include the content of the other resources like the JSP, HTML, or servlet in the current JSP page.	<code>&lt;jsp:include page="header.jsp" /&gt;</code>
<b>Forward Tag</b>	<code>&lt;jsp:forward page="destination.jsp" /&gt;</code>	This tag can be used to forward the current request to another resource like the JSP, HTML, or servlet without the client's knowledge.	<code>&lt;jsp:forward page="success.jsp" /&gt;</code>

Tag Name	Syntax	Description	Example
<b>setProperty Tag</b>	<code>&lt;jsp:setProperty name="beanName" property="propertyName" value="propertyValue" /&gt;</code>	This tag can be used to set the properties of the JavaBean component of the JSP pages.	<code>&lt;jsp:setProperty name="user" property="username" value="John" /&gt;</code>
<b>getProperty</b>	<code>&lt;jsp:getProperty name="beanName" property="propertyName" /&gt;</code>	This tag can be used to retrieve the properties of the JavaBean component of the JSP pages.	<code>&lt;jsp:getProperty name="user" property="username" /&gt;</code>
<b>useBean Tag</b>	<code>&lt;jsp:useBean id="beanId" class="packageName.class Name" /&gt;</code>	This tag can be used to instantiate the JavaBean component or retrieve the existing instance of the JSP pages.	<code>&lt;jsp:useBean id="user" class="com.example.Us er" /&gt;</code>
<b>plugin Tag</b>	<code>&lt;jsp:plugin type="pluginType" code="pluginCode" /&gt;</code>	It can be used to generates the HTML code for the browser specific plugin.	<code>&lt;jsp:plugin type="applet" code="MyApplet.class" /&gt;</code>
<b>attribute tag</b>	<code>&lt;jsp:attribute name="attributeName" value="attributeValue" /&gt;</code>	This tag can be used to defines the attributes values for the custom actions of the JSP pages.	<code>&lt;jsp:attribute name="color" value="blue" /&gt;</code>
<b>Body Tag</b>	<code>&lt;jsp:body&gt; &lt;!-- Body Content --&gt; &lt;/jsp:body&gt;</code>	This tag can be used to defines the body content for the custom actions of the JSP pages.	<code>&lt;jsp:body&gt; &lt;p&gt;This is the body content.&lt;/p&gt; &lt;/jsp:body&gt;</code>

### Directive

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	<b>&lt;%@ page ... %&gt;</b> Defines page-dependent attributes, such as scripting language, error page, and buffering requirements.
2	<b>&lt;%@ include ... %&gt;</b> Includes a file during the translation phase.
3	<b>&lt;%@ taglib ... %&gt;</b> 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 –

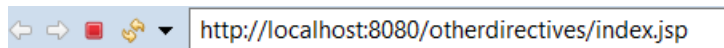
S.No.	Attribute & Purpose
1	<b>buffer</b> Specifies a buffering model for the output stream.
2	<b>autoFlush</b> Controls the behavior of the servlet output buffer.
3	<b>contentType</b> Defines the character encoding scheme.
4	<b>errorPage</b> Defines the URL of another JSP that reports on Java unchecked runtime exceptions.
5	<b>isErrorPage</b> Indicates if this JSP page is a URL specified by another JSP page's errorPage attribute.
6	<b>extends</b> Specifies a superclass that the generated servlet must extend.

7	<b>import</b> Specifies a list of packages or classes for use in the JSP as the Java import statement does for Java classes.
8	<b>info</b> Defines a string that can be accessed with the servlet's <b>getServletInfo()</b> method.
9	<b>isThreadSafe</b> Defines the threading model for the generated servlet.
10	<b>language</b> Defines the programming language used in the JSP page.
11	<b>session</b> Specifies whether or not the JSP page participates in HTTP sessions
12	<b>isELIgnored</b> Specifies whether or not the EL expression within the JSP page will be ignored.
13	<b>isScriptingEnabled</b> Determines if the scripting elements are allowed for use.

Check for more details related to all the above attributes at [Page Directive](#).

<%-- JSP code to demonstrate how to use page directive to import a package --%>

```
<%@page import = "java.util.Date"%>
<%Date d = new Date();%>
<%=d%>
```



Fri Jul 13 17:11:18 IST 2018

### What are JSP Directives?

- JSP directives are the messages to JSP container. They provide global information about an entire JSP page.
- JSP directives are used to give special instruction to a container for translation of JSP to servlet code.
- In JSP life cycle phase, JSP has to be converted to a servlet which is the translation phase.

- They give instructions to the container on how to handle certain aspects of JSP processing
- Directives can have many attributes by comma separated as key-value pairs.
- In JSP, directive is described in `<%@ %>` tags.

### Syntax of Directive:

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

There are three types of directives:

1. Page directive
2. Include directive
3. Taglib directive

Each one of them is described in detail below with examples:

### JSP Page directive

#### Syntax of Page directive:

`<%@ page...%>`

- It provides attributes that get applied to entire JSP page.
- It defines page dependent attributes, such as scripting language, error page, and buffering requirements.
- It is used to provide instructions to a container that pertains to current JSP page.

Following are its list of attributes associated with page directive:

1. Language
2. Extends
3. Import
4. contentType
5. info
6. session
7. isThreadSafe
8. autoflush
9. buffer
10. isErrorPage
11. pageEncoding
12. errorPage
13. isELIgnored

More details about each attribute

**1) language:** It defines the [programming language](#) (underlying language) being used in the page.

#### Syntax of language:

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

Here value is the programming language (underlying language)

**Example:**

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
```

**Explanation of code:** In the above example, attribute language value is [Java](#) which is the underlying language in this case. Hence, the code in expression tags would be compiled using java compiler.

**2) Extends:** This attribute is used to extend (inherit) the class like JAVA does

**Syntax of extends:**

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

Here the value represents class from which it has to be inherited.

**Example:**

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
```

```
<%@ page extends="demotest.DemoClass" %>
```

**Explanation of the code:** In the above code JSP is extending DemoClass which is within demotest package, and it will extend all class features.

**3) Import:** This attribute is most used attribute in page directive attributes. It is used to tell the container to import other java classes, interfaces, enums, etc. while generating servlet code. It is similar to import statements in java classes, interfaces.

**Syntax of import:**

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

Here value indicates the classes which have to be imported.

**Example:**

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    import="java.util.Date" pageEncoding="ISO-8859-1"%>
```

**Explanation of the code:**

In the above code, we are importing Date class from java.util package (all utility classes), and it can use all methods of the following class.

**4) contentType:**

- It defines the character encoding scheme i.e. it is used to set the content type and the character set of the response
- The default type of contentType is “text/html; charset=ISO-8859-1”.

### **Syntax of the contentType:**

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

#### **Example:**

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
```

#### **Explanation of the code:**

In the above code, the content type is set as text/html, it sets character encoding for JSP and for generated response page.

### **5) info**

- It defines a string which can be accessed by getServletInfo() method.
- This attribute is used to set the servlet description.

### **Syntax of info:**

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

Here, the value represents the servlet information.

#### **Example:**

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    info="Guru Directive JSP" pageEncoding="ISO-8859-1"%>
```

#### **Explanation of the code:**

In the above code, string "Guru Directive JSP" can be retrieved by the servlet interface using getServletInfo()

### **6) Session**

- JSP page creates session by default.
- Sometimes we don't need a session to be created in JSP, and hence, we can set this attribute to false in that case. The default value of the session attribute is true, and the session is created. When it is set to false, then we can indicate the compiler to not create the session by default.

### **Syntax of session:**

```
<%@ page session="true/false"%>
```

Here in this case session attribute can be set to true or false

#### **Example:**

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    session="false"%>
```

#### **Explanation of code:**

In the above example, session attribute is set to “false” hence we are indicating that we don’t want to create any session in this JSP

### 7) isThreadSafe:

- It defines the threading model for the generated servlet.
- It indicates the level of thread safety implemented in the page.
- Its default value is true so simultaneous
- We can use this attribute to implement SingleThreadModel interface in generated servlet.
- If we set it to false, then it will implement SingleThreadModel and can access any shared objects and can yield inconsistency.

### Syntax of isThreadSafe:

```
<% @ page isThreadSafe="true/false" %>
```

Here true or false represents if synchronization is there then set as true and set it as false.

### Example:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    isThreadSafe="true"%>
```

### Explanation of the code:

In the above code, isThreadSafe is set to “true” hence synchronization will be done, and multiple threads can be used.

### 8) AutoFlush:

This attribute specifies that the buffered output should be flushed automatically or not and default value of that attribute is true.

If the value is set to false the buffer will not be flushed automatically and if its full, we will get an exception.

When the buffer is none then the false is illegitimate, and there is no buffering, so it will be flushed automatically.

### Syntax of autoFlush:

```
<% @ page autoFlush="true/false" %>
```

Here true/false represents whether buffering has to be done or not

### Example:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    autoFlush="false"%>
```

### Explanation of the code:

In the above code, the autoflush is set to false and hence buffering won't be done and it has manually flush the output.

### 9) Buffer:

- Using this attribute the output response object may be buffered.
- We can define the size of buffering to be done using this attribute and default size is 8KB.
- It directs the servlet to write the buffer before writing to the response object.

#### Syntax of buffer:

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

Here the value represents the size of the buffer which has to be defined. If there is no buffer, then we can write as none, and if we don't mention any value then the default is 8KB

#### Example:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    buffer="16KB"%>
```

#### Explanation of the code:

In the above code, buffer size is mentioned as 16KB wherein the buffer would be of that size

### 10) isErrorPage:

- It indicates that JSP Page that has an errorPage will be checked in another JSP page
- Any JSP file declared with "isErrorPage" attribute is then capable to receive exceptions from other JSP pages which have error pages.
- Exceptions are available to these pages only.
- The default value is false.

#### Syntax of isErrorPage:

```
<%@ page isErrorPage="true/false"%>
```

#### Example:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    isErrorPage="true"%>
```

#### Explanation of the code:

In the above code, isErrorPage is set as true. Hence, it will check any other JSPs has errorPage (described in the next attribute) attribute set and it can handle exceptions.

### 11) PageEncoding:

The "pageEncoding" attribute defines the character encoding for JSP page.

The default is specified as "ISO-8859-1" if any other is not specified.

### Syntax of pageEncoding:

```
<%@ page pageEncoding="vaue" %>
```

Here value specifies the charset value for JSP

### Example:

```
<%@ page language="java" contentType="text/html;" pageEncoding="ISO-8859-1"
  isErrorPage="true"%>
```

### Explanation of the code:

In the above code “pageEncoding” has been set to default charset ISO-8859-1

### 12) errorPage:

This attribute is used to set the error page for the JSP page if JSP throws an exception and then it redirects to the exception page.

### Syntax of errorPage:

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

Here value represents the error JSP page value

### Example:

```
<%@ page language="java" contentType="text/html;" pageEncoding="ISO-8859-1"
  errorPage="errorHandler.jsp"%>
```

### Explanation of the code:

In the above code, to handle exceptions we have erroHandler.jsp

### 13) isELIgnored:

- IsELIgnored is a flag attribute where we have to decide whether to ignore EL tags or not.
- Its datatype is java enum, and the default value is false hence EL is enabled by default.

### Syntax of isELIgnored:

```
<%@ page isELIgnored="true/false" %>
```

Here, true/false represents the value of EL whether it should be ignored or not.

### Example:

```
<%@ page language="java" contentType="text/html;" pageEncoding="ISO-8859-1"
  isELIgnored="true"%>
```

### Explanation of the code:

In the above code, isELIgnored is true and hence [Expression Language \(EL\)](#) is ignored here.



In the below example we are using four attributes(code line 1-2)

### Example with four attributes

```
<%@ page language="java" contentType="text/html;" pageEncoding="ISO-8859-1"
  isELIgnored="false"%>
<%@page import="java.util.Date" %>

<!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>Directive Guru JSP1</title>
</head>
<body>
<a>Date is:</a>
<%= new java.util.Date() %>
</body>
</html>
```

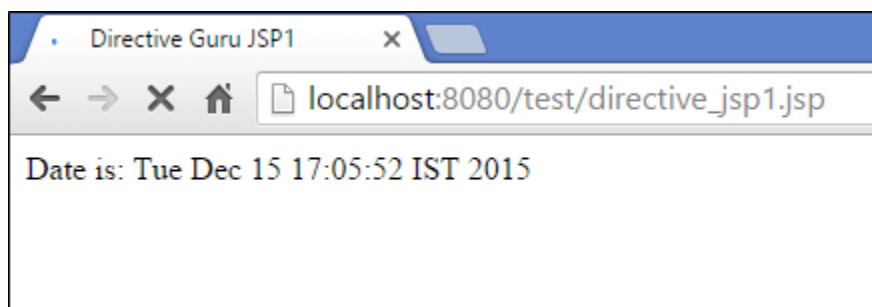
### Explanation of the code:

**Code Line 1-2:** Here we have defined four attributes i.e.

- Language: It is set as [Java](#) as programming language
- contentType: set as text/html to tell the compiler that html has to be format
- pageEncoding: default charset is set in this attribute
- isELIgnored: Expression Tag is false hence it is not ignored

**Code Line 3:** Here we have used import attribute, and it is importing “Date class” which is from Java util package, and we are trying to display current date in the code.

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



### Output:

- Date is: Current date using the date method of the date class

### JSP Include directive

- JSP “include directive”( codeline 8 ) is used to include one file to the another file

- This included file can be HTML, JSP, text files, etc.
- It is also useful in creating templates with the user views and break the pages into header&footer and sidebar actions.
- It includes file during translation phase

### Syntax of include directive:

```
<%@ include...%>
```

#### Example:

Directive\_jsp2.jsp (Main file)

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
    <%@ include file="directive_header_jsp3.jsp" %>
<!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 Directive JSP2</title>
</head>
<body>
<a>This is the main file</a>
</body>
</html>
```

Directive\_header\_jsp3.jsp (which is included in the main file)

```
<%@ 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">

</head>
<body>
<a>Header file : </a>
<%int count =1; count++;
out.println(count);%> :
</body>
</html>
```

#### Explanation of the code:

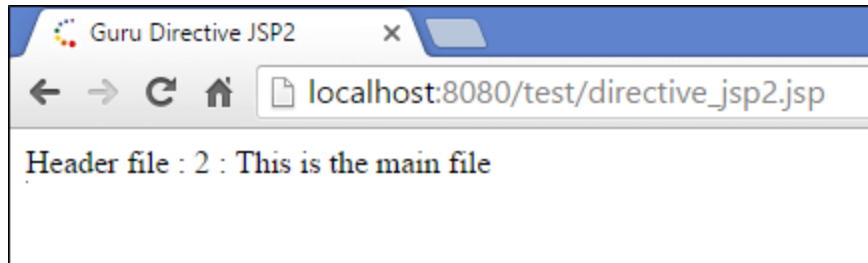
#### Directive\_jsp2.jsp:

**Code Line 3:** In this code, we use include tags where we are including the file directive\_header\_jsp3.jsp into the main file(jsp2.jsp)and gets the output of both main file and included file.

## Directive\_header\_jsp3.jsp:

**Code Line 11-12:** We have taken a variable count initialized to 1 and then incremented it. This will give the output in the main file as shown below.

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



## Output:

- The output is Header file: 2 : This is the main file
- The output is executed from the directive\_jsp2.jsp file while the directive\_header\_jsp3.jsp included file will be compiled first.
- After the included file is done, the main file is executed, and the output will be from the main file "This is the main file". So you will get the output as "Header file: 2" from \_jsp3.jsp and "This is main file" from \_jsp2.jsp.

## JSP Taglib Directive

- JSP taglib directive is used to define the tag library with "taglib" as the prefix, which we can use in [JSP](#).
- More detail will be covered in JSP Custom Tags section
- JSP taglib directive is used in the JSP pages using the JSP standard tag libraries
- It uses a set of custom tags, identifies the location of the library and provides means of identifying custom tags in JSP page.

## Syntax of taglib directive:

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

Here "uri" attribute is a unique identifier in tag library descriptor and "prefix" attribute is a tag name.

## Example:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
    <%@ taglib prefix="gurutag" uri="http://java.sun.com/jsp/jstl/core" %>
<!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 Directive JSP</title>
```

```
<gurutag:hello/>
</head>
<body>
</body>
</html>
```

### Explanation of the code:

**Code Line 3:** Here “taglib” is defined with attributes uri and prefix.

**Code Line 9:** “gurutag” is the custom tag defined and it can be used anywhere

### Example of JSP Custom Tag

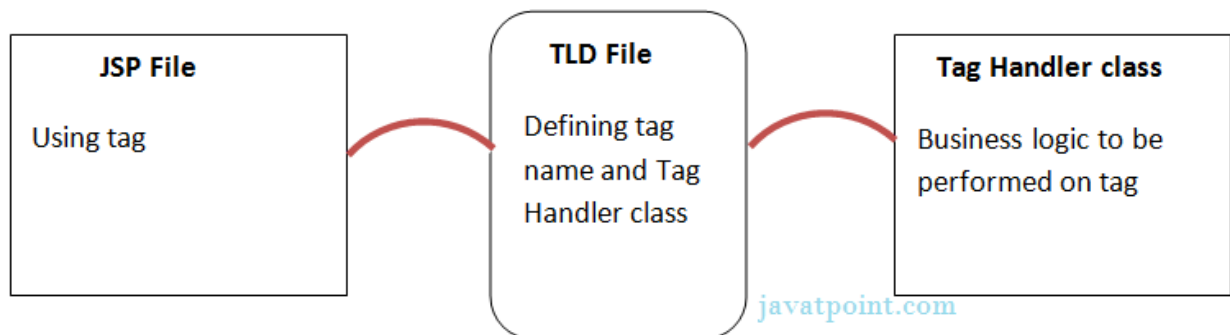
In this example, we are going to create a **custom tag that prints the current date and time**. We are performing action at the start of tag.

For creating any custom tag, we need to follow following steps:

1. **Create the Tag handler class** and perform action at the start or at the end of the tag.
2. **Create the Tag Library Descriptor (TLD) file** and define tags
3. **Create the JSP file that uses the Custom tag defined in the TLD file**

---

### Understanding flow of custom tag in jsp



#### 1) Create the Tag handler class

To create the Tag Handler, we are inheriting the **TagSupport** class and overriding its method **doStartTag()**. To write data for the jsp, we need to use the **JspWriter** class.

The **PageContext** class provides **getOut()** method that returns the instance of JspWriter class. TagSupport class provides instance of pageContext by default.

*File: MyTagHandler.java*

```

package com.javatpoint.sonoo;
import java.util.Calendar;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.tagext.TagSupport;
public class MyTagHandler extends TagSupport {

public int doStartTag() throws JspException {
    JspWriter out=pageContext.getOut();//returns the instance of JspWriter
    try {
        out.print(Calendar.getInstance().getTime());//printing date and time using JspWriter
    }catch(Exception e){System.out.println(e);}
    return SKIP_BODY;//will not evaluate the body content of the tag
}
}

```

## 2) Create the TLD file

**Tag Library Descriptor (TLD)** file contains information of tag and Tag Handler classes. It must be contained inside the **WEB-INF** directory.

*File: mytags.tld*

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
    "http://java.sun.com/j2ee/dtd/web-jsptaglibrary_1_2.dtd">

```

```
<taglib>
```

```

<tlib-version>1.0</tlib-version>
<jsp-version>1.2</jsp-version>
<short-name>simple</short-name>
<uri>http://tomcat.apache.org/example-taglib</uri>

```

```
<tag>
```

```

<name>today</name>
<tag-class>com.javatpoint.sonoo.MyTagHandler</tag-class>
</tag>
</taglib>

```

## 3) Create the JSP file

Let's use the tag in our jsp file. Here, we are specifying the path of tld file directly. But it is recommended to use the uri name instead of full path of tld file. We will learn about uri later.

It uses **taglib** directive to use the tags defined in the tld file.

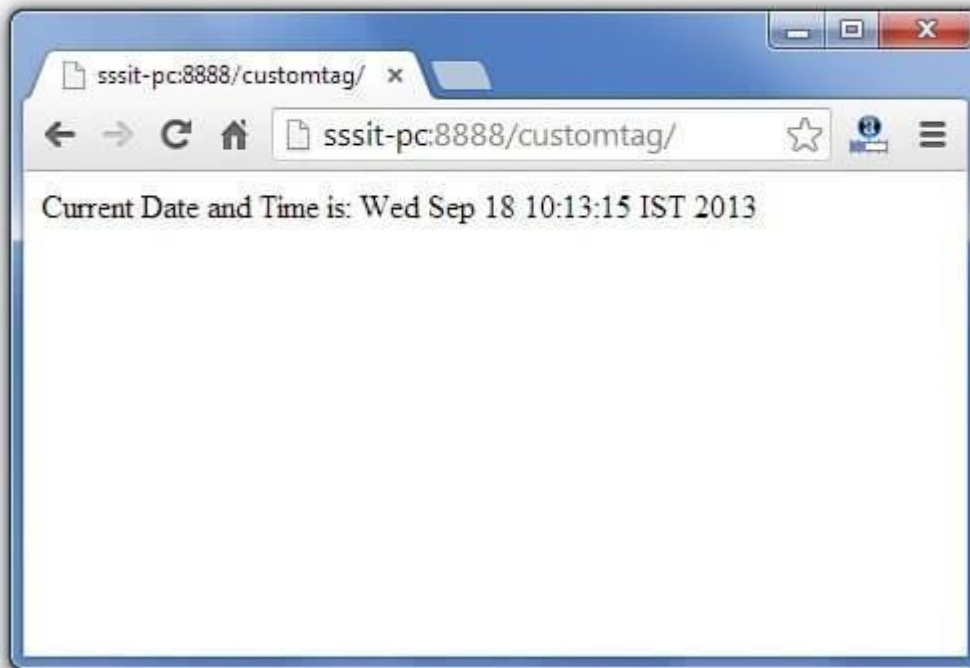
*File: index.jsp*

1. <%@ taglib uri="WEB-INF/mytags.tld" prefix="m" %>
2. Current Date and Time is: <m:today/>

[download this example](#)

---

*Output*



## UNIT-V

### RECENT JAVA TOOLS

9

**Spring Boot** - Deploying a Spring-Boot application running with Java8 - Hibernate: Introduction to Hibernate 3.0 - Hibernate Architecture - First Hibernate Application. Java Server Faces - Installing application - writing - deploying and testing application - Request Process life cycle - Basic JSF Tags - Expression Language.

#### Spring Boot

Spring Boot is a Java framework that makes it easier to create and run Java applications. It simplifies the configuration and setup process, allowing developers to focus more on writing code for their applications.

Spring Boot, a module of the Spring framework, facilitates **Rapid Application Development** (RAD) capabilities.

This **Spring tutorial** includes basic to advanced topics of Spring Boot, like Basics of Spring Boot, Spring Boot core, Spring Boot REST API, Spring Boot with Microservices, Spring Boot with Kafka, Spring Boot with Database and Data JPA, etc.

#### **What is Spring Boot?**

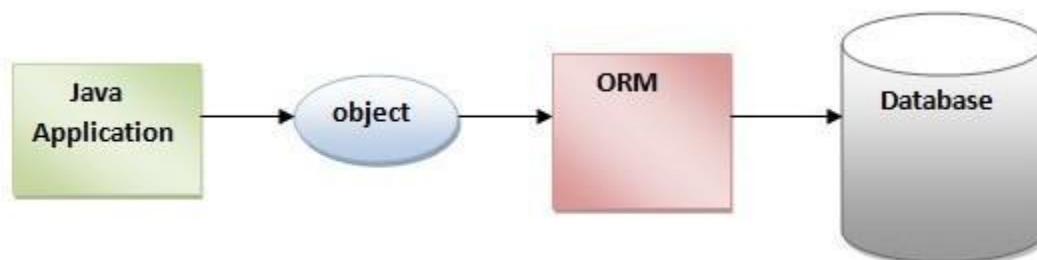
**Spring Boot** is an open-source Java framework used to create a Micro Service. Spring boot is developed by Pivotal Team, and it provides a faster way to set up and an easier, configure, and run both simple and web-based applications. It is a combination of Spring Framework and Embedded Servers. The main goal of Spring Boot is to reduce development, unit test, and integration test time and in Spring Boot, there is no requirement for XML configuration.

#### Hibernate Framework

Hibernate is a Java framework that simplifies the development of Java application to interact with the database. It is an open source, lightweight, ORM (Object Relational Mapping) tool. Hibernate implements the specifications of JPA (Java Persistence API) for data persistence.

#### ORM Tool

An ORM tool simplifies the data creation, data manipulation and data access. It is a programming technique that maps the object to the data stored in the database.



The ORM tool internally uses the JDBC API to interact with the database.

#### **What is JPA?**

Java Persistence API (JPA) is a Java specification that provides certain functionality and standard to ORM tools. The javax.persistence package contains the JPA classes and interfaces.

## **Advantages of Hibernate Framework**

Following are the advantages of hibernate framework:

### **1) Open Source and Lightweight**

Hibernate framework is open source under the LGPL license and lightweight.

### **2) Fast Performance**

The performance of hibernate framework is fast because cache is internally used in hibernate framework. There are two types of cache in hibernate framework first level cache and second level cache. First level cache is enabled by default.

### **3) Database Independent Query**

HQL (Hibernate Query Language) is the object-oriented version of SQL. It generates the database independent queries. So you don't need to write database specific queries. Before Hibernate, if database is changed for the project, we need to change the SQL query as well that leads to the maintenance problem.

### **4) Automatic Table Creation**

Hibernate framework provides the facility to create the tables of the database automatically. So there is no need to create tables in the database manually.

### **5) Simplifies Complex Join**

Fetching data from multiple tables is easy in hibernate framework.

### **6) Provides Query Statistics and Database Status**

Hibernate supports Query cache and provide statistics about query and database status.

## **Introduction to Hibernate ORM Framework**

What is ORM?

ORM (Object-relational mapping) is a programming technique for mapping application domain model objects to relational database tables. Hibernate is a Java-based ORM tool that provides a framework for mapping application domain objects to relational database tables and vice versa.

What is the Java Persistence API (JPA)?

The Java Persistence API (JPA) is a Java specification for accessing, persisting, and managing data between Java objects/classes and a relational database. JPA acts as a bridge between object-oriented domain models and relational database systems, making it easier for developers to work with data in their applications.

JPA allows developers to map Java objects to database tables and vice versa using annotations or XML configuration files. This abstracts the complexities in converting data between its object-oriented form in the application and its relational form in the database.



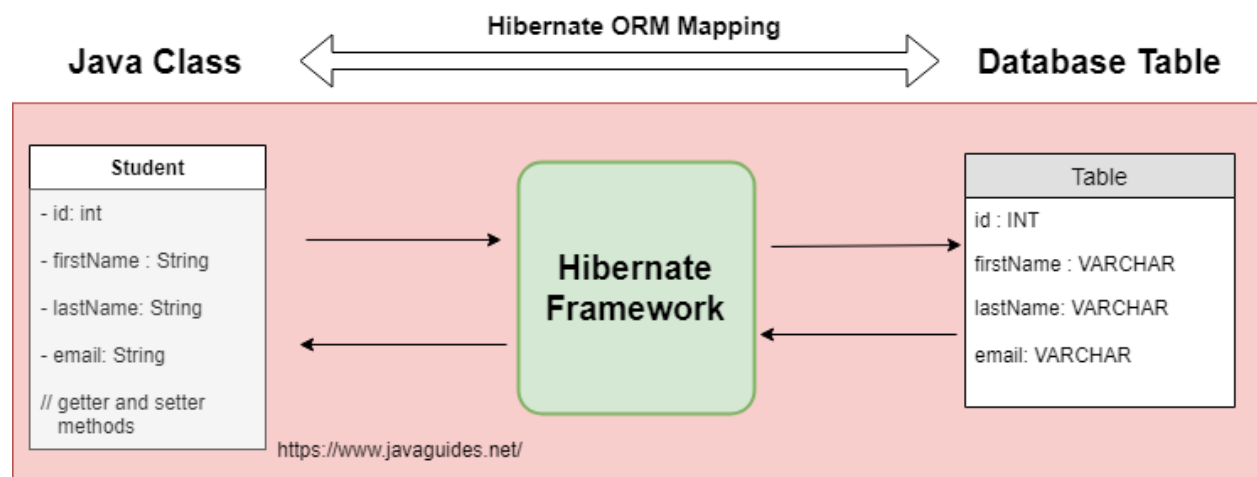
JPA is not an implementation but a specification. Various ORM tools, such as Hibernate, EclipseLink, and Apache OpenJPA, provide implementations of the JPA specification. This allows developers to switch between these implementations if needed without changing the application code that uses JPA.

### What is the Hibernate Framework?

Hibernate is a Java-based ORM tool that provides a framework for mapping application domain objects to relational database tables and vice versa.

Hibernate is the most popular JPA implementation and one of the most popular Java ORM frameworks. Hibernate is an additional layer on top of JDBC and enables you to implement a database-independent persistence layer. It provides an object-relational mapping implementation that maps your database records to Java objects and generates the required SQL statements to replicate all operations to the database.

Example: The diagram below shows an Object-Relational Mapping between the Student Java class and the student's table in the database.



### Key Features of Hibernate

**Transparent Persistence:** Hibernate manages the persistence of objects without requiring significant changes to how those objects are designed.

**Database Independence:** Applications built with Hibernate are portable across databases with minimal changes.

**Performance Optimization:** Features like caching and lazy loading help optimize performance by reducing database access.

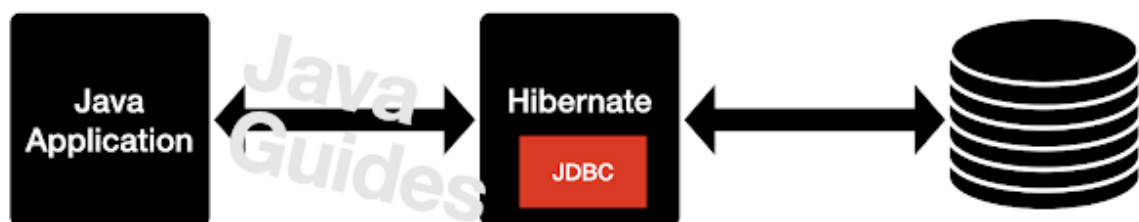
**Powerful Query Language:** Hibernate Query Language (HQL) offers an object-oriented extension to SQL, easing data manipulation and retrieval.

Automatic Schema Generation: Hibernate can generate database schemas based on the object model, simplifying initial setup and migrations

### How does Hibernate relate to JDBC?

Hibernate internally uses JDBC for all database communications.

Hibernate acts as an additional layer on top of JDBC and enables you to implement a database-independent persistence layer:



### Hibernate Architecture

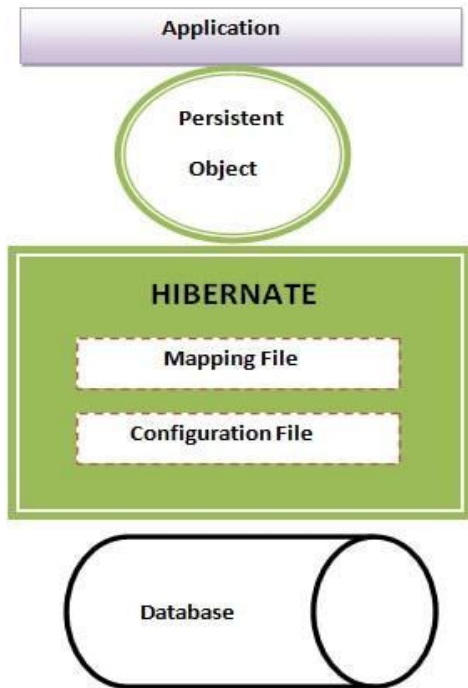
Hibernate architecture consists of several layers, including the Java application layer, Hibernate framework, backhand API, and the database layer. Let's break down the core components:

The Hibernate architecture includes many objects such as persistent object, session factory, transaction factory, connection factory, session, transaction etc.

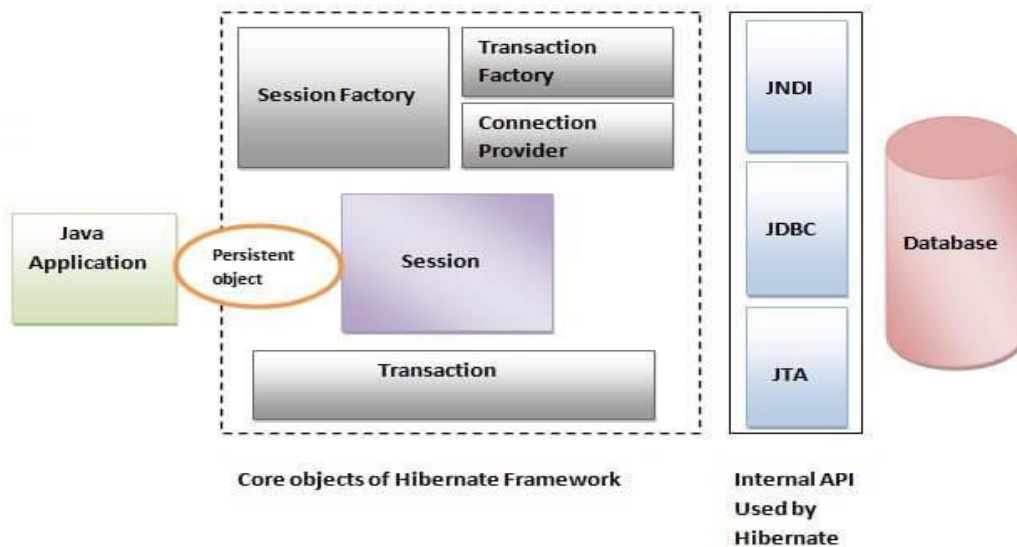
The Hibernate architecture is categorized in four layers.

- Java application layer
- Hibernate framework layer
- Backhand api layer
- Database layer

Let's see the diagram of hibernate architecture:



This is the high level architecture of Hibernate with mapping file and configuration file.



### Elements of Hibernate Architecture

For creating the first hibernate application, we must know the elements of Hibernate architecture. They are as follows:

#### SessionFactory

A thread-safe, immutable cache of compiled mappings for a single database. SessionFactory is a heavyweight object, usually created during application initialization and kept for later use.

### **Session**

A single-threaded, short-lived object representing a conversation between the application and the database. It acts as a factory for Transaction instances and holds a first-level cache of retrieved data.

### **Transaction**

A unit of work with the database represents an abstraction of the application from the underlying transaction implementation (JTA or JDBC).

### **ConnectionProvider**

Manages the database connections needed by Hibernate sessions. It abstracts the application from underlying connection management mechanisms.

### **TransactionFactory**

Creates Transaction instances, hiding the underlying transaction implementation details from the application.

### **What are the important benefits of using the Hibernate Framework?**

**Code Efficiency:** Hibernate significantly reduces boilerplate code associated with JDBC, allowing developers to concentrate on business logic and speeding up development time.

**Flexibility in Code:** By supporting both XML configurations and JPA annotations, Hibernate ensures code independence from the implementation, enhancing portability across different database systems.

**Advanced Query Capabilities:** HQL (Hibernate Query Language) offers an object-oriented alternative to SQL, seamlessly integrating with Java's object-oriented features like inheritance, polymorphism, and associations.

**Community and Documentation:** As an open-source project backed by the Red Hat Community, Hibernate benefits from widespread use, a shallow learning curve, extensive documentation, and robust community support.

**Integration with Java EE Frameworks:** Hibernate's popularity and support make it easily integrated with other Java EE frameworks, notably Spring, which offers built-in Hibernate integration for seamless development.

**Performance Optimization:** Features like lazy loading, where database operations are deferred until necessary, and caching mechanisms significantly improve application performance.

Vendor-Specific Features: Hibernate allows for native SQL queries, providing flexibility to utilize database-specific optimizations and features when needed.

Comprehensive ORM Tool: With its extensive feature set addressing nearly all ORM tool requirements, Hibernate stands out as a leading choice in the market for object-relational mapping solutions.

### **What are the advantages of Hibernate over JDBC?**

Simplified Code: Hibernate significantly reduces boilerplate code required in JDBC, making the codebase cleaner and more readable.

Advanced Mapping Features: Unlike JDBC, Hibernate fully supports object-oriented features such as inheritance, associations, and collections.

Transaction Management: Hibernate seamlessly handles transaction management, requiring transactions for most operations, which contrasts with JDBC's manual transaction handling through commit and rollback.

Exception Handling: Hibernate abstracts boilerplate try-catch blocks by converting JDBC's checked SQLExceptions into unchecked JDBCException or HibernateException, simplifying error handling.

Object-Oriented Query Language: HQL (Hibernate Query Language) offers an object-oriented API, which aligns it more with Java programming concepts than JDBC's need for native SQL queries.

Caching for Performance: Hibernate's support for caching enhances performance, a feature not available with JDBC, where queries are directly executed without caching.

Database Synchronization: Hibernate can automatically generate database tables, offering greater flexibility than JDBC, which requires pre-existing tables.

Flexible Connection Management: Hibernate allows for both JDBC-like connections and JNDI DataSource connections with pooling, which is essential for enterprise applications and not supported by JDBC.

ORM Tool Independence: By supporting JPA annotations, Hibernate-based applications are not tightly bound to Hibernate and can switch ORM tools more easily than JDBC-based applications, which are closely coupled with the database.

### **Hibernate Example using XML in Eclipse**

Here, we are going to create a simple example of hibernate application using eclipse IDE. For creating the first hibernate application in Eclipse IDE, we need to follow the following steps:

- Create the java project
- Add jar files for hibernate
- Create the Persistent class
- Create the mapping file for Persistent class
- Create the Configuration file

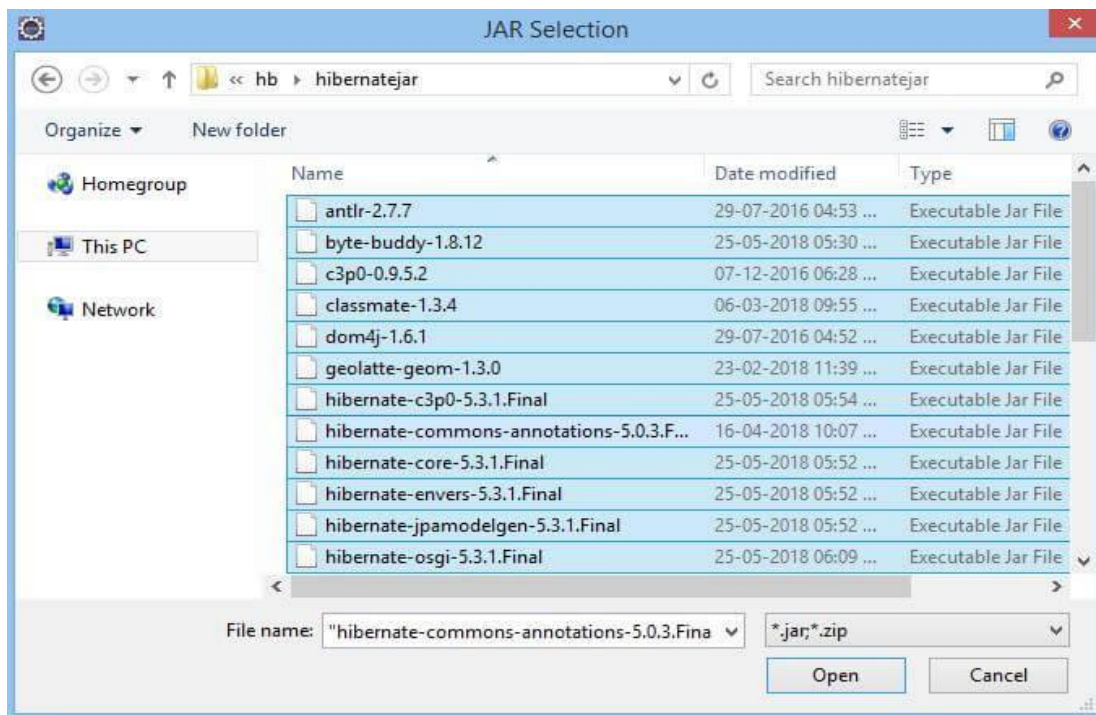
- Create the class that retrieves or stores the persistent object
- Run the application

### 1) Create the java project

Create the java project by File Menu - New - project - java project . Now specify the project name e.g. firsthb then next - finish .

### 2) Add jar files for hibernate

To add the jar files Right click on your project - Build path - Add external archives. Now select all the jar files as shown in the image given below then click open.



In this example, we are connecting the application with oracle database. So you must add the ojdbc14.jar file.

### 3) Create the Persistent class

Here, we are creating the same persistent class which we have created in the previous topic. To create the persistent class, Right click on src - New - Class - specify the class with package name (e.g. com.javatpoint.mypackage) - finish .

Employee.java

```
package com.javatpoint.mypackage;
```

```
public class Employee {
```

```
private int id;
```

```
private String firstName,lastName;
```

```
public int getId() {
```

```

    return id;
}
public void setId(int id) {
    this.id = id;
}
public String getFirstName() {
    return firstName;
}
public void setFirstName(String firstName) {
    this.firstName = firstName;
}
public String getLastName() {
    return lastName;
}
public void setLastName(String lastName) {
    this.lastName = lastName;
} }

```

#### 4) Create the mapping file for Persistent class

Here, we are creating the same mapping file as created in the previous topic. To create the mapping file, Right click on src - new - file - specify the file name (e.g. employee.hbm.xml) - ok. It must be outside the package.

##### employee.hbm.xml

```

<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 5.3//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-5.3.dtd">
<hibernate-mapping>
<class name="com.javatpoint.mypackage.Employee" table="emp1000">
<id name="id">
<generator class="assigned"></generator>
</id>

```

```
<property name="firstName"></property>
<property name="lastName"></property>
</class>
</hibernate-mapping>
```

## 5) Create the Configuration file

The configuration file contains all the informations for the database such as connection\_url, driver\_class, username, password etc. The hbm2ddl.auto property is used to create the table in the database automatically. We will have in-depth learning about Dialect class in next topics. To create the configuration file, right click on src - new - file. Now specify the configuration file name e.g. hibernate.cfg.xml.

### hibernate.cfg.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 5.3//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-5.3.dtd">

<hibernate-configuration>
    <session-factory>
        <property name="hbm2ddl.auto">update</property>
        <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
        <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
        <property name="connection.username">system</property>
        <property name="connection.password">oracle</property>
        <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
        <mapping resource="employee.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

## 6) Create the class that retrieves or stores the persistent object

In this class, we are simply storing the employee object to the database.

```
package com.javatpoint.mypackage;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
```



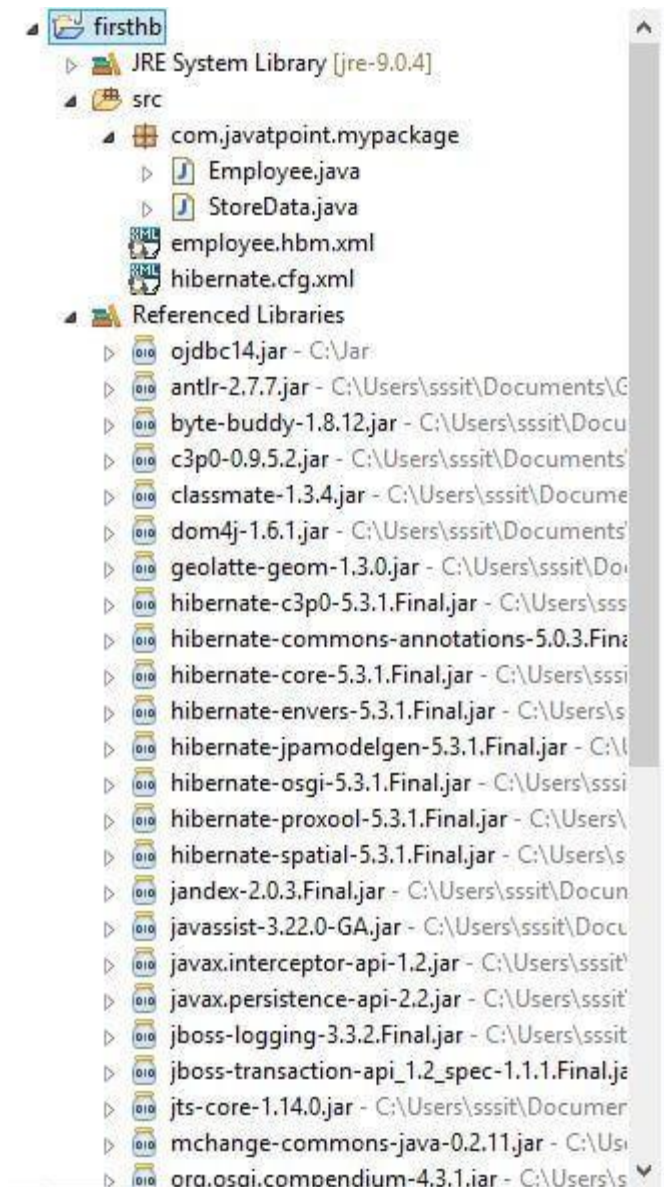
```

import org.hibernate.Transaction;
import org.hibernate.boot.Metadata;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
public class StoreData {
    public static void main( String[] args )
    {
        StandardServiceRegistry ssr = new
StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
        Metadata meta = new MetadataSources(ssr).getMetadataBuilder().build();
        SessionFactory factory = meta.getSessionFactoryBuilder().build();
        Session session = factory.openSession();
        Transaction t = session.beginTransaction();
        Employee e1=new Employee();
        e1.setId(1);
        e1.setFirstName("Gaurav");
        e1.setLastName("Chawla");
        session.save(e1);
        t.commit();
        System.out.println("successfully saved");
        factory.close();
        session.close();
    }
}

```

## 7) Run the application

Before running the application, determine that directory structure is like this.



To run the hibernate application, right click on the Store Data class - Run As - Java Application.

## JAVASERVER FACES

It is a server side component based user interface framework. It is used to develop web applications. It provides a well-defined programming model and consists of rich API and tag libraries. The latest version JSF 2 uses Facelets as its default templating system. It is written in Java.

The JSF API provides components (inputText, commandButton etc) and helps to manage their states. It also provides server-side validation, data conversion, defining page navigation, provides extensibility, supports for internationalization, accessibility etc.

The JSF Tag libraries are used to add components on the web pages and connect components with objects on the server. It also contains tag handlers that implements the component tag.

With the help of these features and tools, you can easily and effortlessly create server-side user interface.

## JSF Features

Latest version of JSF 2.2 provides the following features.

### Component Based Framework

Implements Facelets Technology

Integration with Expression Language

Support HTML5

Ease and Rapid web Development.

Support Internationalization

Bean Annotations

Default Exception Handling

Templating

Inbuilt AJAX Support

Security

### **Component Based Framework**

JSF is a server side component-based framework. It provides inbuilt components to build web application. You can use HTML5, Facelets tags to create web pages.

### **Facelets Technology**

Facelets is an open source Web template system. It is a default view handler technology for JavaServer Faces (JSF). The language requires valid input XML documents to work. Facelets supports all of the JSF UI components and focuses completely on building the view for a JSF application.

### **Expression Language**

Expression Language provides an important mechanism for creating the user interface (web pages) to communicate with the application logic (managed beans). The EL represents a union of the expression languages offered by JavaServer Faces technology.

### **HTML 5**

HTML5 is the new standard for writing web pages. JavaServer Faces version 2.2 offers an easy way for including new attributes of HTML 5 to JSF components and provides HTML5 friendly markup.

## **Ease and Rapid web Development.**

JSF provides rich set of inbuilt tools and libraries so that you can easily and rapidly develop we application.

## **Support Internationalization**

JSF supports internationalization for creating World Class web application. You can create applications in the different-different languages. With the help of JSF you can make the application adaptable to various languages and regions.

## **Bean Annotations**

JSF provides annotations facility in which you can perform validation related tasks in Managed Bean. It is good because you can validate your data in bean rather than in HTML validation.

## **Exception Handling**

JSF provide default Exception handling so you can develop exception and bug free web application.

## **Templating**

Introducing template in new version of JSF provides reusability of components. In JSF application, you can create new template, reuse template and treat it as component for application.

## **AJAX Support**

JSF provides inbuilt AJAX support. So, you can render application request to server side without refreshing the web page. JSF also support partial rendering by using AJAX.

## **Security**

JSF provides implicit protection against this when state is saved on the server and no stateless views are used, since a post-back must then contain a valid javax.faces.ViewState hidden parameter. Contrary to earlier versions, this value seems sufficiently random in modern JSF implementations. Note that stateless views and saving state on the client does not have this implicit protection.

## **JavaServer Faces Lifecycle**

JavaServer Faces application framework manages lifecycle phases automatically for simple applications and also allows you to manage that manually. The lifecycle of a JavaServer Faces application begins when the client makes an HTTP request for a page and ends when the server responds with the page.

The JSF lifecycle is divided into two main phases:

Execute Phase

Render Phase

1) Execute Phase

In execute phase, when first request is made, application view is built or restored. For other subsequent requests other actions are performed like request parameter values are applied, conversions and validations are performed for component values, managed beans are updated with component values and application logic is invoked.

The execute phase is further divided into following subphases.

Restore View Phase

Apply Request Values Phase

Process Validations Phase

Update Model Values Phase

Invoke Application Phase

Render Response Phase

### **Restore View Phase**

When a client requests for a JavaServer Faces page, the JavaServer Faces implementation begins the restore view phase. In this phase, JSF builds the view of the requested page, wires event handlers and validators to components in the view and saves the view in the FacesContext instance.

If the request for the page is a postback, a view corresponding to this page already exists in the FacesContext instance. During this phase, the JavaServer Faces implementation restores the view by using the state information saved on the client or the server.

### **Apply Request Values Phase**

In this phase, component tree is restored during a postback request. Component tree is a collection of form elements. Each component in the tree extracts its new value from the request parameters by using its decode (processDecodes()) method. After that value is stored locally on each component.

If any decode methods or event listeners have called the renderResponse method on the current FacesContext instance, the JavaServer Faces implementation skips to the Render Response phase.

If any events have been queued during this phase, the JavaServer Faces implementation broadcasts the events to interested listeners.

If the application needs to redirect to a different web application resource or generate a response that does not contain any JavaServer Faces components, it can call the `FacesContext.responseComplete()` method.

If the current request is identified as a partial request, the partial context is retrieved from the `FacesContext`, and the partial processing method is applied.

### **Process Validations Phase**

In this phase, the JavaServer Faces processes all validators registered on the components by using its `validate()` method. It examines the component attributes that specify the rules for the validation and compares these rules to the local value stored for the component. The JavaServer Faces also completes conversions for input components that do not have the `immediate` attribute set to `true`.

If any `validate` methods or event listeners have called the `renderResponse` method on the current `FacesContext`, the JavaServer Faces implementation skips to the Render Response phase.

If the application needs to redirect to a different web application resource or generate a response that does not contain any JavaServer Faces components, it can call the `FacesContext.responseComplete` method.

If events have been queued during this phase, the JavaServer Faces implementation broadcasts them to interested listeners.

If the current request is identified as a partial request, the partial context is retrieved from the `FacesContext`, and the partial processing method is applied.

### **Update Model Values Phase**

After ensuring that the data is valid, it traverses the component tree and sets the corresponding server-side object properties to the components' local values. The JavaServer Faces implementation updates only the bean properties pointed at by an input component's value attribute. If the local data cannot be converted to the types specified by the bean properties, the lifecycle advances directly to the Render Response phase so that the page is re-rendered with errors displayed.

If any `updateModels` methods or any listeners have called the `renderResponse()` method on the current `FacesContext` instance, the JavaServer Faces implementation skips to the Render Response phase.

If the application needs to redirect to a different web application resource or generate a response that does not contain any JavaServer Faces components, it can call the `FacesContext.responseComplete()` method.

If any events have been queued during this phase, the JavaServer Faces implementation broadcasts them to interested listeners.

If the current request is identified as a partial request, the partial context is retrieved from the `FacesContext`, and the partial processing method is applied.

## **Invoke Application Phase**

In this phase, JSF handles application-level events, such as submitting a form or linking to another page.

Now, if the application needs to redirect to a different web application resource or generate a response that does not contain any JSF components, it can call the `FacesContext.responseComplete()` method.

After that, the JavaServer Faces implementation transfers control to the Render Response phase.

## **Render Response Phase**

This is last phase of JSF life cycle. In this phase, JSF builds the view and delegates authority to the appropriate resource for rendering the pages.

If this is an initial request, the components that are represented on the page will be added to the component tree.

If this is not an initial request, the components are already added to the tree and need not to be added again.

If the request is a postback and errors were encountered during the Apply Request Values phase, Process Validations phase, or Update Model Values phase, the original page is rendered again during this phase.

If the pages contain `h:message` or `h:messages` tags, any queued error messages are displayed on the page.

After rendering the content of the view, the state of the response is saved so that subsequent requests can access it. The saved state is available to the Restore View phase.

### **2) Render**

In this phase, the requested view is rendered as a response to the client browser. View rendering is a process in which output is generated as HTML or XHTML. So, user can see it at the browser.

The following steps are taken during the render process.

Application is compiled, when a client makes an initial request for the `index.xhtml` web page.

Application executes after compilation and a new component tree is constructed for the application and placed in a `FacesContext`.

The component tree is populated with the component and the managed bean property associated with it, represented by the EL expression.

Based on the component tree. A new view is built.

The view is rendered to the requesting client as a response.

The component tree is destroyed automatically.

On subsequent requests, the component tree is rebuilt, and the saved state is applied.

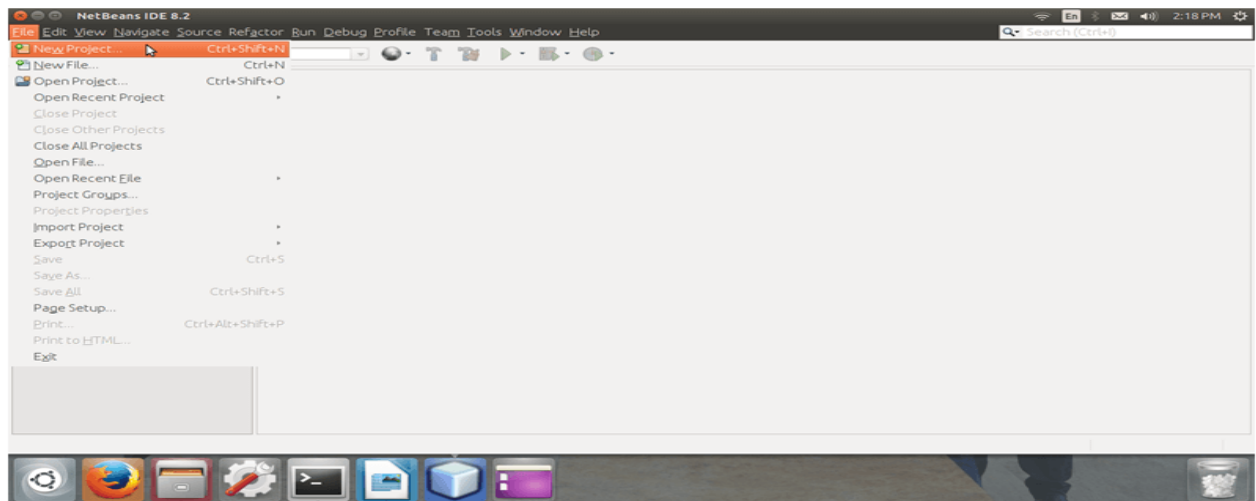
## A Simple JavaServer Faces Application

To create a JSF application, we are using NetBeans IDE 8.2. You can also refer to other Java IDEs.

Here, we are creating a project after that we will run to test it's configuration settings. So, let's create a new project fist.

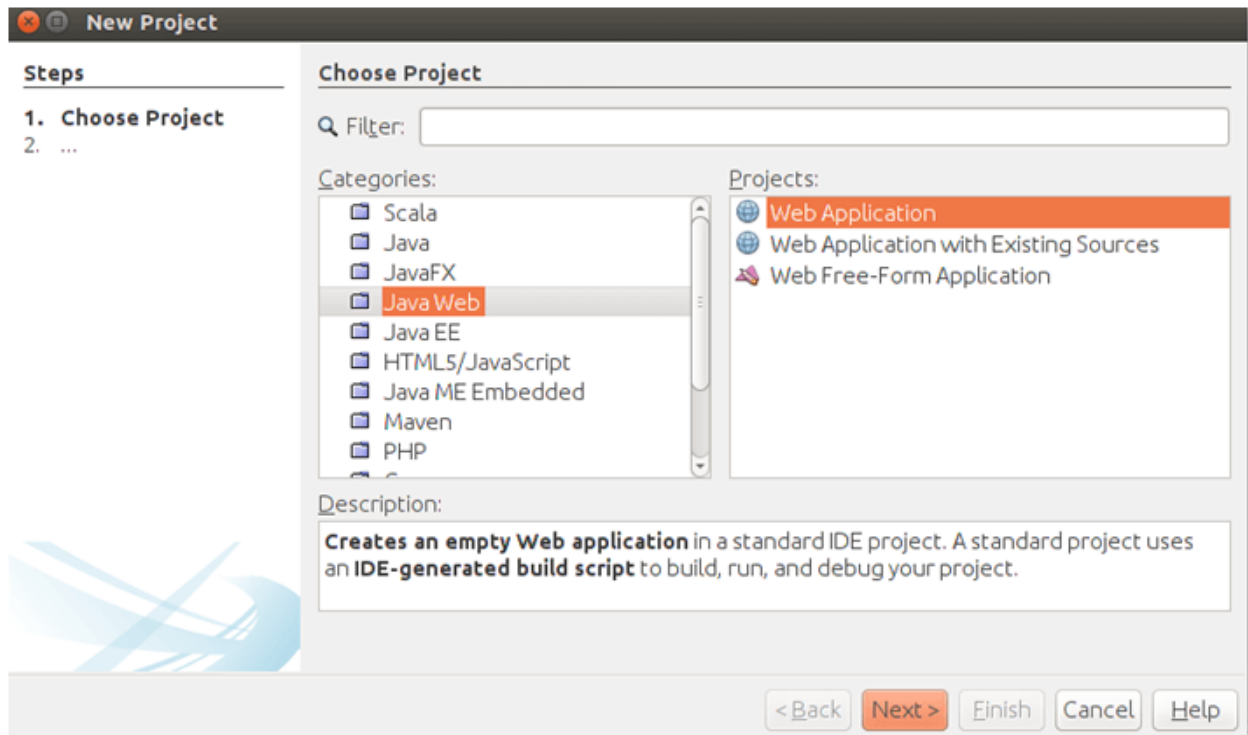
Step 1: Create a New Project

Go to file menu and select new Project.

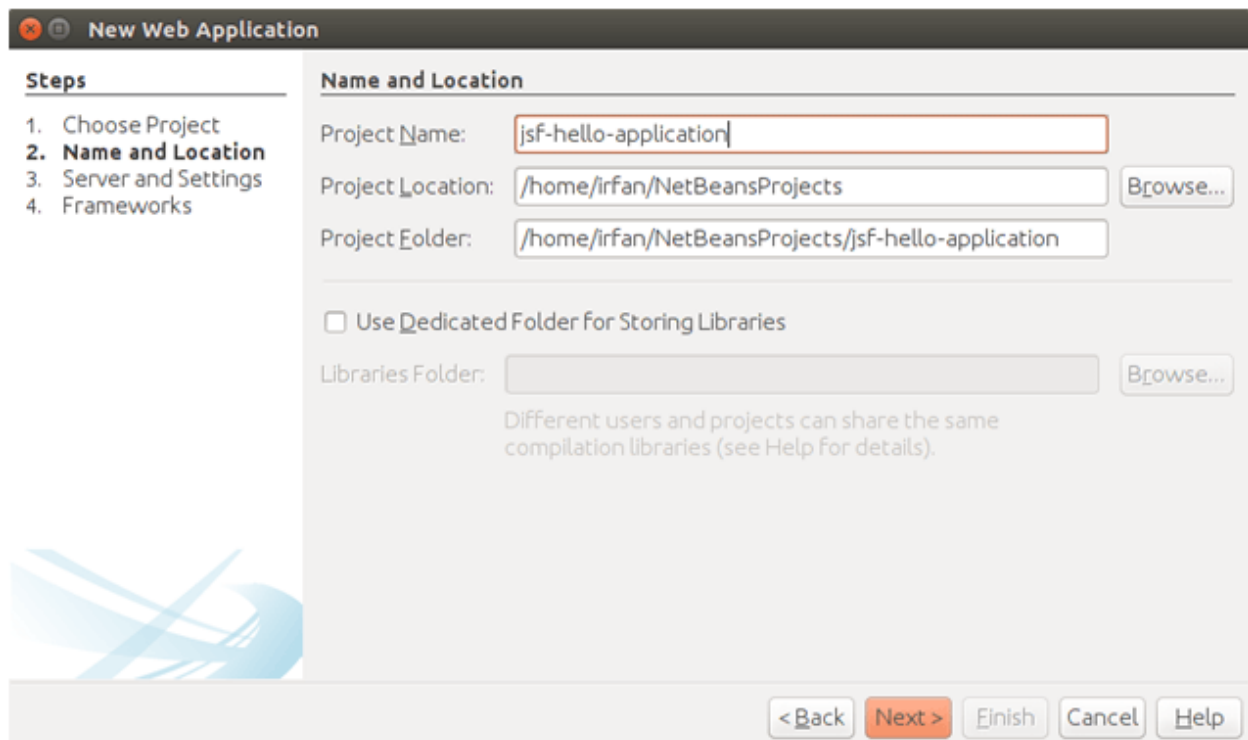


Select Category Java Web and Project Web Application

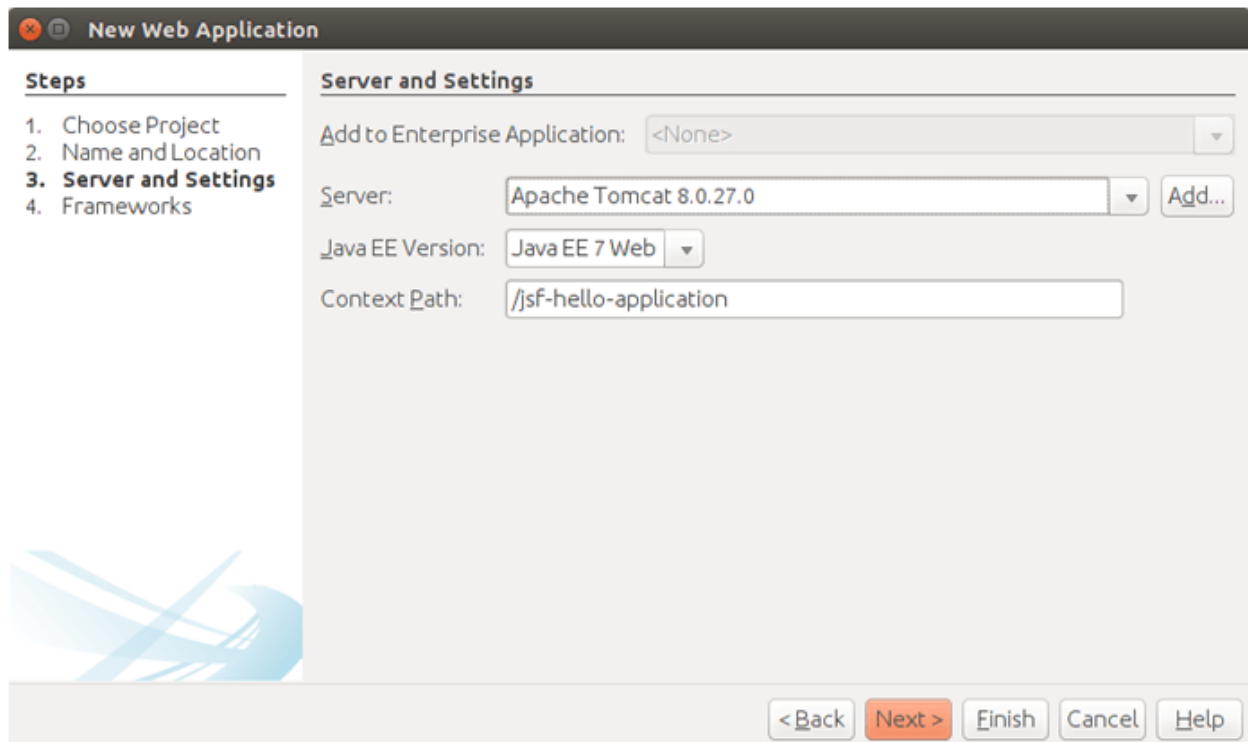




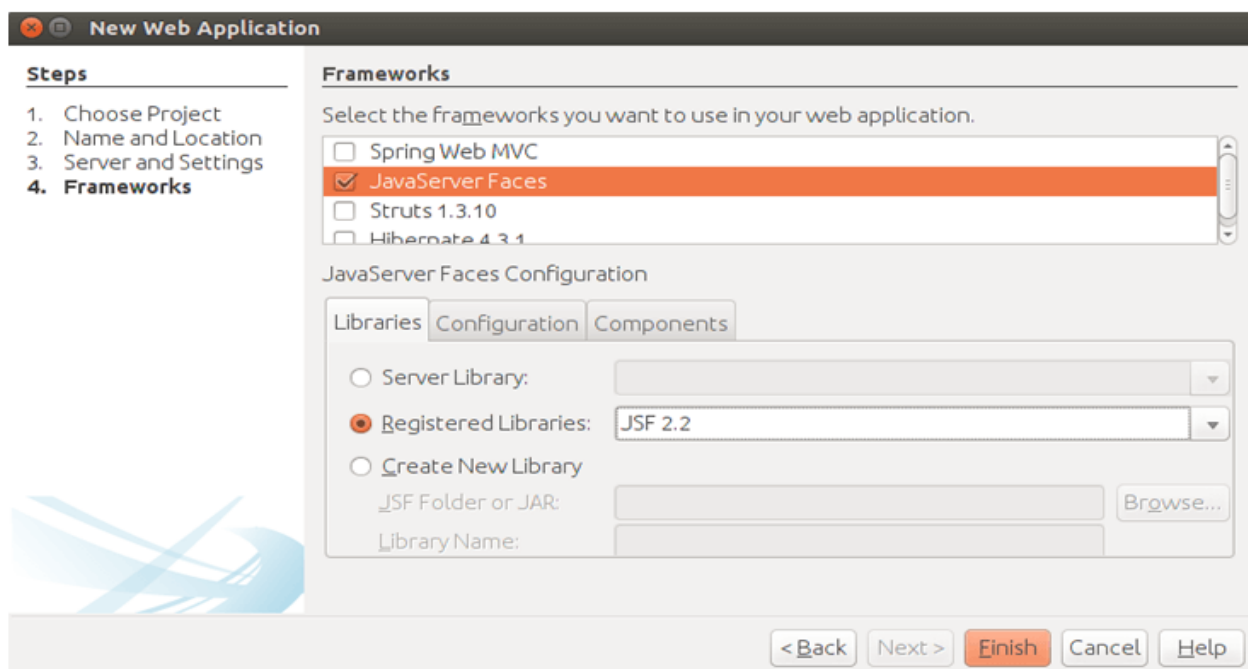
Enter project name.



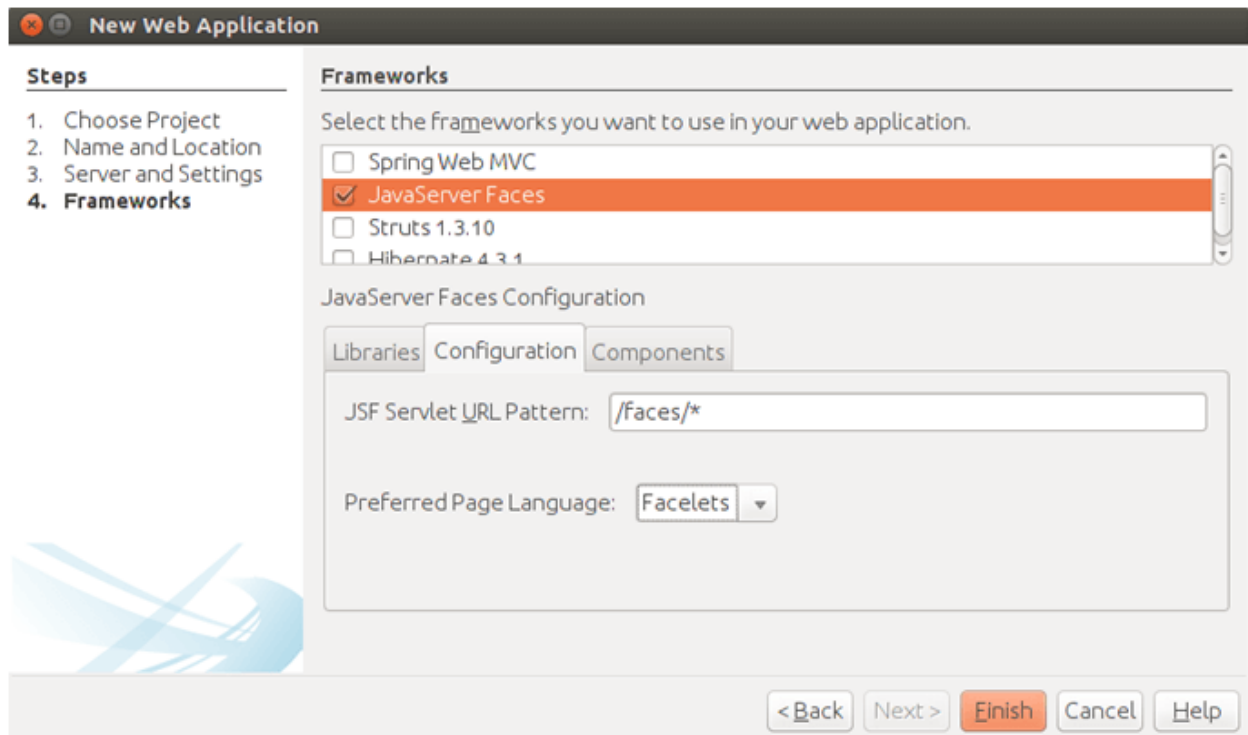
Select Server and Java EE Version.



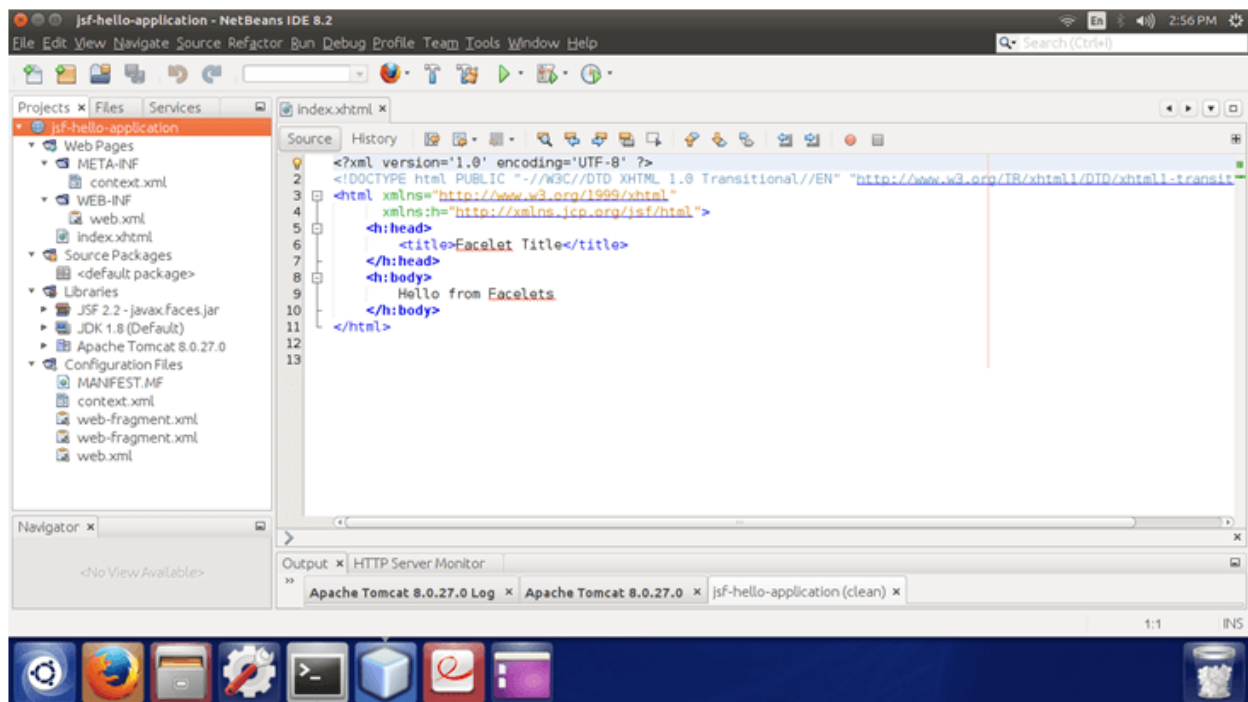
### Select JSF Framework



elect Preferred Page Language: Earlier versions of JSF framework are default to JSP for presentation pages. Now, in latest version 2.0 and later JSF has included powerful tool "Facelets". So, here we have selected page language as facelets. We will talk about facelets in more details in next chapter.



Index.xhtml Page: After finishing, IDE creates a JSF project for you with a default index.xhtml file. Xhtml is an extension of html and used to create facelets page.



Run: Now, you can run your application by selecting run option after right click on the project. It will produce a default message "Hello from Facelets".

We have created JSF project successfully. This project includes following files:

**index.xhtml**: inside the Web Pages directory

**web.xml:** inside the WEB-INF directory

Whenever we run the project, it renders index.xhtml as output. Now, we will create an application which contains two web pages, one bean class and a configuration file.

It requires the following steps in order to develop new application:

- Creating user interface
- Creating managed beans
- Configuring and managing FacesServlet

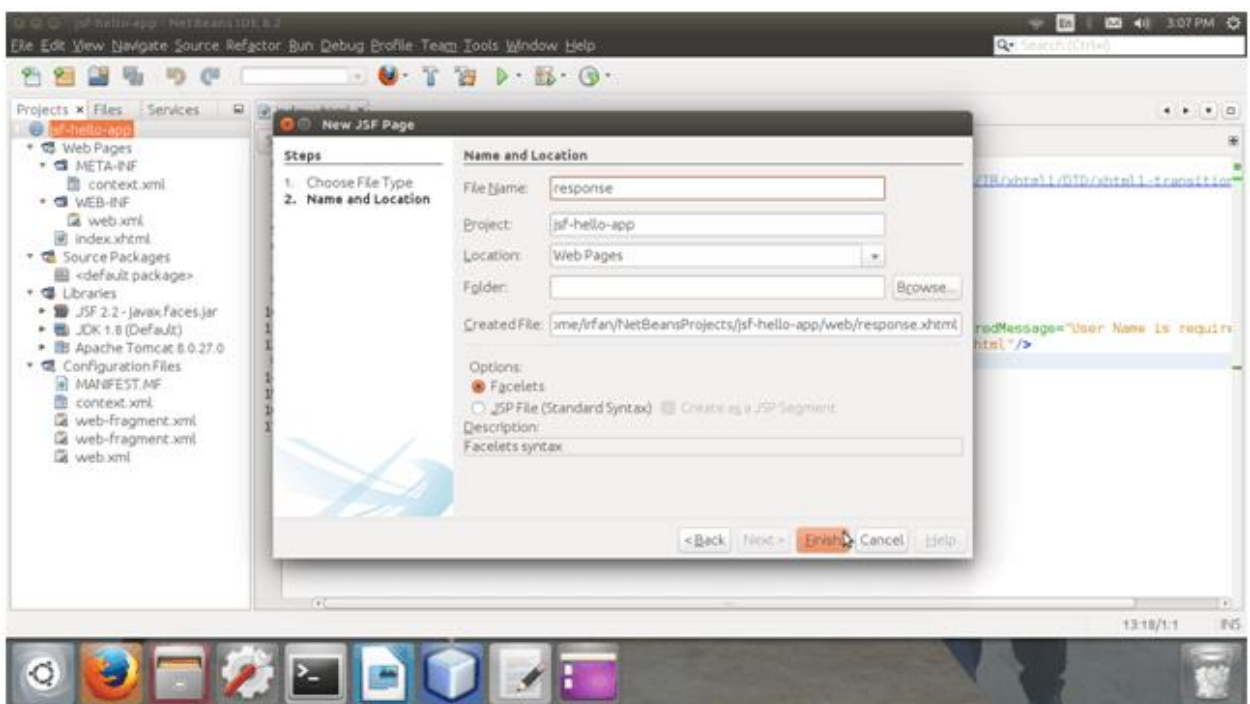
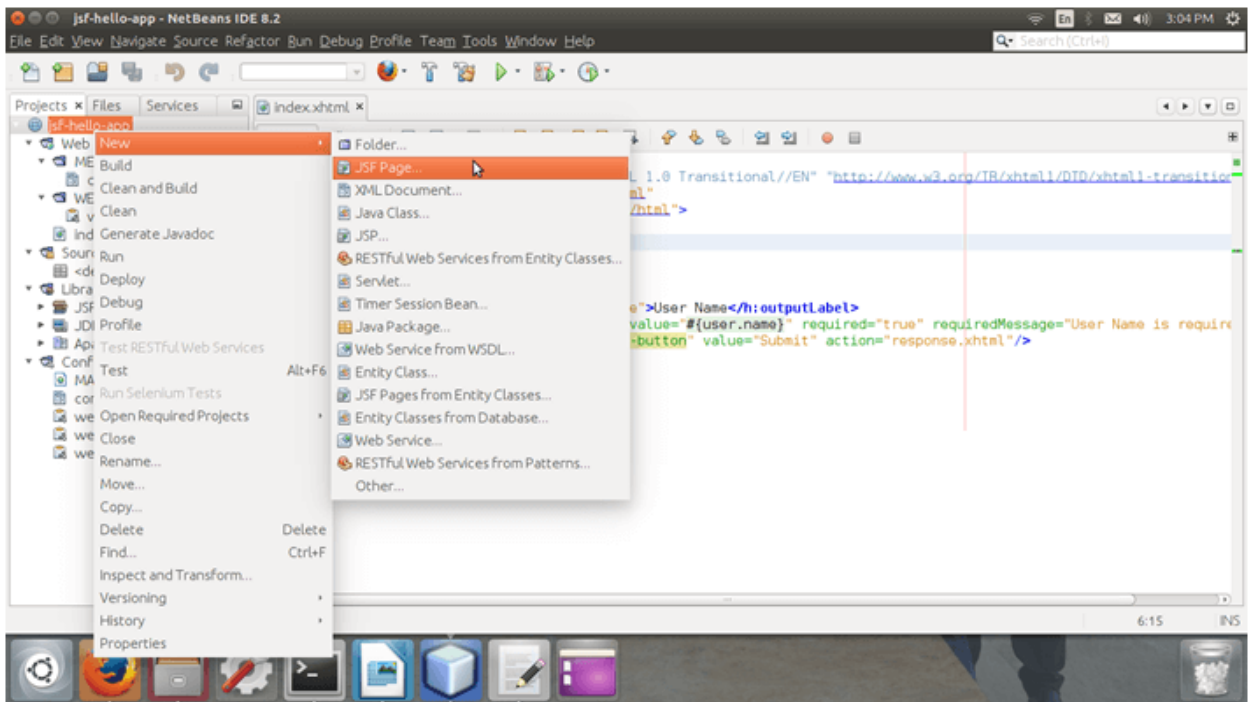
#### 1) Create User Interface

We will use default page index.xhtml to render input web page. Modify your index.xhtml source code as the given below.

// **index.xhtml**

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Transitional//EN""http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://xmlns.jcp.org/jsf/html">
<h:head>
<title>User Form</title>
</h:head>
<h:body>
<h:form>
<h:outputLabel for="username">User Name</h:outputLabel>
<h:inputText id="username" value="#{user.name}" required="true" requiredMessage="User
Name is required" /><br/>
<h:commandButton id="submit-button" value="Submit" action="response.xhtml"/>
</h:form>
</h:body>
</html>
```

Create a second web page which produce the output.



After creating response.xhtml page. Now, modify it's source code as the given below.

```
// response.xhtml
```

```
<?xml version='1.0' encoding='UTF-8' ?>
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml"
```

```
xmlns:h="http://xmlns.jcp.org/jsf/html">
```

```
<h:head>
```

```
<title>Welcome Page</title>
```

```
</h:head>
```

```
<h:body>
```

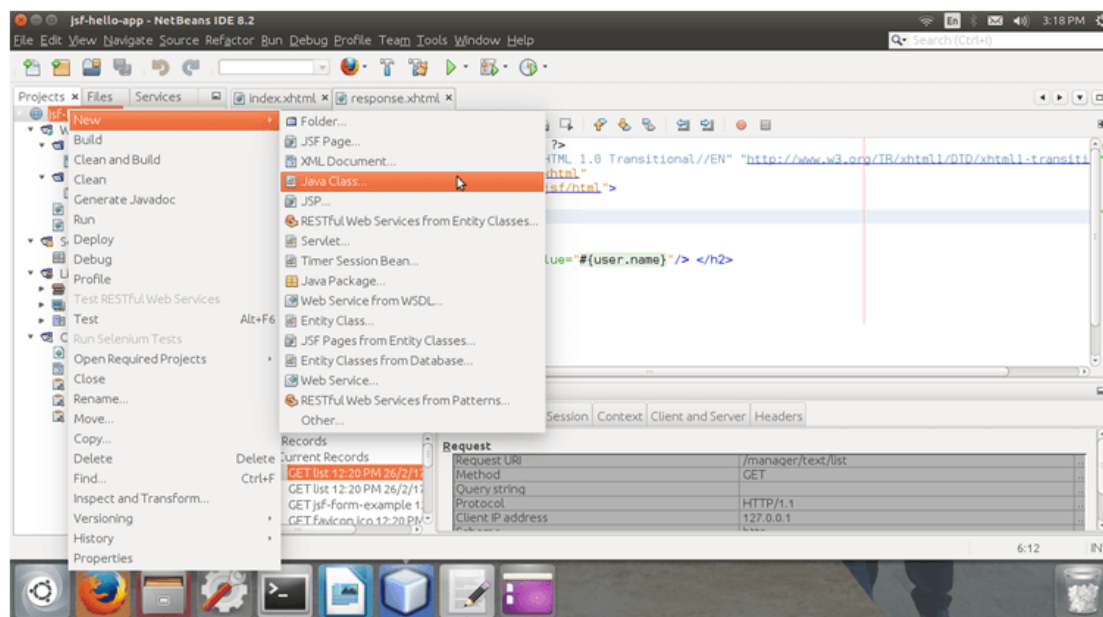
```
<h2>Hello, <h:outputText value="#{user.name}"></h:outputText></h2>
```

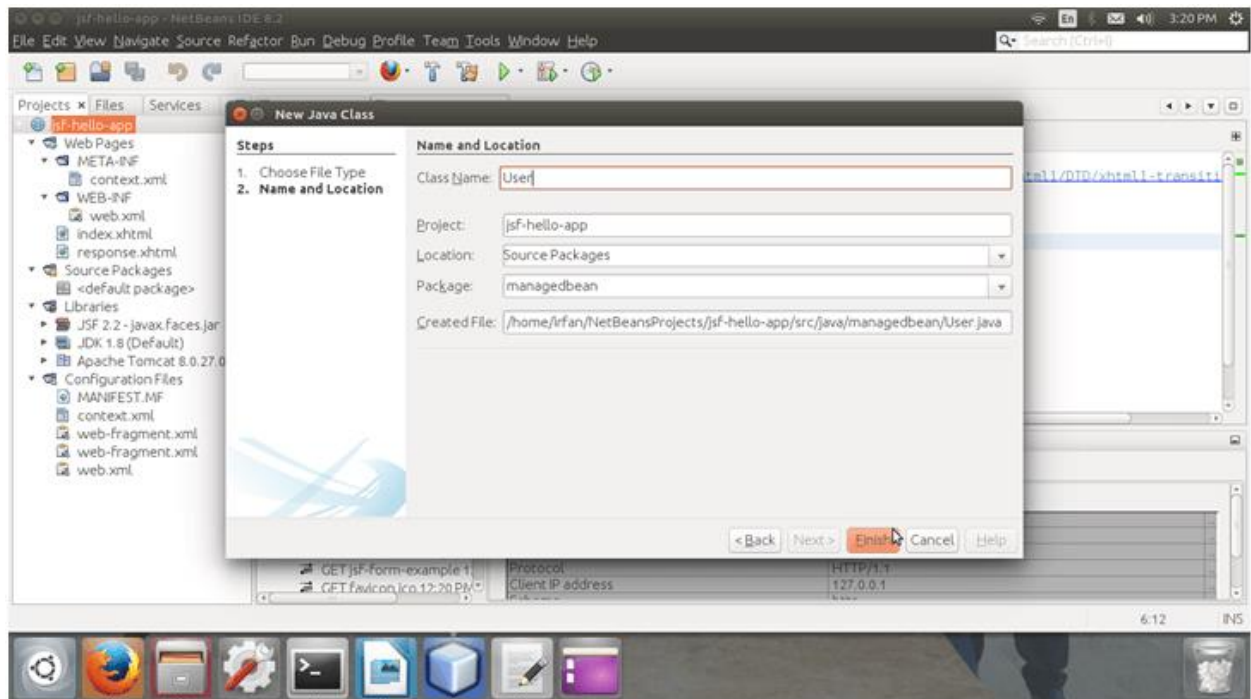
```
</h:body>
```

```
</html>
```

## 2) Create a Managed Bean

It is a Java class which contains properties and getter setter methods. JSF uses it as a Model. So, you can use it to write your business logic also.





After creating a Java class put the below code into your User.java file.

```
// User.java
```

```
package managedbean;
```

```
Facelet Title
```

```
import javax.faces.bean.ManagedBean;
```

```
import javax.faces.bean.RequestScoped;
```

```
@ManagedBean
```

```
@RequestScoped
```

```
public class User {
```

```
String name;
```

```
public String getName() {
```

```
return name;
```

```
}
```

```
public void setName(String name) {
```

```
this.name = name;
```

```
} }
```

### 3) Configure Application

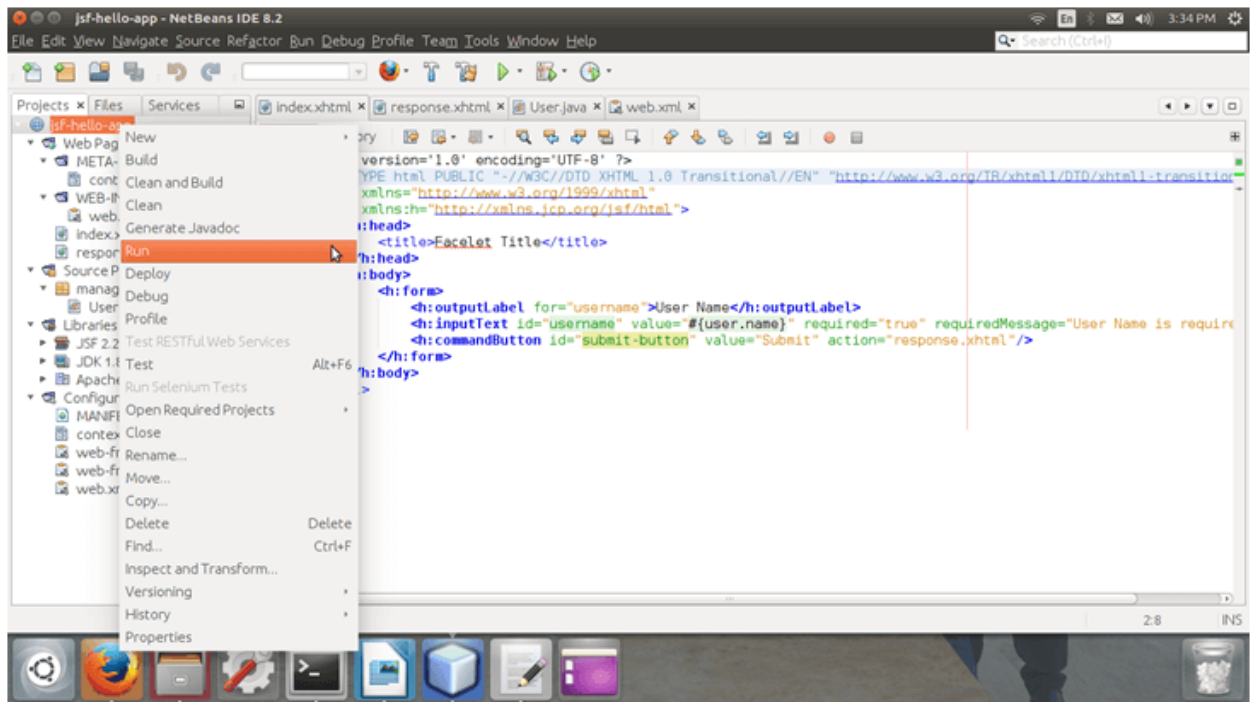
To configure application, project contains a web.xml file which helps to set FacesServlet instances. You can also set your application welcome page and any more.

Below is the code of web.xml code for this application.

```
// web.xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
<context-param>
<param-name>javax.faces.PROJECT_STAGE</param-name>
<param-value>Development</param-value>
</context-param>
<servlet>
<servlet-name>Faces Servlet</servlet-name>
<servlet-class>javax.faces.webapp.FaceletTitle.FacesServlet</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>Faces Servlet</servlet-name>
<url-pattern>/faces/*</url-pattern>
</servlet-mapping>
<session-config>
<session-timeout>
30
</session-timeout>
</session-config>
<welcome-file-list>
<welcome-file>faces/index.xhtml</welcome-file>
</welcome-file-list>
</web-app>
```

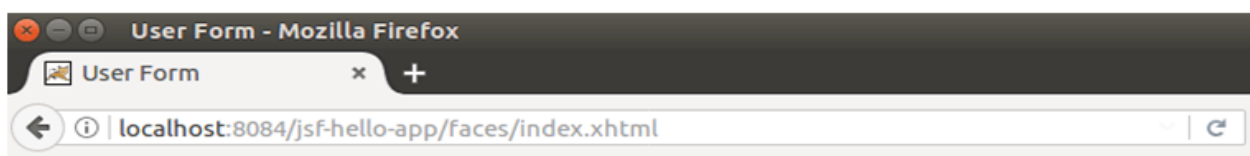
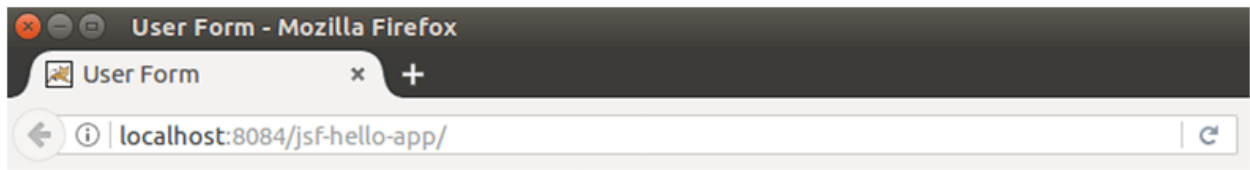
Well! All set. Now run the application.



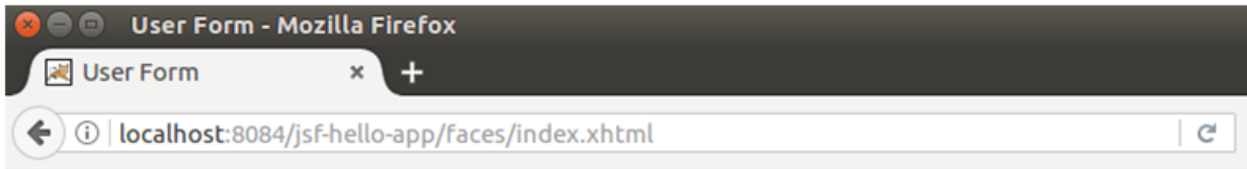


Output:

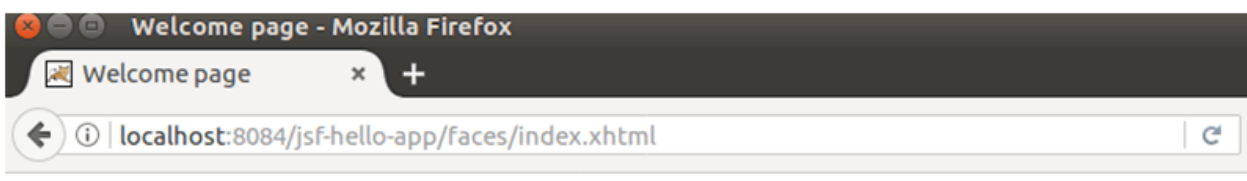
This is index page of the application.



• User Name is required



User Name



**Hello, javatpoint**

### JSF - Basic Tags

SF provides a standard HTML tag library. These tags get rendered into corresponding html output.

For these tags you need to use the following namespaces of URI in html node.

```
<html  
  xmlns = "http://www.w3.org/1999/xhtml"  
  xmlns:h = "http://java.sun.com/jsf/html">
```

Following are the important Basic Tags in JSF 2.0.

S.No	Tag & Description
1	<u><a href="#">h:inputText</a></u> Renders a HTML input of type="text", text box.
2	<u><a href="#">h:inputSecret</a></u> Renders a HTML input of type="password", text box.
3	<u><a href="#">h:inputTextarea</a></u> Renders a HTML textarea field.
4	<u><a href="#">h:inputHidden</a></u> Renders a HTML input of type="hidden".

5	<b><u><a href="#">h:selectBooleanCheckbox</a></u></b> Renders a single HTML check box.
6	<b><u><a href="#">h:selectManyCheckbox</a></u></b> Renders a group of HTML check boxes.
7	<b><u><a href="#">h:selectOneRadio</a></u></b> Renders a single HTML radio button.
8	<b><u><a href="#">h:selectOneListbox</a></u></b> Renders a HTML single list box.
9	<b><u><a href="#">h:selectManyListbox</a></u></b> Renders a HTML multiple list box.
10	<b><u><a href="#">h:selectOneMenu</a></u></b> Renders a HTML combo box.
11	<b><u><a href="#">h:outputText</a></u></b> Renders a HTML text.
12	<b><u><a href="#">h:outputFormat</a></u></b> Renders a HTML text. It accepts parameters.
13	<b><u><a href="#">h:graphicImage</a></u></b> Renders an image.
14	<b><u><a href="#">h:outputStylesheet</a></u></b> Includes a CSS style sheet in HTML output.
15	<b><u><a href="#">h:outputScript</a></u></b> Includes a script in HTML output.
16	<b><u><a href="#">h:commandButton</a></u></b> Renders a HTML input of type="submit" button.
17	<b><u><a href="#">h:Link</a></u></b> Renders a HTML anchor.
18	<b><u><a href="#">h:commandLink</a></u></b> Renders a HTML anchor.
19	<b><u><a href="#">h:outputLink</a></u></b> Renders a HTML anchor.
20	<b><u><a href="#">h:panelGrid</a></u></b> Renders an HTML Table in form of grid.
21	<b><u><a href="#">h:message</a></u></b> Renders message for a JSF UI Component.

22	<b><u>h:messages</u></b> Renders all message for JSF UI Components.
23	<b><u>f:param</u></b> Pass parameters to JSF UI Component.
24	<b><u>f:attribute</u></b> Pass attribute to a JSF UI Component.
25	<b><u>f:setPropertyActionListener</u></b> Sets value of a managed bean's property.

### JSF - h:inputText

The h:inputText tag renders an HTML input element of the type "text".

JSF Tag

```
<h:inputText value = "Hello World!" />
```

Rendered Output

```
<input type = "text" name = "j_idt6:j_idt8" value = "Hello World!" />
```

### Tag Attributes

S.No	Attribute & Description
1	<b>id</b> Identifier for a component
2	<b>binding</b> Reference to the component that can be used in a backing bean
3	<b>rendered</b> A boolean; false suppresses rendering
4	<b>styleClass</b> Cascading stylesheet (CSS) class name
5	<b>value</b> A component's value, typically a value binding
6	<b>valueChangeListener</b> A method binding to a method that responds to value changes
7	<b>converter</b>

	Converter class name
8	<b>validator</b> Class name of a validator that's created and attached to a component
9	<b>required</b> A boolean; if true, requires a value to be entered in the associated field
10	<b>accesskey</b> A key, typically combined with a system-defined metakey, that gives focus to an element
11	<b>accept</b> Comma-separated list of content types for a form
12	<b>accept-charset</b> Comma- or space-separated list of character encodings for a form. The <b>accept-charset</b> attribute is specified with the JSF HTML attribute named <b>acceptcharset</b> .
13	<b>alt</b> Alternative text for nontextual elements such as images or applets
14	<b>border</b> Pixel value for an element's border width
15	<b>charset</b> Character encoding for a linked resource
16	<b>coords</b> Coordinates for an element whose shape is a rectangle, circle, or polygon
17	<b>dir</b> Direction for text. Valid values are <b>ltr</b> (left to right) and <b>rtl</b> (right to left).
18	<b>disabled</b> Disabled state of an input element or button
19	<b>hreflang</b> Base language of a resource specified with the <b>href</b> attribute; <b>hreflang</b> may only be used with <b>href</b>
20	<b>lang</b> Base language of an element's attributes and text
21	<b>maxlength</b> Maximum number of characters for text fields
22	<b>readonly</b> Read-only state of an input field; the text can be selected in a readonly field but not edited
23	<b>style</b> Inline style information

24	<b>tabindex</b> Numerical value specifying a tab index
25	<b>target</b> The name of a frame in which a document is opened
26	<b>title</b> A title, used for accessibility, that describes an element. Visual browsers typically create tooltips for the title's value
27	<b>type</b> Type of a link; for example, <b>stylesheet</b>
28	<b>width</b> Width of an element
29	<b>onblur</b> Element loses focus
30	<b>onchange</b> Element's value changes
31	<b>onclick</b> Mouse button is clicked over the element
32	<b>ondblclick</b> Mouse button is double-clicked over the element
33	<b>onfocus</b> Element receives focus
34	<b>onkeydown</b> Key is pressed
35	<b>onkeypress</b> Key is pressed and subsequently released
36	<b>onkeyup</b> Key is released
37	<b>onmousedown</b> Mouse button is pressed over the element
38	<b>onmousemove</b> Mouse moves over the element
39	<b>onmouseout</b> Mouse leaves the element's area
40	<b>onmouseover</b> Mouse moves onto an element

41	<b>onmouseup</b> Mouse button is released
42	<b>onreset</b> Form is reset
43	<b>onselect</b> Text is selected in an input field
44	<b>immediate</b> Process validation early in the life cycle



### Example Application

Let us create a test JSF application to test the above tag.

Step	Description
1	Create a project with a name <i>helloworld</i> under a package <i>com.tutorialspoint.test</i> as explained in the <i>JSF - First Application</i> chapter.
2	Modify <i>home.xhtml</i> as explained below. Keep the rest of the files unchanged.
3	Compile and run the application to make sure business logic is working as per the requirements.
4	Finally, build the application in the form of war file and deploy it in Apache Tomcat Webserver.
5	Launch your web application using appropriate URL as explained below in the last step.

#### home.xhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns = "http://www.w3.org/1999/xhtml">
  <head>
    <title>JSF Tutorial!</title>
  </head>

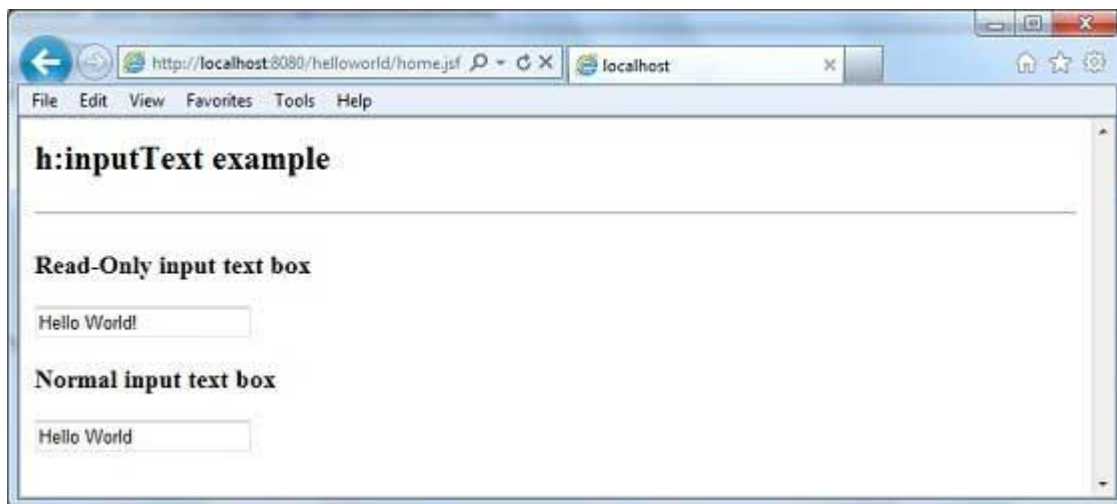
  <body>
    <h2>h:inputText example</h2>
    <hr />

    <h:form>
      <h3>Read-Only input text box</h3>
      <h:inputText value = "Hello World!" readonly = "true"/>
    </h:form>
  </body>
</html>
```

```
<h3>Read-Only input text box</h3>
<h:inputText value = "Hello World"/>
</h:form>

</body>
</html>
```

Once you are ready with all the changes done, let us compile and run the application as we did in JSF - First Application chapter. If everything is fine with your application, this will produce the following result.



## JSF EXPRESSION

JSF provides a rich expression language. We can write normal operations using `#{operation-expression}` notation. Following are some of the advantages of JSF Expression languages.

- Can reference bean properties where bean can be an object stored in request, session or application scope or is a managed bean.
- Provides easy access to elements of a collection which can be a list, map or an array.
- Provides easy access to predefined objects such as a request.
- Arithmetic, logical and relational operations can be done using expression language.
- Automatic type conversion.
- Shows missing values as empty strings instead of `NullPointerException`.

## Example Application

Let us create a test JSF application to test expression language.



Step	Description
1	Create a project with a name helloworld under a package com.tutorialspoint.test as explained in the JSF - First Application chapter.
2	Modify UserData.java under package com.tutorialspoint.test as explained below.
3	Modify home.xhtml as explained below. Keep the rest of the files unchanged.
4	Compile and run the application to make sure the business logic is working as per the requirements.
5	Finally, build the application in the form of war file and deploy it in Apache Tomcat Webserver.
6	Launch your web application using appropriate URL as explained below in the last step.

**UserData.java**

```

package com.tutorialspoint.test;

import java.io.Serializable;
import java.util.Date;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean(name = "userData", eager = true)
@SessionScoped

public class UserData implements Serializable {
    private static final long serialVersionUID = 1L;
    private Date createTime = new Date();
    private String message = "Hello World!";

    public Date getCreateTime() {
        return(createTime);
    }

    public String getMessage() {
        return(message);
    }
}

```

**home.xhtml**

```
<?xml version = "1.0" encoding = "UTF-8"?>
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns = "http://www.w3.org/1999/xhtml"
xmlns:f = "http://java.sun.com/jsf/core"
xmlns:h = "http://java.sun.com/jsf/html">
  <h:head>
    <title>JSF Tutorial!</title>
  </h:head>
  <h:body>
    <h2>Expression Language Example</h2>
    Creation time:
    <h:outputText value = "#{userData.createTime}"/>
    <br/><br/>
    Message:
    <h:outputText value = "#{userData.message}"/>
  </h:body>
</html>
```

Once you are ready with all the changes done, let us compile and run the application as we did in JSF - First Application chapter. If everything is fine with your application, this will produce the following result.

#### JSF Expression Language Result

