

# Java RMI

Remote method invocation(RMI) allow a java object to invoke method on an object running on another machine. RMI provide remote communication between java program. RMI is used for building distributed application.

---

## Concept of RMI application

A RMI application can be divided into two part, **Client** program and **Server** program. A **Server** program creates some remote object, make their references available for the client to invoke method on it. A **Client** program make request for remote objects on server and invoke method on them. **Stub** and **Skeleton** are two important object used for communication with remote object.

---

## Stub

In RMI, a stub is an object that is used as a Gateway for the client-side. All the outgoing request are sent through it. When a client invokes the method on the stub object following things are performed internally:

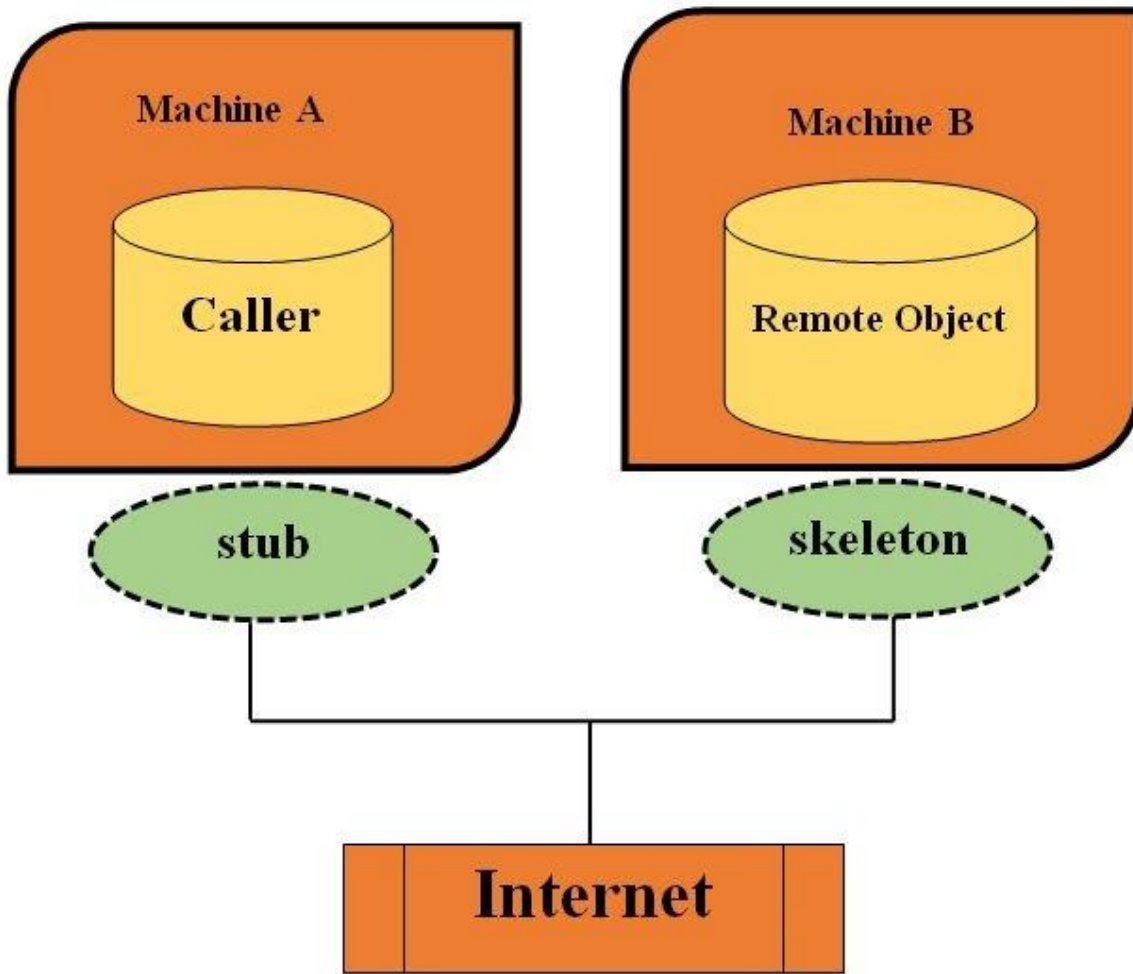
1. A connection is established using Remote Virtual Machine.
2. It then transmits the parameters to the Remote Virtual Machine.  
This is also known as Marshals
3. After the 2nd step, it then waits for the output.
4. Now it reads the value or exception which is come as an output.

5. At last, it returns the value to the client.
- 

### Skeleton

In RMI, a skeleton is an object that is used as a Gateway for the server-side. All the incoming requests are sent through it. When a Server invokes the method on the skeleton object, the following things are performed internally:

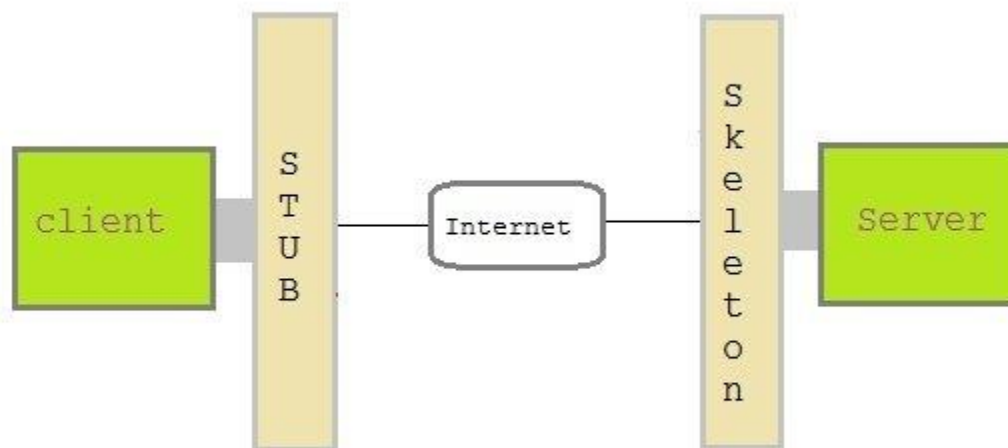
1. All the Parameters are read for the remote method.
2. The method is invoked on the remote object.
3. It then writes and transmits the parameters for the result. This is also known as Marshals.



---

### Stub and Skeleton

**Stub** act as a gateway for Client program. It resides on Client side and communicate with **Skeleton** object. It establish the connection between remote object and transmit request to it.



Skeleton object resides on server program. It is responsible for passing request from **Stub** to remote object.

---

Creating a Simple RMI application involves following steps

- Define a remote interface.
  - Implementing remote interface.
  - create and start remote application
  - create and start client application
- 

Define a remote interface

A remote interface specifies the methods that can be invoked remotely by a client. Clients program communicate to remote interfaces, not to classes implementing it. To be a remote interface, a interface must extend the **Remote** interface of **java.rmi** package.

```
import java.rmi.*;

public interface AddServerInterface extends Remote
{

public int sum(int a,int b);

}
```

Copy

---

Implementation of remote interface

For implementation of remote interface, a class must either extend **UnicastRemoteObject** or use exportObject() method of **UnicastRemoteObject** class.

```
import java.rmi.*;

import java.rmi.server.*;

public class Adder extends UnicastRemoteObject
implements AddServerInterface
{

    Adder() throws RemoteException{

        super();
    }

}
```

```

public int sum(int a,int b)

{

    return a+b;

}

}

```

Copy

---

Create AddServer and host rmi service

You need to create a server application and host rmi service **Adder** in it. This is done using `rebind()` method of **java.rmi.Naming** class. `rebind()` method take two arguments, first represent the name of the object reference and second argument is reference to instance of **Adder**

```

import java.rmi.*;

import java.rmi.registry.*;

public class AddServer {

    public static void main(String args[]) {

        try {

            AddServerInterface addService=new
Adder();

```

```

        Naming.rebind("AddService",addService);
        //addService object is hosted with name
AddService

    }

    catch(Exception e) {

        System.out.println(e);

    }

}

}

```

Copy

---

Create client application

Client application contains a java program that invokes the **lookup()** method of the **Naming** class. This method accepts one argument, the **rmi** URL and returns a reference to an object of type **AddServerInterface**. All remote method invocation is done on this object.

```

import java.rmi.*;

public class Client {

```

```
public static void main(String args[]) {  
  
    try{  
  
        AddServerInterface st =  
(AddServerInterface)Naming.lookup("rmi://" + args[0] +  
"/AddService");  
  
        System.out.println(st.sum(25, 8));  
  
    }  
  
    catch(Exception e) {  
  
        System.out.println(e);  
  
    }  
  
}  
  
}
```

Copy

---

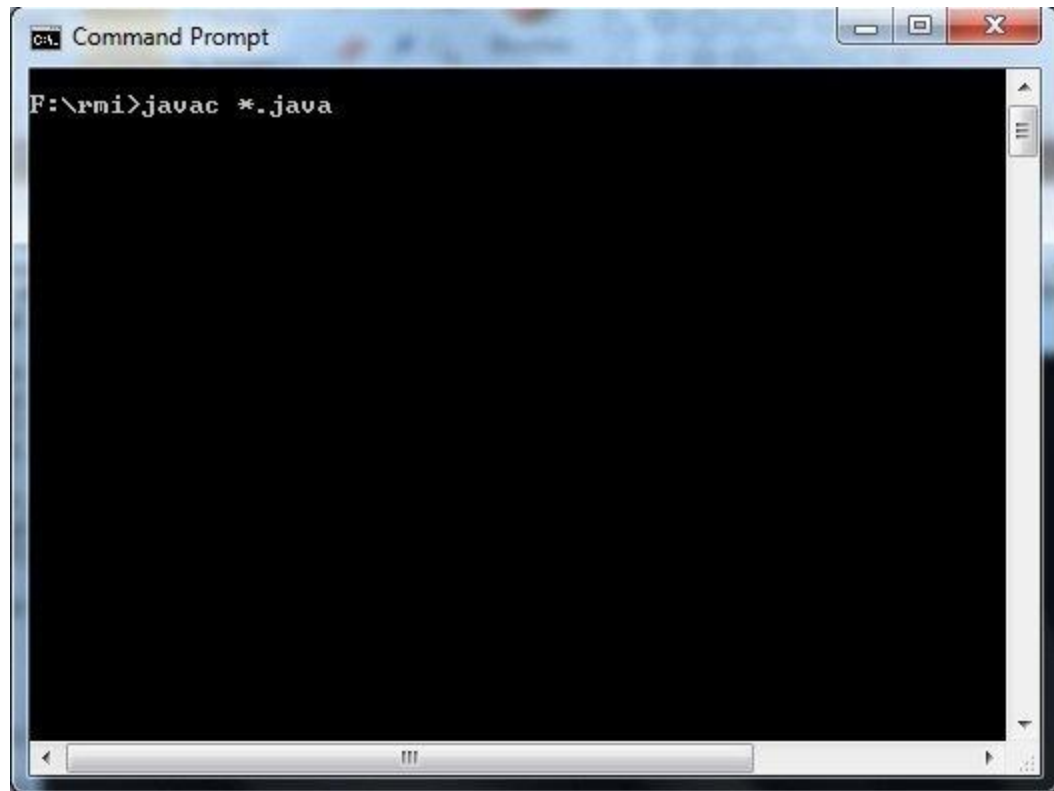
Steps to run this RMI application

Save all the above java file into a directory and name it as "rmi"

- compile all the java files

```
javac *.java
```





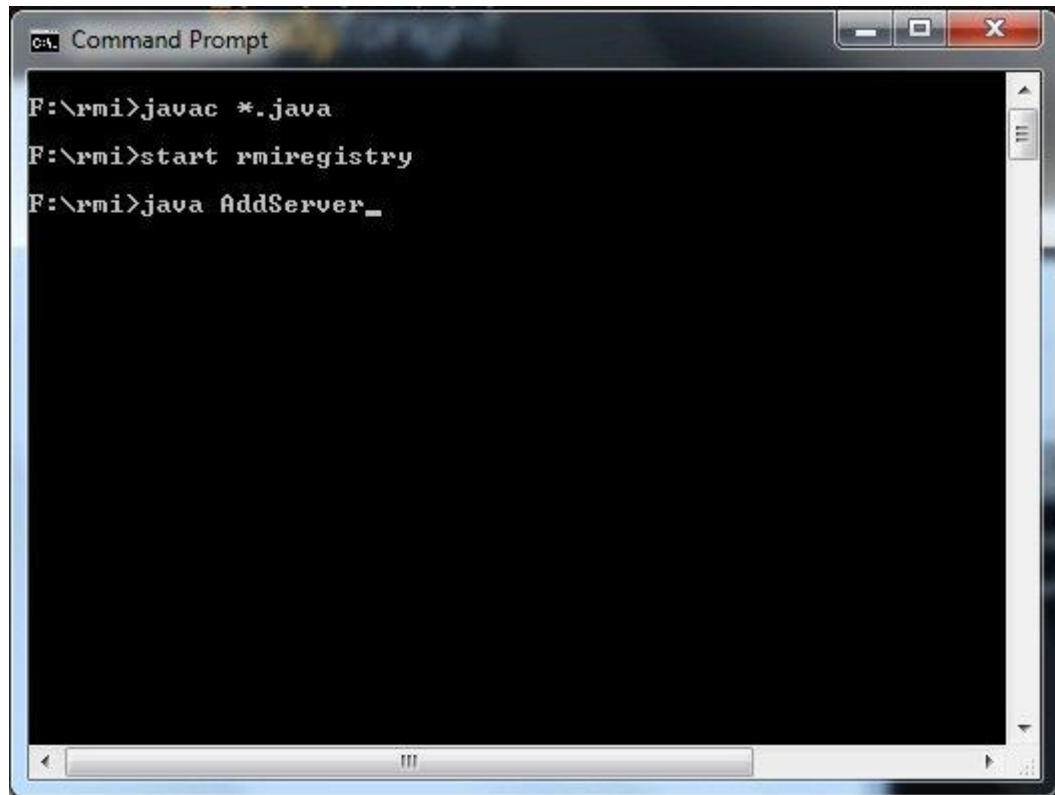
- Start RMI registry

```
start rmiregistry
```



- Run Server file

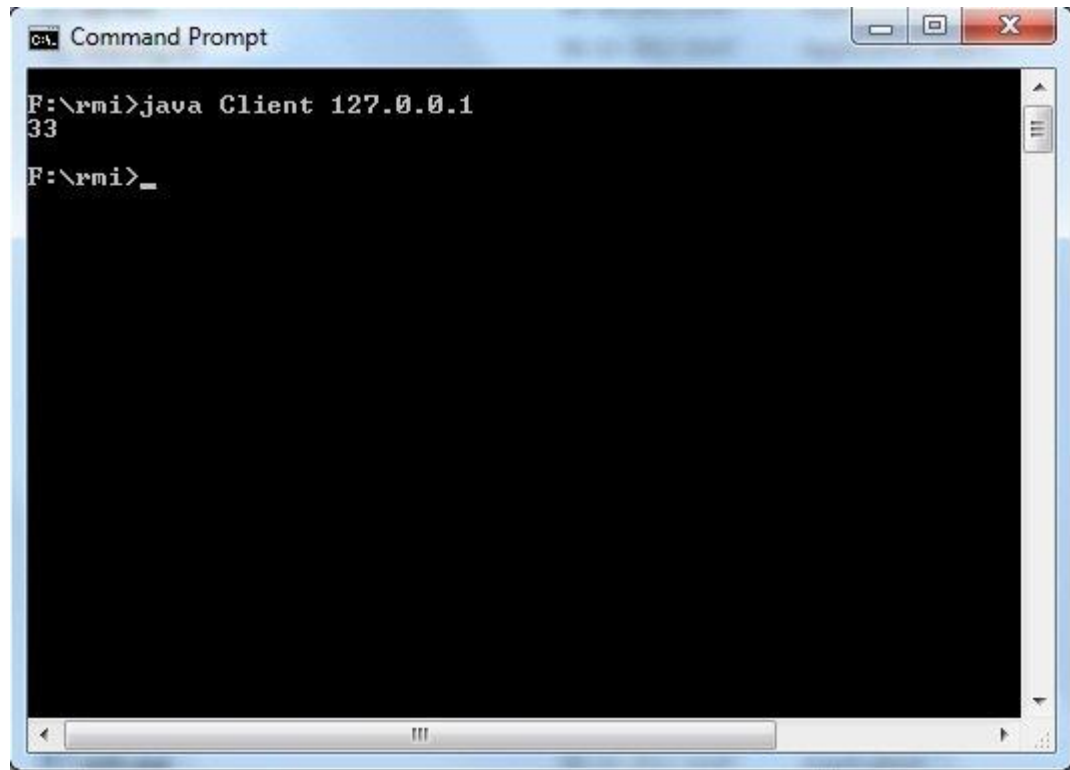
```
java AddServer
```



```
Command Prompt
F:\rmi>javac *.java
F:\rmi>start rmiregistry
F:\rmi>java AddServer_
```

- Run Client file in another command prompt abd pass local host port number at run time

```
java Client 127.0.0.1
```



---

Example:

### **Program: Power.java**

```
import java.rmi.*;

public interface Power extends Remote
{
    public int power1()throwsRemoteException;
}
```

Copy

**Program: PowerRemote.java**

```
import java.rmi.*;

import java.rmi.server.*;

import java.util.Scanner;

public class PowerRemote extends
UnicastRemoteObject implements Power
{

PowerRemote() throws RemoteException
{

    super();

}

public int power1(int z)
{

int z;

Scanner sc = new Scanner(System.in);

System.out.println("Enter the base number ::");

int x = sc.nextInt();
```

```
System.out.println("Enter the exponent number ::");  
  
int y = sc.nextInt();  
  
        z=y^x;  
  
System.out.println(z);  
  
}  
  
}
```

Copy

**MyServer.java**

```
import java.rmi.*;  
  
import java.rmi.registry.*;  
  
public class MyServer  
{  
  
public static void main(String args[])  
  
{  
  
try  
  
{  
  
Power stub=new PowerRemote();
```

```
Naming.rebind("rmi://localhost:1995/shristee", stub)
;

}

catch(Exception e)

{

System.out.println(e);

}

}

}
```

Copy

**MyClient.java**

```
import java.rmi.*;

public class MyClient

{

public static void main(String args[])

{

try
```

```
{  
  
Power  
stub=(Power)Naming.lookup("rmi://localhost:1995/shr  
istee");  
  
System.out.println(stub.power1());  
  
}  
  
catch(Exception e){}  
  
}  
  
}
```

Copy



```

E:\java\rmi>javac *.java
E:\java\rmi>rmic PowerRemote
E:\java\rmi>rmiregistry 5000

E:\>cd java
E:\java>cd rmi
E:\java\rmi>java MyServer

E:\>cd java
E:\java>cd rmi
E:\java\rmi>java MyClient
256.0

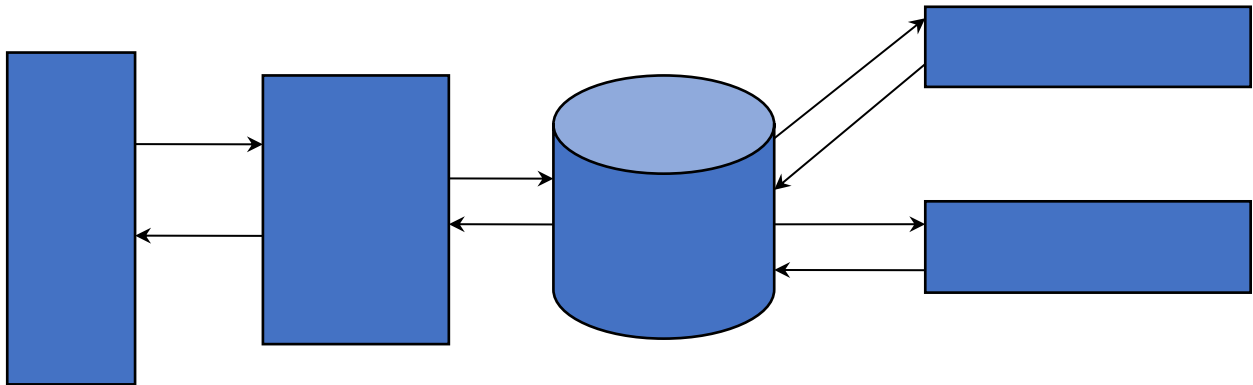
```

### What is JSP?

- JSP is Sun's solution for developing dynamic web sites. JSPs provide a way to separate the generation of dynamic content (java) from its presentation (html).
- JSP provide excellent server side scripting support for creating database driven web applications.
- JSP enable the developers to directly insert java code into jsp file, this makes the development process very simple and its maintenance also becomes very easy.
- Servlets generate the content as well as the necessary HTML syntax to present them to the browser. JSPs differentiate content from presentation.
- JSPs appear like an HTML document, embedded with JSP specific tags and have a file extension “.jsp”.
- The JSP directives are responsible for generating dynamic content while HTML part takes care of formatting and presentation.
- JSPs are ultimately implemented as Servlets.
- A “.jsp” file is always compiled to servlets when they are loaded for the first time, and subsequently whenever the JSP Page is modified.
- JSPs are much simpler than servlets and are easier to develop

### What happens during a JSP Page Request?

- When a request is mapped to a JSP page, the Web container first checks whether the JSP page's servlet is older than the JSP page.
- If the servlet is older, then the Web container translates the JSP page into a servlet class and compiles the class.
- During development, one of the advantages of JSP pages over servlets is that the build process is performed automatically.

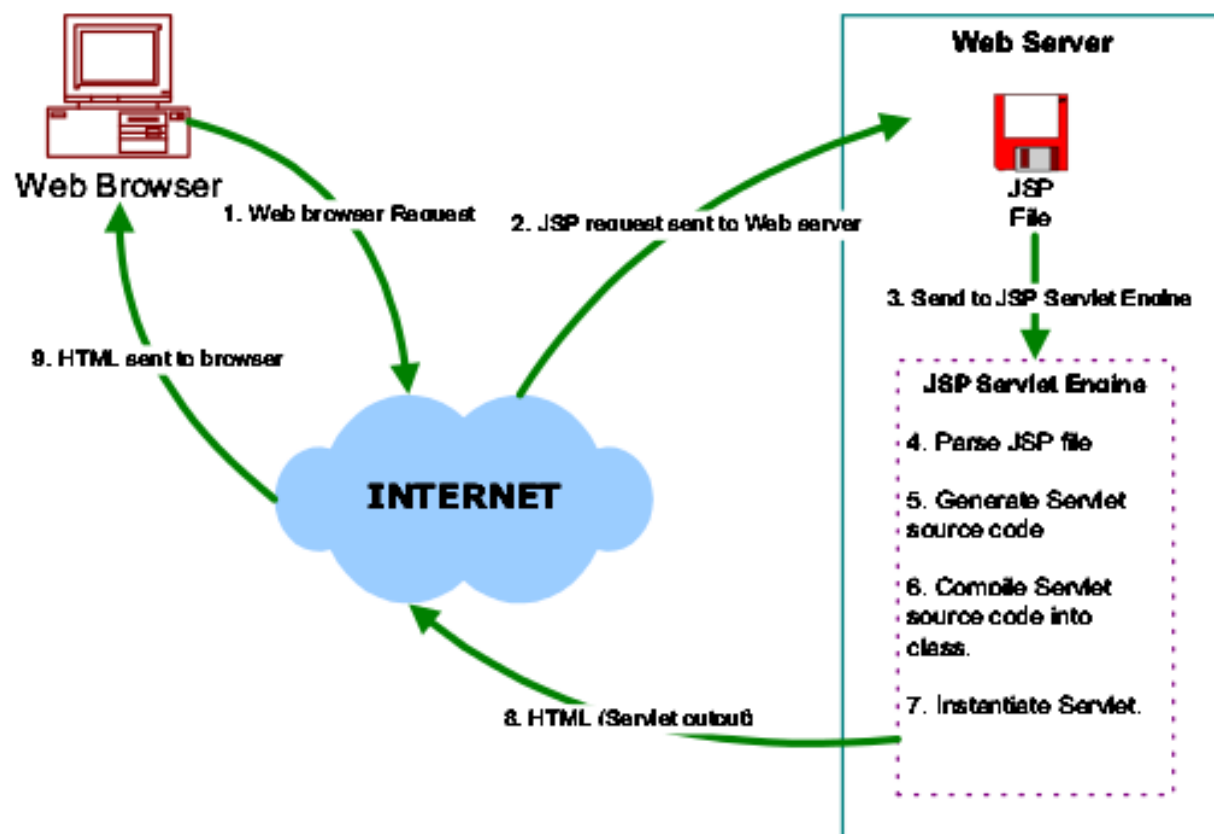


## JSP Architecture/Life Cycle:

1. JSP pages contain HTML elements with JSP tags embedded. JSP tags can contain java code. The JSP file extension is .jsp rather than .html
2. The JSP engine parses the .jsp and creates a Java servlet source file. It then compiles the source file into a class file, this is done the first time page is requested and so JSP is probably slower the first time it is accessed.
3. Any time after this the special compiled servlet is executed and is therefore faster.

**Steps required for a JSP request:**

1. The user goes to a web site made using JSP. The user goes to a JSP page (ending with .jsp). The web browser makes the request via the Internet.
2. The JSP request gets sent to the Web server.
3. The Web server recognises that the file required is special (.jsp), therefore passes the JSP file to the JSP Servlet Engine.
4. If the JSP file has been called the first time, the JSP file is parsed, otherwise go to step 7.
5. The next step is to generate a special Servlet from the JSP file. All the HTML required is converted to print statements.
6. The Servlet source code is compiled into a class.
7. The Servlet is instantiated, calling the *init* and *service* methods.
8. HTML from the Servlet output is sent via the Internet.
9. HTML results are displayed on the user's web browser.



### JSP Tags:

There are 5 main JSP Tags:

1. Declaration Tag
2. Expression Tag
3. Directives Tag
4. Scriptlet Tag
5. Action Tag

## Declaration tag ( <%! %> )

This tag allows the developer to declare variables or methods.

Before the declaration you must have <%!

At the end of the declaration, the developer must have %>

Code placed in this tag must end in a semicolon ( ; ).

Declarations do not generate output so are used with JSP expressions or scriptlets.

For Example,

```
<%!
    private int counter = 0 ;
    private String get Account ( int accountNo ) :
%>
```

## Expression tag ( <%= %> )

This tag allows the developer to embed any Java expression and is short for out.println().

A semicolon ( ; ) does not appear at the end of the code inside the tag.

For example, to show the current date and time.

```
Date : <%= new java.util.Date() %>
```

## Scriptlet tag ( <% ... %> )

Between <% and %> tags, any valid Java code is called a Scriptlet. This code can access any variable or bean declared.

For example, to print a variable.

```
<%
    String username = "visualbuilder" ;
    out.println ( username ) :
%>
```

# Java RMI - Introduction

RMI stands for **Remote Method Invocation**. It is a mechanism that allows an

object residing in one system (JVM) to access/invoke an object running on another JVM.

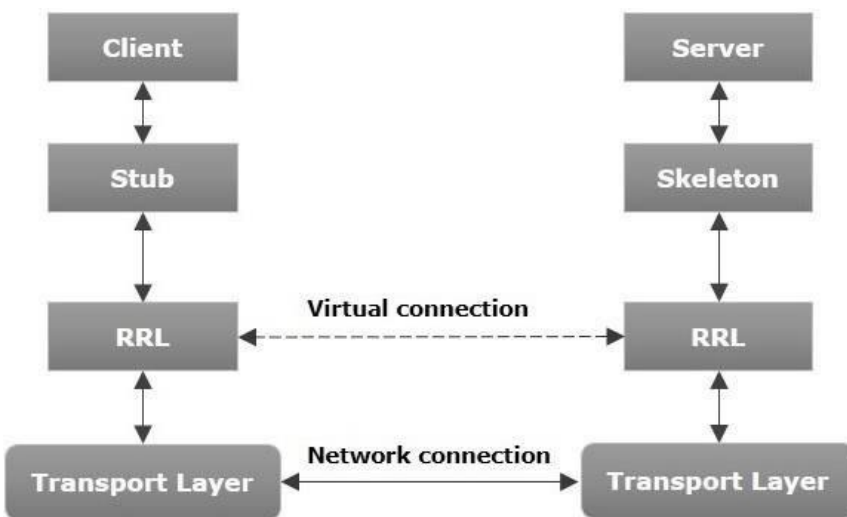
RMI is used to build distributed applications; it provides remote communication between Java programs. It is provided in the package **java.rmi**.

## Architecture of an RMI Application

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

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

The following diagram shows the architecture of an RMI application.



Let us now discuss the components of this architecture.

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

## Working of an RMI Application

The following points summarize how an RMI application works –

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

## Marshalling and Unmarshalling

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

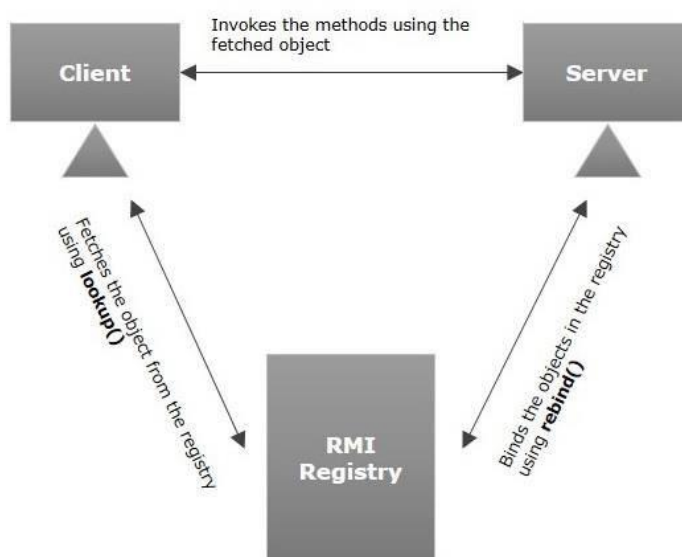
At the server side, the packed parameters are unbundled and then the required method is invoked. This process is known as **unmarshalling**.

## RMI Registry

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

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

The following illustration explains the entire process –



## Goals of RMI

Following are the goals of RMI –

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

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

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

## Defining the Remote Interface

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

To create a remote interface –

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

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

```
import java.rmi.Remote;  
import java.rmi.RemoteException;  
  
// Creating Remote interface for our application  
public interface Hello extends Remote {
```

## Developing the Implementation Class (Remote Object)

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

To develop an implementation class –



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

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

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

    // Implementing the interface method
    public void printMsg() {

        System.out.println("This is an example RMI program");
    }
}
```

## Developing the Server Program

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

To develop a server program –

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

Following is an example of an RMI server program.

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

public class Server extends ImplExample {
    public Server() {}

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

            // Instantiating the implementation class
            ImplExample obj = new ImplExample();
        }
    }
}
```

```

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

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

        registry.bind("Hello", stub);
        System.err.println("Server ready");

    } catch (Exception e) {

        System.err.println("Server exception: " + e.toString());
        e.printStackTrace();
    }
}

```

## Developing the Client Program

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

To develop a client program –

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

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

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

Following is an example of an RMI client program.

```

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

public class Client {
    private Client() {}

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

            // Getting the registry

            Registry registry = LocateRegistry.getRegistry(null);

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

```

```

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

```

## Compiling the Application

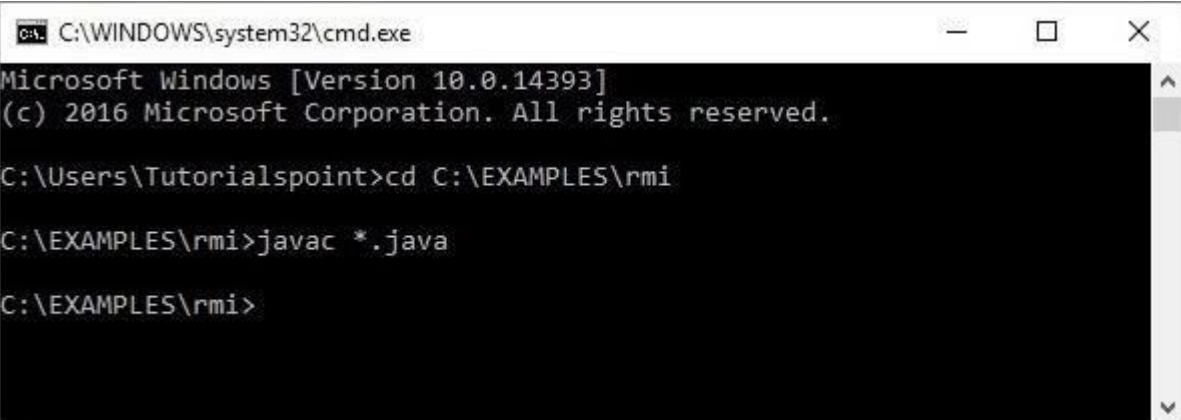
To compile the application –

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

Or,

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

```
Javac *.java
```



The screenshot shows a Windows Command Prompt window titled "C:\WINDOWS\system32\cmd.exe". The text inside the window is as follows:

```

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

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

C:\EXAMPLES\rmi>javac *.java

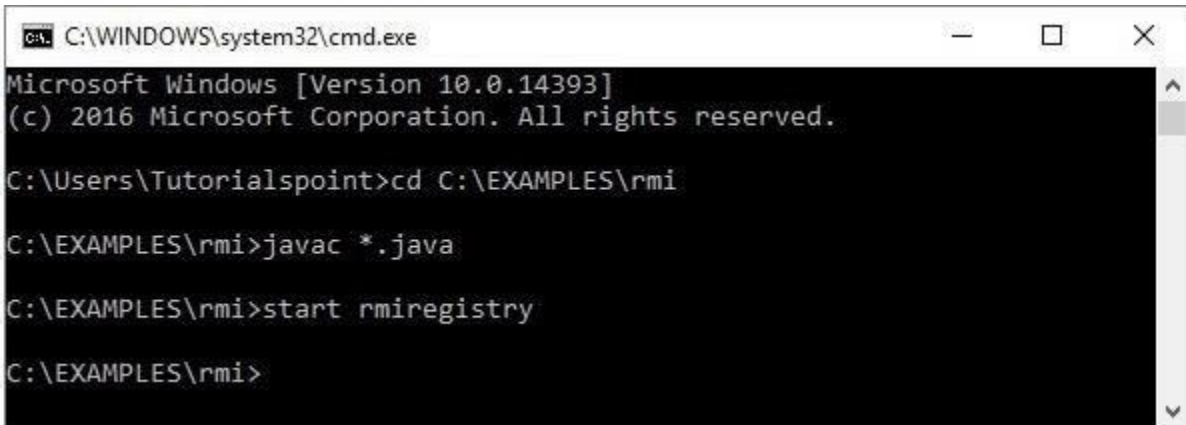
C:\EXAMPLES\rmi>

```

## Executing the Application

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

```
start rmiregistry
```



```

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

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

C:\EXAMPLES\rmi>javac *.java

C:\EXAMPLES\rmi>start rmiregistry

C:\EXAMPLES\rmi>

```

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



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

Java Server



```

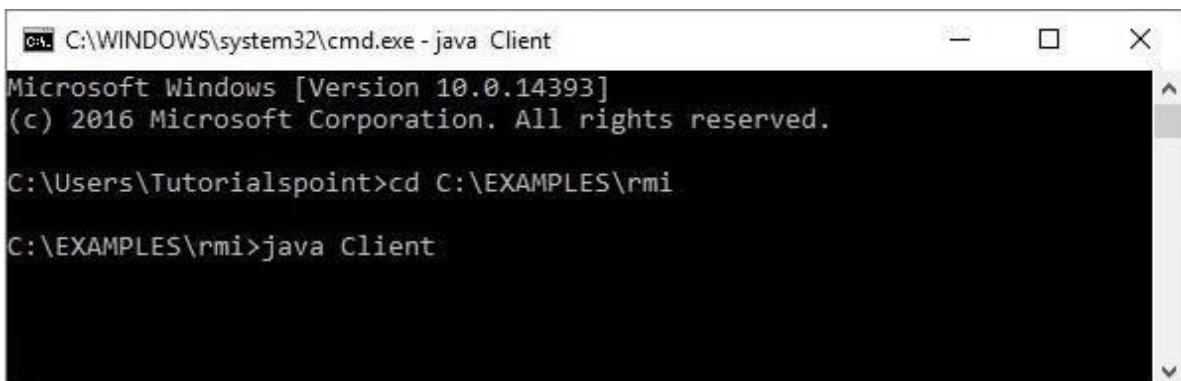
C:\WINDOWS\system32\cmd.exe - java Server

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

```

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

java Client



```

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

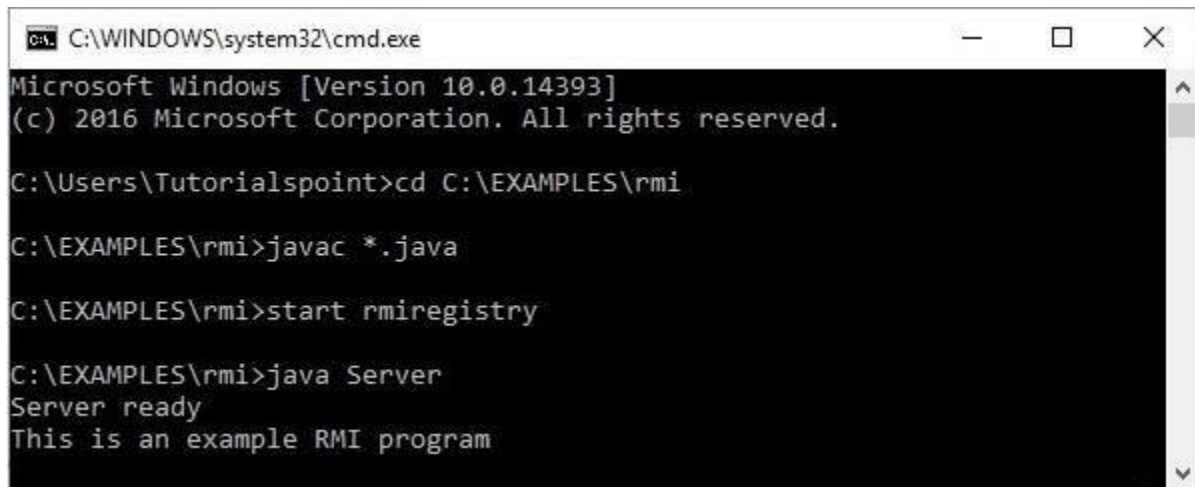
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

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

C:\EXAMPLES\rmi>java Client

```

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



```

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

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

C:\EXAMPLES\rmi>javac *.java

C:\EXAMPLES\rmi>start rmiregistry

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

## RMI applications

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

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

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

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

### Marshalling and Unmarshalling

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

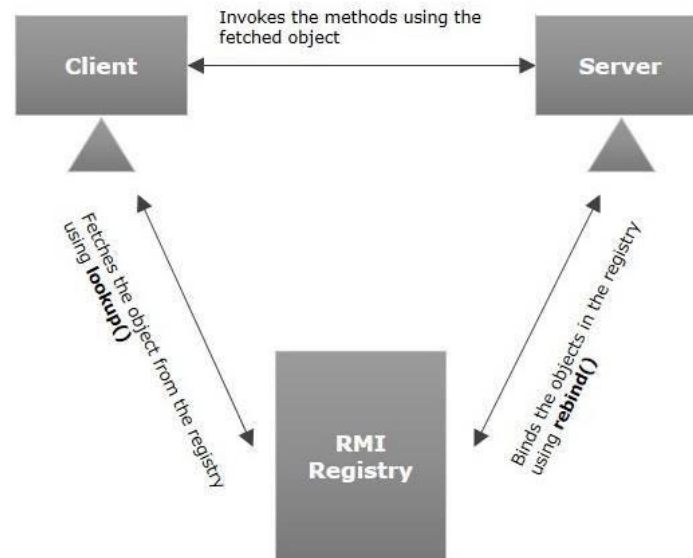
At the server side, the packed parameters are unbundled and then the required method is invoked. This process is known as **unmarshalling**.

### RMI Registry

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

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

The following illustration explains the entire process –



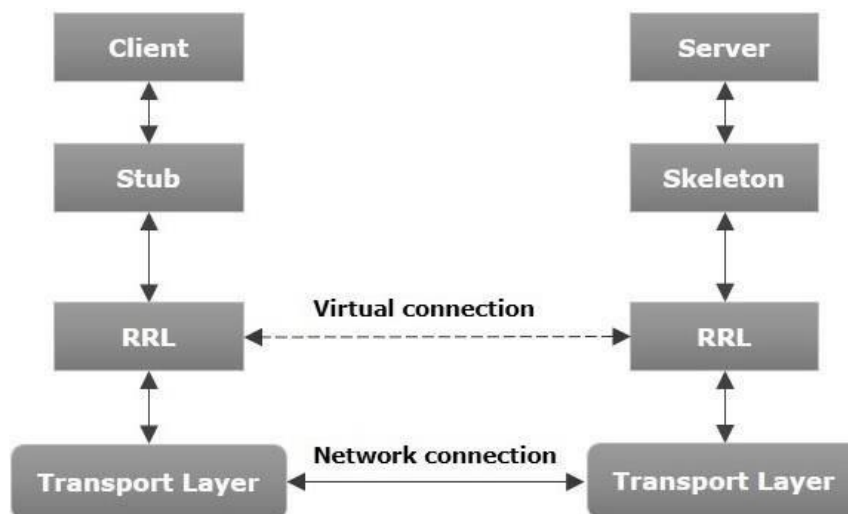
## Components of RMI

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

**Stub** – A stub is a representation (proxy) of the remote object at client. It resides in the client system; it acts as a gateway for the client program.

**Skeleton** – This is the object which resides on the server side. stub communicates with this skeleton to pass request to the remote object.

**RRL(Remote Reference Layer)** – It is the layer which manages the references made by the client to the remote object.



## Implementation class for remote object of RMI

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

To develop an implementation class –

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

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

```
// Implementing the remote interface
```

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

## Unit V JSP - JavaBeans

A JavaBean is a specially constructed Java class written in the Java and coded according to the JavaBeans API specifications.

Following are the unique characteristics that distinguish a JavaBean from other Java classes –

- It provides a default, no-argument constructor.
- It should be serializable and that which can implement the **Serializable** interface.
- It may have a number of properties which can be read or written.
- It may have a number of "**getter**" and "**setter**" methods for the properties.

### JavaBeans Properties

A JavaBean property is a named attribute that can be accessed by the user of the object. The attribute can be of any Java data type, including the classes that you define.

A JavaBean property may be **read**, **write**, **read only**, or **write only**. JavaBean properties are accessed through two methods in the JavaBean's implementation class

–

S.No.	Method & Description
1	<b>getPropertyName()</b> For example, if property name is <i>firstName</i> , your method name would be <b>getFirstName()</b> to read that property. This method is called accessor.
2	<b>setPropertyName()</b> For example, if property name is <i>firstName</i> , your method name would be <b>setFirstName()</b> to write that property. This method is called mutator.

A read-only attribute will have only a **getPropertyName()** method, and a write-only attribute will have only a **setPropertyName()** method.

### JavaBeans Example

Consider a student class with few properties –

```
package com.tutorialspoint;
```

```
public class StudentsBean implements java.io.Serializable {
    private String firstName = null;

    private String lastName = null;
```



```

private int age = 0;

public StudentsBean() {
}

public String getFirstName(){
    return firstName;
}

public String getLastName(){
    return lastName;
}

public int getAge(){
    return age;
}

public void setFirstName(String firstName){
    this.firstName = firstName;
}

public void setLastName(String lastName){
    this.lastName = lastName;
}

public void setAge(Integer age){
    this.age = age;
}
}

```

## Accessing JavaBeans

The **useBean** action declares a JavaBean for use in a JSP. Once declared, the bean becomes a scripting variable that can be accessed by both scripting elements and other custom tags used in the JSP. The full syntax for the useBean tag is as follows –

```
<jsp:useBean id = "bean's name" scope = "bean's scope" typeSpec/>
```

Here values for the scope attribute can be a **page**, **request**, **session** or **application** based on your requirement. The value of the **id** attribute may be any value as long as it is a unique name among other **useBean declarations** in the same JSP.

Following example shows how to use the useBean action –

```

<html>

<head>

```

```
<title>useBean Example</title>
</head>

<body>
  <jsp:useBean id = "date" class = "java.util.Date" />
  <p>The date/time is <%= date %>
</body>
</html>
```

You will receive the following result – –

The date/time is Thu Sep 30 11:18:11 GST 2010

## Accessing JavaBeans Properties

Along with **<jsp:useBean...>** action, you can use the **<jsp:getProperty/>** action to access the get methods and the **<jsp:setProperty/>** action to access the set methods. Here is the full syntax –

```
<jsp:useBean id = "id" class = "bean's class" scope = "bean's scope">

    <jsp:setProperty name = "bean's id" property = "property name"
        value = "value"/>

    <jsp:getProperty name = "bean's id" property = "property name"/>

    .....

</jsp:useBean>
```

The name attribute references the id of a JavaBean previously introduced to the JSP by the useBean action. The property attribute is the name of the **get** or the **set** methods that should be invoked.

Following example shows how to access the data using the above syntax –

```
<html>

<head>

    <title>get and set properties Example</title>

</head>

<body>

    <jsp:useBean id = "students" class = "com.tutorialspoint.StudentsBean">

        <jsp:setProperty name = "students" property = "firstName" value = "Zara"/>

        <jsp:setProperty name = "students" property = "lastName" value = "Ali"/>

        <jsp:setProperty name = "students" property = "age" value = "10"/>

    </jsp:useBean>

    <p>Student First Name:

        <jsp:getProperty name = "students" property = "firstName"/>

    </p>

    <p>Student Last Name:

        <jsp:getProperty name = "students" property = "lastName"/>
```

</p>

<p>Student Age:

<jsp:getProperty name = "students" property = "age"/>

</p>

</body>

</html>

Let us make the **StudentsBean.class** available in CLASSPATH. Access the above JSP. the following result will be displayed –

Student First Name: Zara

Student Last Name: Ali

Student Age: 10

## JSP - Custom Tags

A custom tag is a user-defined JSP language element. When a JSP page containing a custom tag is translated into a servlet, the tag is converted to operations on an object called a tag handler. The Web container then invokes those operations when the JSP page's servlet is executed.

JSP tag extensions lets you create new tags that you can insert directly into a JavaServer Page. The JSP 2.0 specification introduced the Simple Tag Handlers for writing these custom tags.

To write a custom tag, you can simply extend **SimpleTagSupport** class and override the **doTag()** method, where you can place your code to generate content for the tag.

## Create "Hello" Tag

Consider you want to define a custom tag named `<ex:Hello>` and you want to use it in the following fashion without a body –

```
<ex:Hello />
```

To create a custom JSP tag, you must first create a Java class that acts as a tag handler. Let us now create the **HelloTag** class as follows –

```
package com.tutorialspoint;

import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
import java.io.*;

public class HelloTag extends SimpleTagSupport {

    public void doTag() throws JspException, IOException {
        JspWriter out = getJspContext().getOut();
        out.println("Hello Custom Tag!");
    }
}
```

The above code has simple coding where the **doTag()** method takes the current **JspContext** object using the **getJspContext()** method and uses it to send **"Hello Custom Tag!"** to the current **JspWriter** object

Let us compile the above class and copy it in a directory available in the environment variable CLASSPATH. Finally, create the following tag library file: **<Tomcat- Installation-Directory>webapps\ROOT\WEB-INF\custom.tld**.

```
<taglib>

  <tlib-version>1.0</tlib-version>

  <jsp-version>2.0</jsp-version>

  <short-name>Example TLD</short-name>


  <tag>

    <name>Hello</name>

    <tag-class>com.tutorialspoint.HelloTag</tag-class>

    <body-content>empty</body-content>

  </tag>

</taglib>
```

Let us now use the above defined custom tag **Hello** in our JSP program as follows –

```
<%@ taglib prefix = "ex" uri = "WEB-INF/custom.tld"%>


<html>

  <head>

    <title>A sample custom tag</title>

  </head>


  <body>

    <ex:Hello/>

  </body>

</html>
```

Call the above JSP and this should produce the following result –

Hello Custom Tag!

## JSP - Expression Language (EL)

JSP Expression Language (EL) makes it possible to easily access application data stored in JavaBeans components. JSP EL allows you to create expressions both **(a)** arithmetic and **(b)** logical. Within a JSP EL expression, you can use **integers, floating point numbers, strings, the built-in constants true and false** for boolean values, and null.

## Simple Syntax

Typically, when you specify an attribute value in a JSP tag, you simply use a string. For example –

```
<jsp:setProperty name = "box" property = "perimeter" value =  
"100"/>
```

JSP EL allows you to specify an expression for any of these attribute values. A simple syntax for JSP EL is as follows –

```
${expr}
```

Here **expr** specifies the expression itself. The most common operators in JSP EL are `.` and `[]`. These two operators allow you to access various attributes of Java Beans and built-in JSP objects.

For example, the above syntax **<jsp:setProperty>** tag can be written with an expression like –

```
<jsp:setProperty name = "box" property = "perimeter"
    value = "${2*box.width+2*box.height}"/>
```

When the JSP compiler sees the **`${}`** form in an attribute, it generates code to evaluate the expression and substitutes the value of expression.

You can also use the JSP EL expressions within template text for a tag. For example, the **<jsp:text>** tag simply inserts its content within the body of a JSP. The following **<jsp:text>** declaration inserts **<h1>Hello JSP!</h1>** into the JSP output –

```
<jsp:text>
    <h1>Hello JSP!</h1>
</jsp:text>
```

You can now include a JSP EL expression in the body of a **<jsp:text>** tag (or any other tag) with the same **`${}`** syntax you use for attributes. For example –

```
<jsp:text>
    Box Perimeter is: ${2*box.width + 2*box.height}
</jsp:text>
```

EL expressions can use parentheses to group subexpressions. For example, **`${(1 + 2) * 3}` equals 9, but `${1 + (2 * 3)}` equals 7.**

To deactivate the evaluation of EL expressions, we specify the **isELIgnored** attribute of the page directive as below –

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

The valid values of this attribute are true and false. If it is true, EL expressions are ignored when they appear in static text or tag attributes. If it is false, EL expressions are evaluated by the container.

## Basic Operators in EL

JSP Expression Language (EL) supports most of the arithmetic and logical operators supported by Java. Following table lists out the most frequently used operators –

S.No.	Operator & Description
-------	------------------------



1	<b>.</b> Access a bean property or Map entry
2	<b>[]</b> Access an array or List element
3	<b>()</b> Group a subexpression to change the evaluation order
4	<b>+</b> Addition
5	<b>-</b> Subtraction or negation of a value
6	<b>*</b> Multiplication
7	<b>/ or div</b> Division
8	<b>% or mod</b> Modulo (remainder)
9	<b>== or eq</b> Test for equality
10	<b>!= or ne</b> Test for inequality
11	<b>&lt; or lt</b> Test for less than
12	<b>&gt; or gt</b> Test for greater than

13	<b>&lt;= or le</b> Test for less than or equal
14	<b>&gt;= or ge</b> Test for greater than or equal
15	<b>&amp;&amp; or and</b> Test for logical AND
16	<b>   or or</b> Test for logical OR
17	<b>! or not</b> Unary Boolean complement
18	<b>empty</b> Test for empty variable values

## Functions in JSP EL

JSP EL allows you to use functions in expressions as well. These functions must be defined in the custom tag libraries. A function usage has the following syntax –

```
${ns:func(param1, param2, ...)}
```

Where **ns** is the namespace of the function, **func** is the name of the function and **param1** is the first parameter value. For example, the function **fn:length**, which is part of the JSTL library. This function can be used as follows to get the length of a string.

```
${fn:length("Get my length")}
```

To use a function from any tag library (standard or custom), you must install that library on your server and must include the library in your JSP using the **<taglib>** directive as explained in the JSTL chapter.

## JSP EL Implicit Objects

The JSP expression language supports the following implicit objects –

S.No	Implicit object & Description
------	-------------------------------

1	<b>pageScope</b> Scoped variables from page scope
2	<b>requestScope</b> Scoped variables from request scope
3	<b>sessionScope</b> Scoped variables from session scope
4	<b>applicationScope</b> Scoped variables from application scope
5	<b>param</b> Request parameters as strings
6	<b>paramValues</b> Request parameters as collections of strings
7	<b>header</b> HTTP request headers as strings
8	<b>headerValues</b> HTTP request headers as collections of strings
9	<b>initParam</b> Context-initialization parameters
10	<b>cookie</b> Cookie values
11	<b>pageContext</b> The JSP PageContext object for the current page

You can use these objects in an expression as if they were variables. The examples that follow will help you understand the concepts –

## The pageContext Object

The `pageContext` object gives you access to the `pageContext` JSP object. Through the `pageContext` object, you can access the request object. For example, to access the incoming query string for a request, you can use the following expression –

```
${pageContext.request.queryString}
```

## The Scope Objects

The **`pageScope`**, **`requestScope`**, **`sessionScope`**, and **`applicationScope`** variables provide access to variables stored at each scope level.

For example, if you need to explicitly access the `box` variable in the application scope, you can access it through the `applicationScope` variable as **`applicationScope.box`**.

## The param and paramValues Objects

The `param` and `paramValues` objects give you access to the parameter values normally available through the **`request.getParameter`** and **`request.getParameterValues`** methods.

For example, to access a parameter named `order`, use the expression **`${param.order}`** or **`${param["order"]}`**.

Following is the example to access a request parameter named `username` –

```
<%@ page import = "java.io.*,java.util.*" %>
<%String title = "Accessing Request Param";%>

<html>
<head>
<title><% out.print(title); %></title>
</head>

<body>
<center>
<h1><% out.print(title); %></h1>
</center>

<div align = "center">
<p>${param["username"]}</p>
</div>
</body>
```

</html>

The `param` object returns single string values, whereas the `paramValues` object returns string arrays.

## header and headerValues Objects

The header and headerValues objects give you access to the header values normally available through the **request.getHeader** and the **request.getHeaders** methods.

For example, to access a header named user-agent, use the expression **`${header.user-agent}`** or **`${header["user-agent"]}`**.

Following is the example to access a header parameter named user-agent –

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

<%String title = "User Agent Example";%>

<html>

<head>

<title><% out.print(title); %></title>

</head>

<body>

<center>

<h1><% out.print(title); %></h1>

</center>

<div align = "center">

<p>${header["user-agent"]}</p>

</div>

</body>

</html>
```

## What is Java Networking?

Networking supplements a lot of power to simple programs. With networks, a single program can regain information stored in millions of computers positioned anywhere in the world.

Java is the leading programming language composed from scratch with networking in mind.

Java Networking is a notion of combining two or more computing devices together to share resources.

All the Java program communications over the network are done at the application layer.

The **java.net** package of the J2SE APIs comprises various classes and interfaces that execute the low-level communication features, enabling the user to formulate programs that focus on resolving the problem.

## Common Network Protocols

As stated earlier, the **java.net** package of the Java programming language includes various classes and interfaces that provide an easy-to-use means to access network resources.

Other than classes and interfaces, the **java.net** package also provides support for the two well-known network protocols. These are:

1. **Transmission Control Protocol (TCP)** – TCP or Transmission Control Protocol allows secure communication between different applications. TCP is a connection-oriented protocol which means that once a connection is established, data can be transmitted in two directions. This protocol is typically used over the Internet Protocol. Therefore, TCP is also referred to as TCP/IP. TCP has built-in methods to examine for errors and ensure the delivery of data in the order it was sent, making it a complete protocol for transporting information like still images, data files, and web pages.
2. **User Datagram Protocol (UDP)** – UDP or User Datagram Protocol is a connection-less protocol that allows data packets to be transmitted between different applications. UDP is a simpler Internet protocol in which error-checking and recovery services are not required. In UDP, there is no overhead for opening a connection, maintaining a connection, or terminating a connection. In UDP, the data is continuously sent to the recipient, whether they receive it or not.

## Java Networking Terminology

In Java Networking, many terminologies are used frequently. These widely used Java Networking Terminologies are given as follows:

1. **IP Address** – An IP address is a unique address that distinguishes a device on the internet or a local network. IP stands for “Internet Protocol.” It comprises a set of rules governing the format of data sent via the internet or local network. IP Address is referred to as a logical address that can be modified. It is composed of octets. The range of each octet varies from 0 to 255.
  - Range of the IP Address – 0.0.0.0 to 255.255.255.255
  - For Example – 192.168.0.1
2. **Port Number** – A port number is a method to recognize a particular process connecting internet or other network information when it reaches a server. The port number is used to identify different applications uniquely. The port number behaves as a communication endpoint among applications. The port number is correlated with the IP address for transmission and communication among two applications. There are 65,535 port numbers, but not all are used every day.
3. **Protocol** – A network protocol is an organized set of commands that define how data is transmitted between different devices in the same network. Network protocols are the reason through which a user can easily communicate with people all over the world and thus play a critical role in modern digital communications. For Example – TCP, FTP, POP, etc.
4. **MAC Address** – MAC address stands for Media Access Control address. It is a bizarre identifier that is allocated to a NIC (Network Interface Controller/ Card). It contains a 48 bit or 64-bit address, which is combined with the network adapter. MAC address can be in hexadecimal composition. In simple words, a MAC address is a unique number that is used to track a device in a network.
5. **Socket** – A socket is one endpoint of a two-way communication connection between the two applications running on the network. The socket mechanism presents a method of inter-process communication (IPC) by setting named contact points between which the communication occurs. A socket is tied to a port number so that the TCP layer can recognize the application to which the data is intended to be sent.
6. **Connection-oriented and connection-less protocol** – In a connection-oriented service, the user must establish a connection before starting the communication. When the connection is



established, the user can send the message or the information, and after this, they can release the connection. However, In connectionless protocol, the data is transported in one route from source to destination without verifying that the destination is still there or not or if it is ready to receive the message. Authentication is not needed in the connectionless protocol.

- Example of Connection-oriented Protocol – Transmission Control Protocol (TCP)
- Example of Connectionless Protocol – User Datagram Protocol (UDP)

## Java Networking classes

The **java.net** package of the Java programming language includes various classes that provide an easy-to-use means to access network resources. The classes covered in the **java.net** package are given as follows –

1. **CacheRequest** – The CacheRequest class is used in java whenever there is a need to store resources in ResponseCache. The objects of this class provide an edge for the OutputStream object to store resource data into the cache.
2. **CookieHandler** – The CookieHandler class is used in Java to implement a callback mechanism for securing up an HTTP state management policy implementation inside the HTTP protocol handler. The HTTP state management mechanism specifies the mechanism of how to make HTTP requests and responses.
3. **CookieManager** – The CookieManager class is used to provide a precise implementation of CookieHandler. This class separates the storage of cookies from the policy surrounding accepting and rejecting cookies. A CookieManager comprises a CookieStore and a CookiePolicy.
4. **DatagramPacket** – The DatagramPacket class is used to provide a facility for the connectionless transfer of messages from one system to another. This class provides tools for the production of datagram packets for connectionless transmission applying the datagram socket class.
5. **InetAddress** – The InetAddress class is used to provide methods to get the IP address of any hostname. An IP address is expressed by a

32-bit or 128-bit unsigned number. InetAddress can handle both IPv4 and IPv6 addresses.

6. **Server Socket** – The ServerSocket class is used for implementing system-independent implementation of the server-side of a client/server Socket Connection. The constructor for ServerSocket class throws an exception if it can't listen on the specified port. For example – it will throw an exception if the port is already being used.
7. **Socket** – The Socket class is used to create socket objects that help the users in implementing all fundamental socket operations. The users can implement various networking actions such as sending, reading data, and closing connections. Each Socket object built using **java.net.Socket** class has been connected exactly with 1 remote host; for connecting to another host, a user must create a new socket object.
8. **DatagramSocket** – The DatagramSocket class is a network socket that provides a connection-less point for sending and receiving packets. Every packet sent from a datagram socket is individually routed and delivered. It can further be practiced for transmitting and accepting broadcast information. Datagram Sockets is Java's mechanism for providing network communication via UDP instead of TCP.
9. **Proxy** – A proxy is a changeless object and a kind of tool or method or program or system, which serves to preserve the data of its users and computers. It behaves like a wall between computers and internet users. A Proxy Object represents the Proxy settings to be applied with a connection.
10. **URL** – The URL class in Java is the entry point to any available sources on the internet. A Class URL describes a Uniform Resource Locator, which is a signal to a "resource" on the World Wide Web. A source can denote a simple file or directory, or it can indicate a more difficult object, such as a query to a database or a search engine.
11. **URLConnection** – The URLConnection class in Java is an abstract class describing a connection of a resource as defined by a similar URL. The URLConnection class is used for assisting two distinct yet interrelated purposes. Firstly it provides control on interaction with a server(especially an HTTP server) than a URL class. Furthermore, with a URLConnection, a user can verify the header transferred by the

server and can react consequently. A user can also configure header fields used in client requests using URLConnection.

## Java Networking Interfaces

The **java.net** package of the Java programming language includes various interfaces also that provide an easy-to-use means to access network resources. The interfaces included in the **java.net** package are as follows:

1. **CookiePolicy** – The CookiePolicy interface in the **java.net** package provides the classes for implementing various networking applications. It decides which cookies should be accepted and which should be rejected. In CookiePolicy, there are three pre-defined policy implementations, namely ACCEPT\_ALL, ACCEPT\_NONE, and ACCEPT\_ORIGINAL\_SERVER.
2. **CookieStore** – A CookieStore is an interface that describes a storage space for cookies. CookieManager combines the cookies to the CookieStore for each HTTP response and recovers cookies from the CookieStore for each HTTP request.
3. **FileNameMap** – The FileNameMap interface is an uncomplicated interface that implements a tool to outline a file name and a MIME type string. FileNameMap charges a filename map ( known as a mimetable) from a data file.
4. **SocketOption** – The SocketOption interface helps the users to control the behavior of sockets. Often, it is essential to develop necessary features in Sockets. SocketOptions allows the user to set various standard options.
5. **SocketImplFactory** – The SocketImplFactory interface defines a factory for SocketImpl instances. It is used by the socket class to create socket implementations that implement various policies.
6. **ProtocolFamily** – This interface represents a family of communication protocols. The ProtocolFamily interface contains a method known as name(), which returns the name of the protocol family.

## Socket Programming

**Java Socket programming** is practiced for communication between the applications working on different JRE. Sockets implement the communication

tool between two computers using TCP. Java Socket programming can either be connection-oriented or connection-less. In Socket Programming, Socket and ServerSocket classes are managed for connection-oriented socket programming. However, DatagramSocket and DatagramPacket classes are utilized for connection-less socket programming.

A client application generates a socket on its end of the communication and strives to combine that socket with a server. When the connection is established, the server generates an object of socket class on its communication end. The client and the server can now communicate by writing to and reading from the socket.

The **java.net.Socket** class describes a socket, and the **java.net.ServerSocket** class implements a tool for the server program to host clients and build connections with them.

### **Steps to establishing a TCP connection between two computing devices using Socket Programming**

The following are the steps that occur on establishing a TCP connection between two computers using socket programming are given as follows:

**Step 1** – The server instantiates a ServerSocket object, indicating at which port number communication will occur.

**Step 2** – After instantiating the ServerSocket object, the server requests the accept() method of the ServerSocket class. This program pauses until a client connects to the server on the given port.

**Step 3** – After the server is idling, a client instantiates an object of Socket class, defining the server name and the port number to connect to.

**Step 4** – After the above step, the constructor of the Socket class strives to connect the client to the designated server and the port number. If communication is authenticated, the client forthwith has a Socket object proficient in interacting with the server.

**Step 5** – On the server-side, the accept() method returns a reference to a new socket on the server connected to the client's socket.

After the connections are stabilized, communication can happen using I/O streams. Each object of a socket class has both an OutputStream and an InputStream. The client's OutputStream is correlated to the server's InputStream, and the client's InputStream is combined with the server's OutputStream. Transmission Control Protocol (TCP) is a two-way communication protocol. Hence information can be transmitted over both streams at the corresponding time.

### **Socket Class**

The **Socket class** is used to create socket objects that help the users in implementing all fundamental socket operations. The users can implement various networking actions such as sending, reading data, and closing connections. Each Socket object created using **java.net.Socket** class has been correlated specifically with 1 remote host. If a user wants to connect to another host, then he must build a new socket object.

### Methods of Socket Class

In Socket programming, both the client and the server have a Socket object, so all the methods under the Socket class can be invoked by both the client and the server. There are many methods in the Socket class.

Socket programming in [Java](#) is used for communication between the applications that are running on different [JRE](#).

It can be either connection-oriented or connectionless.

On the whole, a socket is a way to establish a connection between a client and a server.

## What is Socket Programming in Java?

*Socket programming* is a way of connecting two nodes on a network to communicate with each other.

One **socket** (node) listens on a particular port at an IP, while other *socket* reaches out to the other in order to form a connection.

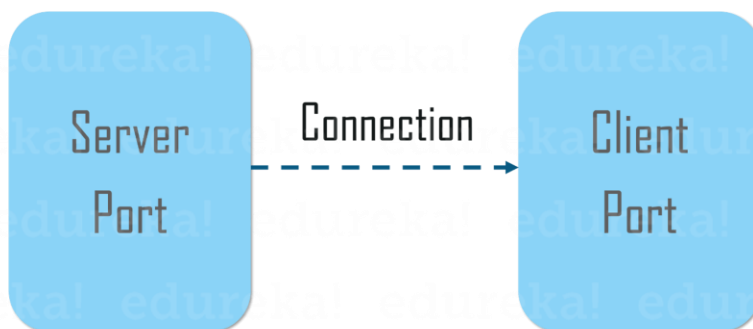


The server forms the listener *socket* while the client reaches out to the server. Socket and Server Socket [classes](#) are used for connection-oriented socket programming.

## What is a Socket in Java?

A **socket** in [Java](#) is one endpoint of a two-way communication link between two programs running on the network.

A **socket** is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to.



An endpoint is a combination of an IP address and a port number.

The package in the Java platform provides a class, Socket that implements one side of a two-way connection between your Java program and another program on the network.

The class sits on top of a platform-dependent implementation, hiding the details of any particular system from your Java program.

By using the class instead of relying on native code, your [Java programs](#) can communicate over the network in a platform-independent fashion.

Now that you know, what is Socket in Java, let's move further and understand how does client communicates with the server and how the server responds back.

## Client Side Programming

In the case of client-side programming, the client will first wait for the server to start.

Once the server is up and running, it will send the requests to the server. After that, the client will wait for the response from the server.

So, this is the whole logic of client and server communication.

Now let's understand the client side and server side programming in detail.

In order to initiate a clients request, you need to follow the below-mentioned steps:

### 1. Establish a Connection



The very first step is to establish a socket connection.

A socket connection implies that the two machines have information about each other's network location (IP Address) and TCP port.

You can create a Socket with the help of a below statement:

```
Socket socket = new Socket("127.0.0.1", 5000)
```

- Here, the first argument represents the **IP address of Server**.
- The second argument represents the **TCP Port**. (It is a number that represents which application should run on a server.)

## 2. Communication

In order to communicate over a socket connection, streams are used for both input and output the data.

After establishing a connection and sending the requests, you need to close the connection.

## 3. Closing the connection

The socket connection is closed explicitly once the message to the server is sent.

Now let's see how to write a Java program to implement socket connection at client side.

---

```

1 // A Java program for a ClientSide
2 import java.net.*;
3 import java.io.*;
4 public class ClientProgram
5 {
6 // initialize socket and input output streams
7 private Socket socket = null;
8 private DataInputStream input = null;
9 private DataOutputStream out = null;
10 // constructor to put ip address and port
11 public Client(String address, int port)
12 {
13 // establish a connection
14 try
15 {
16 socket = new Socket(address, port);
17 System.out.println("Connected");
18 // takes input from terminal

```



```

18input = new DataInputStream(System.in);
19// sends output to the socket
20out = new DataOutputStream(socket.getOutputStream());
21}
22catch(UnknownHostException u)
23{
24System.out.println(u);
25}
26catch(IOException i)
27{
28System.out.println(i);
29}
30// string to read message from input
31String line = "";
32// keep reading until "Over" is input
33while (!line.equals("Over"))
34{
35try
36{
37line = input.readLine();
38out.writeUTF(line);
39}
40catch(IOException i)
41{
42System.out.println(i);
43}
44}
45// close the connection
46try
47{
48input.close();
49out.close();
50socket.close();
51}
52catch(IOException i)
53{
54System.out.println(i);
55}
56}
57}
58
59
60

```

Now, let's implement server-side programming and then arrive at the output.

## Server Side Programming

Basically, the server will instantiate its object and wait for the client request. Once the client sends the request, the server will communicate back with the response.

In order to code the server-side application, you need two sockets and they are as follows:

- A **ServerSocket** which waits for the client requests (when a client makes a new `Socket()`)
- A plain old **socket** for communication with the client.

After this, you need to communicate with the client with the response.

### Communication

**getOutputStream()** method is used to send the output through the socket.

### Close the Connection

It is important to close the connection by closing the socket as well as input/output streams once everything is done.

Now let's see how to write a Java program to implement socket connection at server side.

---

```
// A Java program for a Serverside
1import java.net.*;
2import java.io.*;
3public class ServerSide
4{
5    //initialize socket and input stream
6    private Socket socket = null;
7    private ServerSocket server = null;
8    private DataInputStream in = null;
9    // constructor with port
10   public Server(int port)
11   {
12       // starts server and waits for a connection
13       try{
14           server = new ServerSocket(port);
15           System.out.println("Server started");
16           System.out.println("Waiting for a client ...");
17           socket = server.accept();
```

```

17System.out.println("Client accepted");
18// takes input from the client socket
19in = new DataInputStream(
20new BufferedInputStream(socket.getInputStream()));
21String line = "";
22// reads message from client until "Over" is sent
23while (!line.equals("Over"))
24{
25    try
26    {
27        line = in.readUTF();
28        System.out.println(line);
29
30
31    }
32    catch(IOException i)
33    {
34        System.out.println(i);
35    }
36}
37System.out.println("Closing connection");
38// close connection
39socket.close();
40in.close();
41}
42catch(IOException i){
43    System.out.println(i);
44}
45}
46public static void main(String args[]){
47    Server server = new Server(5000);
48}
49
50
51
52

```

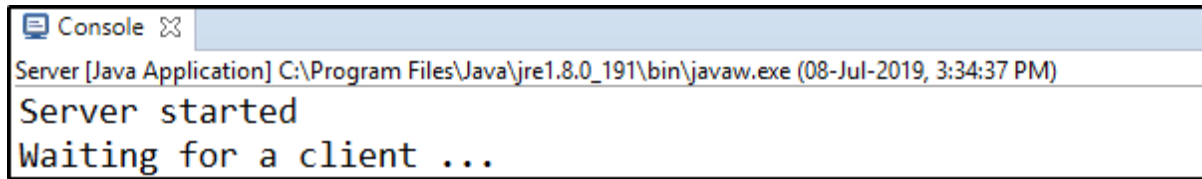
After configuring both client and server end, you can execute the server side program first.

After that, you need to run client side program and send the request.

As soon as the request is sent from the client end, server will respond back.

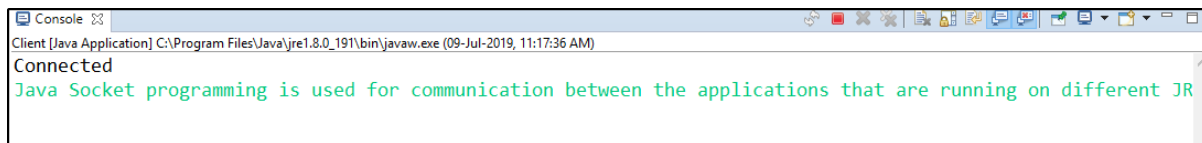
Below snapshot represents the same.

1. When you run the server side script, it will start and wait for the client to get started.

A screenshot of a Windows console window titled "Console". The text inside shows the execution of a Java application: "Server [Java Application] C:\Program Files\Java\jre1.8.0\_191\bin\javaw.exe (08-Jul-2019, 3:34:37 PM)", followed by "Server started" and "Waiting for a client ...".

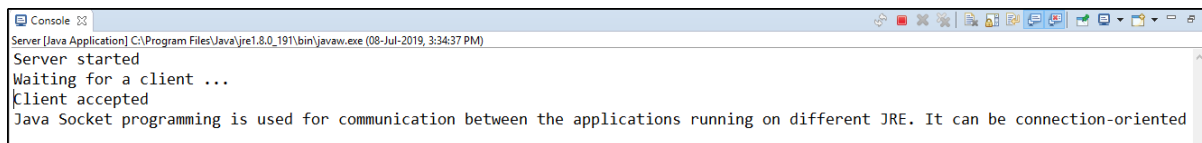
```
Console
Server [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe (08-Jul-2019, 3:34:37 PM)
Server started
Waiting for a client ...
```

2. Next, the client will get connected and inputs the request in the form of a string.

A screenshot of a Windows console window titled "Console". The text shows a client connection: "Client [Java Application] C:\Program Files\Java\jre1.8.0\_191\bin\javaw.exe (09-Jul-2019, 11:17:36 AM)", followed by "Connected" and a green-colored line of text: "Java Socket programming is used for communication between the applications that are running on different JR".

```
Console
Client [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe (09-Jul-2019, 11:17:36 AM)
Connected
Java Socket programming is used for communication between the applications that are running on different JR
```

3. When the client sends the request, the server will respond back.

A screenshot of a Windows console window titled "Console". The text shows the server's response to the client: "Server [Java Application] C:\Program Files\Java\jre1.8.0\_191\bin\javaw.exe (08-Jul-2019, 3:34:37 PM)", followed by "Server started", "Waiting for a client ...", "Client accepted", and "Java Socket programming is used for communication between the applications running on different JRE. It can be connection-oriented".

```
Console
Server [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe (08-Jul-2019, 3:34:37 PM)
Server started
Waiting for a client ...
Client accepted
Java Socket programming is used for communication between the applications running on different JRE. It can be connection-oriented
```

## What is Proxy Server?

Proxy server refers to a server that acts as an intermediary between the request made by clients, and a particular server for some services or requests for some resources.

There are different types of proxy servers available that are put into use according to the purpose of a request made by the clients to the servers.

The basic purpose of Proxy servers is to protect the direct connection of Internet clients and internet resources.

The proxy server also prevents the identification of the client's IP address when the client makes any request is made to any other servers.

- **Internet Client and Internet resources:** For internet clients, Proxy servers also act as a shield for an internal network against the request coming from a client to access the data stored on the server. It makes the original IP address of the node remains hidden while accessing data from that server.
- **Protects true host identity:** In this method, outgoing traffic appears to come from the proxy server rather than internet navigation. It must be configured to the specific application such as HTTPs or FTP.  
For example, organizations can use a proxy to observe the traffic of its employees to get the work efficiently done.  
It can also be used to keep a check on any kind of highly confidential data leakage. Some can also use it to increase their websites rank.

## Need Of Private Proxy:

1. **Defeat Hackers:** To protect organizations data from malicious use, passwords are used and different architects are setup, but still, there may be a possibility that this information can be hacked in case the IP address is accessible easily. To prevent such kind of misuse of Data Proxy servers are set up to prevent tracking of original IP addresses instead data is shown to come from a different IP address.
2. **Filtering of Content:** By caching the content of the websites, Proxy helps in fast access to the data that has been accessed very often.

3. **Examine Packet headers and Payloads:** Payloads and packet headers of the requests made by the user nodes in the internal server to access to social websites can be easily tracked and restricted.
4. **To control internet usage of employees and children:** In this, the Proxy server is used to control and monitor how their employees or kids use the internet.  
Organizations use it, to deny access to a specific website and instead redirecting you with a nice note asking you to refrain from looking at said sites on the company network.
5. **Bandwidth savings and improved speeds:** Proxy helps organizations to get better overall network performance with a good proxy server.
6. **Privacy Benefits:** Proxy servers are used to browse the internet more privately. It will change the IP address and identify the information the web request contains.
7. **Security:** Proxy server is used to encrypt your web requests to keep prying eyes from reading your transactions as it provides top-level security.

## Types Of Proxy Server

1. **Reverse Proxy Server:** The job of a reverse proxy server to listen to the request made by the client and redirect to the particular web server which is present on different servers.  
Example – Listen for TCP port 80 website connections which are normally placed in a demilitarized zone (DMZ) zone for publicly accessible services but it also protects the true identity of the host. Moreover, it is transparent to external users as external users will not be able to identify the actual number of internal servers.  
So, it is the prime duty of reverse proxy to redirect the flow depending upon the configurations of internal servers.  
The request that is made to pass through the private network protected by firewalls will need a proxy server that is not abiding by any of the local policies.  
Such types of requests from the clients are completed using reverse proxy servers.  
This is also used to restrict the access of the clients to the confidential data residing on the particular servers.
2. **Web Proxy Server:** Web Proxy forwards the HTTP requests, only URL is passed instead of a path.  
The request is sent to particular the proxy server responds. Examples, Apache, HAP Proxy.

3. **Anonymous Proxy Server:** This type of proxy server does not make an original IP address instead these servers are detectable still provides rational anonymity to the client device.
4. **Highly Anonymity Proxy:** This proxy server does not allow the original IP address and it as a proxy server to be detected.
5. **Transparent Proxy:** This type of proxy server is unable to provide any anonymity to the client, instead, the original IP address can be easily detected using this proxy.  
But it is put into use to act as a cache for the websites.  
A transparent proxy when combined with gateway results in a proxy server where the connection requests are sent by the client , then IP are redirected.  
Redirection will occurs without the client IP address configuration.  
HTTP headers present on the server-side can easily detect its redirection .
6. **CGI Proxy:** CGI proxy server developed to make the websites more accessible. It accepts the requests to target URLs using a web form and after processing its result will be returned to the web browser.  
It is less popular due to some privacy policies like VPNs but it still receives a lot of requests also.  
Its usage got reduced due to excessive traffic that can be caused to the website after passing the local filtration and thus leads to damage to the organization.
7. **Suffix Proxy:** Suffix proxy server basically appends the name of the proxy to the URL.  
This type of proxy doesn't preserve any higher level of anonymity. It is used for bypassing the web filters.  
It is easy to use and can be easily implemented but is used less due to the more number of web filter present in it.
8. **Distorting Proxy:** Proxy servers are preferred to generate an incorrect original IP address of clients once being detected as a proxy server.  
To maintain the confidentiality of the Client IP address HTTP headers are used.
9. **Tor Onion Proxy:** This server aims at online anonymity to the user's personal information.  
It is used to route the traffic through various networks present worldwide to arise difficulty in tracking the users' address and prevent the attack of any anonymous activities.  
It makes it difficult for any person who is trying to track the original address.  
In this type of routing, the information is encrypted in a multi-folds layer.

At the destination, each layer is decrypted one by one to prevent the information to scramble and receive original content.

This software is open-source and free of cost to use.

10. **I2P Anonymous Proxy:** It uses encryption to hide all the communications at various levels.  
This encrypted data is then relayed through various network routers present at different locations and thus I2P is a fully distributed proxy. This software is free of cost and open source to use, It also resists the censorship.
11. **DNS Proxy:** DNS proxy take requests in the form of DNS queries and forward them to the Domain server where it can also be cached, moreover flow of request can also be redirected.

## How Does The Proxy Server Operates?

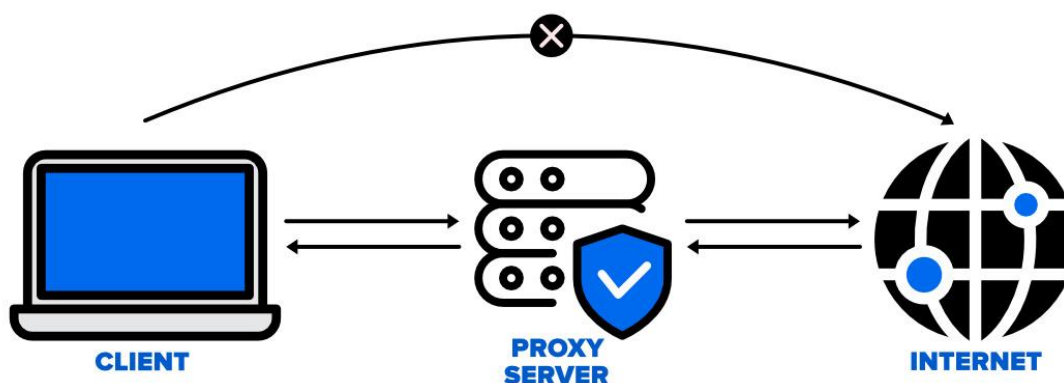
Every computer has its unique IP address which it uses to communicate with another node.

Similarly, the proxy server has its IP address that your computer knows. When a web request is sent, your request goes to the proxy server first.

The Proxy sends a request on your behalf to the internet and then collect the data and make it available to you.

A proxy can change your IP address So, the webserver will be unable to fetch your location in the world.

It protects data from getting hacked too. Moreover, it can block some web pages also.



## Disadvantages of Proxy Server



1. **Proxy Server Risks:** Free installation does not invest much in backend hardware or encryption.  
It will result in performance issues and potential data security issues. If you install a “free” proxy server, treat very carefully, some of those might steal your credit card numbers.
2. **Browsing history log:** The proxy server stores your original IP address and web request information in possibly unencrypted form and saved locally.  
Always check if your proxy server logs and saves that data – and what kind of retention or law enforcement cooperation policies they follow while saving data.
3. **No encryption:** No encryption means you are sending your requests as plain text.  
Anyone will be able to pull usernames and passwords and account information easily.  
Keep a check that proxy provides full encryption whenever you use it.

## Creating an Server-Client Application using the DatagramPacket and DatagramSocket classes

To create an application that uses UDP to establish the connection between a client and server, we need to perform the following steps:

- Create a server program
- Create a client program
- Execute the client and server program

Let's perform the steps in the following subsections:

### Creating the Server Program

Let's create the server class, named `UDPServerEx` which takes messages from a user and sends the messages (datagrams) to the clients. Listing 1 shows the code of the **`UDPServerEx.java`** file:

**Filename: `UDPServerEx.java`**

- Java

```
// A server that sends messages to the client
```

```
import java.net.*;
```

```
class UDPServerEx {
```

```

public static DatagramSocket mySocket;
public static byte myBuffer[] = new byte[2000];

public static void serverMethod() throws Exception
{
    int position = 0;
    while (true) {
        int charData = System.in.read();
        switch (charData) {
            case -1:
                System.out.println(
                    "The execution of "
                    + "the server has been terminated");
                return;
            case '\r':
                break;
            case '\n':
                mySocket.send(
                    new DatagramPacket(
                        myBuffer,
                        position,
                        InetAddress.getLocalHost(),
                        777));
                position = 0;
                break;
            default:
                myBuffer[position++]
                    = (byte)charData;
        }
    }
}

public static void main(String args[]) throws Exception
{
    System.out.println("Please enter some text here");
    mySocket = new DatagramSocket(888);
    serverMethod();
}
}

```

**To compile the UDPServerEx.java file:**

D:\UDPExample>javac UDPServerEx.java

***Note: The path may vary according to where you save file.***

## Creating the ClientProgram

Let's create a client class, named UDPClient, which accepts the messages sent from the server, UDPServerEx class. The client then displays the messages received in the Command Prompt. Listing 2 shows the code of the **UDPClient.java** file:

**Filename: UDPClient.java**

- Java

```
// UDPClient that receives and
// displays messages sent from the server

import java.net.*;
class UDPClient {

    public static DatagramSocket mySocket;
    public static byte myBuffer[] = new byte[2000];

    public static void clientMethod() throws Exception
    {
        while (true) {
            DatagramPacket dataPacket
                = new DatagramPacket(myBuffer,
                                     myBuffer.length);
            mySocket.receive(dataPacket);
            System.out.println("Message Received :");
            System.out.println(
                new String(
                    dataPacket.getData(),
                    0,
                    dataPacket.getLength()));
        }
    }
    public static void main(String args[]) throws Exception
    {
        System.out.println(
            "You need to press CTRL+C"
            + " in order to quit.");
        mySocket = new DatagramSocket(777);
        clientMethod();
    }
}
```

**Use the following command to compile the UDPClient.java file:**

D:\UDPExample>javac UDPClient.java

## Output

**Note:** To execute the UDPServerEx and UDPClient classes, run the UDPServerEx.java and UDPClient.java in two separate Command Prompt windows. Remember, the UDPServerEx class is executed before the UDPClient class. Figure 1 shows the output of the UDP Server java and UDPClient.java files:

```

Command Prompt - java UDPServerEx
Microsoft Windows [Version 10.0.18362.356]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\aaashr>cd /d D:\UDPEXample

D:\UDPEXample>set path=C:\Program Files\Java\jdk-13\bin

D:\UDPEXample>java UDPServerEx
Please enter some text here
hi
how are you?

Command Prompt - java UDPClient
Microsoft Windows [Version 10.0.18362.356]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\aaashr>cd /d D:\UDPEXample

D:\UDPEXample>set path=C:\Program Files\Java\jdk-13\bin

D:\UDPEXample>java UDPClient
You need to press CTRL+C in order to quit.
Message Recieved :
hi
Message Recieved :
how are you?

```

Showing the Output of the UDPServerEx and UDPClient Classes

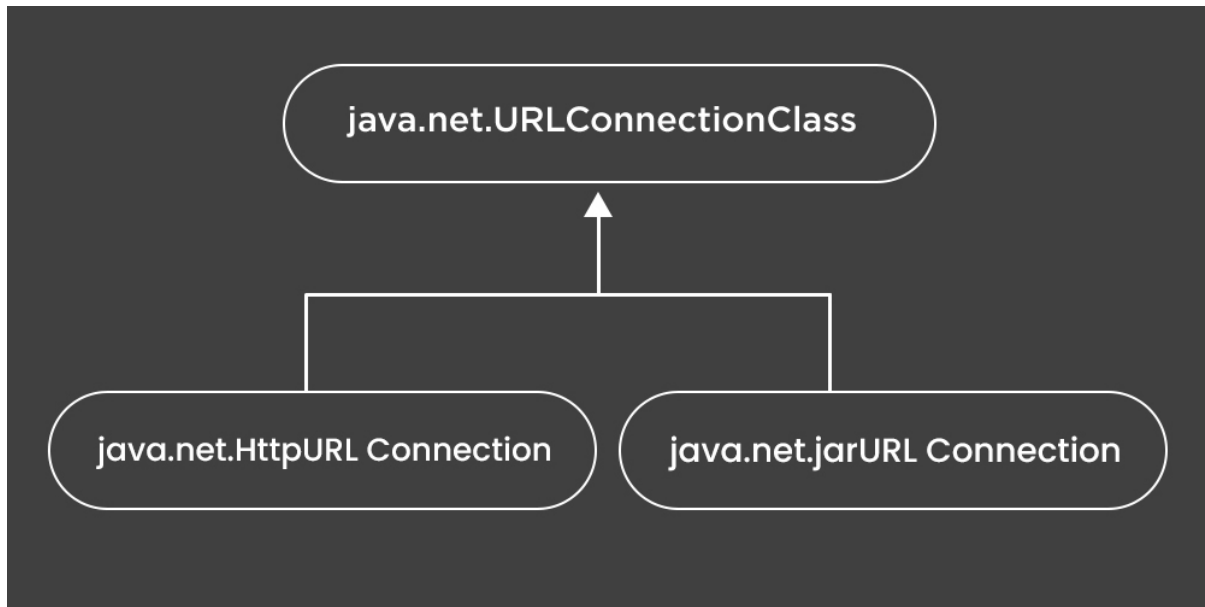
## java.net.URLConnection Class in Java

**URLConnection Class** in Java is an abstract class that represents a connection of a resource as specified by the corresponding URL. It is imported by the *java.net* package.

The [URLConnection class](#) is utilized for serving two different yet related purposes, Firstly it provides control on interaction with a server(especially an HTTP server) than URL class.

Secondly, with a URLConnection we can check the header sent by the server and respond accordingly, we can configure header fields used in client requests.

We can also download binary files by using URLConnection.



Method	Description
<code>addRequestProperty(String key, String value)</code>	This method is used for adding a general request property specified by key-value pair.
<code>connect()</code>	This method is used for establishing connection to the resource specified by URL, if such connection has not already been established.

---

<code>getAllowUserInteraction()</code>	This method returns the <code>allowUserInteraction</code> field for the object.
<code>getConnectTimeout()</code>	This method returns setting for connection time-out.
<code>getContent()</code>	This method is used for retrieving contents of <code>URLConnection</code> .
<code>getContent(Class[] classes)</code>	Retrieves the contents of this URL connection.

`getContentTypeEncoding()`

Returns the value of the content-encoding header field

`getContentTypeLength()`

Returns the value of the content-length header field.

`getContentTypeLengthLong()`

Returns the value of the content-length header field as a long.

`getContentType()`

Returns the value of the content-type header field

`getDate()`

Returns the value of the date header field.

`getDefaultAllowUserInteraction()`

Returns the default value of the `allowUserInteraction` field.

`getDefaultRequestProperty(String key)`

The instance specific `getRequestProperty` method should be used after an appropriate instance of `URLConnection` is obtained.



`getDefaultUseCaches()`

Returns the default value of a `URLConnection`'s `useCaches` flag.

`getDoInput()`

Returns the value of this `URLConnection`'s `doInput` flag.

`getDoOutput()`

Returns the value of this `URLConnection`'s `doOutput` flag.

`getExpiration()`

Returns the value of the expires header field.

---

## Example

- Java

```
// Java Program to demonstrate URLConnection class
```

```
// Importing input output classes
```

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

```
// Importing java.net package
```

```
// consisting of all network classes
```

```
import java.net.*;
```

```
// Main class
```

```
// URLConnectionExample
```

```

public class demo {

    // Main driver method
    public static void main(String[] args) throws Exception
    {
        // Try block to check for exceptions
        try {

            // Creating an object of URL class

            // Custom input URL is passed as an argument
            URL u = new URL("www.mitwpu.com");

            // Creating an object of URLConnection class to
            // communicate between application and URL
            URLConnection urlconnect = u.openConnection();

            // Creating an object of InputStream class
            // for our application streams to be read
            InputStream stream
                = urlconnect.getInputStream();

            // Declaring an integer variable
            int i;

            // Till the time URL is being read
            while ((i = stream.read()) != -1) {

                // Continue printing the stream
                System.out.print((char)i);
            }
        }

        // Catch block to handle the exception
        catch (Exception e) {

            // Print the exception on the console
            System.out.println(e);
        }
    }
}

```

### Output

```
java.net.MalformedURLException: no protocol: www.mitwpu.com
```

## Introducing Threads in Socket Programming in Java

### **Why to use threads in network programming?**

The reason is simple, we don't want only a single client to connect to server at a particular time but many clients simultaneously.

We want our architecture to **support multiple clients at the same time**.

For this reason, we must use threads on server side so that whenever a client request comes, a separate thread can be assigned for handling each request.

Let us take an example, suppose a Date-Time server is located at a place, say X.

Being a generic server, it does not serve any particular client, rather to a whole set of generic clients.

Also suppose at a particular time, two requests arrives at the server. With our basic server-client program, the request which comes even a nano-second first would be able to connect to the server and the other request would be rejected as no mechanism is provided for handling multiple requests simultaneously.

To overcome this problem, we use threading in network programming.

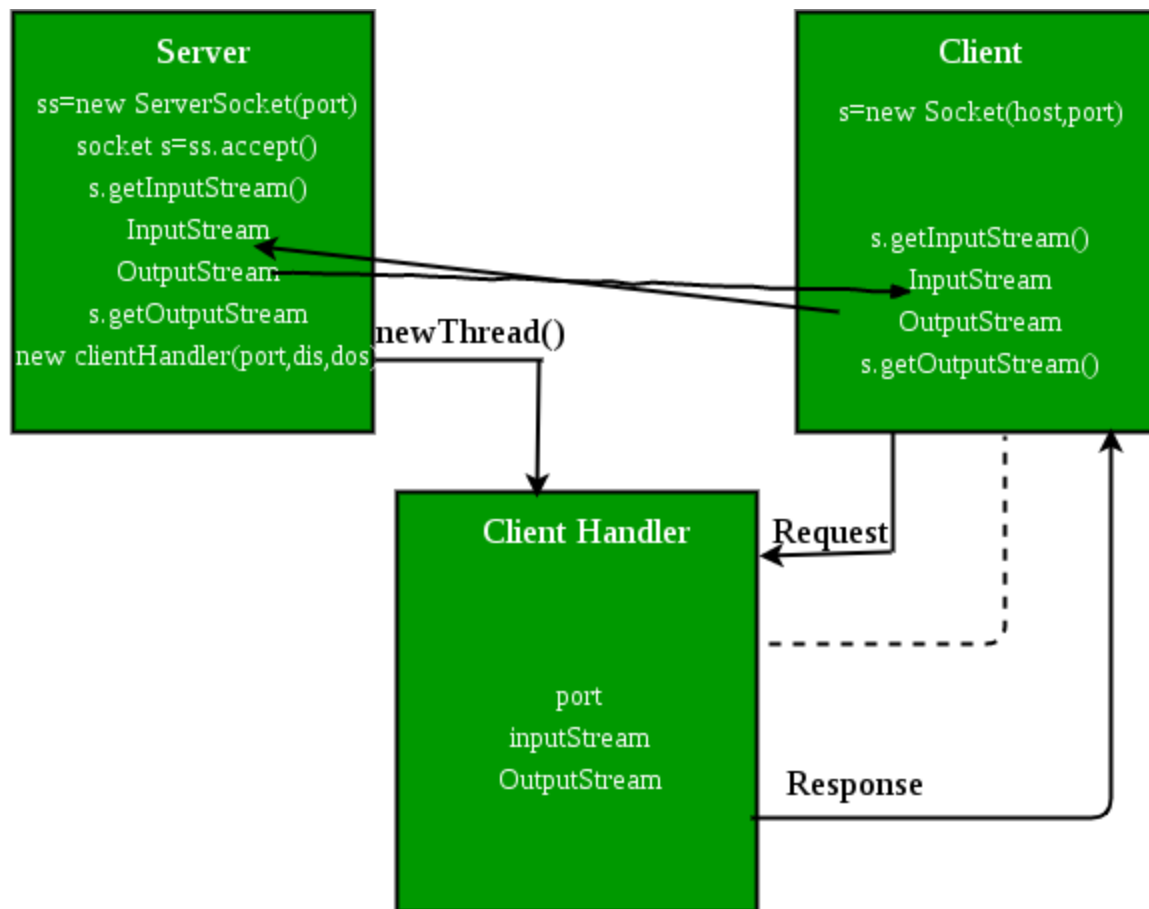
We shall see **creating a simple Date-Time server for handling multiple client requests** at the same time.

### Quick Overview

As normal, we will create two java files, **Server.java** and **Client.java**.

Server file contains two classes namely **Server** (public class for creating server) and **ClientHandler** (for handling any client using multithreading).

Client file contain only one public class **Client** (for creating a client). Below is the flow diagram of how these three classes interact with each other.



### Server Side Programming(Server.java)

- **Server class** : The steps involved on server side are similar to the article [Socket Programming in Java](#) with a slight change to create the thread object after obtaining the streams and port number.
  1. **Establishing the Connection:** Server socket object is initialized and inside a while loop a socket object continuously accepts incoming connection.
  2. **Obtaining the Streams:** The inputstream object and outputstream object is extracted from the current requests' socket object.

3. **Creating a handler object:** After obtaining the streams and port number, a new clientHandler object (the above class) is created with these parameters.
4. **Invoking the [start\(\)](#) method :** The start() method is invoked on this newly created thread object.
- **ClientHandler class :** As we will be using separate threads for each request, let's understand the working and implementation of the ClientHandler class extending Threads. An object of this class will be instantiated each time a request comes.
  1. First of all this class extends [Thread](#) so that its objects assume all properties of Threads.
  2. Secondly, the constructor of this class takes three parameters, which can uniquely identify any incoming request, i.e. a **Socket**, a [DataInputStream](#) to read from and a [DataOutputStream](#) to write to. Whenever we receive any request of client, the server extracts its port number, the DataInputStream object and DataOutputStream object and creates a new thread object of this class and invokes [start\(\)](#) method on it.  
*Note : Every request will always have a triplet of socket, input stream and output stream. This ensures that each object of this class writes on one specific stream rather than on multiple streams.*
  3. Inside the **run()** method of this class, it performs three operations: request the user to specify whether time or date needed, read the answer from input stream object and accordingly write the output on the output stream object.

### Client Side Programming (Client.java)

Client side programming is similar as in general socket programming program with the following steps-

1. **Establish a Socket Connection**
2. **Communication**

### **How these programs works together?**

1. When a client, say client1 sends a request to connect to server, the server assigns a new thread to handle this request. The newly assigned thread is given the access to streams for communicating with the client.
2. After assigning the new thread, the server via its while loop, again comes into accepting state.
3. When a second request comes while first is still in process, the server accepts this requests and again assigns a new thread for processing it. In this way, multiple requests can be handled even when some requests are in process.

### **How to test the above program on your system?**

Save the two programs in same package or anywhere. Then first run the Server.java followed by the Client.java. You can either copy the client program in two three separate files and run them individually, or if you have an IDE like eclipse, run multiple instances from the same program. The output shown above is from a single client program, the similar results will be achieved if multiple clients are used.

## Multi-threaded chat Application in Java

A simple date time server was created which handled multiple user requests at the same time using threading. It explains the basic concepts of threading in network programming. The same concepts can be used with very slight modification to extend the above idea and create a chatting application similar to facebook messenger, whatsapp, etc.

### Server Side Programming(Server.java)

**1. Server class :** The main server implementation is easy and similar to the previous article. The following points will help understand Server implementation :

1. The server runs an infinite loop to keep accepting incoming requests.
2. When a request comes, it assigns a new thread to handle the communication part.
3. The server also stores the client name into a vector, to keep a track of connected devices. The vector stores the thread object corresponding to the current request. The helper class uses this [vector](#) to find the name of recipient to which message is to be delivered. As this vector holds all the streams, handler class can use it to successfully deliver messages to specific clients.
4. Invoke the [start\(\)](#) method.

**2. ClientHandler class :** Similar to previous article, we create a helper class for handling various requests. This time, along with the socket and streams, we introduce a name variable. This will hold the name of the





```

// It contains two classes : Server and ClientHandler
// Save file as Server.java

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

// Server class
public class Server
{
    // Vector to store active clients
    static Vector<ClientHandler> ar = new Vector<>();

    // counter for clients
    static int i = 0;

    public static void main(String[] args) throws IOException
    {
        // server is listening on port 1234
        ServerSocket ss = new ServerSocket(1234);

        Socket s;

        // running infinite loop for getting
        // client request
        while (true)
        {
            // Accept the incoming request
            s = ss.accept();

            System.out.println("New client request received : " + s);

            // obtain input and output streams
            DataInputStream dis = new DataInputStream(s.getInputStream());
            DataOutputStream dos = new
DataOutputStream(s.getOutputStream());

            System.out.println("Creating a new handler for this client...");

            // Create a new handler object for handling this request.
            ClientHandler mtch = new ClientHandler(s,"client " + i, dis,
dos);

            // Create a new Thread with this object.

```

```

        Thread t = new Thread(mtch);

        System.out.println("Adding this client to active client list");

        // add this client to active clients list
        ar.add(mtch);

        // start the thread.
        t.start();

        // increment i for new client.
        // i is used for naming only, and can be replaced
        // by any naming scheme
        i++;
    }
}

// ClientHandler class
class ClientHandler implements Runnable
{
    Scanner scn = new Scanner(System.in);
    private String name;
    final DataInputStream dis;
    final DataOutputStream dos;
    Socket s;
    boolean isloggedin;

    // constructor
    public ClientHandler(Socket s, String name,
                        DataInputStream dis, DataOutputStream dos) {
        this.dis = dis;
        this.dos = dos;
        this.name = name;
        this.s = s;
        this.isloggedin=true;
    }

    @Override
    public void run() {

        String received;
        while (true)
        {

```

```

try
{
    // receive the string
    received = dis.readUTF();

    System.out.println(received);

    if(received.equals("logout")){
        this.isloggedin=false;
        this.s.close();
        break;
    }

    // break the string into message and recipient part
    StringTokenizer st = new StringTokenizer(received, "#");
    String MsgToSend = st.nextToken();
    String recipient = st.nextToken();

    // search for the recipient in the connected devices list.
    // ar is the vector storing client of active users
    for (ClientHandler mc : Server.ar)
    {
        // if the recipient is found, write on its
        // output stream
        if (mc.name.equals(recipient) && mc.isloggedin==true)
        {
            mc.dos.writeUTF(this.name+" : "+MsgToSend);
            break;
        }
    }
} catch (IOException e) {

    e.printStackTrace();
}

}
try
{
    // closing resources
    this.dis.close();
    this.dos.close();

} catch (IOException e){
    e.printStackTrace();
}

```

```
    }  
}
```

### Output:

```
New client request received :  
Socket[addr=/127.0.0.1,port=61818,localport=1234]  
Creating a new handler for this client...  
Adding this client to active client list  
New client request received :  
Socket[addr=/127.0.0.1,port=61819,localport=1234]  
Creating a new handler for this client...  
Adding this client to active client list
```

Implementation of client program for the multi-threaded chat application. Till now all examples in socket programming assume that client first sends some information and then server or other clients responds to that information.

In real world, this might not be the case. It is not required to send someone a message in order to be able to receive one. A client should readily receive a message whenever it is delivered to it i.e sending and receiving must be **implemented as separate activities rather than sequential**.

There is a very simple solution which uses threads to achieve this functionality. In the client side implementation we will be creating two threads:

1. **SendMessage** : This thread will be used for sending the message to other clients. The working is very simple, it takes input the message to send and the recipient to deliver to. Note that this implementation assumes the message to be of the format **message # recipient**, where recipient is the name of the recipient. It then writes the message on its output stream which is connected to the handler for this client. The handler breaks the message and recipient part and deliver to particular recipient. Lets look at how this thread can be implemented.

```
Thread sendMessage = new Thread(new Runnable() {
    @Override
    public void run() {
        while (true) {

            // read the message to deliver.
            String msg = sc.nextLine();
            try {

                // write on the output stream
                dos.writeUTF(msg);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
})
```

```
});
```

2. **readMessage** : A similar approach is taken for creating a thread for receiving the messages. When any client tries to write on this clients input stream, we use readUTF() method to read that message. The following snippet of how this thread is implemented is shown below-

```
Thread readMessage = new Thread(new Runnable() {

    @Override
    public void run() {

        while (true) {
            try {

                // read the message sent to this
                client

                String msg = dis.readUTF();
                System.out.println(msg);
            } catch (IOException e) {

                e.printStackTrace();
            }
        }
    }
});
```

The remaining steps of client side programming are similar to previous examples. A brief explanation is as follows –

1. **Establish a Socket Connection**
2. **Communication**

Communication occurs with the help of the readMessage and sendMessage threads. Separate threads for reading and writing ensures simultaneous sending and receiving of messages.

```

// Java implementation for multithreaded chat client
// Save file as Client.java

import java.io.*;
import java.net.*;
import java.util.Scanner;

public class Client
{
    final static int ServerPort = 1234;

    public static void main(String args[]) throws
UnknownHostException, IOException
    {
        Scanner scn = new Scanner(System.in);

        // getting localhost ip
        InetAddress ip = InetAddress.getByName("localhost");

        // establish the connection
        Socket s = new Socket(ip, ServerPort);

        // obtaining input and out streams
        DataInputStream dis = new
DataInputStream(s.getInputStream());
        DataOutputStream dos = new
DataOutputStream(s.getOutputStream());

        // sendMessage thread
        Thread sendMessage = new Thread(new Runnable()

```



```

{
    @Override
    public void run() {
        while (true) {

            // read the message to deliver.
            String msg = scn.nextLine();

            try {
                // write on the output stream
                dos.writeUTF(msg);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
});

// readMessage thread
Thread readMessage = new Thread(new Runnable()
{
    @Override
    public void run() {

        while (true) {
            try {
                // read the message sent to this client
                String msg = dis.readUTF();
                System.out.println(msg);
            } catch (IOException e) {

                e.printStackTrace();
            }
        }
    }
});

sendMessage.start();
readMessage.start();

}
}

```

**Output :****From client 0 :**

hello#client 1

client 1 : heya

how are you#client 1

client 1 : fine..how about you

logout

**From client 1 :**

client 0 : hello

heya#client 0

client 0 : how are you

fine..how about you#client 0

logout

**Important points :**

- To send a message from any client, type the message, followed by a “#” and then the name of the recipient client. Please note that this implementation gives names as “client 0”, “client 1”....“client n” and so carefully names must be appended int the end. After that press Enter key.
- Once a message is sent, the handler for this client will receive the message and it will be delivered to the specified client.
- If any client sends a message to this client, the readMessage thread will automatically print the message on the console.
- Once a client is done with chatting, he can send a “logout” message without any recipient name so that server would know that this client has logged off the system. It is recommended to send a logout message before closing the terminal for the client to avoid any errors.

**How to run the above program ?**

Similar to previous examples, first run the server and then run multiple instances of the client. From each of the client, try sending message to each other. Please make sure you send message to only a valid client, i.e. to the client available on active list.

### Suggested Improvements

This was only the explanation part as to how threads and socket programming can be used to create powerful programs. There are some suggested improvements to above implementations for the interested readers-

- Create a graphical user interface for clients for sending and receiving messages. A tool such as Netbeans can be used to quickly design an interface
- Currently the names are hard-coded as client 0, client 1. This can be improved to use user given nicknames.
- This implementation can be further enhanced to provide client the list of current active users so that he can know who all of his friends are online. A simple method can be implemented for this purpose which when invoked prints the names in active list.



*Collection*

---

*Framework*



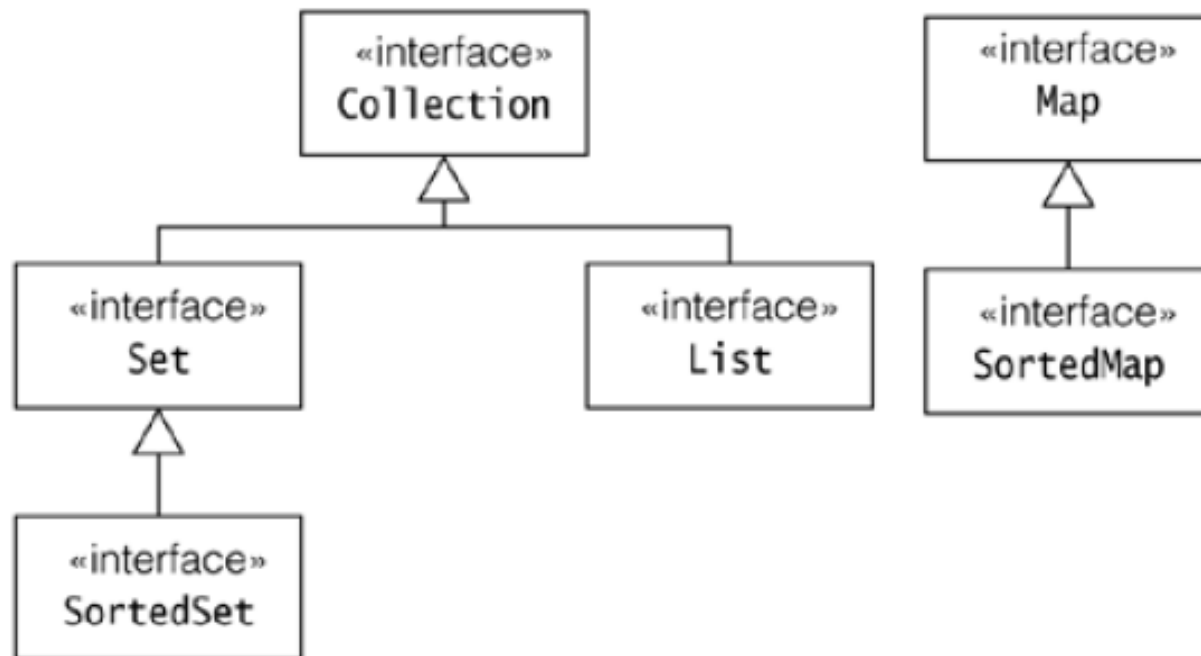


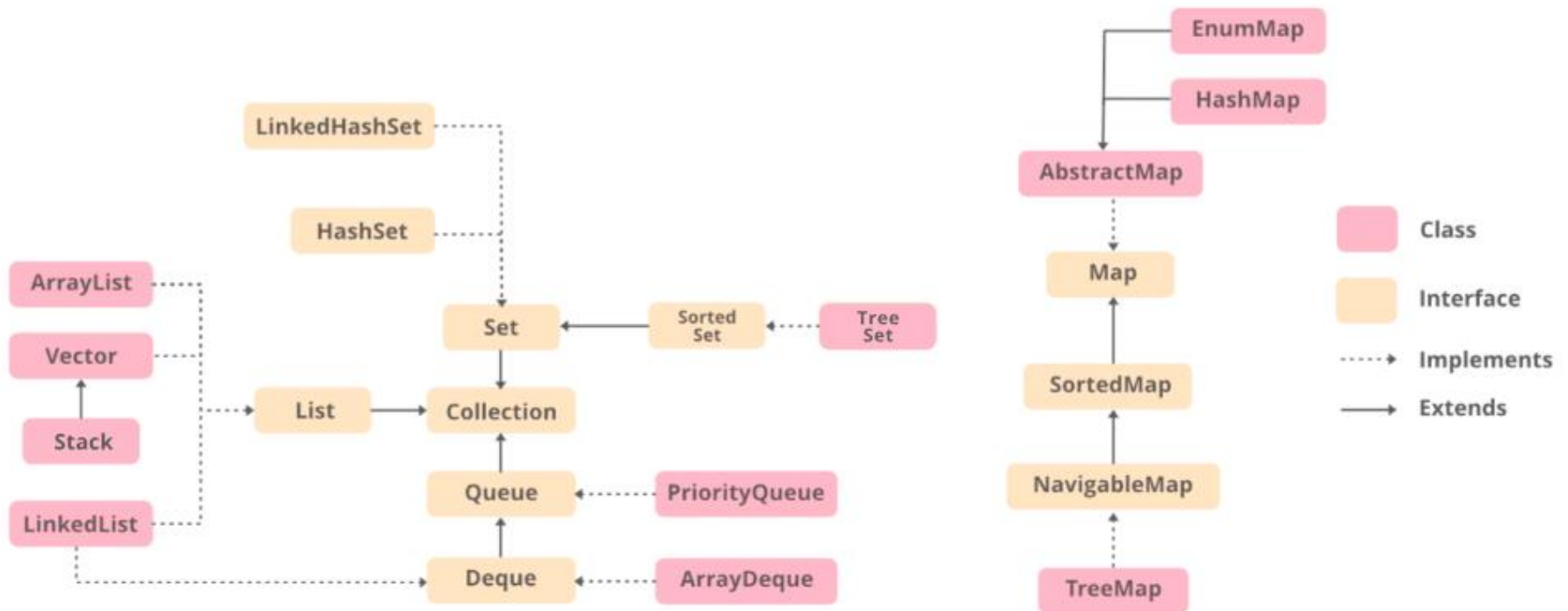




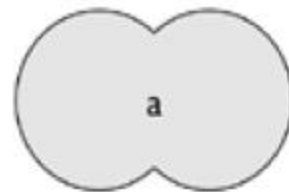
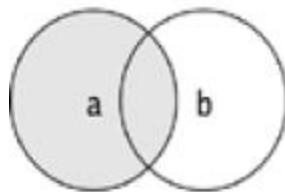








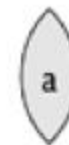




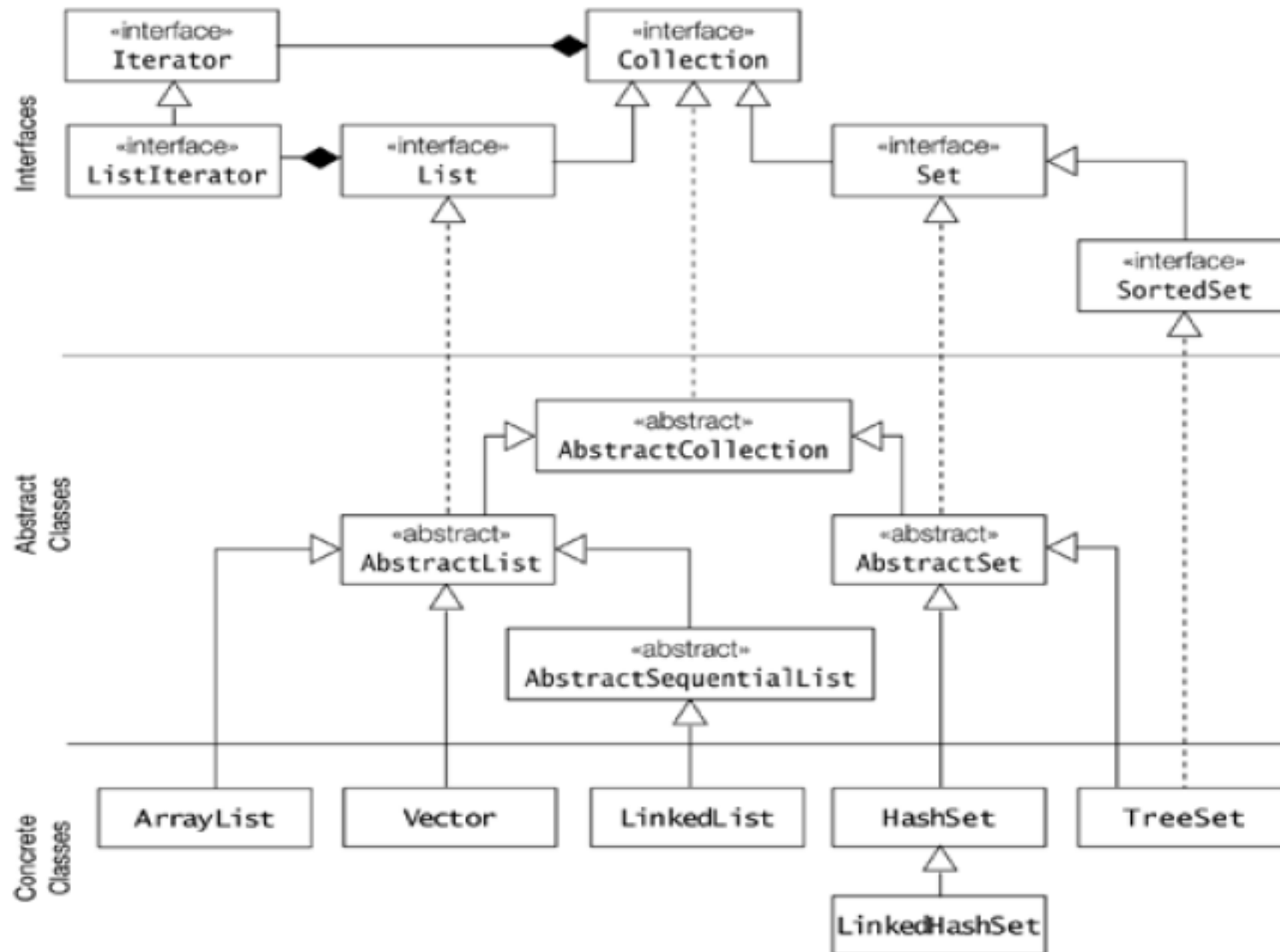
`a.addAll(b)`



`a.removeAll(b)`



`a.retainAll(b)`















---





---







---







---





---











---

■

■

■

■

■

■

■

■

■

■

■

■

■

■

■

■



---

■

■

■

■

■

■

■

■

■

■

■

■





---







---













---







---







---







---







---









---





10<sup>4</sup>









10€







# *Enumeration*

---

```
public static void
main(String[] args) {
    Vector<String> v=new
    Vector<String>();
    v.add("Amit");
    v.add("Raj");
    v.add("Pathak");
    v.add("Sumit");
    v.add("Aron");
    v.add("Trek");
    Enumeration<String>
    en=v.elements();
    while(en.hasMoreEleme
```

```
System.out.println(value  
) ;
```

```
}
```

```
}
```

```
*
```

```
47
```



## *Java Database Con (JDBC)*

\*

1



# *JDBC*

---

- Allows the programmer to connect to a database and query or update it through a Java application.
- Programs developed using Java & JDBC API are platform & vendor independent.
- JDBC API is implemented in **java.sql package**.
- JDBC helps you to write Java applications that manage these three programming activities:
  - Connect to a data source, like a database
  - Send queries and update statements to the database
  - Retrieve and process the results received from the database in answer to your query

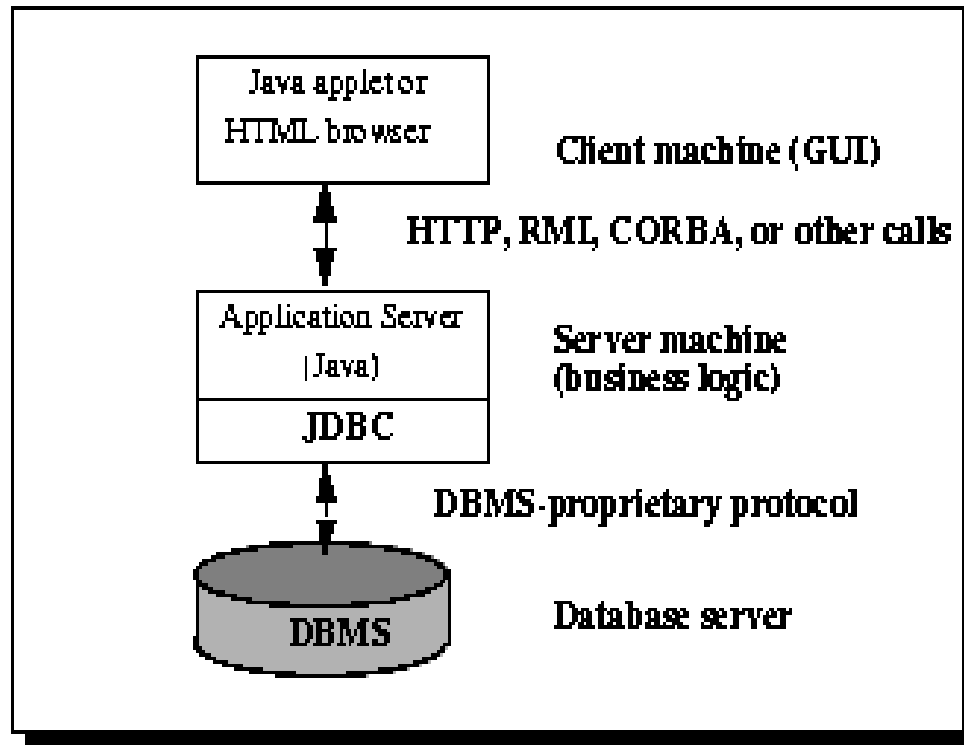


# *JDBC Architecture*

---

- In the three-tier model, commands are sent to a "middle tier" of services, which then sends the commands to the data source.
- The data source processes the commands and sends the results back to the middle tier, which then sends them to the user.
- The three-tier model is very attractive because the middle tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data.
- Another advantage is that it simplifies the deployment of applications. Finally, in many cases, the three-tier architecture can provide performance advantages.
- JDBC API is being used more and more in the middle tier of three-tier architecture. It also allows access to a data source from a Java middle tier.

# JDBC Architecture







## Two-tier model

---

A java application communicates directly to the data source. The JDBC driver enables the communication between the application and the data source.

When a user sends a query to the data source, the answers for those queries are sent back to the user in the form of results.

The data source can be located on a different machine on a network to which a user is connected.

This is known as a client/server configuration, where the user's machine acts as a client, and the machine has the data source running acts as the server.



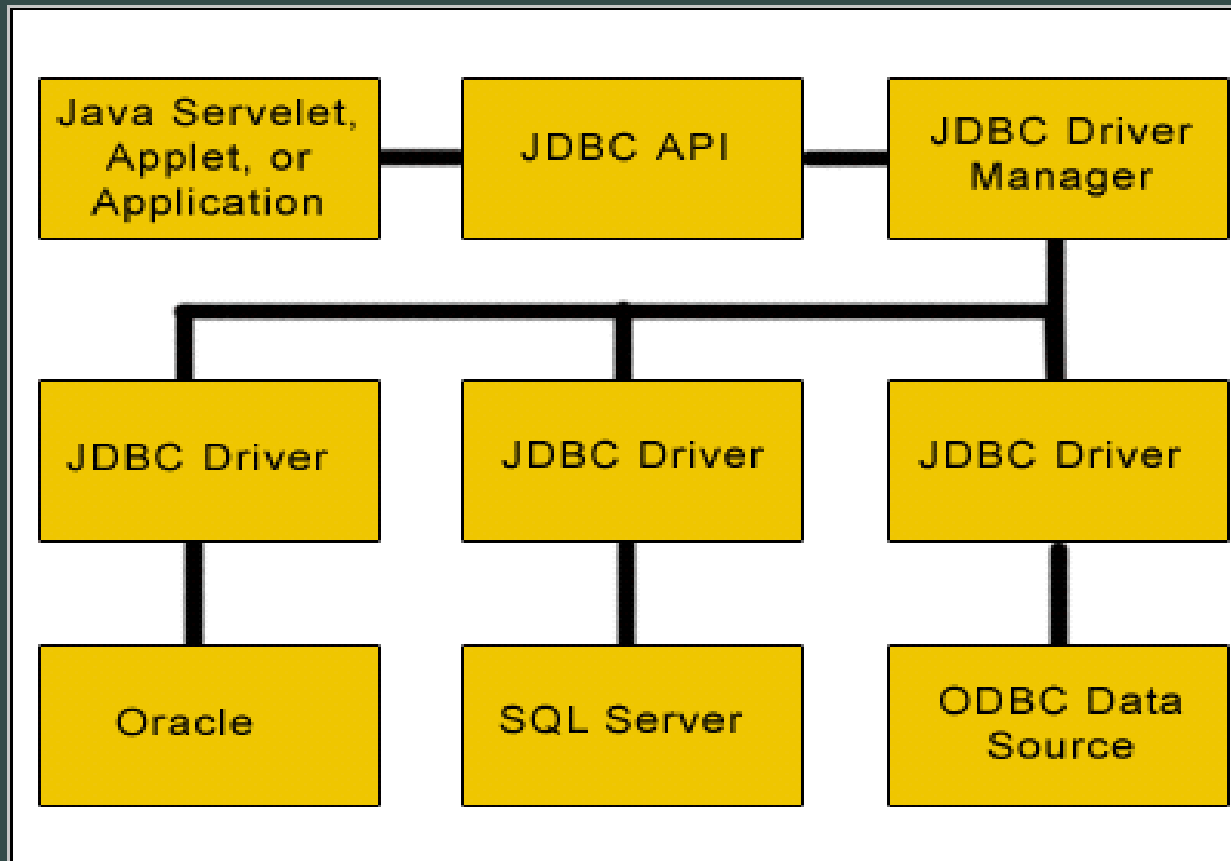
## Three-tier model

---

In this, the user's queries are sent to middle-tier services, from which the commands are again sent to the data source. The results are sent back to the middle tier, and from there to the user.

This type of model is found very useful by management information system directors.

# *JDBC Components (3-tier Architecture)*





# *JDBC Components*

---

JDBC includes four components:

- **The JDBC API:**

- The JDBC API gives programmatic access to relational data from Java. Using the JDBC API, applications can execute SQL statements, retrieve results, and propagate changes back to an underlying data source.
- The JDBC API can also interact with multiple data sources in a distributed, heterogeneous environment.
- The JDBC API is part of the Java platform; it includes the Java Standard Edition (Java SE) and Java Enterprise Edition (JEE). The JDBC 4.0 API is divided into two packages: `java.sql` and `javax.sql`. Both packages are included in the Java SE and Java EE platforms.



# *JDBC Components*

---

## ■ JDBC Driver Manager :

- The JDBC DriverManager class defines objects which can connect Java applications to a JDBC driver.
- DriverManager has traditionally been the backbone of the JDBC architecture. It is quite small and simple.

## ■ JDBC Test Suite:

- The JDBC driver test suite helps JDBC drivers to run your program.
- These tests are not comprehensive or exhaustive, but they do exercise many of the important features in the JDBC API.



---

Open Database Connectivity (ODBC) is an open standard Application Programming Interface (API) for accessing a database.

ODBC is an SQL-based Application Programming Interface (API) created by Microsoft that is used by Windows software applications to access databases via SQL.

JDBC is an SQL-based API created by Sun Microsystems to enable Java applications to use SQL for database access.



# *JDBC Components*

---

## ■ JDBC-ODBC Bridge:

- The Java Software bridge provides JDBC access via ODBC drivers. Note that you need to load ODBC binary code onto each client machine that uses this driver.
- As a result, the ODBC driver is most appropriate on a corporate network where client installations are not a major problem, or for application server code written in Java in three-tier architecture.



# Interfaces of JDBC API

---

A list of popular interfaces of JDBC API is given below:

Driver interface

Connection interface

Statement interface

PreparedStatement interface

CallableStatement interface

ResultSet interface

ResultSetMetaData interface

DatabaseMetaData interface

RowSet interface





# *JDBC Drivers*

---

## ■ What is JDBC Driver ?

- ❑ JDBC drivers implement the defined interfaces in the JDBC API for interacting with your database server.
- ❑ For example, using JDBC drivers enable you to open database connections and to interact with it by sending SQL or database commands then receiving results with Java.
- ❑ The *java.sql* package that ships with JDK contains various classes with their behaviours defined and their actual implementations are done in third-party drivers.
- ❑ Third party vendors implements the *java.sql.Driver* interface in their database driver.



# *JDBC Drivers*

---

- There are four distinct types of JDBC drivers.
  - JDBC-ODBC Bridge (Type 1)
  - Native-API partly Java Driver (Type 2)
  - Net-Protocol All-Java Driver (Type 3)
  - Native Protocol Pure Java Driver (Type 4)



# *Type 1 Driver – JDBC-ODBC Bridge*

---

- The **JDBC- ODBC** Bridge provides JDBC access using most standard ODBC drivers. In a Type 1 driver, a JDBC bridge is used to access ODBC drivers installed on each client machine.
- Using ODBC requires configuring on your system a Data Source Name (DSN) that represents the target database.
- The driver converts JDBC method calls into ODBC function calls. The driver is platform-dependent as it makes use of ODBC which in turn depends on native libraries of the underlying operating system the JVM is running upon.
- Also, use of this driver leads to other installation dependencies; for example, ODBC must be installed on the computer having the driver and the database must support an ODBC driver.

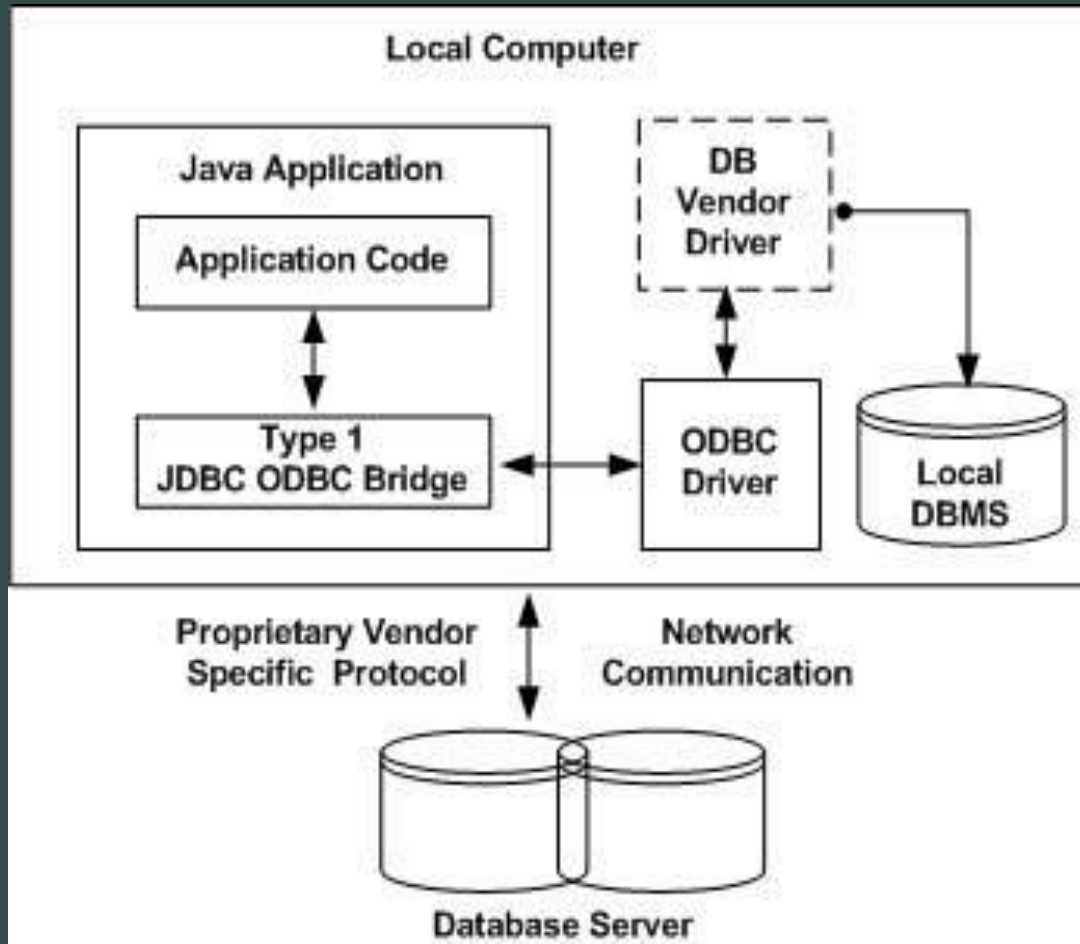


# *Type 1 Driver – JDBC-ODBC Bridge*

---

- When Java first came out, this was a useful driver because most databases only supported ODBC access but now this type of driver is recommended only for experimental use or when no other alternative is available.
- Advantages:
  - Almost any database for which ODBC driver is installed, can be accessed.
- Disadvantages:
  - Performance overhead since the calls have to go through the JDBC overhead bridge to the ODBC driver, then to the native database connectivity interface.
  - The ODBC driver needs to be installed on the client machine

# Type 1 Driver – JDBC-ODBC Bridge





## *Type 2 Driver - JDBC-Native API*

---

- In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls which are unique to the database.
- It converts the JDBC calls into a database specific call for databases such as SQL, ORACLE etc. This driver communicates directly with the database server.
- It requires some native code to connect to the database.
- These drivers typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge, the vendor-specific driver must be installed on each client machine.

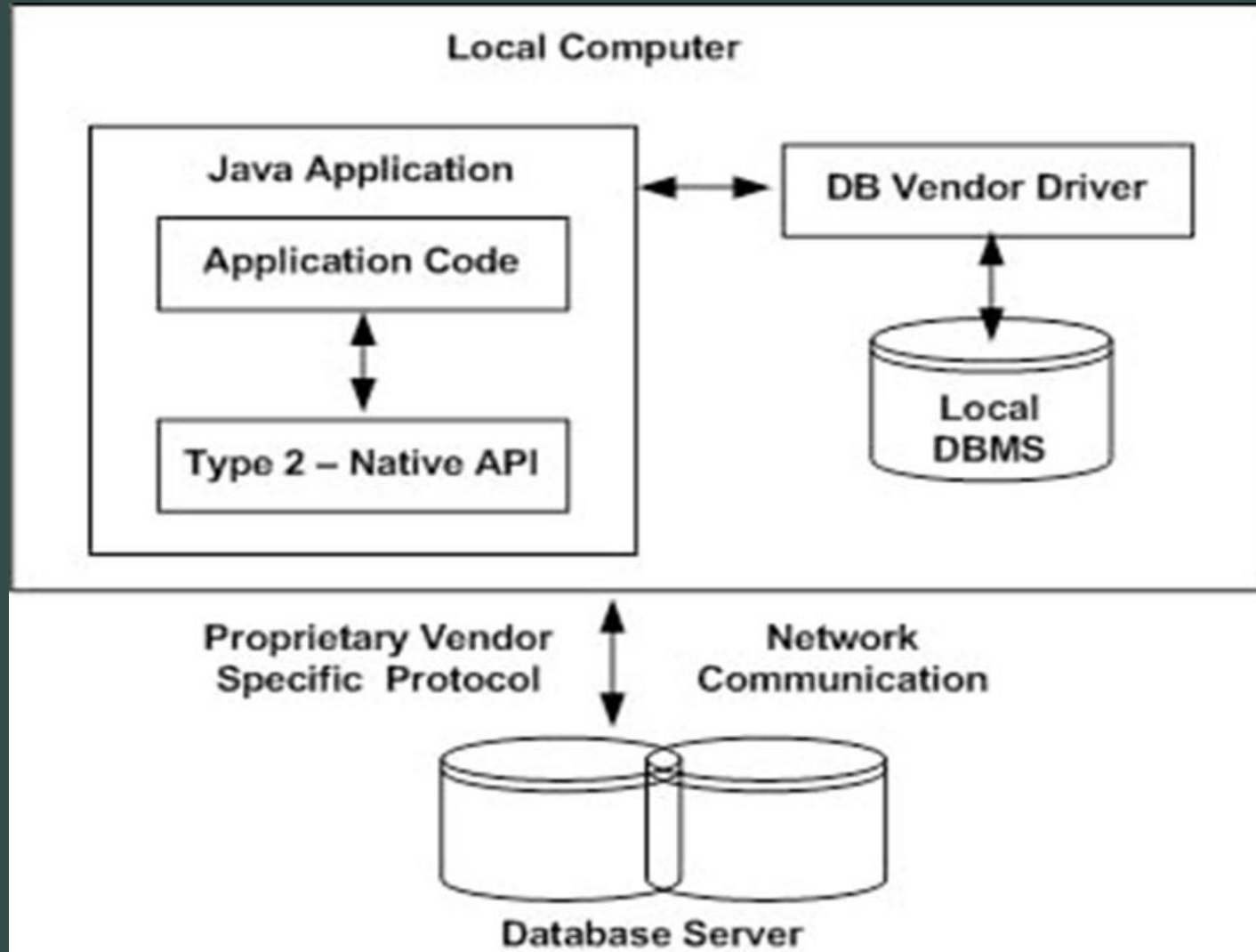


# *Type 2 Driver - JDBC-Native API*

---

- If we change the Database we have to change the native API as it is specific to a database and they are mostly obsolete now but you may realize some speed increase with a Type 2 driver, because it eliminates ODBC's overhead.
- Advantages:
  - Better performance than Type 1 since no JDBC to ODBC translation is needed.
- Disadvantages
  - The vendor client library needs to be installed on the client machine.
  - Cannot be used in internet due the client side software needed.
  - Not all databases give the client side library.

# Type 2 Driver - *JDBC-Native API*







## *Type 3 Driver - Net-Protocol All-Java*

---

- In a Type 3 driver, a three-tier approach is used to accessing databases. The JDBC clients use standard network sockets to communicate with an middleware application server. The socket information is then translated by the middleware application server into the call format required by the DBMS, and forwarded to the database server.
- This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases.
- You can think of the application server as a JDBC "proxy," meaning that it makes calls for the client application. As a result, you need some knowledge of the application server's configuration in order to effectively use this driver type.



# *Type 3 Driver - Net-Protocol All-Java*

---

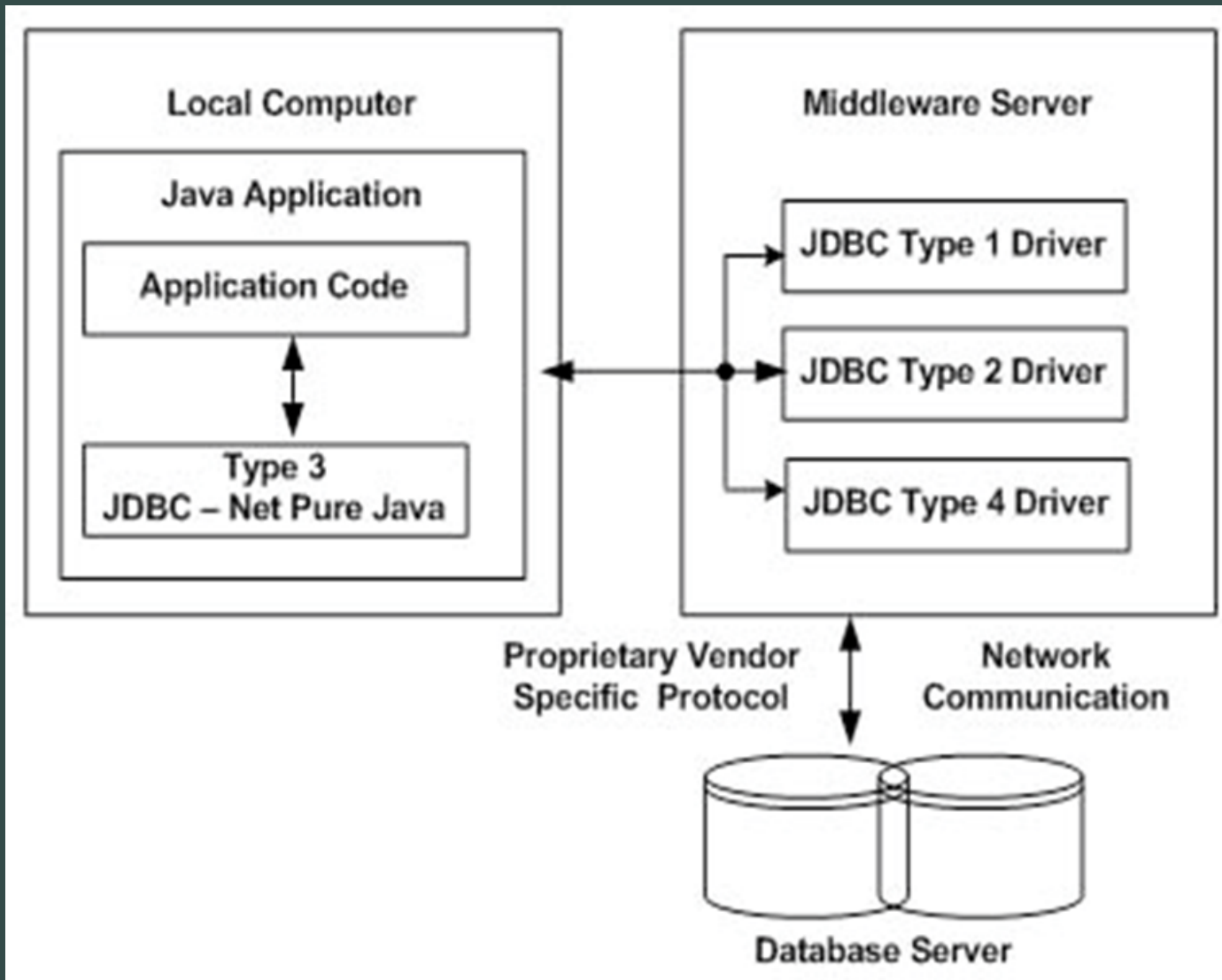
## ■ Advantages

- Since the communication between client and the middleware server is database independent, there is no need for the vendor db library on the client machine. Also the client to middleware need'nt be changed for a new database.
- The Middleware Server can provide typical middleware services like caching, load balancing, logging, auditing etc.
- Can be used in internet since there is no client side software needed.
- At client side a single driver can handle any database.

## ■ Disadvantages:

- Requires database-specific coding to be done in the middle tier.
- An extra layer added may result in a time-bottleneck. But typically this is overcome by providing efficient middleware services.

# Type 3 Driver - Net-Protocol All-Java





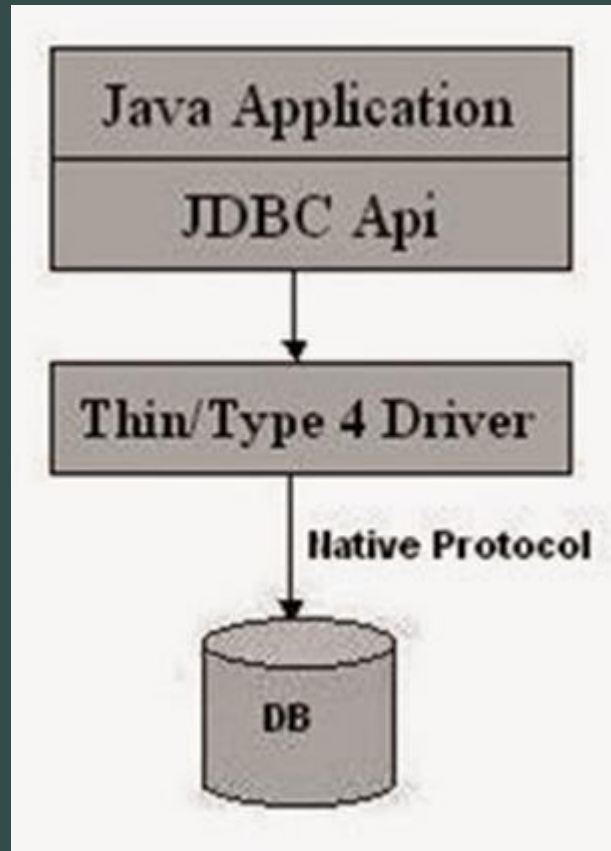
# *Type 4 Driver - Native Protocol Pure Java*

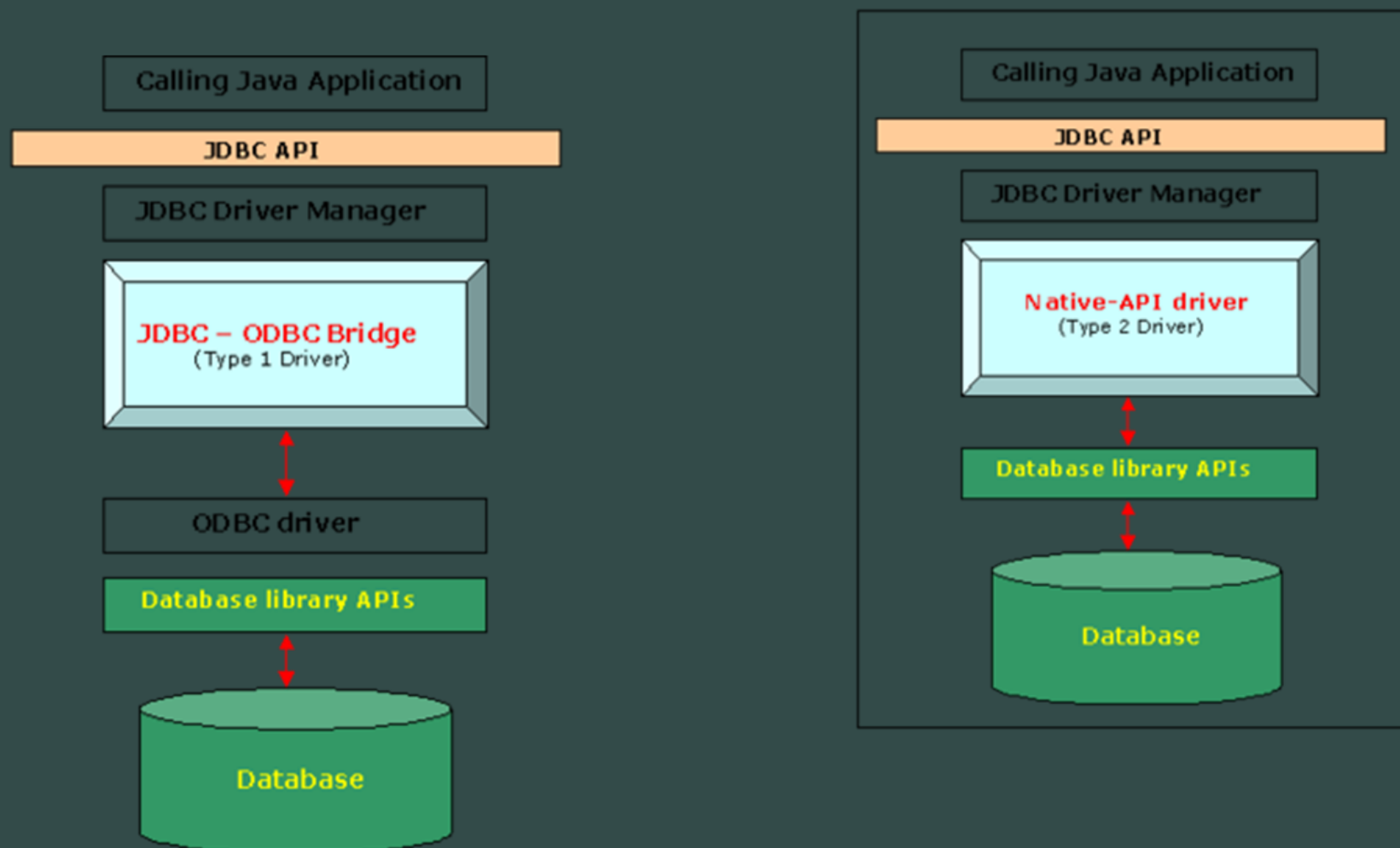
---

- In a Type 4 driver, a pure Java-based driver that communicates directly with vendor's database through socket connection. This is the highest performance driver available for the database and is usually provided by the vendor itself.
- This kind of driver is extremely flexible, you don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.
- Advantages:
  - These drivers don't translate the requests into db request to ODBC or pass it to client api for the db, nor do they need a middleware layer for request indirection. Thus the performance is considerably improved.
- **Disadvantages :**
  - At client side, a separate driver is needed for each database.

# *Type 4 Driver - Native Protocol Pure Java*

---







Calling Java Application

JDBC API

JDBC Driver Manager

**Network-Protocol driver**  
(Type 3 Driver)

**Middleware**

(Application server)



Different database vendors

Calling Java Application

JDBC API

JDBC Driver Manager

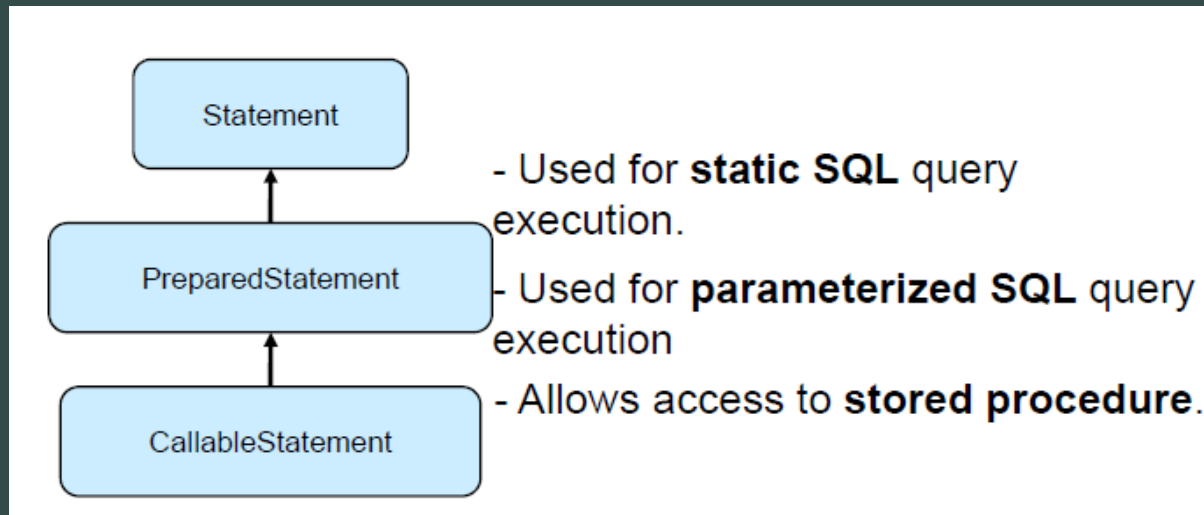
**Native-Protocol driver**  
(Type 4 Driver)

direct calls using  
specific database protocol



**Database**

# Statements in JDBC



Interfaces	Recommended Use
<b>Statement</b>	Use statement for general-purpose access to your database. Useful when you are using static SQLstatements at runtime. The Statement interface cannot accept parameters.
<b>PreparedStatement</b>	Use when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
<b>CallableStatement</b>	Use when you want to access database stored procedures. The CallableStatement interface can also accept runtime input parameters.





# Statements in JDBC

---

- Once a connection is obtained we can interact with the database.
- The JDBC *Statement*, *CallableStatement*, and *PreparedStatement* interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database.
- They also define methods that help bridge data type differences between Java and SQL data types used in a database.




## 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:

Connect to the database

Execute queries and update statements to the database

Retrieve the result received from the database.



---

## How to make connect using JDBC?

Handling a connection requires following steps:

Load the driver

Open database connection

Close database connection



---

## 1) Load JDBC driver

The easiest way to do this is to use `Class.forName()` on the class that implements the `java.sql.Driver` interface.

With MySQL Connector, the name of this class is `com.mysql.jdbc.Driver`. With this method, you could use an external configuration file to supply the driver class name and driver parameters to use when connecting to a database.



---

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

As part of its initialization, the DriverManager class will attempt to load the driver classes referenced in the “jdbc.drivers” system property. This allows a user to customize the JDBC Drivers used by their applications.



## 2) Open database connection

---

After the driver has been registered with the Driver Manager, you can obtain a Connection instance that is connected to a particular database by calling `DriverManager.getConnection()`:

```
Connection connection =  
DriverManager.getConnection("jdbc:mysql://localhost:3306/  
/db_name", "root", "password");
```

Once a Connection is established, it can be used to create Statement and Prepared Statement objects, as well as retrieve metadata about the database.



### 3) Close database connection

---

This step is as much important as opening a connection. Any connection left open is waste of resource and lead to various exceptions.

```
try
{
    if(connection != null)
        connection.close();
    System.out.println("Connection closed !!");
} catch (SQLException e) {
    e.printStackTrace();
}
*
```





---

```
import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.SQLException;
```

```
public class ConnectionDemo {  
    public static void main(String[] argv) {
```

```
        System.out.println("----- MySQL JDBC Connection  
Example -----");
```



---

```
try
{
    Class.forName("com.mysql.jdbc.Driver");
}
catch (ClassNotFoundException e) {
    System.out.println("MySQL JDBC Driver not found
!!");
    return;
}
System.out.println("MySQL JDBC Driver
Registered!");
```




---

```
Connection connection = null;
    try {
        connection = DriverManager

.getConnection("jdbc:mysql://localhost:3306/Jdb_name",
"root", "password");
        System.out.println("SQL Connection to database
established!");

    } catch (SQLException e) {
        System.out.println("Connection Failed! Check
output console");
        return;
```



---

```
} finally {  
    try  
    {  
        if(connection != null)  
            connection.close();  
        System.out.println("Connection closed !!");  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
}  
}
```





---


Output:

----- MySQL JDBC Connection Example -----

MySQL JDBC Driver Registered!

SQL Connection to database established!

Connection closed !!




---

The statement interface is used to create SQL basic statements in Java it provides methods to execute queries with the database. There are different types of statements that are used in JDBC as follows:

Create Statement

Prepared Statement

Callable Statement



---

1. Create a Statement: From the connection interface, you can create the object for this interface. It is generally used for general-purpose access to databases and is useful while using static SQL statements at runtime.

Syntax:

```
Statement statement = connection.createStatement();
```





Implementation: Once the Statement object is created, there are three ways to execute it.

---

`boolean execute(String SQL):`


If the ResultSet object is retrieved, then it returns true else false is returned. Is used to execute SQL DDL statements or for dynamic SQL.

`int executeUpdate(String SQL):`

Returns number of rows that are affected by the execution of the statement, used when you need a number for INSERT, DELETE or UPDATE statements.

`ResultSet executeQuery(String SQL):`

Returns a ResultSet object. Used similarly as SELECT is used in SQL.




2. Prepared Statement represents a recompiled SQL statement, that can be executed many times. This accepts parameterized SQL queries. In this, “?” is used instead of the parameter, one can pass the parameter dynamically by using the methods of PREPARED STATEMENT at run time.

Illustration:

Considering in the people database if there is a need to INSERT some values, SQL statements such as these are used:

```
INSERT INTO people VALUES ("Ayan",25);  
INSERT INTO people VALUES("Kriya",32);
```



---

To do the same in Java, one may use Prepared Statements and set the values in the ? holders, setXXX() of a prepared statement is used as shown:

```
String query = "INSERT INTO people(name, age)VALUES(?, ?)";
```

```
Statement pstmt = con.prepareStatement(query);
```

```
pstmt.setString(1,"Ayan");
```

```
ptstmt.setInt(2,25);
```

```
* // where pstmt is an object name
```




Implementation: Once the PreparedStatement object is created, there are three ways to execute it:

---

`execute()`: This returns a boolean value and executes a static SQL statement that is present in the prepared statement object.

`executeQuery()`: Returns a ResultSet from the current prepared statement.

`executeUpdate()`: Returns the number of rows affected by the DML statements such as INSERT, DELETE, and more that is present in the current Prepared Statement.



3. Callable Statement are stored procedures which are a group of statements that we compile in the database for some task, they are beneficial when we are dealing with multiple tables with complex scenario & rather than sending multiple queries to the database, we can send the required data to the stored procedure & lower the logic executed in the database server itself.

The Callable Statement interface provided by JDBC API helps in executing stored procedures.

Syntax: To prepare a CallableStatement

```
CallableStatement cstmt = con.prepareCall("{call  
Procedure_name(?, ?)}");
```



---

Implementation: Once the callable statement object is created

`execute()` is used to perform the execution of the statement.



---

## Why DataSource interface is introduced

If a java program wants to obtain a connection with a database then there are two options for the java program

1. A java program can use DriverManager class of java.sql package to obtain a connection.
2. A java program can use DataSource interface of java.sql package to obtain a connection with a database.



## The following are the major drawbacks of DriverManager class

1. DriverManager class takes atleast three or four seconds of time to open database connection in a network so it decreases the burden on or performance of java application.
2. To obtain a connection using DriverManager a programmer has to remember driver class name it's url and username, password so it increase the burden on a programmer.
3. DriverManager class opens a separate connection for each client so number of connections on database server will be increased so it's performance is decrease.
4. DriverManager opens non reusable connections with a database so it increase burden on database server.





## Why connection pooling:

---

When we use driver manager or datasource the connection opened with a datasource is a non reusable. For each client if a new connection is opened then burden on database server will be increase so to reduce the burden on database server we use connection pooling.

## What is connection pooling:

Connection pooling is a mechanism which makes a database connection as a reusable for more than one client so burden on a database server will be reduced. A connection pooling makes database connections as reusable. A connection pool contains set of pooled connections(reusable connections).

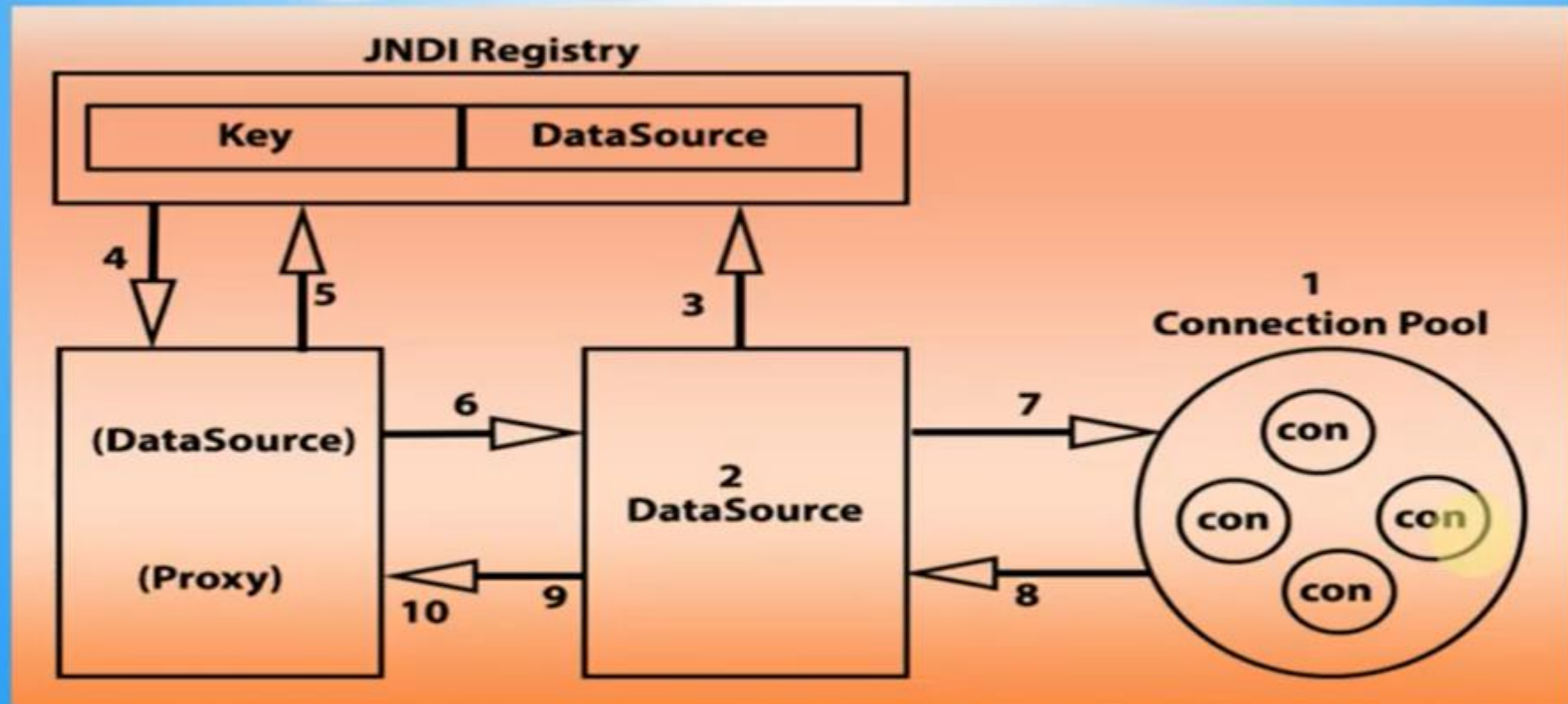


## How connection pooling works:

---

1. A server administrator prepares a connection pool with a setoff connections and also a mediator object as DataSource object to access the pool.
2. An administrator stores DataSource object into JNDI registry (java naming directory interface) .
3. A java application reads DataSource object from JNDI registry and asks for a connection from a pool.
4. A DataSource object takes a connection from a pool and creates a proxy or logical connection do it and then sends the proxy connection to java program.
5. When a java program closes proxy connection the DataSource will done real connection back to the pool.

## JDBC – Connection pooling in JDBC



1 and 2 and steps are done by administrator.

4, 5, 6 steps are done by java programmer.


7, 8, 9, 10 steps are done by DataSource object.

# Java Server Pages (JSP)




Thank You !!!

# Java Server Pages (JSP) Introduction

- 
- JSP is Sun's solution for developing dynamic web sites. JSPs provide a way to separate the generation of dynamic content (java) from its presentation (html).
  - JSP provide excellent server side scripting support for creating database driven web applications.
  - JSP enable the developers to directly insert java code into jsp file, this makes the development process very simple and its maintenance also becomes very easy.
  - Servlets generate the content as well as the necessary HTML syntax to present them to the browser. JSPs differentiate content from presentation.



# Java Server Pages (JSP) Introduction

- 
- JSPs appear like an HTML document, embedded with JSP specific tags and have a file extension “.jsp”.
  - The JSP directives are responsible for generating dynamic content while HTML part takes care of formatting and presentation.
  - JSPs are ultimately implemented as Servlets.
  - A “.jsp” file is always compiled to servlets when they are loaded for the first time, and subsequently whenever the JSP Page is modified.
  - JSPs are much simpler than servlets and are easier to develop



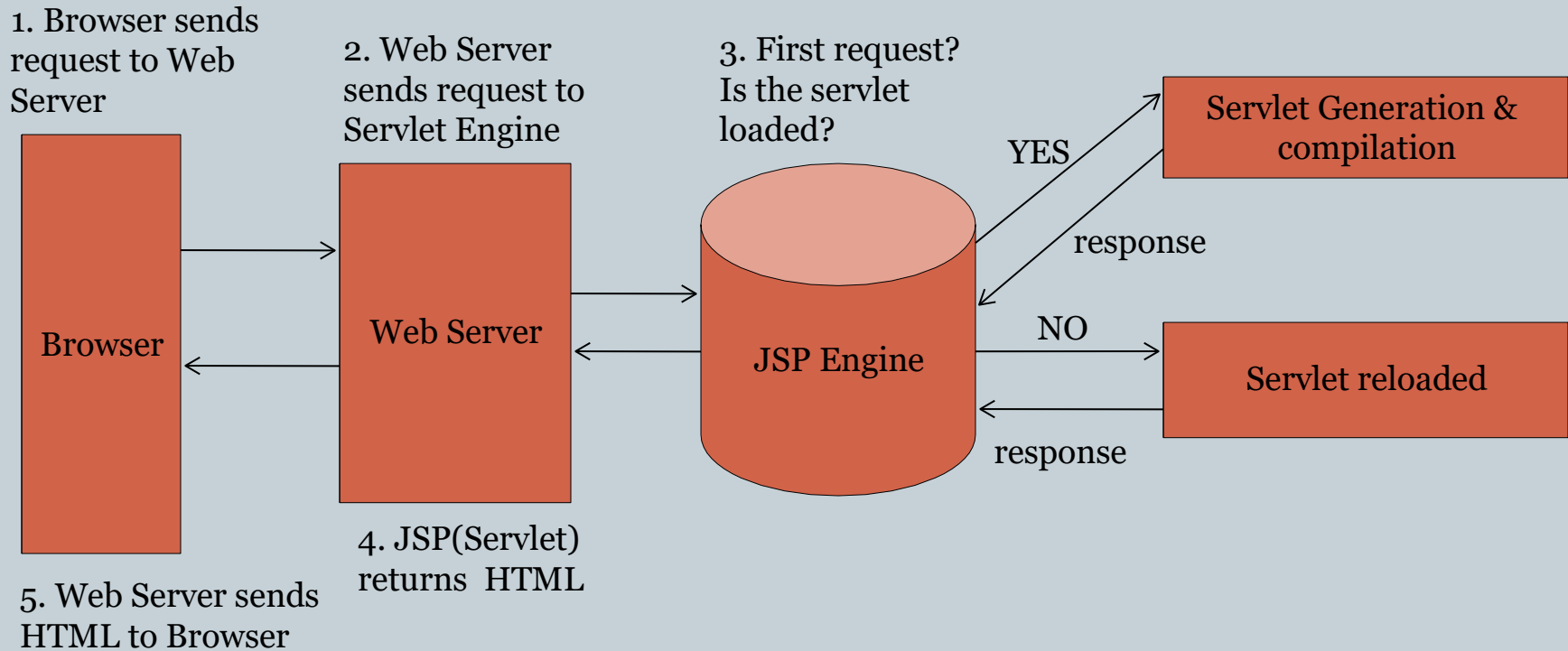


# The Lifecycle of a JSP Page

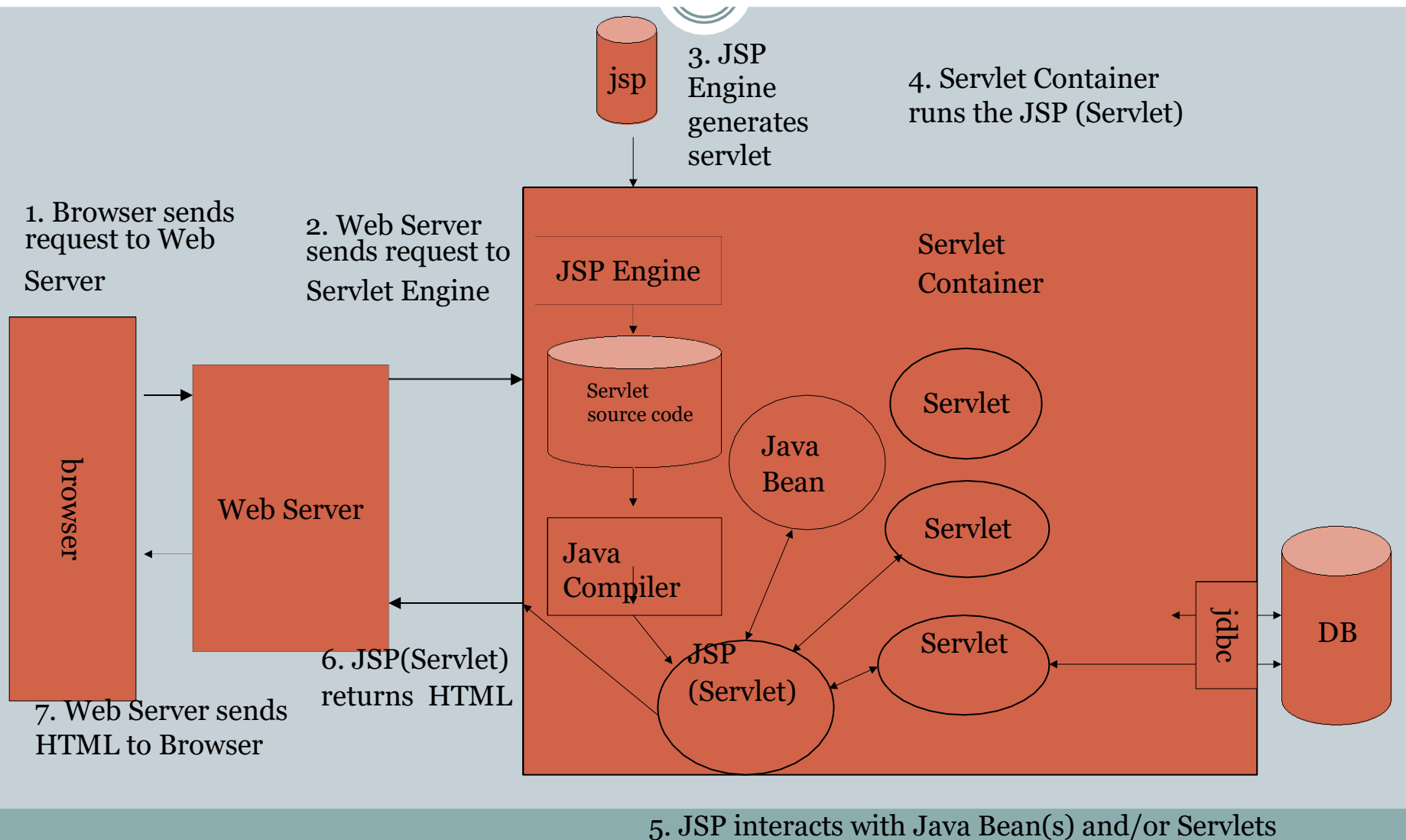
- What happens during a JSP Page Request?
  - When a request is mapped to a JSP page, the Web container first checks whether the JSP page's servlet is older than the JSP page.
  - If the servlet is older, then the Web container translates the JSP page into a servlet class and compiles the class.
  - During development, one of the advantages of JSP pages over servlets is that the build process is performed automatically.
- The lifecycle of a JSP page contains the following phases:
  - Translation and compilation phase
  - Execution phase



# JSP Diagram



# JSP Diagram



# The Lifecycle of a JSP Page

- Translation and compilation phase:
  - During the translation phase, static data is transformed into code that will emit the data into the response stream.
  - JSP elements are treated as follows:
    - ✦ Directives are used to control how the Web container translates and executes the JSP page.
    - ✦ Scripting elements are inserted into the JSP page's servlet class.
    - ✦ Expression language expressions are passed as parameters to calls to the JSP expression evaluator.
    - ✦ `jsp:[set|get]Property` elements are converted into method calls to JavaBeans components.



# The Lifecycle of a JSP Page

- ✦ `jsp:[include|forward]` elements are converted into invocations of the Java Servlet API.
- ✦ The `jsp:plugin` element is converted into browser-specific markup for activating an applet.
- ✦ Custom tags are converted into calls to the tag handler that implements the custom tag.





# The Lifecycle of a JSP Page

## Execution phase:

This phase can control various JSP page execution parameters by using page directives. The following are the two parts of the Execution phase:

- Buffering:
  - When a JSP page is executed, the output written to the response object is automatically buffered.
  - The size of the page buffer can be set using the following page directive: `<%@ page buffer="none|xxxkb" %>`
  - A larger buffer allows more content to be written before anything is actually sent back to the client, thus providing the JSP page with more time to set appropriate status codes and headers or to forward to another web resource. A smaller buffer decreases server memory load.
  - Allows the client to start receiving data more quickly.



# The Lifecycle of a JSP Page

- Handling errors:

- To specify an error, the Web container should forward the control to an error page.
- If an exception occurs, include the following page directive at the beginning of your JSP page:

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

**Example:**

```
<%@ page errorPage="errorpage.jsp"%>
```

The following page directive at the beginning of errorpage.jsp indicates that it is serving as an error page:

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



# Components of JSP

- JSPs are text files that combine standard HTML and new scripting tags. JSPs look like HTML, but they get compiled into Java servlets the first time they are invoked. The resulting servlet is a combination of HTML from the JSP file and embedded dynamic content specified by the new tags.
- JSP is a tag-based language. The tags help the container to interpret the enclosed script suitably. JSP tags are case sensitive. JSP tags fall into the following categories:
  - Directives
  - Scripting Elements
  - Standard Actions



# Directives

- Directives affect the overall structure of the servlet that results from the translation of JSP.
- These are used to set global values for the JSP file as a whole.

## **page directive:**

- Defines the attributes like import, session, and so on, which is common to the entire JSP page.
- The syntax for the page directive is:

```
<% page import="java.rmi.*, java.utils.*"  
session="true"%>
```





# Directives

## **include directive:**

- It directs the container to include the specified JSP, HTML, and other file types, in the current file.
- The specified resource is copied inline. This happens during translation time.
- Any subsequent changes to the included resource will not be reflected in the JSP, unless the JSP undergoes some modification forcing the container to recompile it.
- The file is included as follows:

```
<% include file="/hello.html" %>
```

## **The taglib directive:**

- Allows the page to use custom JSP tags.



# Scripting Elements

- Scripting elements are further subdivided into
  - Declarations:
    - ✦ Declarations enclose Java code that defines class-wide variables and methods.
    - ✦ They are declared using `<% ! javacode %>` tag sets.
    - ✦ Declarations are initialized when the JSP page is initialized.
    - ✦ They are automatically made available to other declarations, expressions, and code within that page.
  - Scriptlets:
    - ✦ A scriptlet is a block of Java code that will be executed during run time (request processing time) by the JSP container.
    - ✦ It is enclosed with `<% java statements %>`



# Scripting Elements

- ✦ Multiple scriptlets in the same JSP page are combined together in the same order when the container generates the servlet.
- ✦ Because the scriptlets allow you to write fully functional Java code, these are powerful tools in the hands of the JSP programmer.
- Expressions:
  - ✦ An expression is a scriptlet that is evaluated by the JSP Container and “sent to the client” for being displayed.
  - ✦ The tags discussed earlier normally instruct the container and are not involved in display.
  - ✦ You enclose an expression in a tag as:  
`<%= "the value of i = " i %>`



# Standard Actions

- Standard actions have the tags that define the behavior of the JSP during run time.
- The result of a standard action command is normally sent to the client.
- A standard action tag is incorporated as follows:  

```
<jsp:include page="myjsp.jsp" flush="true" />
```
- The include action covers the file contents during run time (request processing time) unlike the include directive, which includes the file contents during compile time (translation time).
- So, if the included file is modified subsequently, the include action output will reflect the change, whereas the include directive will not.





# Standard Actions

- Other frequently used standard action tags are:
  - <jsp:useBean>
  - <jsp:setProperty>
  - <jsp:getProperty>
  - <jsp:param>
  - <jsp:include>
  - <jsp:forward> etc.,



# Sample JSP Code

```
<%@ page language = "java"%>
<html>
  <body>

    <%! int count = 0; %>
    <% count++; %>
    Welcome you are visitor no:
    <%=count %>

  </body>
</html>
```



# *Multithreading*

\*

1



# *Multitasking*

---

- Multitasking allows several activities to occur concurrently on the computer.
  - 1) Process-based Multitasking
  - 2) Thread-based Multitasking



# *Process- based multitasking*

---

- Many processes done simultaneously.
- Program is a set of instructions and a “**A process is a running instance of a program.**”
- Here the CPU is shared between different processes.  
E.g. A MS-Word application & MS-Excel application windows opened simultaneously or in general any two **applications running simultaneously is Process- based multitasking.**



# *Multi processing*

---

- The OS provides you with a mechanism called **CONTEXT SWITCHING** which enables it to switch from process to process.
- Here what happens is a program's entire context (including variables, global variables, functions etc..) are stored separately.
- Therefore if one program is running, it is as if other programs aren't existing at all, because they reside in their own contexts.



# *Thread based multitasking*

---

- Different tasks within a task done simultaneously.
- Extend the idea of multitasking by taking it one level lower.
- Individual programs appear to do multiple tasks at the same time.
- Each sub task is called Thread.
- Think of each thread running in a separate context - that is, each thread has its own CPU -with registers, memory & own code.





# Thread

---

- A thread is an entity within a process.
- You can say, “**A thread within a process executes**”.

E.g.

- Lets take a typical application like Word. There are various independent processes going on, like showing GUI, saving into file, typing in text.
- In your java program `main()` is a thread & garbage collector is another thread.



# *Process & Thread*

---

- Each process has a complete set of its own variables.
- It takes more overhead to launch a process
- Inter-process communication is slower and restrictive.
- Threads may share same data.
- It takes much less overhead to create & destroy thread.
- Communication between threads is easier.



# *Scheduling of process/thread*

---

- Scheduling is decided by OS.
- Scheduling algorithm's basically classified as:
  - Preemptive scheduling
    - OS interrupts programs without consulting them eg : win 95, win NT
  - Non-preemptive scheduling
    - programs interrupted only when they are ready to yield control. Eg : Sun Solaris, Win 3.1



# *Multithreading In Context With Java*

---

- Java supports multithreading.
- But Java is platform dependent as far as multithreading goes.
- JVM is running on top of OS.
- Thus, multithreading capability is totally dependent on what scheduling algorithm is followed by the underlying OS !!
- Thus inconsistent behavior is observed in multithreaded programs.



# *Multithreading In Context With Java*

---

- Java has a class Thread
- It's a part of java.lang package
- Constructors of Thread class –
  - `public Thread()`
  - `public Thread(Runnable target)`
  - `public Thread(ThreadGroup group, String name)`
  - `public Thread(Runnable target, String name)`



# *Life Cycle Of Thread*

---

## ■ Methods of Thread class –

- ❑ run() : must be overridden; code should be added which executes in thread.
- ❑ start() : brings the thread into ready to run condition. When CPU is available for this thread, invokes the run().
- ❑ stop() : Forces the thread to stop executing.
- ❑ sleep() : A static method, puts currently executing thread to sleep for specified no. of milliseconds.





# *Multithreading*

---

```
class Demo extends Thread{
    String word;
    public Demo (String w) {
        word = w;
    }
    public void run() {
        try{
            for( ;; ){
                System.out .println(word);
                Thread.sleep( 1000);
            }
        }catch (Exception e) {System.out.println(e);}
    }
}
```





# *Multithreading*

---

```
public static void main(String args[])
{
    Thread t1 = new Demo("Pass");
    t1.start();
    Thread t2 = new Demo("fail");
    t2.start();
}
}
```



# *Thread Priorities*

---

- Thread priorities are used by thread scheduler to decide when each thread should be allowed to run.
- At any given time, when multiple threads are ready to be executed, the runtime system chooses the runnable thread with the highest priority for execution.
- Priorities are integer values from 1 (lowest priority given by the constant `Thread.MIN_PRIORITY`) to 10 (highest priority given by the constant `Thread.MAX_PRIORITY`). The default priority is 5(`Thread.NORM_PRIORITY`).



# Thread Priorities

---

Constant	Description
Thread.MIN_PRIORITY	The maximum priority of any thread (an int value of 10)
Thread.MAX_PRIORITY	The minimum priority of any thread (an int value of 1)
Thread.NORM_PRIORITY	The normal priority of any thread (an int value of 5)

The methods that are used to set the priority of thread shown as:

Method	Description
setPriority()	This is method is used to set the priority of thread.
getPriority()	This method is used to get the priority of thread.



# *Thread Priorities*

---

- In Java runtime system, **preemptive scheduling** algorithm is applied.
- If at the execution time a thread with a higher priority and all other threads are runnable then the runtime system chooses the new **higher priority** thread for execution.
- On the other hand, if two threads of the same priority are waiting to be executed by the CPU then the **round-robin** algorithm is applied in which the scheduler chooses one of them to run according to their round of **time-slice**.



# *Thread Priorities*

---

- In the implementation of threading scheduler usually applies one of the two following strategies:
  - **Preemptive scheduling** ? If the new thread has a higher priority then current running thread leaves the runnable state and higher priority thread enter to the runnable state.
  - **Time-Sliced (Round-Robin) Scheduling** ? A running thread is allowed to be execute for the fixed time, after completion the time, current thread indicates to the another thread to enter it in the runnable state.



# *currentThread()*

---

- **static native Thread currentThread()** - returns a reference to current thread.
- Can be used in run() method to check which thread is currently executing

Eg :

```
public void run() {  
    if (Thread.currentThread( ) == t1 )  
    {  
        // do something  
    }  
}
```



# *Runnable Interface*

---

- In Java a class can extend only one class.
- What if we want multithreading support for a class that is already derived from some other class?
  - Eg class MyFrame extends Frame
- Solution ? Runnable interface.
- Implementing this gives the ability to treat the new class as a Runnable object.



---

```
class Demo implements Runnable{
    String word;
    public Demo(String w){
        word = w;
    }
    public void run(){
        try{
            for( ;; ){
                System.out .println(word);
                Thread.sleep( 1000);
            }
        }catch(Exception e){System.out.println(e);}
    }
}
```





# *Multithreading*

---

```
public static void main(String args[])
{
    Runnable d1 = new Demo("Hi");
    Runnable d2 = new Demo("Hello");

    new Thread( d1 ).start();
    new Thread( d2 ).start();
}
}
```





# Thread Synchronization

---

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called *synchronization*.
- Key to synchronization is the concept of the monitor (also called a *semaphore*).
- *A monitor is an object that is used as a mutually exclusive lock, or mutex.*
- Only one thread can *own a monitor at a given time*. When *a thread* acquires a lock, it is said to have *entered the monitor*.



# Thread Synchronization

---

- All other threads attempting to enter the locked monitor will be suspended until the first thread *exits the monitor*. These other threads are said to be *waiting for the monitor*.



# *Using Synchronized Methods*

---

- To enter an object's monitor, just call a method that has been modified with the **synchronized keyword**.
- While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait.
- While creating synchronized methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases.
- Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use synchronized methods.



# Thread Synchronization

---

- Thus, you can't add synchronized to the appropriate methods within the class.
- How can access to an object of this class be synchronized?
- Fortunately, the solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a synchronized block.
- This is the general form of the synchronized statement:  

```
synchronized(object){    // statements to be synchronized }
```
- Here, *object* is a reference to the object being synchronized.



# *Inter-thread Communication*

---

- Threads also provide a secondary benefit: they do away with polling.
- Polling is usually implemented by a loop that is used to check some condition repeatedly. Once the condition is true, appropriate action is taken. This wastes CPU time.



# *Inter-thread Communication*

---

- For example, consider the classic queuing problem, where one thread is producing some data and another is consuming it.
- To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data.
- In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce.
- Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish.






# *Inter-thread Communication*

---

- Clearly, this situation is undesirable. To avoid polling, Java includes an elegant interprocess communication mechanism via the **wait( )**, **notify( )**, and **notifyAll( )** methods.
- These methods are implemented as **final methods in Object**, so all classes have them. All three methods can be called only from within a **synchronized context**.



---

## *Rules for using wait(), notify() and notifyAll():*

- **wait( )** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify( )**.
- **notify( )** wakes up the first thread that called **wait( )** on the same object.
- **notifyAll( )** wakes up all the threads that called **wait( )** on the same object. The highest priority thread will run first.

\*

30