UNIT 5: Java Servlets

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

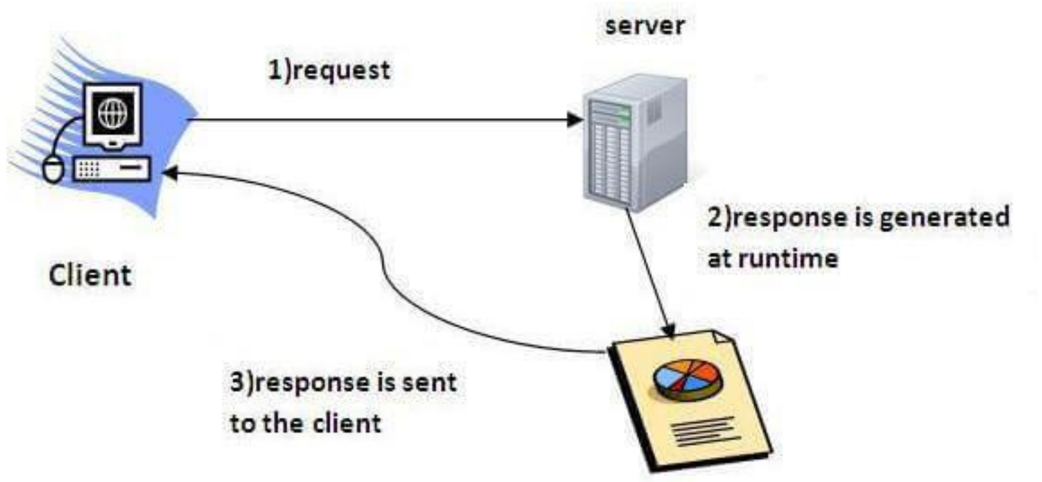
Servlet technology is robust and scalable because of java language. Before Servlet, CGI (Common Gateway Interface) scripting language was common as a server-side programming language. However, there were many disadvantages to this technology. We have discussed these disadvantages below.

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

What is a Servlet?

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

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

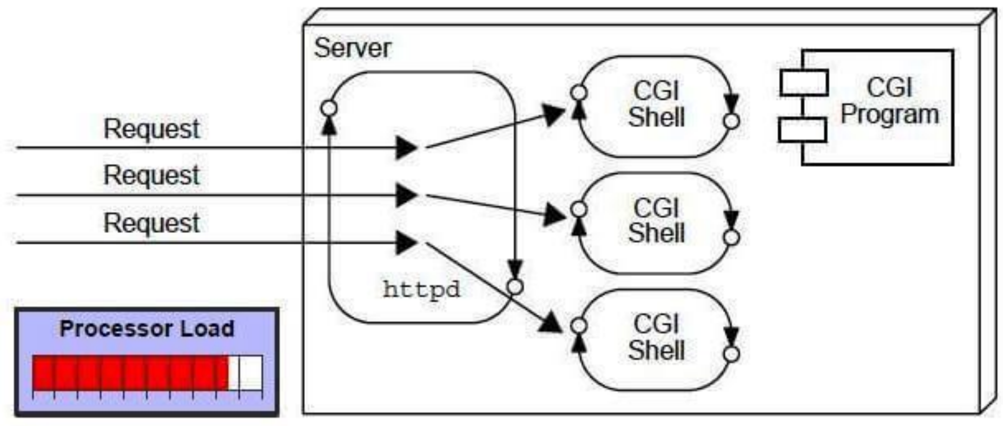


What is a web application?

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

CGI (Common Gateway Interface)

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

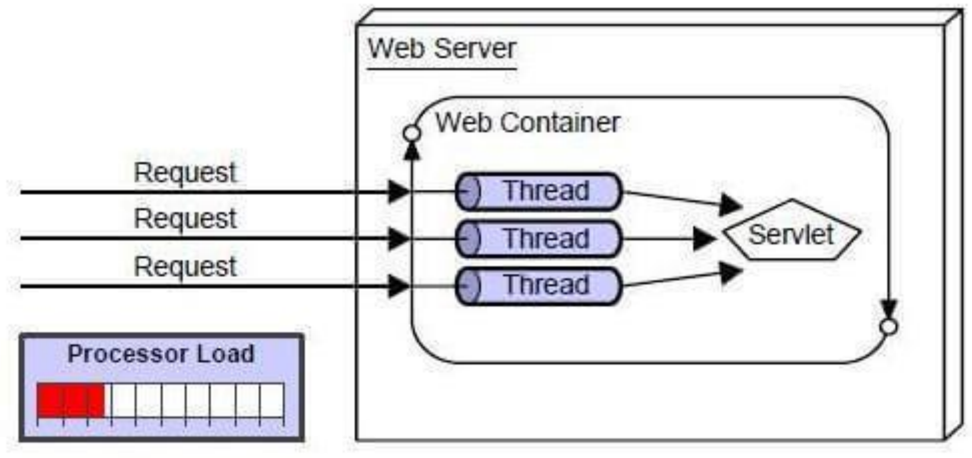


Disadvantages of CGI

There are many problems in CGI technology:

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

Advantages of Servlet



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

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

Servlet API

1. Servlet API
2. Interfaces in javax.servlet package
3. Classes in javax.servlet package
4. Interfaces in javax.servlet.http package
5. Classes in javax.servlet.http package

The javax.servlet and javax.servlet.http packages represent interfaces and classes for servlet api.

The **javax.servlet** package contains many interfaces and classes that are used by the servlet or web container. These are not specific to any protocol.

The **javax.servlet.http** package contains interfaces and classes that are responsible for http requests only.

Let's see what are the interfaces of javax.servlet package.

Interfaces in javax.servlet package

There are many interfaces in javax.servlet package. They are as follows:

1. Servlet
2. ServletRequest
3. ServletResponse
4. RequestDispatcher
5. ServletConfig
6. ServletContext
7. SingleThreadModel
8. Filter
9. FilterConfig
10. FilterChain
11. ServletRequestListener
12. ServletRequestAttributeListener
13. ServletContextListener
14. ServletContextAttributeListener

Classes in javax.servlet package

There are many classes in javax.servlet package. They are as follows:

1. GenericServlet
2. ServletInputStream
3. ServletOutputStream
4. ServletRequestWrapper

5. ServletResponseWrapper
6. ServletRequestEvent
7. ServletContextEvent
8. ServletRequestAttributeEvent
9. ServletContextAttributeEvent
10. ServletException
11. UnavailableException

Interfaces in javax.servlet.http package

There are many interfaces in javax.servlet.http package. They are as follows:

1. HttpServletRequest
2. HttpServletResponse
3. HttpSession
4. HttpSessionListener
5. HttpSessionAttributeListener
6. HttpSessionBindingListener
7. HttpSessionActivationListener
8. HttpSessionContext (deprecated now)

Classes in javax.servlet.http package

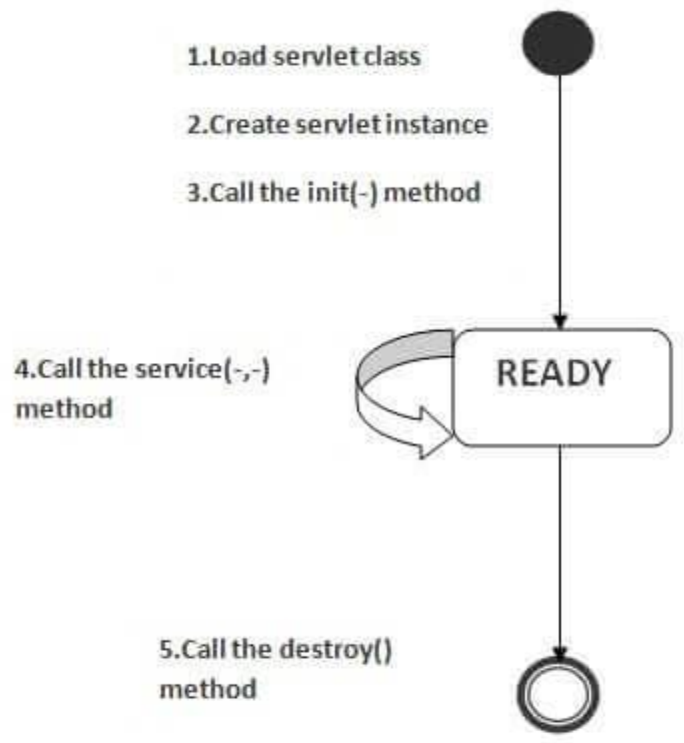
There are many classes in javax.servlet.http package. They are as follows:

1. HttpServlet
2. Cookie
3. HttpServletRequestWrapper
4. HttpServletResponseWrapper
5. HttpSessionEvent
6. HttpSessionBindingEvent
7. HttpUtils (deprecated now)

Life Cycle of a Servlet (Servlet Life Cycle)

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

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


1.Load servlet class
2.Create servlet instance
3.Call the init(-) method
4.Call the service(-,-) method
READY
5.Call the destroy() method

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

1) Servlet class is loaded

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

2) Servlet instance is created

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

3) init method is invoked

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

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

---

4) service method is invoked

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

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

---

5) destroy method is invoked

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

```
1. public void destroy()
```

Servlet – Debugging

One of the most difficult components of building servlets is testing and debugging. Because servlets include a lot of client/server interaction, they're prone to errors– though they're hard to come by. Because servlets operate inside a strongly multithreaded and typically complicated web server, they don't function well with ordinary debuggers, it might be difficult to track down the reason for nonobvious failures.

You may also utilize your server's interactive display to debug a servlet by following these steps:

1. To build your servlet, use the javac -g command in the Qshell Interpreter.
2. Place the source code (.java file) and compiled code (.class file) in a classpath directory.

3. Launch the server.
4. On the job where the servlet runs, perform the Start Service Job (STRSRVJOB) command.
5. STRDBG CLASS(myServlet), where myServlet is your servlet's name. It is necessary to show the source.
6. Press F12 to set a breakpoint in the servlet.
7. To update the modifications, click the Refresh button in the Web Browser if automatic class reloading is enabled, which is the default configuration. The state of your application is not lost.
8. You lose the state of the program if automatic class reloading is not enabled. Restart the server to apply the modifications.

Few hints and suggestions that may aid you in your debugging:

1. By using System.out.println()
2. The message log
3. Using JDB Debugger
4. Use comments
5. Client and Server Headers

Using System.out.println() command

System.out.println() is a marker that is used to see if a given piece of code has been run. The variable's value can also be printed. Furthermore:\

Because System objects are part of the core Java objects, they may be used everywhere without the requirement for extra classes to be installed. Servlets, JSPs, RMIs, EJBs, Common Beans and Classes, and stand-alone apps are all examples of this.

Write to System with various pauses at the breakpoint. It does not interfere with the application's usual flow of execution, which means that timing is critical when it is very beneficial.

Syntax: To use System.out.println() command

```
System.out.println("Debugging message");
```

All of the messages created by the above syntax will be recorded in the log file of the webserver.

message log

Use the proper logging mechanism to capture all debug, warning, and error messages, which is a great idea; log4J is suggested for capturing all messages. The log () function in the Servlet API also provides an easy way to output:

Example:

- Java

```
// Java Program to Illustrate log4J for Capturing

// All Messages Logs



// Importing required classes

import java.io.*;

import javax.servlet.*;

import javax.servlet.http.*;



// Class

// Extending HttpServlet class

public class GFG extends HttpServlet {



   // Method 1

   public void doGet(HttpServletRequest request,

                     HttpServletResponse response)

      throws ServletException, java.io.IOException

   {
```

```
String par = request.getParameter("par1");



// Calling the two ServletContext.log methods

ServletContext context = getServletContext();

if (par == null || par.equals(""))



  // log version with Throwable parameter

  context.log("No message received:",

        new IllegalStateException(

          "Missing parameter"));

else

  context.log("Here is the visitor's message: "

        + par);



response.setContentType("text/html");

java.io.PrintWriter out = response.getWriter();



// Title to be displayed
```

```
        String title = "Geeksforgeeks Context Log program ";



        String docType

            = "<!doctype html public \"-//w3c//dtd html 4.0 "

              + "transitional//en\">\n";

        out.println(

            docType + "<html>\n"

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

            + "<body bgcolor = \"#f0f0f0\">\n"

            + "<h1 align = \"center\">" + title + "</h1>\n"

            + "<h2 align = \"center\">Messages sent successfully </h2>\n"

            + "</body></html>");

    }

}
```

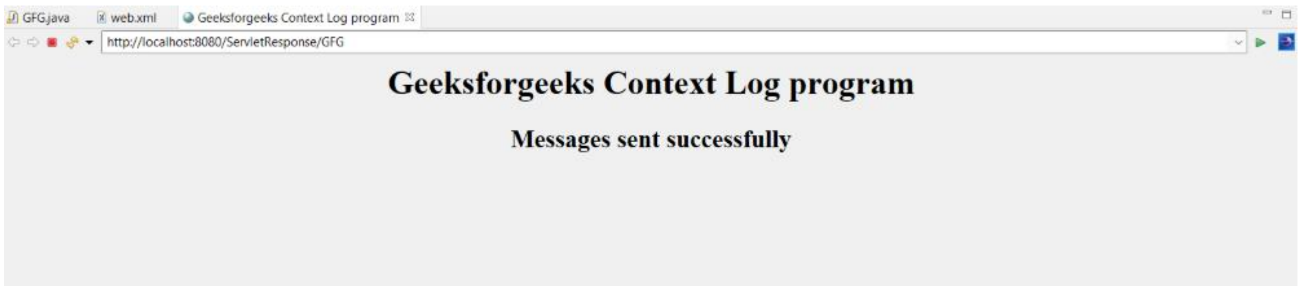
File: web.xml

- XML

```
<web-app>
```

```
<servlet>

 <servlet-name>GFG</servlet-name>

 <servlet-class>GFG</servlet-class>

</servlet>

<servlet-mapping>

 <servlet-name>GFG</servlet-name>

 <url-pattern>/GFG</url-pattern>

</servlet-mapping>

<welcome-file-list>

 <welcome-file>GFG.java</welcome-file>

</welcome-file-list>

</web-app>
```

Output: Run the GFG.java file, you will get the following output:


GFG.java
web.xml
Geeksforgeeks Context Log program
http://localhost:8080/ServletResponse/GFG

Geeksforgeeks Context Log program

Messages sent successfully

Using JDB Debugger

To debug Servlet, use the applet or application debugging ***jdb command***.

We may use sun.servlet.http.HttpServer to debug a Servlet and then run it as HttpServer Servlet to answer HTTP requests on the browser side. This simple debugging applet software is similar. The real application being debugged is ***sun.applet.AppletViewer*** when debugging an applet.

The details of how to debug an applet are usually hidden by default in most debuggers. Similarly, using the debugger, you must perform the following for the servlet:

- Set the classpath of your debugger so that sun.servlet.http.Http-Server and associated classes may be found.
- Set your debugger's classpath to find your servlet and accompanying classes, which are normally located at server root/servlets and server root/classes in Server root/servlets should not be in your classpath since it prevents servlet reloading. This inclusion, on the other hand, is beneficial for debugging. It enables your debugger to set breakpoints in a servlet before HttpServer's proprietary servlet loader loads it.

Start debugging sun.servlet.http.HttpServer after you've specified the right classpath. You may use a web browser to make a request to the HttpServer for the supplied servlet (http://localhost:8080/servlet/ServletToDebug) after setting breakpoints in the servlet you want to debug. At your breakpoints, you should observe execution halt.

Using comments

Debugging will be aided by comments in the code in a variety of ways. Notes may be utilized in a variety of ways to aid with the debugging process.

To temporarily delete some Java code, use Java Servlet comments and single-line comments (//…), multi-line comments (/ *… * /). If the bug goes away, all you have to do now is look at the commented code to figure out what's wrong.

Client and Server Headers

When a servlet fails to operate as intended, it's sometimes helpful to examine the raw HTTP request and response. If you know how HTTP works, you can read the request and response and figure out exactly what's going on with the headers.

Note: *Do remember certain important keypoints for debugging. Here is a list of some other servlet debugging tips:*

1. *Keep in mind that server root/classes do not reload, although server root/servlets very certainly do.*
2. *Request that a browser displays the raw content of the page it is now viewing. This can aid in the detection of formatting issues. Under the View menu, it's generally an option.*
3. *By requiring a full refresh of the website, you can ensure that the browser isn't caching the results of a prior request. Use Shift-Reload in Netscape Navigator and Shift-Refresh in Internet Explorer.*
4. *Check that the init() function of your servlet accepts a ServletConfig argument and immediately calls super.init(config).*

JSP

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

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

Advantages of JSP over Servlet

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

1) Extension to Servlet

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

2) Easy to maintain

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

3) Fast Development: No need to recompile and redeploy

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

4) Less code than Servlet

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

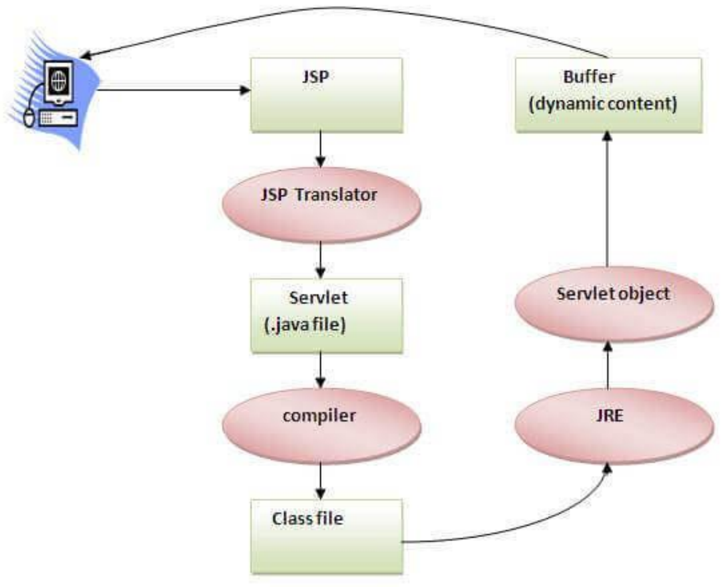
The Lifecycle of a JSP Page

The JSP pages follow these phases:

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

- Destroy (the container invokes jspDestroy() method).

Note: jspInit(), _jspService() and jspDestroy() are the life cycle methods of JSP.



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

Creating a simple JSP Page

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

index.jsp

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

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

4. </body>
5. </html>

It will print **10** on the browser.

How to run a simple JSP Page?

Follow the following steps to execute this JSP page:

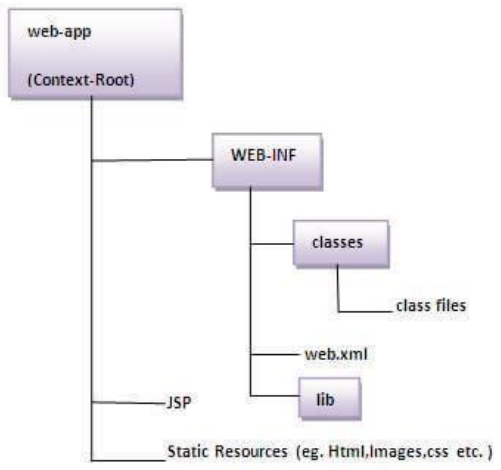
- Start the server
- Put the JSP file in a folder and deploy on the server
- Visit the browser by the URL http://localhost:portno/contextRoot/jspfile, for example, http://localhost:8888/myapplication/index.jsp

Do I need to follow the directory structure to run a simple JSP?

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

The Directory structure of JSP

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



RMI (Remote Method Invocation)

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

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

Understanding stub and skeleton

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

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

stub

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

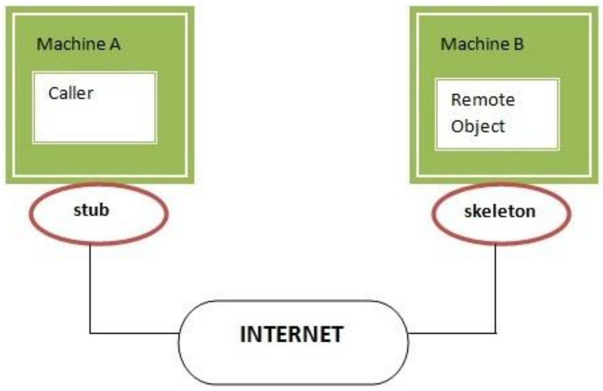
1. It initiates a connection with remote Virtual Machine (JVM),
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads (unmarshals) the return value or exception, and
5. It finally, returns the value to the caller.

skeleton

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

1. It reads the parameter for the remote method
2. It invokes the method on the actual remote object, and
3. It writes and transmits (marshals) the result to the caller.

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



Understanding requirements for the distributed applications

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

.

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

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

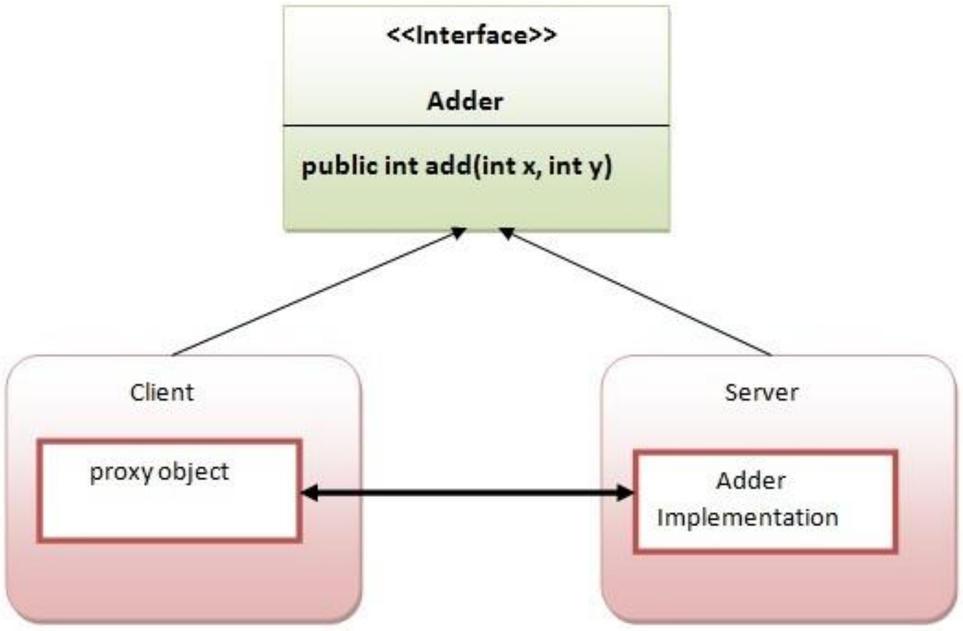
Java RMI Example

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

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

RMI Example

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



1) create the remote interface

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

```
import java.rmi.*;
public interface Adder extends Remote{
public int add(int x,int y)throws RemoteException;
}
```

---

2) Provide the implementation of the remote interface

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

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

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

```
import java.rmi.*;
import java.rmi.server.*;
public class AdderRemote extends UnicastRemoteObject implements Adder{
AdderRemote()throws RemoteException{
super();
}
public int add(int x,int y){return x+y;}
}
```

---

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

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

```
rmic AdderRemote
```

---

4) Start the registry service by the rmiregistry tool

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

```
rmiregistry 5000
```

5) Create and run the server application

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

public static java.rmi.Remote lookup(java.lang.String) throws java.rmi.NotBoundException, java.net.MalformedURLException, java.rmi.RemoteException;	It returns the reference of the remote object.
public static void bind(java.lang.String, java.rmi.Remote) throws java.rmi.AlreadyBoundException, java.net.MalformedURLException, java.rmi.RemoteException;	It binds the remote object with the given name.
public static void unbind(java.lang.String) throws java.rmi.RemoteException, java.rmi.NotBoundException, java.net.MalformedURLException;	It destroys the remote object which is bound with the given name.
public static void rebind(java.lang.String, java.rmi.Remote) throws java.rmi.RemoteException, java.net.MalformedURLException;	It binds the remote object to the new name.
public static java.lang.String[] list(java.lang.String) throws java.rmi.RemoteException, java.net.MalformedURLException;	It returns an array of the names of the remote objects bound in the registry.

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

```
import java.rmi.*;
import java.rmi.registry.*;
public class MyServer{
```

```
public static void main(String args[]){
try{
Adder stub=new AdderRemote();
Naming.rebind("rmi://localhost:5000/sonoo",stub);
}catch(Exception e){System.out.println(e);}
}
}
```

6) Create and run the client application

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

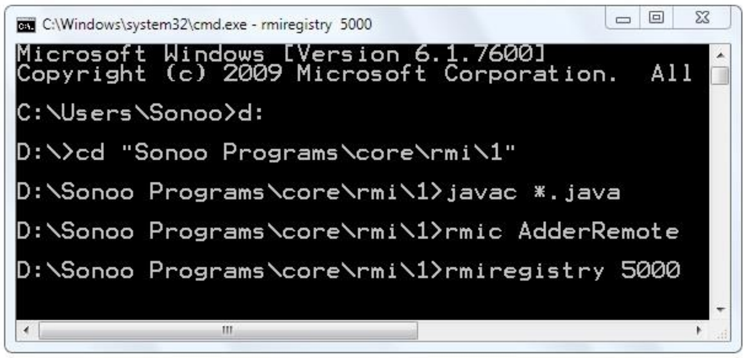
```
import java.rmi.*;
public class MyClient{
public static void main(String args[]){
try{
Adder stub=(Adder)Naming.lookup("rmi://localhost:5000/sonoo");
System.out.println(stub.add(34,4));
}catch(Exception e){}
}
}
```

download this example of rmi

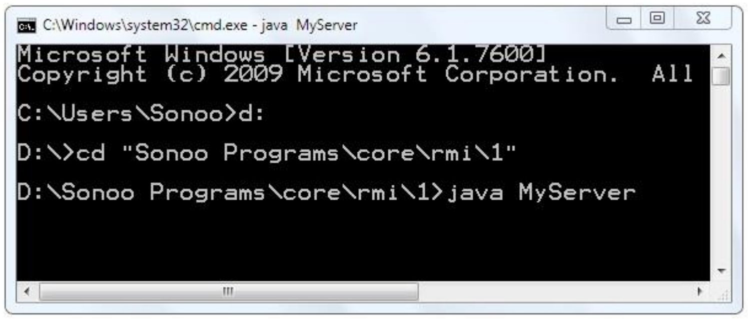
```
For running this rmi example,

1) compile all the java files

javac *.java

2)create stub and skeleton object by rmic tool
```

```

rmic AdderRemote

3)start rmi registry in one command prompt

rmiregistry 5000

4)start the server in another command prompt

java MyServer

5)start the client application in another command prompt

java MyClient
```

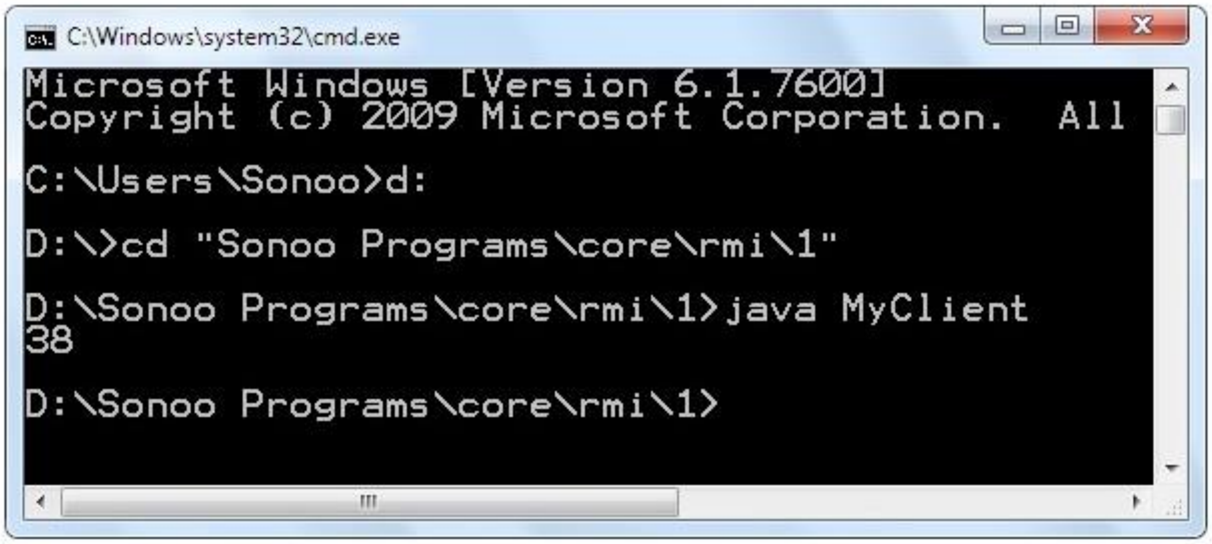
Output of this RMI example



```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All

C:\Users\Sonoo>d:

D:\>cd "Sonoo Programs\core\rmi\1"

D:\Sonoo Programs\core\rmi\1>javac *.java

D:\Sonoo Programs\core\rmi\1>rmic AdderRemote

D:\Sonoo Programs\core\rmi\1>rmiregistry 5000
```



```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All

C:\Users\Sonoo>d:

D:\>cd "Sonoo Programs\core\rmi\1"

D:\Sonoo Programs\core\rmi\1>java MyServer
```



```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All

C:\Users\Sonoo>d:

D:\>cd "Sonoo Programs\core\rmi\1"

D:\Sonoo Programs\core\rmi\1>java MyClient
38

D:\Sonoo Programs\core\rmi\1>
```

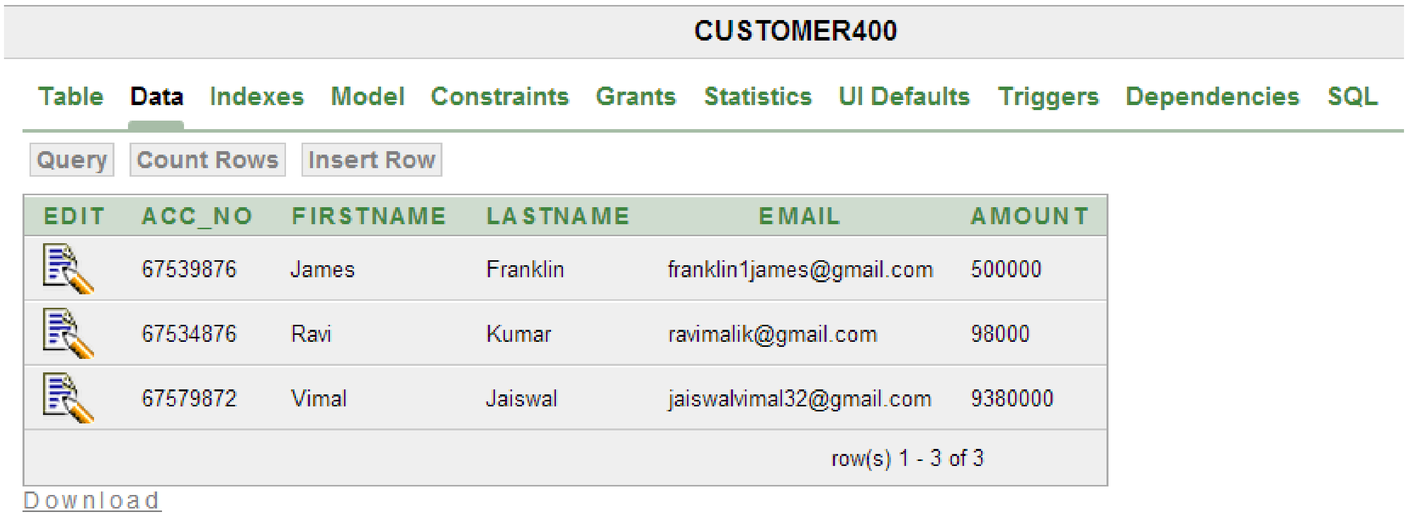
Meaningful example of RMI application with database

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

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

1) Create the table

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



EDIT	ACC_NO	FIRSTNAME	LASTNAME	EMAIL	AMOUNT
	67539876	James	Franklin	franklin1james@gmail.com	500000
	67534876	Ravi	Kumar	ravimalik@gmail.com	98000
	67579872	Vimal	Jaiswal	jaiswalvimal32@gmail.com	9380000

row(s) 1 - 3 of 3

2) Create Customer class and Remote interface

File: Customer.java

```
package com.javatpoint;
public class Customer implements java.io.Serializable{
    private int acc_no;
    private String firstname,lastname,email;
    private float amount;
//getters and setters
}
```

Note: Customer class must be Serializable.

File: Bank.java

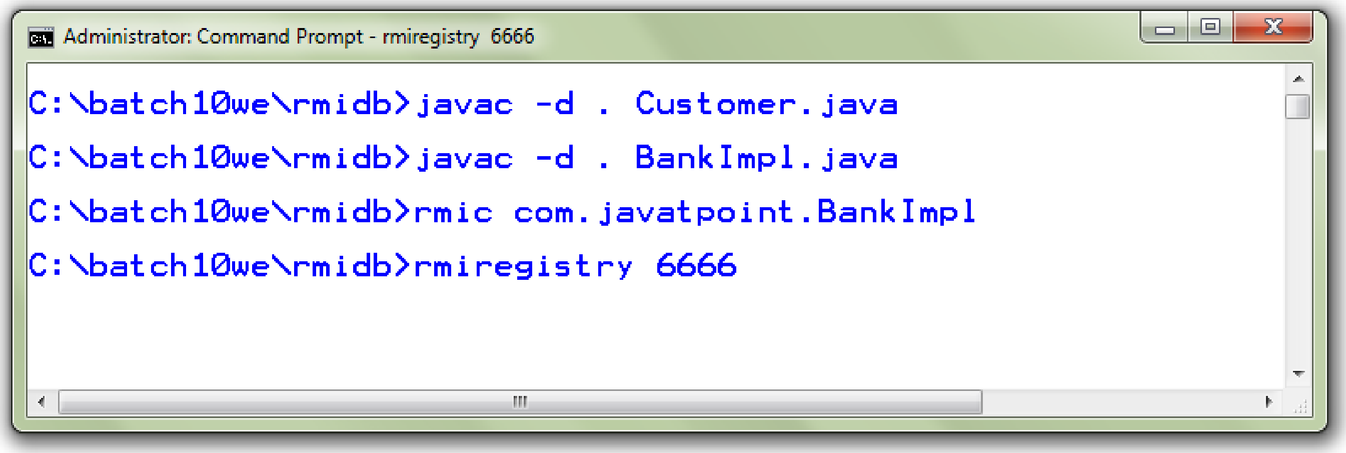
```
package com.javatpoint;
import java.rmi.*;
import java.util.*;
interface Bank extends Remote{
public List<Customer> getCustomers()throws RemoteException;
}
```

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

File: BankImpl.java

```
package com.javatpoint;
import java.rmi.*;
import java.rmi.server.*;
import java.sql.*;
import java.util.*;
class BankImpl extends UnicastRemoteObject implements Bank{
BankImpl()throws RemoteException{}

public List<Customer> getCustomers(){
List<Customer> list=new ArrayList<Customer>();
try{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","
oracle");
PreparedStatement ps=con.prepareStatement("select * from customer400");
ResultSet rs=ps.executeQuery();

while(rs.next()){
Customer c=new Customer();
c.setAcc_no(rs.getInt(1));
c.setFirstname(rs.getString(2));
c.setLastname(rs.getString(3));
c.setEmail(rs.getString(4));
c.setAmount(rs.getFloat(5));
list.add(c);
}

con.close();
}catch(Exception e){System.out.println(e);}
return list;
}//end of getCustomers()
}
```

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



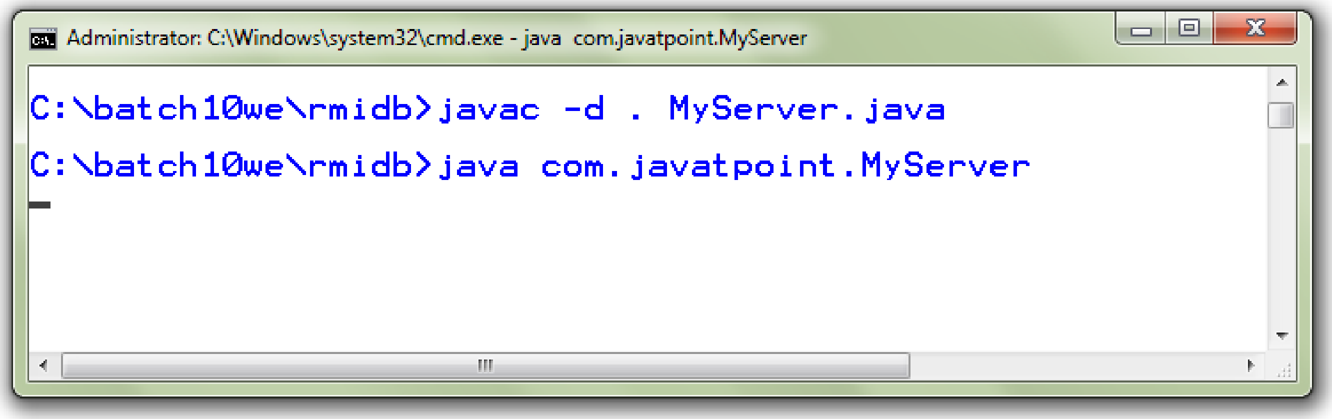
Administrator: Command Prompt - rmiregistry 6666

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

5) Create and run the Server

File: MyServer.java

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



```
C:\batch10we\rmidb>javac -d . MyServer.java

C:\batch10we\rmidb>java com.javatpoint.MyServer
_
```

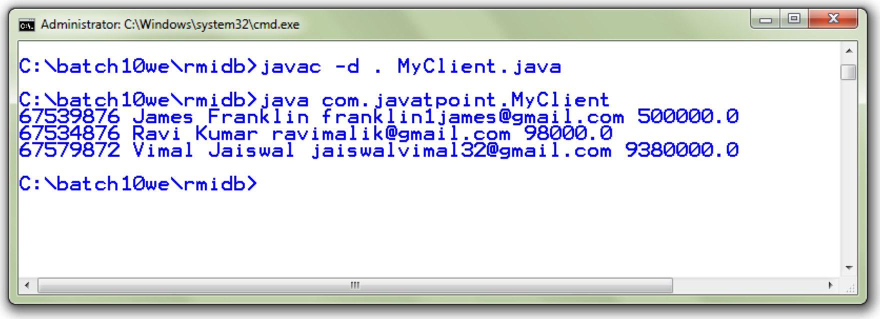
Administrator: C:\Windows\system32\cmd.exe - java com.javatpoint.MyServer

---

6) Create and run the Client

File: MyClient.java

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

List<Customer> list=b.getCustomers();
for(Customer c:list){
System.out.println(c.getAcc_no()+" "+c.getFirstname()+" "+c.getLastname()
+" "+c.getEmail()+" "+c.getAmount());
}

}}
```



```
C:\batch10we\rmidb>javac -d . MyClient.java

C:\batch10we\rmidb>java com.javatpoint.MyClient
67539876 James Franklin franklin1james@gmail.com 500000.0
67534876 Ravi Kumar ravimalik@gmail.com 98000.0
67579872 Vimal Jaiswal jaiswalvimal32@gmail.com 9380000.0

C:\batch10we\rmidb>
```