

JAVA PROGRAMMING

20 MCA 2 2 C

Unit – V

SOFTWARE DEVELOPMENT USING JAVA

FACULTY

Dr. K. ARTHI MCA, M.Phil., Ph.D.,
Assistant Professor,
Postgraduate Department of Computer Applications,
Government Arts College (Autonomous),
Coimbatore – 641 018.

JavaBean

A JavaBean is a Java class that should follow the following conventions:

- It should have a no-arg constructor.
- It should be Serializable.
- It should provide methods to set and get the values of the properties, known as getter and setter methods.

Why use JavaBean?

According to Java white paper, it is a reusable software component. A bean encapsulates many objects into one object so that we can access this object from multiple places. Moreover, it provides easy maintenance.

Simple example of JavaBean class

```
1. //Employee.java
2.
3. package mypack;
4. public class Employee implements java.io.Serializable{
5.     private int id;
6.     private String name;
7.     public Employee(){ }
8.     public void setId(int id){this.id=id;}
9.     public int getId(){return id;}
10.    public void setName(String name){this.name=name;}
11.    public String getName(){return name;}
12. }
```

How to access the JavaBean class?

To access the JavaBean class, we should use getter and setter methods.

```
1. package mypack;
2. public class Test{
3.     public static void main(String args[]){
4.         Employee e=new Employee();//object is created
5.         e.setName("Arjun");//setting value to the object
6.         System.out.println(e.getName()); 7. }}
```

Note: There are two ways to provide values to the object. One way is by constructor and second is by setter method.

JavaBean Properties

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

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

1. getPropertyname ()

For example, if the property name is firstName, the method name would be getFirstName() to read that property. This method is called the accessor.

2. setPropertyName ()

For example, if the property name is firstName, the method name would be setFirstName() to write that property. This method is called the mutator.

Advantages of JavaBean

The following are the advantages of JavaBean:

- The JavaBean properties and methods can be exposed to another application.
- It provides an easiness to reuse the software components.

Disadvantages of JavaBean

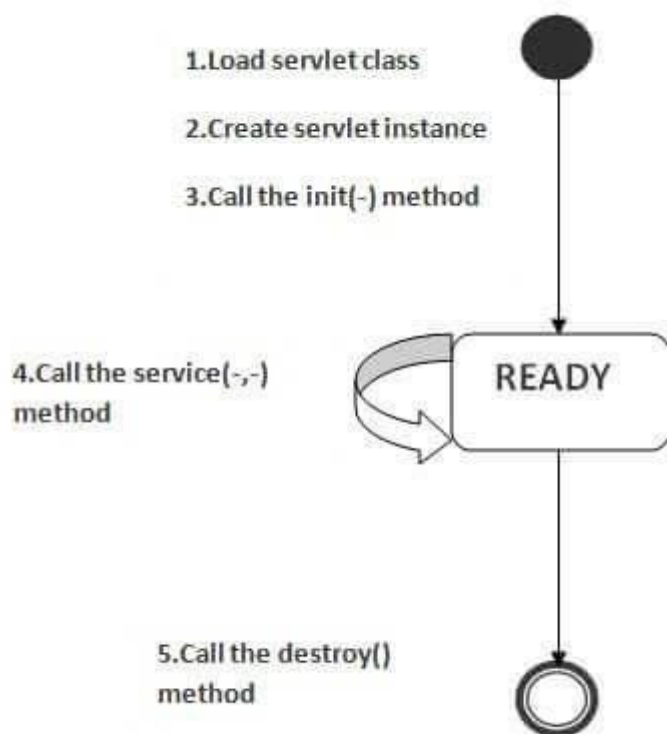
The following are the disadvantages of JavaBean:

- JavaBeans are mutable. So, it can't take advantages of immutable objects.
- Creating the setter and getter method for each property separately may lead to the boilerplate code.

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.

Life Cycle of a Servlet (Servlet Life Cycle)



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

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

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

Steps to create a servlet example

1. [Steps to create the servlet using Tomcat server](#)
1. [Create a directory structure](#)
2. [Create a Servlet](#)
3. [Compile the Servlet](#)
4. [Create a deployment descriptor](#)
5. [Start the server and deploy the application](#)

There are given 6 steps to create a **servlet example**. These steps are required for all the servers.

The servlet example can be created by three ways:

1. By implementing Servlet interface,
2. By inheriting GenericServlet class, (or)
3. By inheriting HttpServlet class

The mostly used approach is by extending `HttpServlet` because it provides http request specific method such as `doGet()`, `doPost()`, `doHead()` etc.

Here, we are going to use **apache tomcat server** in this example. The steps are as follows:

1. Create a directory structure
2. Create a Servlet
3. Compile the Servlet
4. Create a deployment descriptor
5. Start the server and deploy the project
6. Access the servlet

1) Create a directory structures

The **directory structure** defines that where to put the different types of files so that web container may get the information and respond to the client.

The Sun Microsystem defines a unique standard to be followed by all the server vendors. Let's see the directory structure that must be followed to create the servlet.

As you can see that the servlet class file must be in the classes folder. The web.xml file must be under the WEBINF folder.

2) Create a Servlet

There are three ways to create the servlet.

1. By implementing the Servlet interface
2. By inheriting the `GenericServlet` class
3. By inheriting the `HttpServlet` class

The `HttpServlet` class is widely used to create the servlet because it provides methods to handle http requests such as `doGet()`, `doPost`, `doHead()` etc.

In this example we are going to create a servlet that extends the `HttpServlet` class.

In this example, we are inheriting the `HttpServlet` class and providing the implementation of the `doGet()` method. Notice that get request is the default request.

DemoServlet.java

```
1. import javax.servlet.http.*;
2. import javax.servlet.*;
3. import java.io.*;
4. public class DemoServlet extends HttpServlet{
5.     public void doGet(HttpServletRequest req,HttpServletResponse res)
6.     throws ServletException,IOException
7.     {
8.         res.setContentType("text/html");//setting the content type
9.         PrintWriter pw=res.getWriter();//get the stream to write the
           data
10.
11.         //writing html in the stream
```

12. pw.println("<html><body>");
13. pw.println("Welcome to servlet");
14. pw.println("</body></html>");
- 15.
16. pw.close();//closing the stream
17. }}

3)Compile the servlet

For compiling the Servlet, jar file is required to be loaded. Different Servers provide different jar files:

Jar file	Server
1) servlet-api.jar	Apache Tomcat
2) weblogic.jar	Weblogic
3) javaee.jar	Glassfish
4) javaee.jar	JBoss

Two ways to load the jar file

1. set classpath
2. paste the jar file in JRE/lib/ext folder

Put the java file in any folder. After compiling the java file, paste the class file of servlet in **WEBINF/classes** directory.

4)Create the deployment descriptor (web.xml file)

The **deployment descriptor** is an xml file, from which Web Container gets the information about the servlet to be invoked.

The web container uses the Parser to get the information from the web.xml file. There are many xml parsers such as SAX, DOM and Pull.

There are many elements in the web.xml file. Here is given some necessary elements to run the simple servlet program.

web.xml file

1. **<web-app>**
- 2.
3. **<servlet>**
4. **<servlet-name>sonoojaiswal</servlet-name>**
5. **<servlet-class>DemoServlet</servlet-class>**
6. **</servlet>**
- 7.
8. **<servlet-mapping>**
9. **<servlet-name>sonoojaiswal</servlet-name>**
10. **<url-pattern>/welcome</url-pattern>**

11. `</servlet-mapping>`

12.

13. `</web-app>`

Description of the elements of web.xml file

There are too many elements in the web.xml file. Here is the illustration of some elements that is used in the above web.xml file. The elements are as follows:

`<web-app>` represents the whole application.

`<servlet>` is sub element of `<web-app>` and represents the servlet.

`<servlet-name>` is sub element of `<servlet>` represents the name of the servlet.

`<servlet-class>` is sub element of `<servlet>` represents the class of the servlet.

`<servlet-mapping>` is sub element of `<web-app>`. It is used to map the servlet.

`<url-pattern>` is sub element of `<servlet-mapping>`. This pattern is used at client side to invoke the servlet.

5)Start the Server and deploy the project

To start Apache Tomcat server, double click on the startup.bat file under apache-tomcat/bin directory.

One Time Configuration for Apache Tomcat Server

You need to perform 2 tasks:

1. set JAVA_HOME or JRE_HOME in environment variable (It is required to start server).
2. Change the port number of tomcat (optional). It is required if another server is running on same port (8080).

1) How to set JAVA_HOME in environment variable?

To start Apache Tomcat server JAVA_HOME and JRE_HOME must be set in Environment variables.

Go to My Computer properties -> Click on advanced tab then environment variables -> Click on the new tab of user variable -> Write JAVA_HOME in variable name and paste the path of jdk folder in variable value -> ok -> ok -> ok.

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](#))

Handling HTTP Requests

The request sent by the computer to a web server, contains all sorts of potentially interesting information; it is known as HTTP requests.

The HTTP client sends the request to the server in the form of request message which includes following information:

- The Request-line
- The analysis of source IP address, proxy and port
- The analysis of destination IP address, protocol, port and host
- The Requested URI (Uniform Resource Identifier)
- The Request method and Content
- The User-Agent header
- The Connection control header
- The Cache control header

The HTTP request method indicates the method to be performed on the resource identified by the **Requested URI (Uniform Resource Identifier)**. This method is case-sensitive and should be used in uppercase.

The HTTP request methods are:

HTTP Request	Description
GET	Asks to get the resource at the requested URL.
POST	Asks the server to accept the body info attached. It is like GET request with extra info sent with the request.
HEAD	Asks for only the header part of whatever a GET would return. Just like GET but with no body.
TRACE	Asks for the loopback of the request message, for testing or troubleshooting.
PUT	Says to put the enclosed info (the body) at the requested URL.
DELETE	Says to delete the resource at the requested URL.
OPTIONS	Asks for a list of the HTTP methods to which the thing at the request URL can respond

HTTP Response

HTTP Response sent by a server to the client. The response is used to provide the client with the resource it requested. It is also used to inform the client that the action requested has been carried out. It can also inform the client that an error occurred in processing its request.

An HTTP response contains the following things:

1. Status Line
2. Response Header Fields or a series of HTTP headers
3. Message Body

In the request message, each HTTP header is followed by a carriage returns line feed (CRLF). After the last of the HTTP headers, an additional CRLF is used and then begins the message body.

Status Line

In the response message, the status line is the first line. The status line contains three items: **a) HTTP**

Version Number

It is used to show the HTTP specification to which the server has tried to make the message comply.

Example

1. HTTP-Version = HTTP/1.1

b) Status Code

It is a three-digit number that indicates the result of the request. The first digit defines the class of the response. The last two digits do not have any categorization role. There are five values for the first digit, which are as follows:

Code and Description

1xx: Information

It shows that the request was received and continuing the process.

2xx: Success

It shows that the action was received successfully, understood, and accepted.

3xx: Redirection

It shows that further action must be taken to complete the request.

4xx: Client Error

It shows that the request contains incorrect syntax, or it cannot be fulfilled. **5xx: Server**

Error

It shows that the server failed to fulfil a valid request.

c) Reason Phrase

It is also known as the status text. It is a human-readable text that summarizes the meaning of the status code.

An example of the response line is as follows:

1. HTTP/1.1 200 OK

Here,

- HTTP/1.1 is the HTTP version. ○ 200 is the status code. ○ OK is the reason phrase.

Response Header Fields

The HTTP Headers for the response of the server contain the information that a client can use to find out more about the response, and about the server that sent it. This information is used to assist the client with displaying the response to a user, with storing the response for the use of future, and with making further requests to the server now or in the future.

1. response-header = Accept-Ranges
2. | Age
3. | ETag
4. | Location
5. | Proxy-Authenticate
6. | Retry-After
7. | Server
8. | Vary
9. | WWW-Authenticate

The name of the Response-header field can be extended reliably only in combination with a change in the version of the protocol.

Message Body

The response's message body may be referred to for convenience as a response body.

The body of the message is used for most responses. The exceptions are where a server is using certain status codes and where the server is responding to a client request, which asks for the headers but not the response body.

For a response to a successful request, the body of the message contains either some information about the status of the action which is requested by the client or the resource which is requested by the client. For the response to an unsuccessful request, the body of the message might provide further information about some action the client needs to take to complete the request successfully or about the reason for the error.

Session Tracking in Servlets

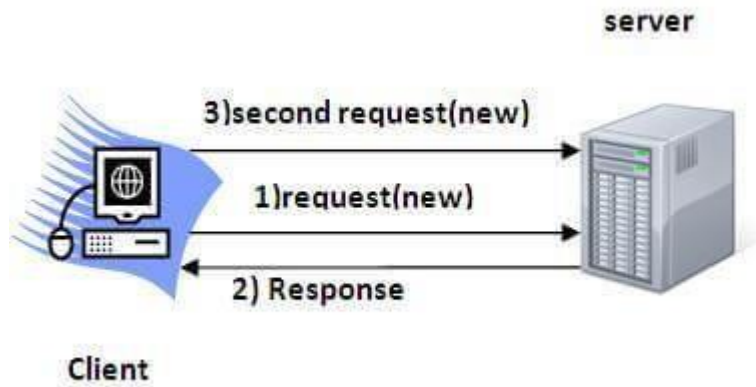
1. [Session Tracking](#)
2. [Session Tracking Techniques](#)

Session simply means a particular interval of time.

Session Tracking is a way to maintain state (data) of an user. It is also known as **session management** in servlet.

Http protocol is a stateless so we need to maintain state using session tracking techniques. Each time user requests to the server, server treats the request as the new request. So we need to maintain the state of an user to recognize to particular user.

HTTP is stateless that means each request is considered as the new request. It is shown in the figure given below:



Why use Session Tracking?

To recognize the user It is used to recognize the particular user.

Session Tracking Techniques

There are four techniques used in Session tracking:

1. **Cookies**
2. **Hidden Form Field**
3. **URL Rewriting**
4. **HttpSession**

Networking basics

RMI (Remote Method Invocation)

1. [Remote Method Invocation \(RMI\)](#)
2. [Understanding stub and skeleton](#)
 1. [stub](#)
 2. [skeleton](#)
3. [Requirements for the distributed applications](#)
4. [Steps to write the RMI program](#)
5. [RMI Example](#)

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

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

Understanding stub and skeleton

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

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

stub

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

1. It initiates a connection with remote Virtual Machine (JVM),

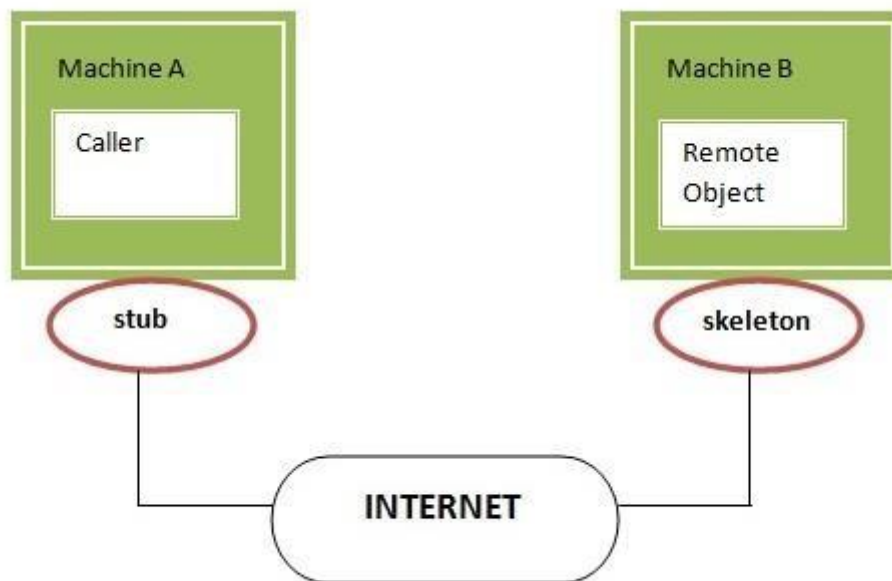
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads (unmarshals) the return value or exception, and
5. It finally, returns the value to the caller.

skeleton

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

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

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



Understanding requirements for the distributed applications

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

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

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

Java RMI Example

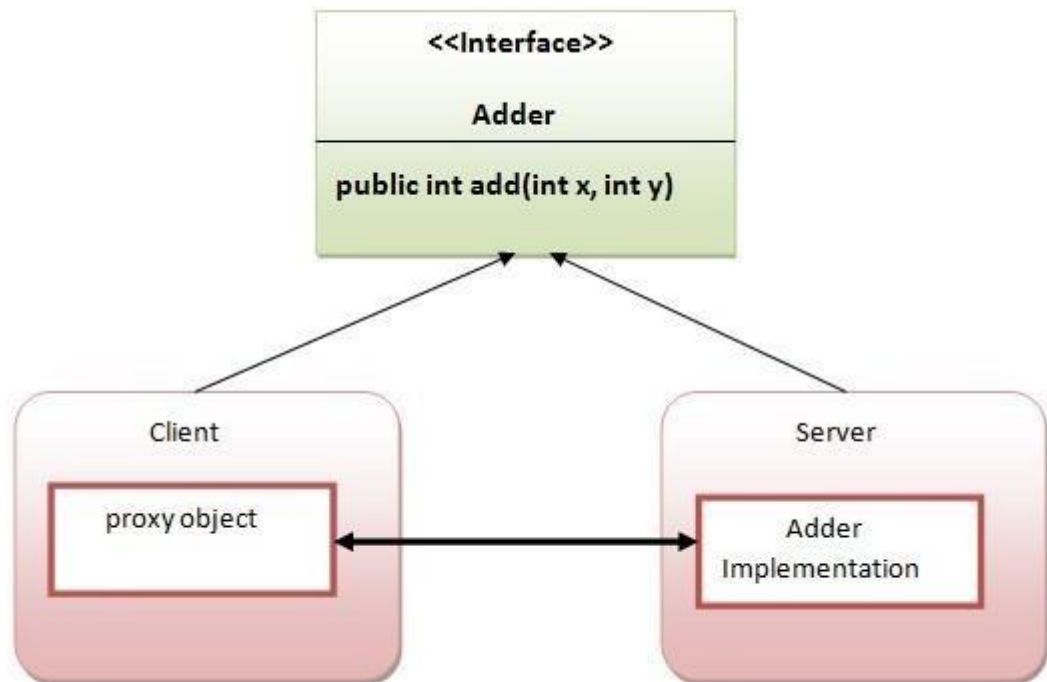
The is given the 6 steps to write the RMI program.

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

5. Create and start the remote application
6. Create and start the client application

RMI Example

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



1) create the remote interface

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

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

2) Provide the implementation of the remote interface

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

we need to ○ Either extend the UnicastRemoteObject class,

- or use the exportObject() method of the UnicastRemoteObject class In case, you extend the UnicastRemoteObject

class, you must define a constructor that declares RemoteException.

1. **import** java.rmi.*;
2. **import** java.rmi.server.*;
3. **public class** AdderRemote **extends** UnicastRemoteObject **implements** Adder{
4. AdderRemote()**throws** RemoteException{
5. **super**();
6. }

```

7. public int add(int x,int y){return x+y;}
8. }

```

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

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

```
1. rmic AdderRemote
```

4) Start the registry service by the rmiregistry tool

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

```
1. rmiregistry 5000
```

5) Create and run the server application

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

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

```

1. import java.rmi.*;
2. import java.rmi.registry.*;
3. public class MyServer{
4. public static void main(String args[]){
5. try{
6. Adder stub=new AdderRemote();
7. Naming.rebind("rmi://localhost:5000/sonoo",stub);

```

```
8. }catch(Exception e){System.out.println(e);}
9. }
10. }
```

6) Create and run the client application

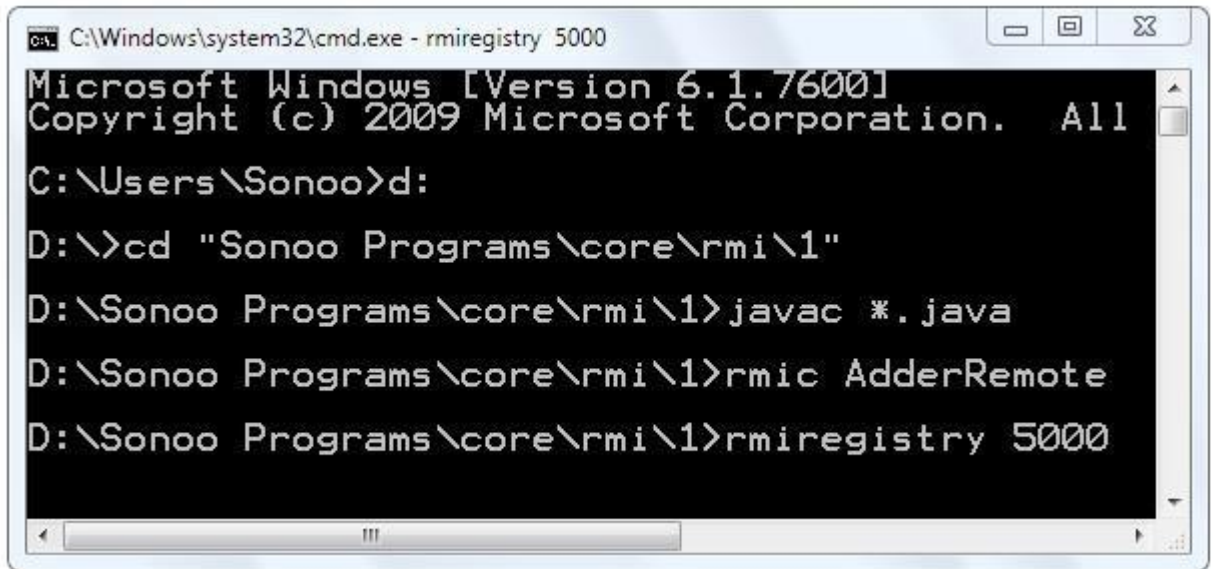
At the client we are getting the stub object by the lookup() method of the Naming class and invoking the method on this object. In this example, we are running the server and client applications, in the same machine so we are using localhost. If you want to access the remote object from another machine,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

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

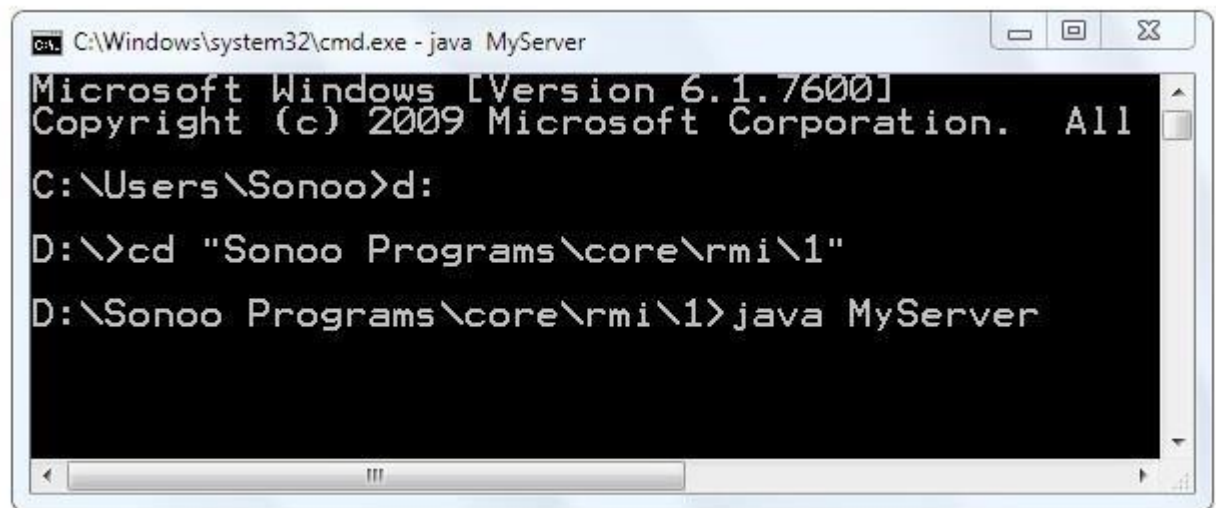
[download this example of rmi](#)

1. For running **this** rmi example,
- 2.
3. **1)** compile all the java files
- 4.
5. javac *.java
- 6.
7. **2)**create stub and skeleton object by rmic tool
- 8.
9. rmic AdderRemote
- 10.
11. **3)**start rmi registry in one command prompt
- 12.
13. rmiregistry **5000**
- 14.
15. **4)**start the server in another command prompt
- 16.
17. java MyServer
- 18.
19. **5)**start the client application in another command prompt
- 20.
21. java MyClient

Output of this RMI example



```
C:\Windows\system32\cmd.exe - rmiregistry 5000
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All
C:\Users\Sonoo>d:
D:\>cd "Sonoo Programs\core\rmi\1"
D:\Sonoo Programs\core\rmi\1>javac *.java
D:\Sonoo Programs\core\rmi\1>rmic AdderRemote
D:\Sonoo Programs\core\rmi\1>rmiregistry 5000
```



```
C:\Windows\system32\cmd.exe - java MyServer
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All
C:\Users\Sonoo>d:
D:\>cd "Sonoo Programs\core\rmi\1"
D:\Sonoo Programs\core\rmi\1>java MyServer
```

Java JDBC

Why Should We Use JDBC

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

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

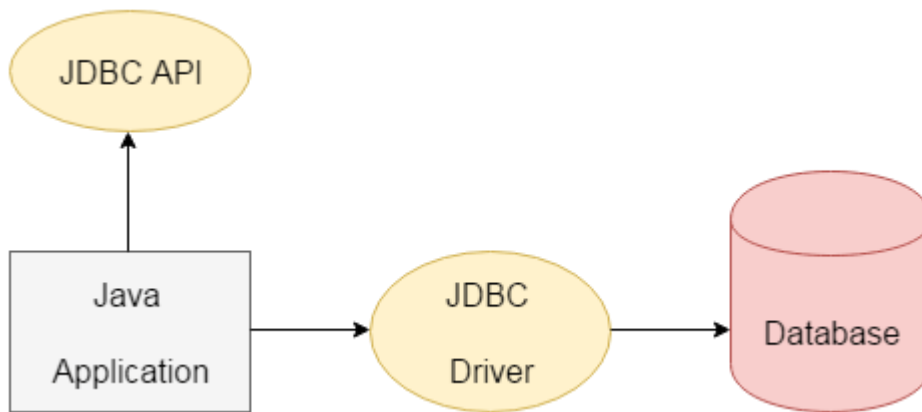
1. Connect to the database
2. Execute queries and update statements to the database
3. Retrieve the result received from the database.

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

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

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

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



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

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

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

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

JDBC Driver

1. [JDBC Drivers](#)
1. [JDBC-ODBC bridge driver](#)
2. [Native-API driver](#)
3. [Network Protocol driver](#)
4. [Thin driver](#)

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

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

1) JDBC-ODBC bridge driver

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

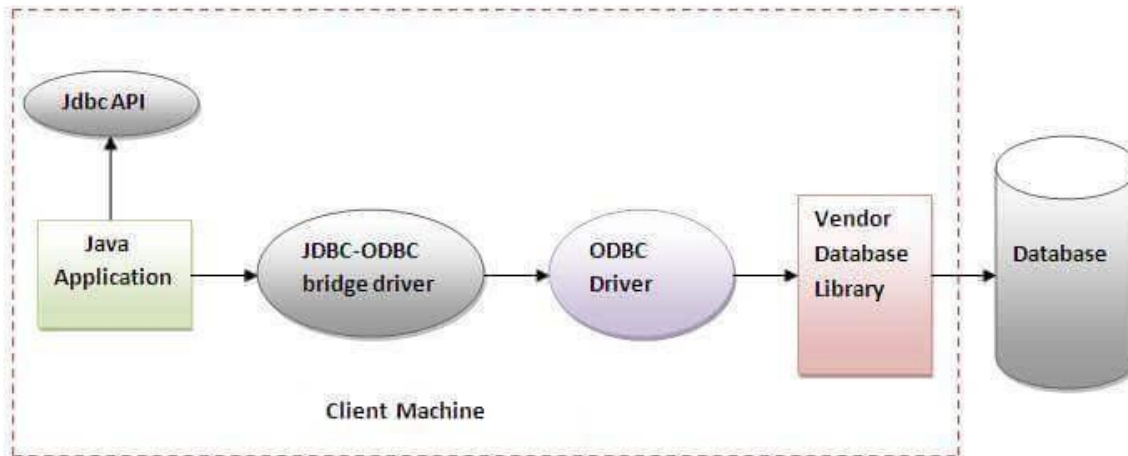


Figure- JDBC-ODBC Bridge Driver

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

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

Advantages:

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

Disadvantages:

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

2) Native-API driver

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

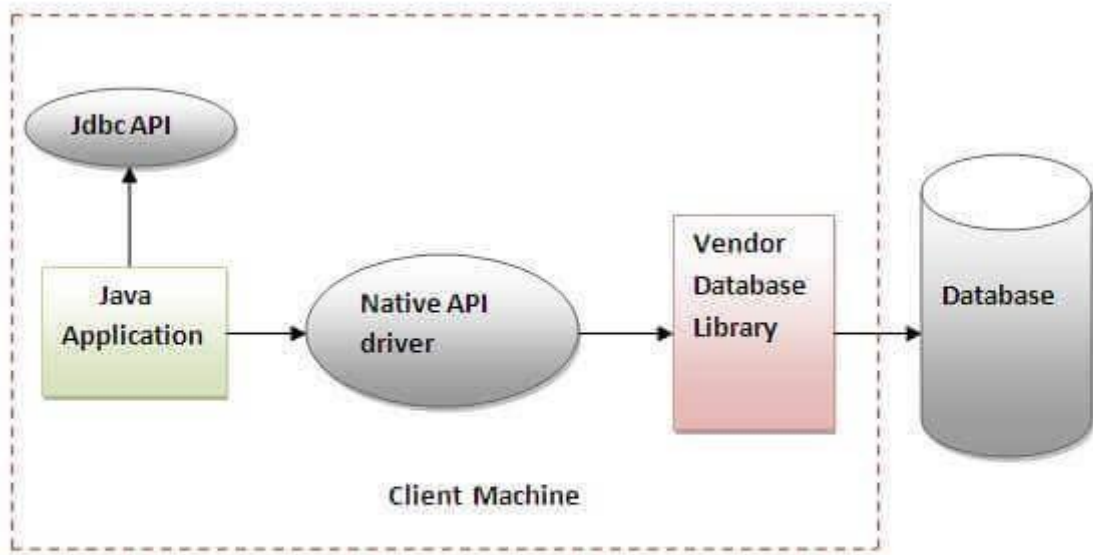


Figure- Native API Driver

Advantage:

- performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

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

3) Network Protocol driver

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

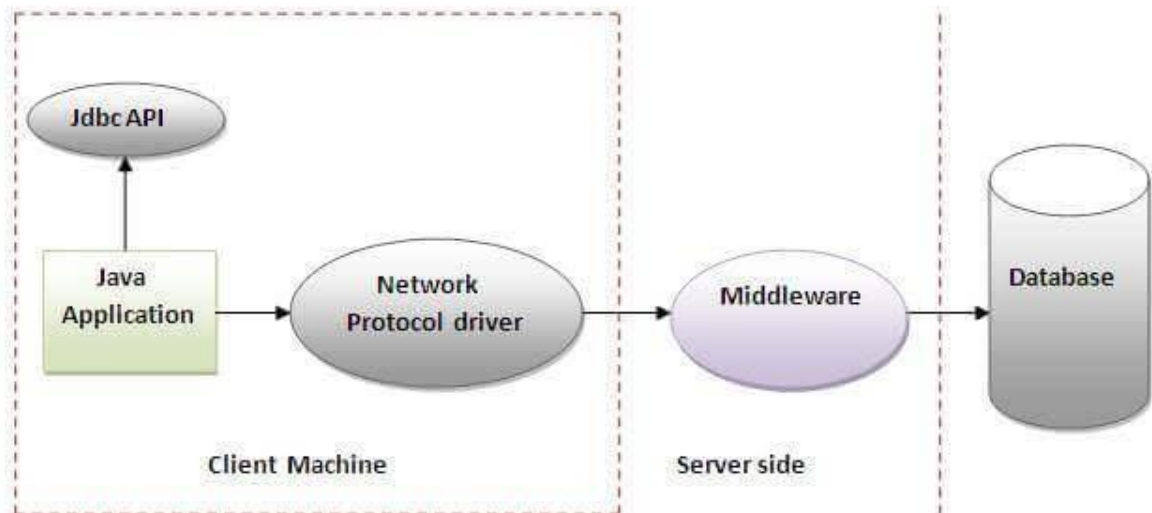


Figure- Network Protocol Driver

Advantage:

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

Disadvantages:

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

4) Thin driver

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

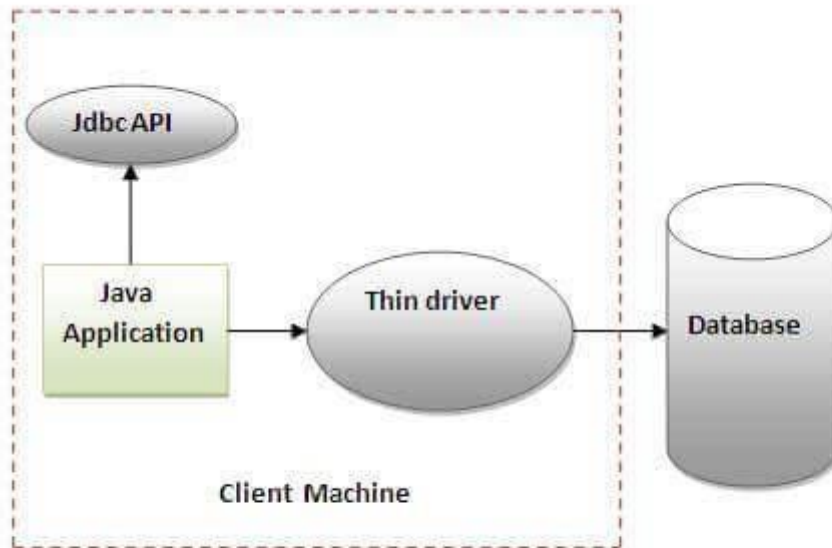


Figure- Thin Driver

Advantage:

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

Disadvantage: ○ Drivers depend on the Database.

Java Database Connectivity with 5 Steps

1. 5 Steps to connect to the database in java
1. Register the driver class
2. Create the connection object
3. Create the Statement object
4. Execute the query
5. Close the connection object

There are 5 steps to connect any java application with the database using JDBC.

These steps are as follows:

- Register the Driver class ○
Create connection ○
Create statement
- Execute queries ○ Close
connection

1) Register the driver class

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

Syntax of forName() method

1. **public static void** forName(String className)**throws** ClassNotFoundException

Note: Since JDBC 4.0, explicitly registering the driver is optional. We just need to put vendor's Jar in the classpath, and then JDBC driver manager can detect and load the driver automatically.

Example to register the OracleDriver class

Here, Java program is loading oracle driver to establish database connection.

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

2) Create the connection object

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

Syntax of getConnection() method

1. **1) public static** Connection getConnection(String url)**throws** SQLException
2. **2) public static** Connection getConnection(String url,String name,String password)
3. **throws** SQLException

Example to establish connection with the Oracle database

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

3) Create the Statement object

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

Syntax of createStatement() method

1. **public** Statement createStatement()**throws** SQLException

Example to create the statement object

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

4) Execute the query

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

Syntax of executeQuery() method

1. **public** ResultSet executeQuery(String sql)**throws** SQLException

Example to execute query

1. `ResultSet rs=stmt.executeQuery("select * from emp");`
2.
3. `while(rs.next()){`
4. `System.out.println(rs.getInt(1)+" "+rs.getString(2));`
5. `}`

5) Close the connection object

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

Syntax of close() method

1. **public void** close()**throws** SQLException

Example to close connection

1. con.close();



THANK YOU

The content for this material is taken from the prescribed
Textbooks and Reference books.

