
# II Year/III Semester

# CS23312- Object Oriented Programming

# Unit-IV

**Prepared By** **Verified By** **Approved By**

# UNIT IV

## JAVA NETWORKING & JDBC

Socket programming in Java-InetAddress and URL classes-TCP and UDP protocols in Java-Server Socket and Socket Classes-Multi-threaded servers-Handling multiple client connections- Introduction to RMI-Creating RMI servers and clients-RMI registry-RMI and object serialization- Overview of JDBC-JDBC drivers-Connecting to databases-Executing SQL queries.

Socket programming in Java is a powerful technique for building networked applications. It allows communication between devices over a network, either using TCP (Transmission Control Protocol) or UDP (User Datagram Protocol). In socket programming, sockets are the endpoints of two-way communication between machines.

Here's a basic introduction to socket programming in Java, along with examples of both a **server** and a **client** using **TCP**.

### Key Classes for Socket Programming in Java

- **`java.net.Socket`**: This class implements a client socket that connects to a server.
- **`java.net.ServerSocket`**: This class is used to create a server that listens for incoming client requests.
- **`java.io.InputStream`** and **`java.io.OutputStream`**: These are used for reading from and writing to the socket.

### Example 1: Simple TCP Server

The server listens for incoming client requests, accepts them, and can send or receive data.

```
java
Copy code
import java.io.*;
import java.net.*;

public class SimpleServer {
    public static void main(String[] args) {
        try {
            // Create a server socket on port 5000
            ServerSocket serverSocket = new ServerSocket(5000);
            System.out.println("Server is listening on port 5000...");

            // Accept client connection
            Socket clientSocket = serverSocket.accept();
            System.out.println("Client connected!");

            // Create input and output streams for communication
            InputStream input = clientSocket.getInputStream();
            BufferedReader reader = new BufferedReader(new
InputStreamReader(input));
            OutputStream output = clientSocket.getOutputStream();
```

```
            PrintWriter writer = new PrintWriter(output, true);

            // Read message from client
            String clientMessage = reader.readLine();
            System.out.println("Client says: " + clientMessage);

            // Send response to client
            writer.println("Hello from server!");

            // Close resources
            clientSocket.close();
            serverSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

## Example 2: Simple TCP Client

The client connects to the server and exchanges data.

```
java
Copy code
import java.io.*;
import java.net.*;

public class SimpleClient {
    public static void main(String[] args) {
        try {
            // Connect to the server running on localhost, port 5000
            Socket socket = new Socket("localhost", 5000);

            // Create input and output streams for communication
            OutputStream output = socket.getOutputStream();
            PrintWriter writer = new PrintWriter(output, true);
            InputStream input = socket.getInputStream();
            BufferedReader reader = new BufferedReader(new
InputStreamReader(input));

            // Send message to server
            writer.println("Hello from client!");

            // Read response from server
            String serverResponse = reader.readLine();
            System.out.println("Server says: " + serverResponse);

            // Close resources
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

## How it Works:

1. **Server:**
    - Listens for connections on port 5000.

    - When a client connects, it establishes a socket and communicates by reading and writing data.
2. **Client:**
    - Connects to the server on the same port (5000).
    - Sends a message and receives a response from the server.

### Steps to Run:

1. Compile both the server and client programs.
2. Run the server program first, which will wait for incoming connections.
3. Run the client program, which will connect to the server, send a message, and display the response.

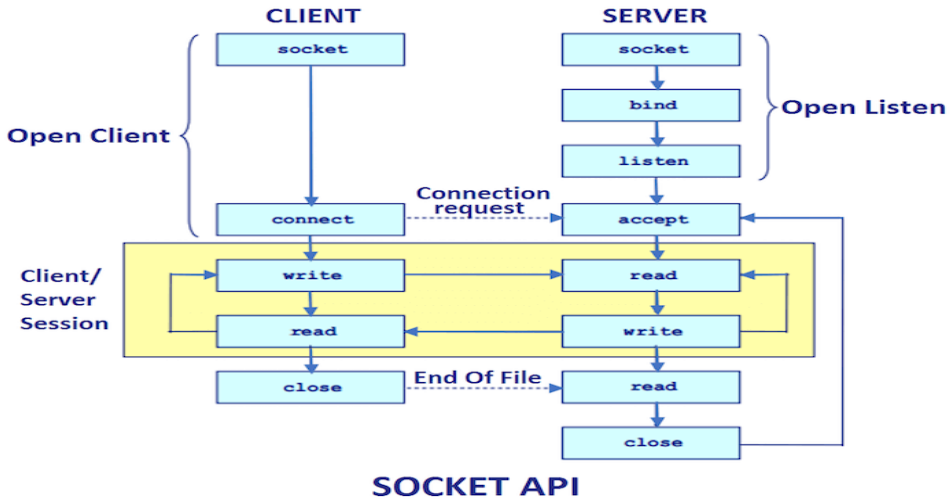
### Key Points:

- **Blocking Behavior**: Both `accept()` and `readLine()` are blocking operations, meaning they will pause execution until a connection or data is available.
- **Multi-threading**: To handle multiple clients, the server should spawn a new thread for each connected client.

## 1) Socket programming in Java

- Java Socket programming is used for communication between the applications running on different JRE.
- Java Socket programming can be connection-oriented or connection-less.
- Socket and ServerSocket classes are used for connection-oriented socket programming and DatagramSocket and DatagramPacket classes are used for connection-less socket programming.
- The client in socket programming must know two information:
    1. IP Address of Server, and
    2. Port number.

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




-

**Client-Side Programming**

Establish a Socket Connection

To connect to another machine we need a socket connection. A socket connection means the two machines have information about each other's network location (IP Address) and TCP port. The java.net.Socket class represents a Socket. To open a socket:

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

- The first argument – IP address of Server. ( 127.0.0.1 is the IP address of localhost, where code will run on the single stand-alone machine).
- The second argument – TCP Port. (Just a number representing which application to run on a server. For example, HTTP runs on port 80. Port number can be from 0 to 65535).

Closing the connection

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

**Socket class:**

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

IMPORTANT METHODS:

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

**ServerSocket class:**

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

IMPORTANT METHODS:

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

**Creating Server:**

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

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

**Creating Client:**

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

```
Socket s=new Socket("localhost",6666);
```

EXAMPLE:

***1)File: MyServer.java***

```
import java.io.*;
import java.net.*;
public class MyServer {
```

```
public static void main(String[] args){
try{
ServerSocket ss=new ServerSocket(6666);
Socket s=ss.accept();//establishes connection
DataInputStream dis=new DataInputStream(s.getInputStream());
String  str=(String)dis.readUTF();
System.out.println("message= "+str);
ss.close();
}catch(Exception e){System.out.println(e);}
}
}
```

***2) File: MyClient.java***

```
import java.io.*;
import java.net.*;
public class MyClient {
public static void main(String[] args) {
try{
Socket s=new Socket("localhost",6666);
DataOutputStream dout=new DataOutputStream(s.getOutputStream());
dout.writeUTF("Hello Server");
dout.flush();
dout.close();
s.close();
}catch(Exception e){System.out.println(e);}
}
}
```

## *2)InetAddress and URL classes*

**Java InetAddress class**

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

An IP address is represented by 32-bit or 128-bit unsigned number. An instance of InetAddress represents the IP address with its corresponding host name. There are two types of addresses: Unicast and Multicast. The Unicast is an identifier for a single interface whereas Multicast is an identifier for a set of interfaces.

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

**IP Address**

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

There are two versions of IP address:

**1. IPv4**

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

**Features of IPv4:**

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

**2. IPv6**

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

**Features of IPv6:**

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

- It has a hierarchical addressing and routing infrastructure.

## TCP/IP Protocol

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

## Java InetAddress Class Methods

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

**EXAMPLE:**

**File:InetDemo.java**

```
import java.io.*;
import java.net.*;
public class InetDemo{
public static void main(String[] args){
try{
InetAddress ip=InetAddress.getByName("www.javatpoint.com");
```

```
System.out.println("Host Name: "+ip.getHostName());
System.out.println("IP Address: "+ip.getHostAddress());
}catch(Exception e){System.out.println(e);}
}
}
```

***Output:***

Host Name: www.javatpoint.com

IP Address: 172.67.196.82

## Java URL

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

For example:

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

A URL contains many information:

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

**Constructors of Java URL class**

**URL(String spec)**

Creates an instance of a URL from the String representation.

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

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

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

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

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

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

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

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

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

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

## Commonly used methods of Java URL class

| Method | Description |
|---|---|
| public String getProtocol() | it returns the protocol of the URL. |
| public String getHost() | it returns the host name of the URL. |
| public String getPort() | it returns the Port Number of the URL. |
| public String getFile() | it returns the file name of the URL. |
| public String getAuthority() | it returns the authority of the URL. |
| public String toString() | it returns the string representation of the URL. |
| public String getQuery() | it returns the query string of the URL. |
| public String getDefaultPort() | it returns the default port of the URL. |
| public URLConnection openConnection() | it returns the instance of URLConnection i.e. associated with this URL. |
| public boolean equals(Object obj) | it compares the URL with the given object. |
| public Object getContent() | it returns the content of the URL. |
| public String getRef() | it returns the anchor or reference of the URL. |
| public URI toURI() | it returns a URI of the URL. |

Example of Java URL class

1)//URLDemo.java

```
import java.net.*;
public class URLDemo{
public static void main(String[] args){
try{
URL url=new URL("http://www.javatpoint.com/java-tutorial");
System.out.println("Protocol: "+url.getProtocol());
System.out.println("Host Name: "+url.getHost());
System.out.println("Port Number: "+url.getPort());
System.out.println("File Name: "+url.getFile());

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

**Output:**

```
Protocol: http
Host Name: www.javatpoint.com
Port Number: -1
File Name: /java-tutorial
```

2)//URLDemo.java

```
import java.net.*;
public class URLDemo{
public static void main(String[] args){
try{
URL url=new URL("https://www.google.com/search?q=javatpoint&oq=javatpoint&s
ourceid=chrome&ie=UTF-8");

System.out.println("Protocol: "+url.getProtocol());
System.out.println("Host Name: "+url.getHost());
System.out.println("Port Number: "+url.getPort());
System.out.println("Default Port Number: "+url.getDefaultPort());
System.out.println("Query String: "+url.getQuery());
System.out.println("Path: "+url.getPath());
System.out.println("File: "+url.getFile());
  }catch(Exception e){System.out.println(e);}
}
 }
```

**Output:**

```
Protocol: https
Host Name: www.google.com
Port Number: -1
Default Port Number: 443
Query String: q=javatpoint&oq=javatpoint&sourceid=chrome&ie=UTF-8
Path: /search
```

File: /search?q=javatpoint&oq=javatpoint&sourceid=chrome&ie=UTF-8

## 3) TCP and UDP protocols in Java

### What is the TCP?

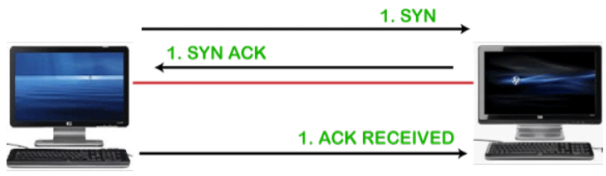
The TCP stands for **Transmission Control Protocol**. If we want the communication between two computers and communication should be good and reliable. For example, we want to view a web page, then we expect that nothing should be missing on the page, or we want to download a file, then we require a complete file, i.e., nothing should be missing either it could be a text or an image. This can only be possible due to the TCP. It is one of the most widely used protocols over the TCP/IP network.

Features of TCP

**The following are the features of the TCP:**

- **Data delivery**
  TCP protocol ensures that the data is received correctly, no data is missing and in order. If TCP protocol is not used, then the incorrect data can be received or out of order. For example, if we try to view the web page or download a file without using TCP, then some data or images could be missing.

- **Protocol**
  TCP is a connection-oriented protocol. Through the word **connection-oriented,** we understand that the computers first establish a connection and then do the communication. This is done by using a three-way handshake. In a **three-way handshake,** the first sender sends the SYN message to the receiver then the receiver sends back the SYN ACK message to confirm that the message has been received. After receiving the **SYN ACK** message, the sender sends the acknowledgment message to the receiver. In this way, the connection is established between the computers. Once the connection is established, the data will be delivered. This protocol guarantees the data delivery means that if the data is not received then the TCP will resend the data.




Connection oriented protocol

### What is UDP?

The UDP stands for **User Datagram Protocol**. Its working is similar to the TCP as it is also used for sending and receiving the message. The main difference is that UDP is a

connectionless protocol. Here, connectionless means that no connection establishes prior to communication. It also does not guarantee the delivery of data packets. It does not even care whether the data has been received on the receiver's end or not, so it is also known as the "fire-and-forget" protocol. It is also known as the **"fire-and-forget"** protocol as it sends the data and does not care whether the data is received or not. UDP is faster than TCP as it does not provide the assurance for the delivery of the packets.

| Factor | TCP | UDP |
|---|---|---|
| Connection type | Requires an established connection before transmitting data | No connection is needed to start and end a data transfer |
| Data sequence | Can sequence data (send in a specific order) | Cannot sequence or arrange data |
| Data retransmission | Can retransmit data if packets fail to arrive | No data retransmitting. Lost data can't be retrieved |
| Delivery | Delivery is guaranteed | Delivery is not guaranteed |
| Check for errors | Thorough error-checking guarantees data arrives in its intended state | Minimal error-checking covers the basics but may not prevent all errors |
| Broadcasting | Not supported | Supported |
| Speed | Slow, but complete data delivery | Fast, but at risk of incomplete data delivery |

## 4)ServerSocket and Socket Classes

**Server socket:**

ServerSocket Class is used for providing system-independent implementation of the server-side of a client/server Socket Connection. The constructor for ServerSocket throws an exception if it can't listen on the specified port (for example, the port is already being used).

It is widely used so the applications of java.net.ServerSocket class which is as follows:

1. In java.nio channel, ServerSocket class is used for retrieving a serverSocket associated with this channel.
2. In java.rmi.Server, ServerSocket class is used to create a server socket on the specified port (port 0 indicates an anonymous port).
3. In javax.net, Server socket is used widely so as to:
    - return an unbound server socket.
    - return a server socket bound to the specified port.
    - return a server socket bound to the specified port, and uses the specified connection backlog.
    - return a server socket bound to the specified port, with a specified listen backlog and local IP.

## Methods of Server Socket Class

There are a lot of methods present in the Server Socket Class in Java. Some of them are discussed below:

**1)The bind() method**
Before using the socket, we must bind it to a specific address. This can be achieved with the help of the bind() method. The address is defined by the combination of IP Address and Port number.

**2)The accept() method**
The accept method makes a blocking call by waiting for the incoming connections on this socket. It is used to accept incoming connections.

**3)The getLocalPort() method**
This method is used to find the local port number on which the socket in discussion is running.

**4)The getInetAddress() method**
This function returns the local address of the server socket.

**5)The close() method**
This method plays a very crucial role in creating programs involving Server Socket Class in Java. It is used to close the socket. It is essential to close the socket so that it can be used again for other purposes.

## Example Program

In this section, we will look at a sample program containing a client and a server. In this program, the client and server establish a connection, and upon successful connection, they exchange a message between them.

**Client-Side Program**

The following is the program for the **Client.java** file.

**Code:**

```
// Java Client Side Program

import java.io.*;

import java.net.*;

public class Client {

public static void main(String[] args){

    try {
```

```
        System.out.println("Client program started");
int PORT = 9000;
String IP = "127.0.0.1";
Socket conn = new Socket(IP, PORT);
DataOutputStream dos = new DataOutputStream(conn.getOutputStream());
String mssge = "Hello Ninja";
 dos.writeUTF(mssge);
System.out.println("Client sent the message: " + mssge);
dos.flush();
dos.close();
conn.close();
System.out.println("Client program closed");
      }
      catch (Exception exp) {
System.out.println(exp);
      }
   }
}
```

**Server Side Program**

The following is the program for the **Server.java** file.

**Code:**

```
// Java Server Side Program
import java.io.*;
import java.net.*;
public class Server {
   public static void main(String[] args){
      try {
 System.out.println("Starting the server");
int PORT = 9000;
ServerSocket sock = new ServerSocket(PORT);
```

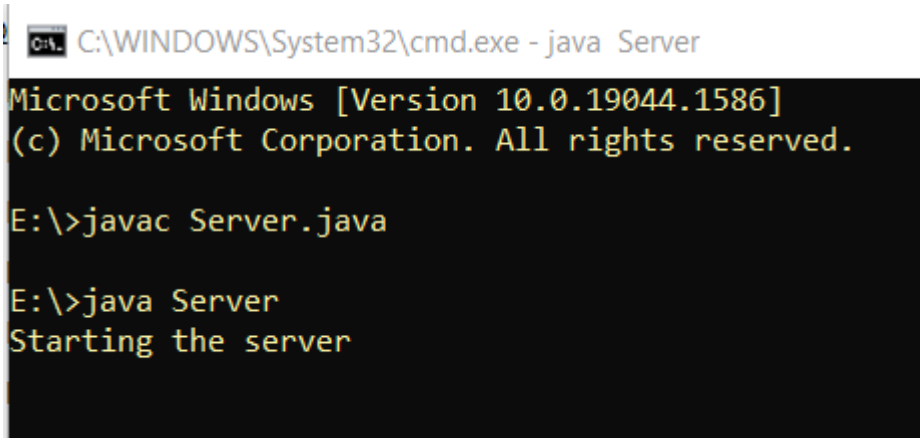
```
Socket conn = sock.accept();
System.out.println("Client Server Connection established");
 DataInputStream dis = new DataInputStream(conn.getInputStream());
System.out.println("Waiting for the client's message");
String msg = (String)dis.readUTF();
System.out.println("Message from client : " + msg);
conn.close();
sock.close();
 System.out.println("Closing the server");
      }
      catch (Exception exp) {
         System.out.println(exp) }
  } }
```

**Steps to run the program and final output**

You need to have **java** installed in your system to run this program. The first step is to run the server. Use the following commands to run the server.

**> javac Server.java**

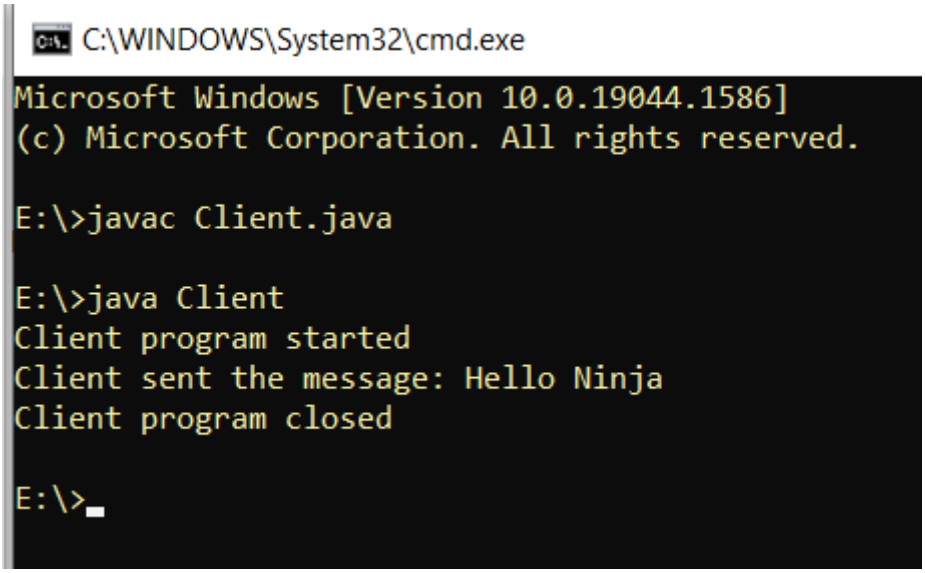
**> java Server**



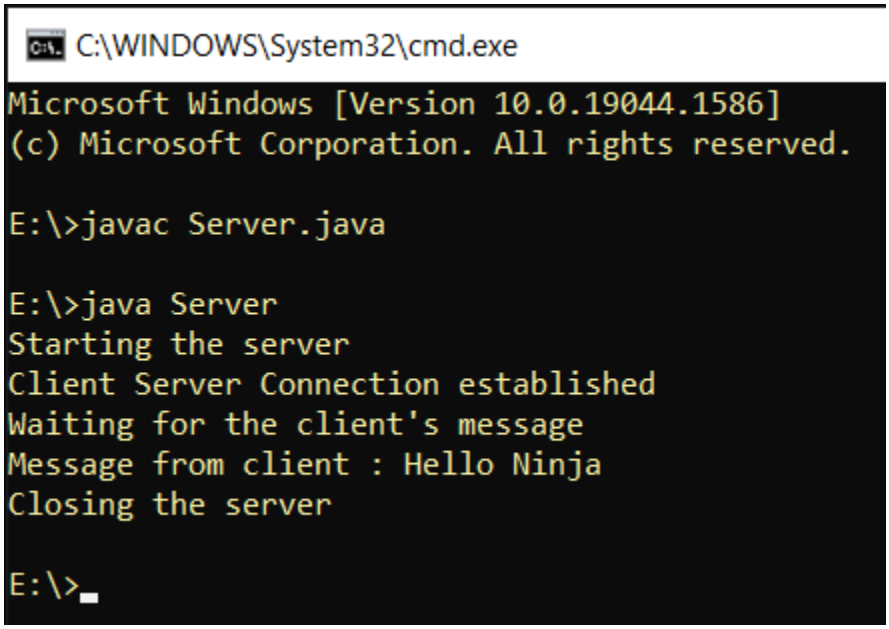

After running the server, open another terminal and run the client using the following commands.

**> javac Client.java**

**> java Client**

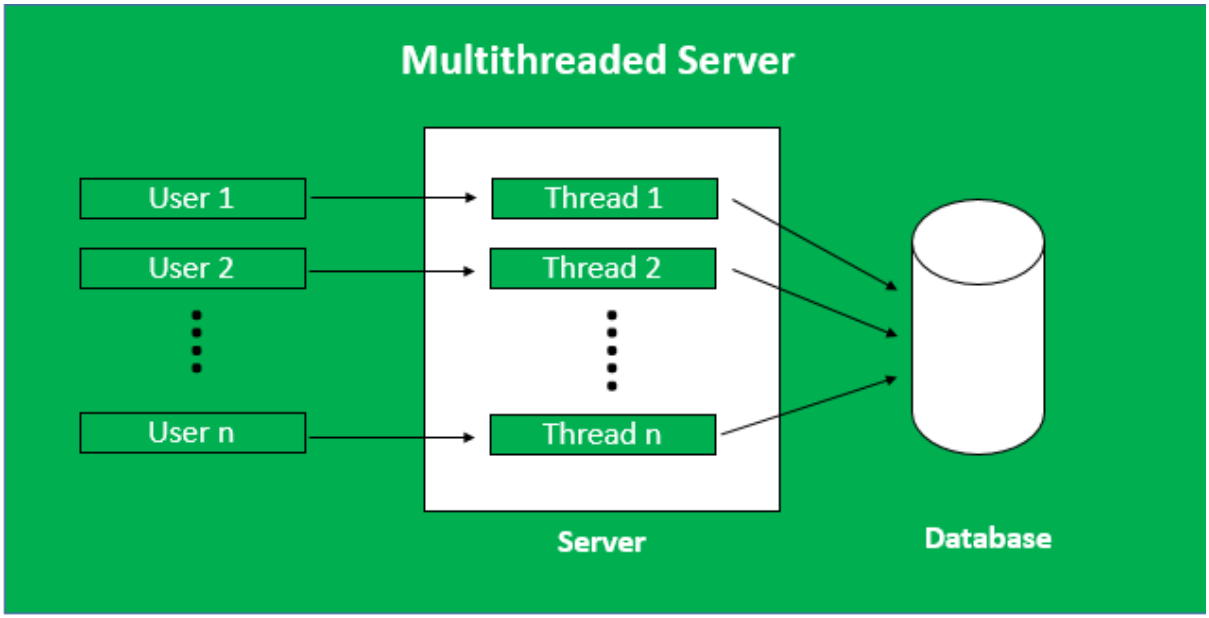



After running the client, you will see the following output on the server terminal.




**Multi-threaded servers:**

A server having more than one thread is known as Multithreaded Server. When a client sends the request, a thread is generated through which a user can communicate with the server. We need to generate multiple threads to accept multiple requests from multiple clients at the same time.



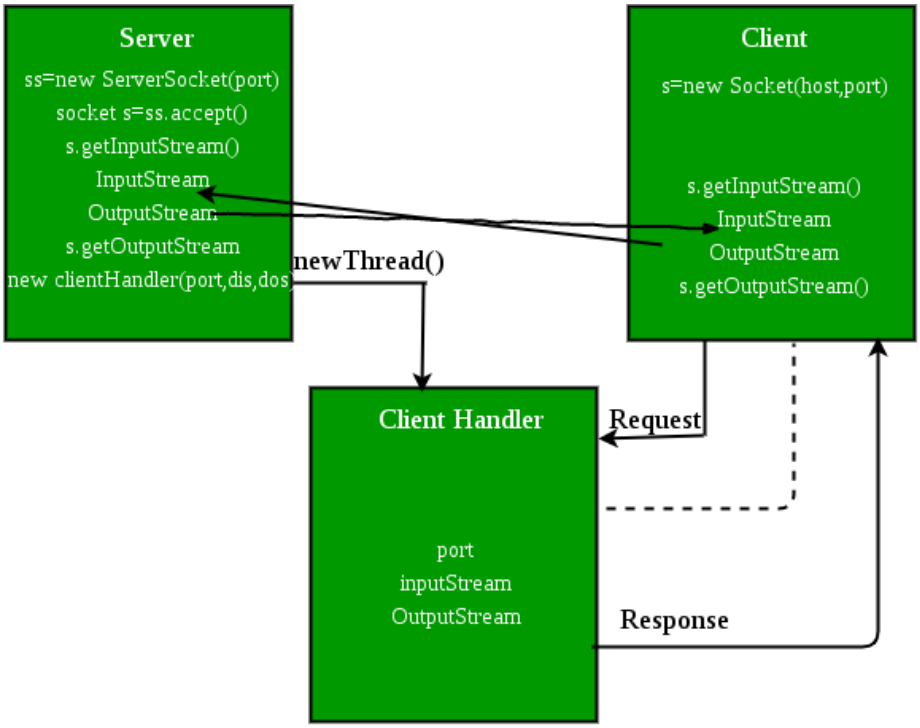

## Advantages of Multithreaded Server:

- **Quick and Efficient:** Multithreaded server could respond efficiently and quickly to the increasing client queries quickly.
- **Waiting time for users decreases:** In a single-threaded server, other users had to wait until the running process gets completed but in multithreaded servers, all users can get a response at a single time so no user has to wait for other processes to finish.
- **Threads are independent of each other:** There is no relation between any two threads. When a client is connected a new thread is generated every time.
- **The issue in one thread does not affect other threads:** If any error occurs in any of the threads then no other thread is disturbed, all other processes keep running normally. In a single-threaded server, every other client had to wait if any problem occurs in the thread.

**Disadvantages of Multithreaded Server:**

- **Complicated Code:** It is difficult to write the code of the multithreaded server. These programs can not be created easily
- **Debugging is difficult:** Analyzing the main reason and origin of the error is difficult.

We create two java files, **Client.java** and **Server.java**. Client file contains only one class **Client** (for creating a client). Server file has two classes, **Server**(creates a server) and **ClientHandler**(handles clients using multithreading).




**Client-Side Program:** A client can communicate with a server using this code. This involves

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

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

Threads can be created by using two mechanisms :

1. Extending the Thread class
2. Implementing the Runnable Interface

**Thread creation by extending the Thread class**
We create a class that extends the **java.lang.Thread** class. This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

**Java**

```
// Java code for thread creation by extending
// the Thread class
class MultithreadingDemo extends Thread {
    public void run()
    {
        try {
            // Displaying the thread that is running
            System.out.println(
                "Thread " + Thread.currentThread().getId()
                + " is running");
        }
        catch (Exception e) {
            // Throwing an exception
            System.out.println("Exception is caught");
        }
    }
}

// Main Class
public class Multithread {
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i = 0; i < n; i++) {
            MultithreadingDemo object
                = new MultithreadingDemo();
            object.start();
        }
    }
```

```
}
```

**Output**

Thread 15 is running

Thread 14 is running

Thread 16 is running

Thread 12 is running

Thread 11 is running

Thread 13 is running

Thread 18 is running

Thread 17 is running.

## *6)Handling multiple client connections*

To handle multiple clients, the server must be written so that it starts a separate thread for each client connection. This allows the main server thread to continue waiting for new client connections while the separate threads processes individual clients.

Here's an example server that can handle mutliple clients:

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

public class Server
{
   public static void main(String[] args) throws IOException
   {
      ServerSocket serverSocket = null;
      boolean listening = true;

      try
      {
         serverSocket = new ServerSocket(4444);
      }
      catch (IOException e)
      {
         System.err.println("Could not listen on port: 4444.");
         System.exit(-1);
      }

      while (listening)
         new Thread (new ClientHandler (serverSocket.accept()))).start();
```

```
        serverSocket.close();
    }
}
```

In this server code, the class ClientHandler is simply a class that implements Runnable and does something with the socket connection in the run method. As written, this server could be used for any purpose, since the details of what the server actually does are hidden inside the ClientHandler class.

## 7)Introduction to RMI-Creating RMI servers and clients

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:

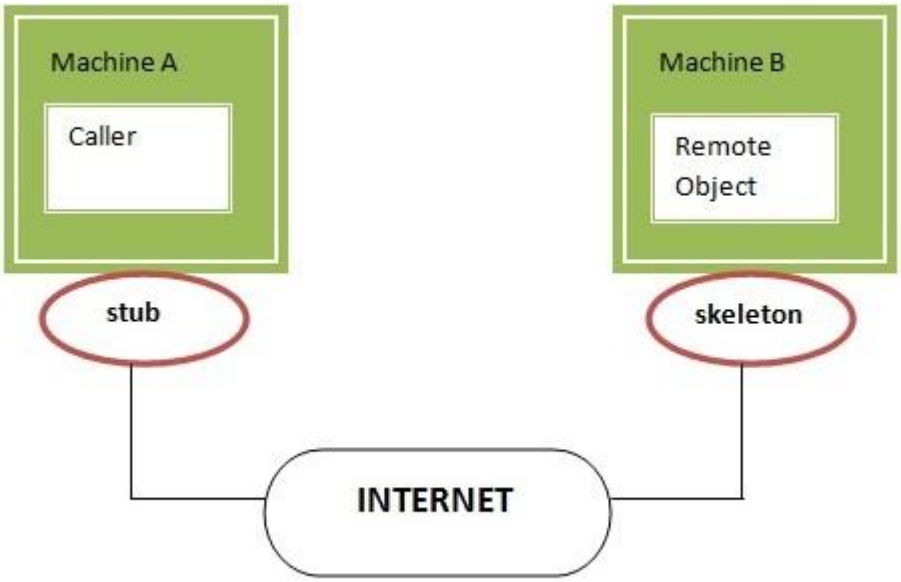
1. It initiates a connection with remote Virtual Machine (JVM),
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads (unmarshals) the return value or exception, and
5. It finally, returns the value to the caller.

## 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.



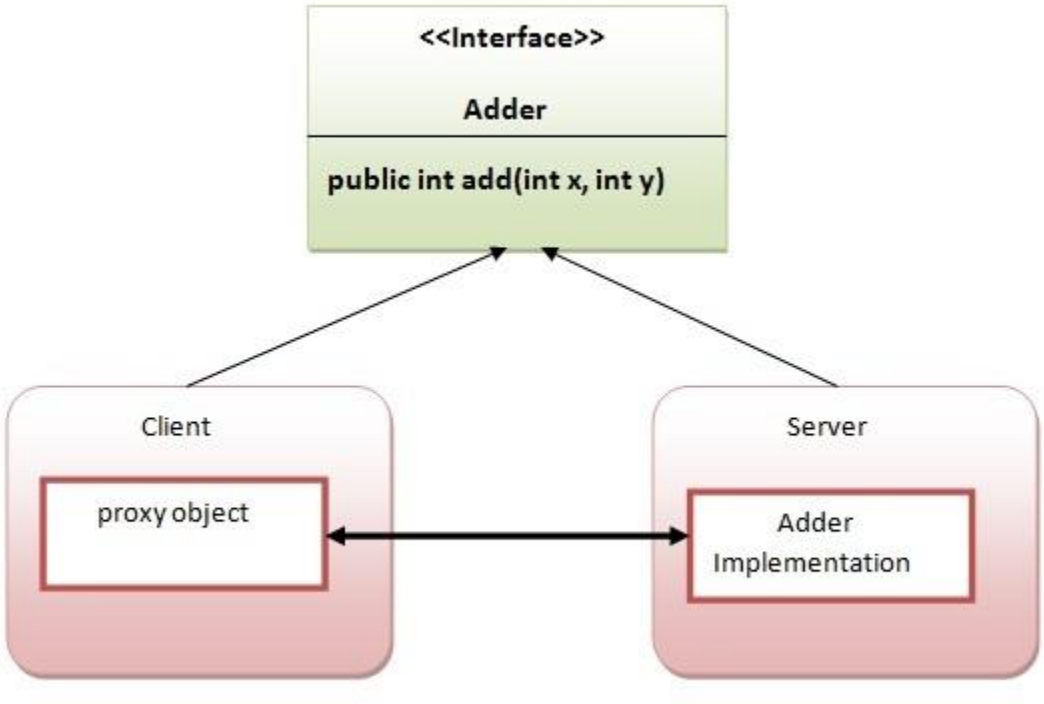

**Java RMI Example**

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){}
}
}
```

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
```

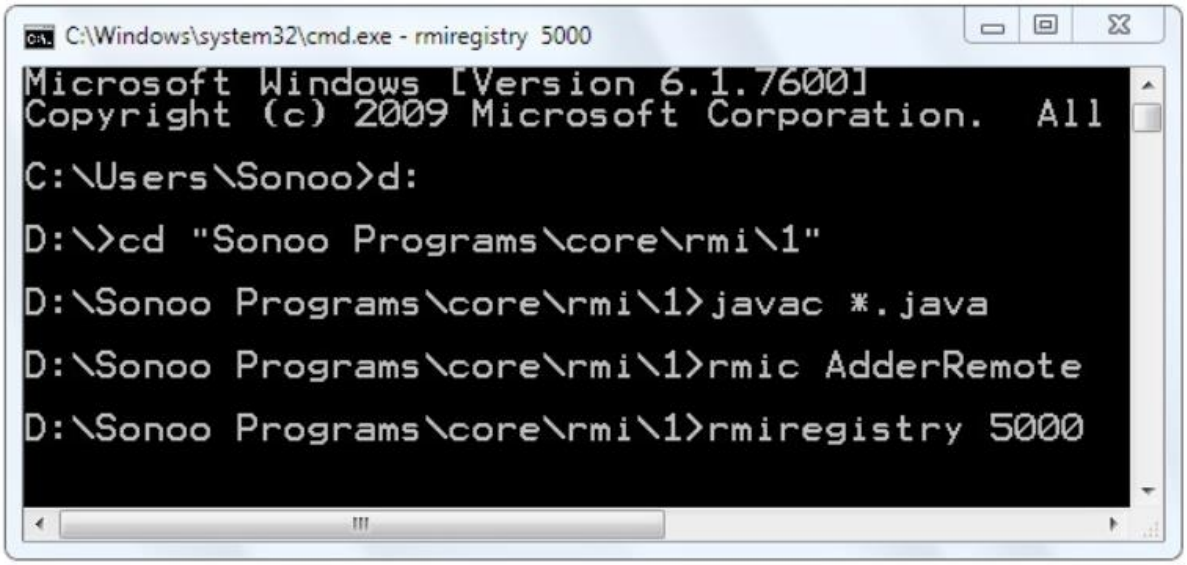
4)start the server in another command prompt

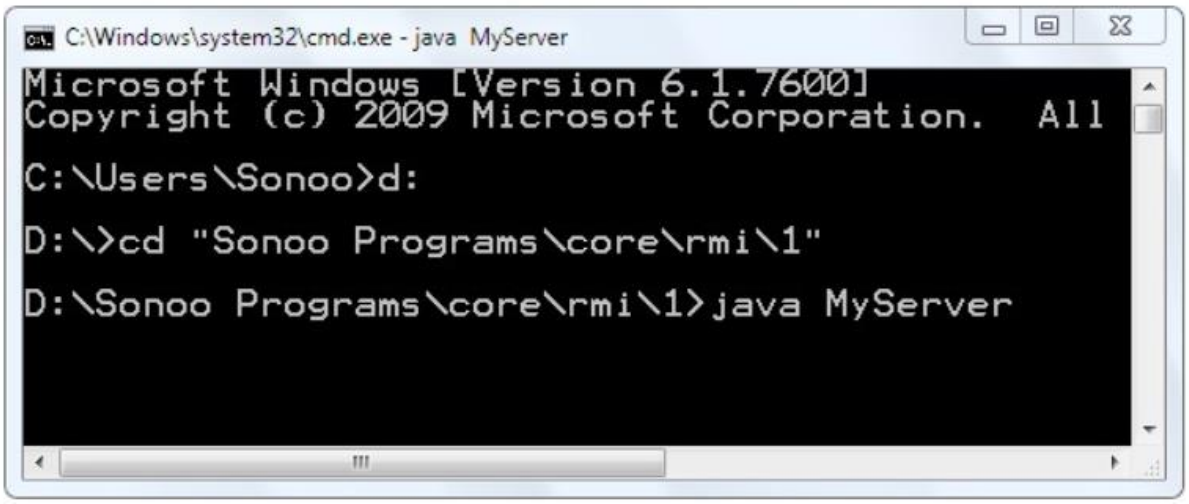
```
java MyServer
```

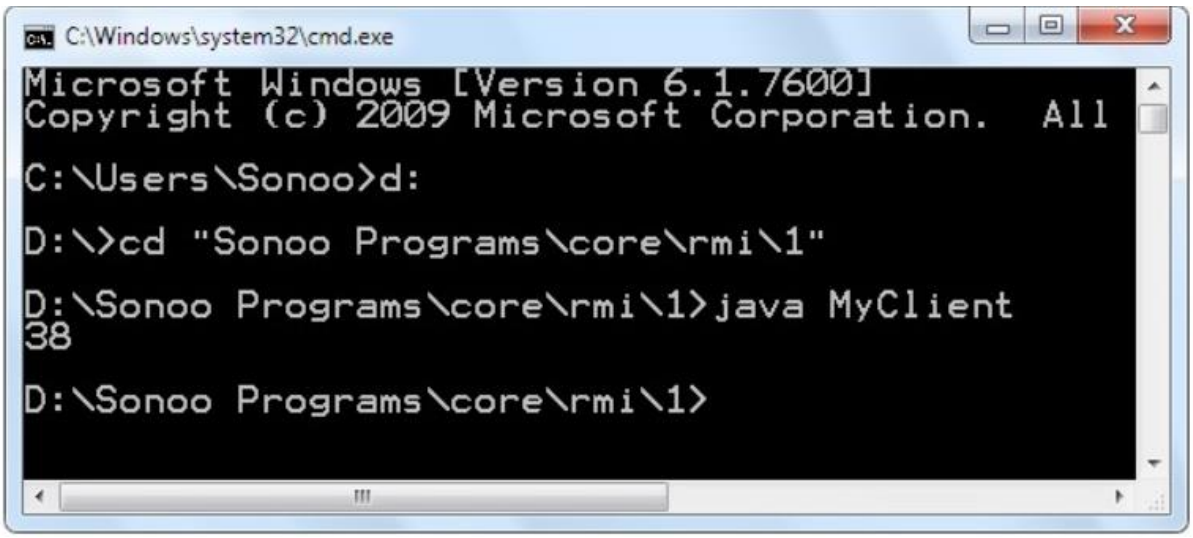
5)start the client application in another command prompt

```
java MyClient
```

**Output of this RMI example**






**8)RMI registry**

A Java RMI registry is a simplified name service that allows clients to get a reference (a stub) to a remote object. In general, a registry is used (if at all) only to locate the first remote object a client needs to use. Then, typically, that first object would in turn provide application-specific support for finding other objects. For example, the reference can be obtained as a parameter to, or a return value from, another remote method call. For a discussion on how this works, please take a look at *Applying the Factory Pattern to Java RMI*.

Once a remote object is registered on the server, callers can look up the object by name, obtain a remote object reference, and then invoke remote methods on the object.

The following code in the server obtains a stub for a registry on the local host and default registry port and then uses the registry stub to bind the name "Hello" to the remote object's stub in that registry:

```
Registry registry = LocateRegistry.getRegistry();
registry.bind("Hello", stub);
```

The static method LocateRegistry.getRegistry that takes no arguments returns a stub that implements the remote interface java.rmi.registry.Registry and sends invocations to the registry on server's local host on the default registry port of 1099. The bind method is then invoked on the registry stub in order to bind the remote object's stub to the name "Hello" in the registry.

**Implement the client**

The client program obtains a stub for the registry on the server's host, looks up the remote object's stub by name in the registry, and then invokes the sayHello method on the remote object using the stub.

Here is the source code for the client:

```
package example.hello;

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

public class Client {

    private Client() {}

    public static void main(String[] args) {

        String host = (args.length < 1) ? null : args[0];
        try {
            Registry registry = LocateRegistry.getRegistry(host);
            Hello stub = (Hello) registry.lookup("Hello");
            String response = stub.sayHello();
            System.out.println("response: " + response);
        } catch (Exception e) {
            System.err.println("Client exception: " + e.toString());
```

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

This client first obtains the stub for the registry by invoking the static LocateRegistry.getRegistry method with the hostname specified on the command line. If no hostname is specified, then null is used as the hostname indicating that the local host address should be used.

Next, the client invokes the remote method lookup on the registry stub to obtain the stub for the remote object from the server's registry.

Finally, the client invokes the sayHello method on the remote object's stub, which causes the following actions to happen:

- The client-side runtime opens a connection to the server using the host and port information in the remote object's stub and then serializes the call data.
- The server-side runtime accepts the incoming call, dispatches the call to the remote object, and serializes the result (the reply string "Hello, world!") to the client.
- The client-side runtime receives, deserializes, and returns the result to the caller.

The response message returned from the remote invocation on the remote object is then printed to System.out.

**Compile the source files**

The source files for this example can be compiled as follows:

javac -d ***destDir*** Hello.java Server.java Client.java

where ***destDir*** is the destination directory to put the class files in.

**Note:** For details on how to deploy your application along with pregenerated stub classes, see the codebase tutorial.

**Start the Java RMI registry, server, and client**

To run this example, you will need to do the following:

- Start the Java RMI registry
- Start the server
- Run the client

**Start the Java RMI registry**

To start the registry, run the rmiregistry command on the server's host. This command produces no output (when successful) and is typically run in the background. For more information, see the tools documentation for rmiregistry [UNIX, Windows].

For example, on the Solaris Operating System:

rmiregistry &

Or, on Windows platforms:

start rmiregistry

By default, the registry runs on TCP port 1099. To start a registry on a different port, specify the port number from the command line. For example, to start the registry on port 2001 on a Windows platform:

start rmiregistry 2001

If the registry will be running on a port other than 1099, you'll need to specify the port number in the calls to LocateRegistry.getRegistry in the Server and Client classes. For example, if the registry is running on port 2001 in this example, the call to getRegistry in the server would be:

Registry registry = LocateRegistry.getRegistry(2001);

**Start the server**

To start the server, run the Server class using the java command as follows:

On the Solaris Operating System:

java -classpath ***classDir*** -Djava.rmi.server.codebase=file:***classDir***/ example.hello.Server &

On Windows platforms:

start java -classpath ***classDir*** -Djava.rmi.server.codebase=file:***classDir***/ example.hello.Server

where ***classDir*** is the root directory of the class file tree. Setting the java.rmi.server.codebase system property ensures that the registry can load the remote interface definition (note that the trailing slash is important);

The output from the server should look like this:

Server ready

The server remains running until the process is terminated by the user (typically by killing the process).

**Run the client**

Once the server is ready, the client can be run as follows:

java -classpath ***classDir*** example.hello.Client

where ***classDir*** is the root directory of the class file tree.

The output from the client is the following message:

response: Hello, world!

## *9*)RMI and object serialization

**Serialization in java:**

In java serialization is way used to convert an object into a byte stream which can be transported to any other running JVM through a network or can be persisted into disk and that object can be rebuilt again. Java provides serialization API for this.

**Java Serialisation with RMI**

**Sample code**

```
public class FSFile implements Serializable
{
public static final int READ    = 0;
public static final int WRITE   = 1;
private int     flag;
private String  filename;
private transient BufferedWriter  writer;
private transient BufferedReader  reader;
...
private void writeObject(ObjectOutputStream stream) throws IOException
{
   stream.defaultWriteObject();
   stream.writeObject(writer);
   stream.writeObject(reader);
}
private void readObject(ObjectInputStream stream) throws IOException,
ClassNotFoundException
{
   stream.defaultReadObject();
   writer      = (BufferedWriter) stream.readObject();
   reader      = (BufferedReader) stream.readObject();
```

}

}

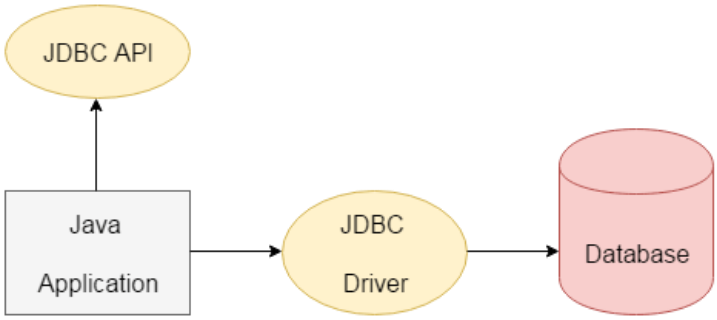
## ***10) Overview of JDBC***

JDBC stands for Java Database Connectivity. JDBC is a Java API or tool used in Java applications to interact with the database. It is a specification from Sun Microsystems that provides APIs for Java applications to communicate with different databases. Interfaces and Classes for JDBC API comes under java.sql package

There are four types of JDBC drivers:

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

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




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

- Driver interface
- Connection interface
- Statement interface
- PreparedStatement interface
- CallableStatement interface
- ResultSet interface
- ResultSetMetaData interface
- DatabaseMetaData interface
- RowSet interface
- A list of popular *classes* of JDBC API are given below:
- DriverManager class
- Blob class
- Clob class

- Types class

**Why Should We Use JDBC?**

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

We can use JDBC API to handle database using Java program and can perform the following activities:
1)Connect to the database
2)Execute queries and update statements to the database
3)Retrieve the result received from the database.

## *11)JDBC Drivers*

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)

## JDBC-ODBC bridge driver

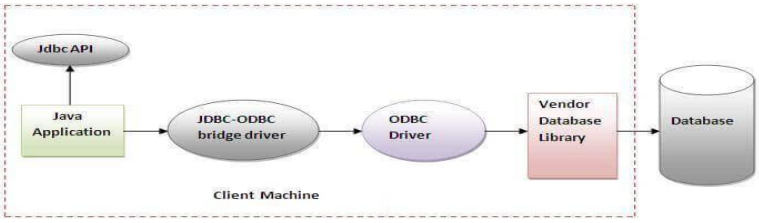



Figure- JDBC-ODBC Bridge Driver

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.

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.

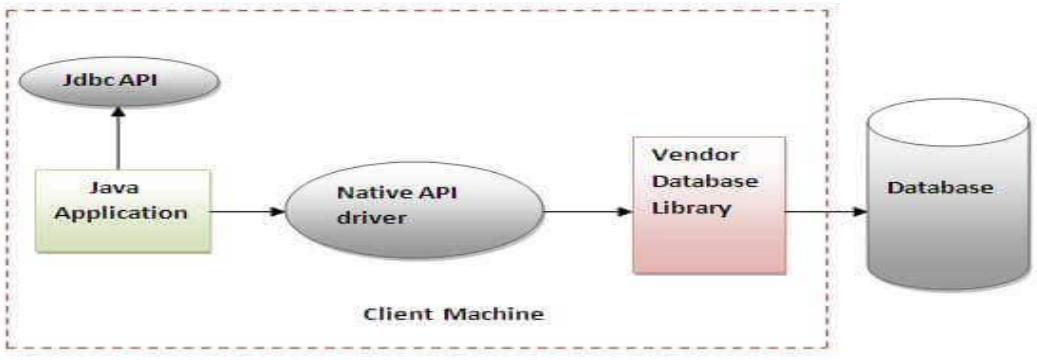



Figure- Native API Driver

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.

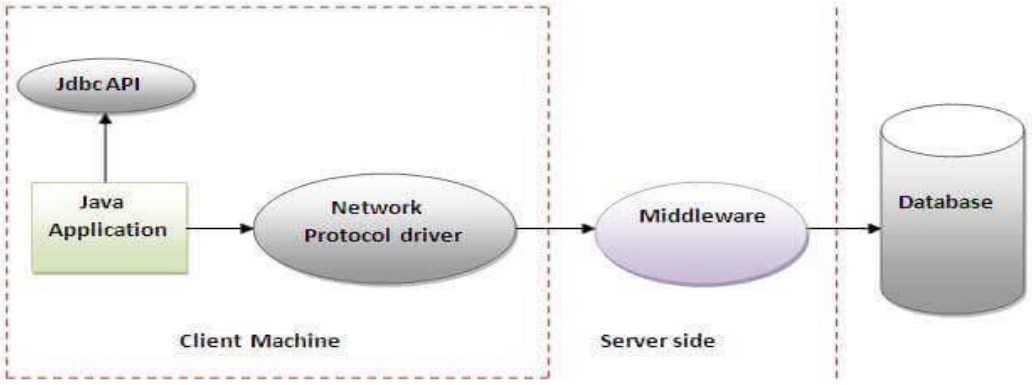



Figure- Network Protocol Driver

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 kno fully written in Java language.

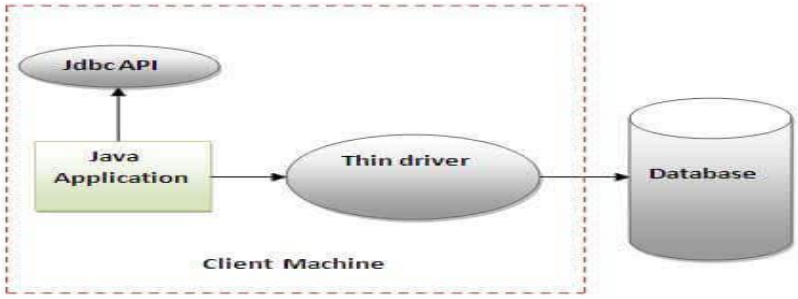



Figure- Thin Driver

Advantage:

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

Disadvantage:

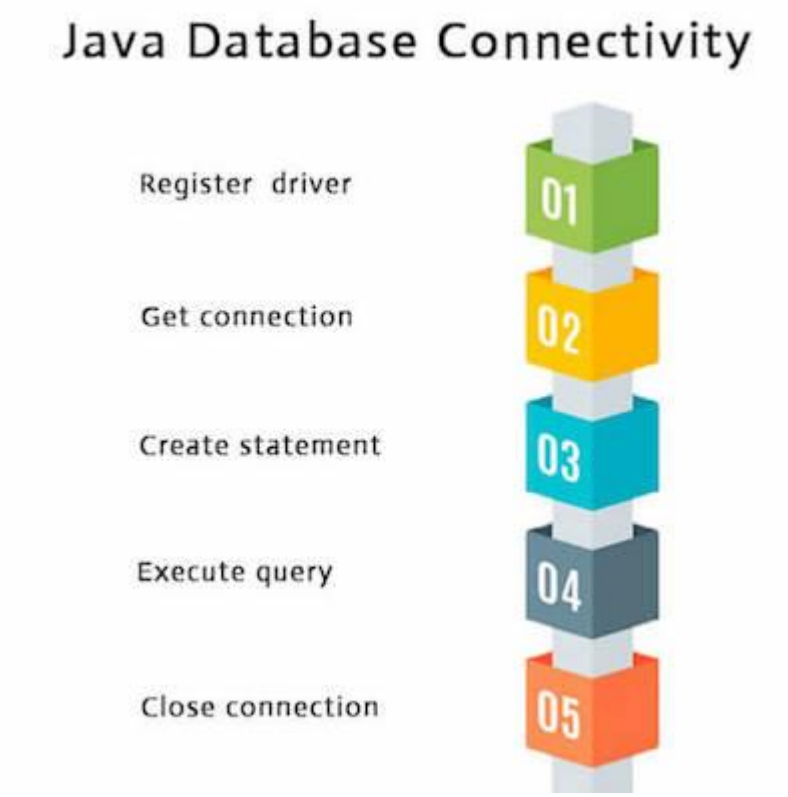
- Drivers depend on the Database.

## *12)Connecting to databases*

***Java Database Connectivity with 5 Steps***

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 dynam
class.

Syntax of forName() method

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

Note: Since JDBC 4.0, explicitly registering the driver is optional. We just need to put vender'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 esteblish database connection.

```
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) **public static** Connection getConnection(String url)**throws** SQLException

2) **public static** Connection getConnection(String url,String name,String password)

**throws** SQLException

Example to establish connection with the Oracle database

```
Connection con=DriverManager.getConnection(
"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

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

Example to create the statement object

```
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

```
public ResultSet executeQuery(String sql)throws SQLException
```

Example to execute query

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

**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();

Note: Since Java 7, JDBC has ability to use try-with-resources statement to automatically close resources of type Connection, ResultSet, and Statement.

## 13)Executing SQL queries.

**How to Execute a SQL Query Using JDBC?**

Java programming provides a lot of packages for solving problems in our case we need to execute SQL queries by using JDBC. We can execute a SQL query in two different approaches by using **PreparedStatement** and **Statement.** These two interfaces are available in **java.sql** package. When comparing both of them the PreparedStatement approach is secure.

**Approaches to Execute a SQL Query using JDBC**

We have two different approaches to executing a SQL query using JDBC. Below is the list and we will explain them with examples to understand the concept correctly.

- Using Statement
- Using PreparedStatement

**Statement in JDBC**

The Statement is an interface that is available in **java.sql** package with JDBC.

- This interface is part of JDBC API and can execute simple SQL queries without parameters.
- We can create a Statement by using **createStatement().**
- This method is available in the Connection class.

**Example:**

In this example, we will write an SQL query to fetch all data from the table in the database. We have already some data in the table. Now we will write an SQL query for fetching that data using Statement.

For this, we have used a database named **books** and the table name is a **book.**

**Example:**

```
package geeksforgeeks;


import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.Statement;


public class RetrieveDataExample {


	public static void main(String[] args) {

		try {

			// load the MySQL JDBC driver

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


			// establish connection with the database

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

```
                if (con != null) {
                        // SQL query to retrieve data from the 'book' table
                        String selectQuery = "SELECT * FROM book";
                        Statement statement = con.createStatement();

                        // execute the query and get the result set
                        ResultSet resultSet = statement.executeQuery(selectQuery);
                        System.out.println("The Available Data\n");

                        // iterate through the result set and print the data
                        while (resultSet.next()) {
                                int id = resultSet.getInt("id");
                                String author_name = resultSet.getString("author");
                                String book_name = resultSet.getString("name");
                                String book_price = resultSet.getString("price");

                                // print the retrieved data
                                System.out.println("ID: " + id + ", Author_Name: " +
author_name + ", Book_Name: " + book_name
                                                + ", Book_Price " + book_price);
                        }
                } else {
                        System.out.println("Not Connected...");
                }
        } catch (Exception e) {
                // handle any exceptions that occur
                System.out.println("Exception is " + e.getMessage());
        }
    }
}
```

## Processing SQL Statements with JDBC

In general, to process any SQL statement with JDBC, you follow these steps:

1. Establishing a connection.
2. Create a statement.
3. Execute the query.
4. Process the ResultSet object.
5. Close the connection.

This page uses the following method, CoffeesTable.viewTable, from the tutorial sample to demonstrate these steps. This method outputs the contents of the table COFFEES. This method will be discussed in more detail later in this tutorial:

```
public static void viewTable(Connection con) throws SQLException {
  String query = "select COF_NAME, SUP_ID, PRICE, SALES, TOTAL from COFFEES";
  try (Statement stmt = con.createStatement()) {
    ResultSet rs = stmt.executeQuery(query);
    while (rs.next()) {
      String coffeeName = rs.getString("COF_NAME");
      int supplierID = rs.getInt("SUP_ID");
      float price = rs.getFloat("PRICE");
      int sales = rs.getInt("SALES");
      int total = rs.getInt("TOTAL");
      System.out.println(coffeeName + ", " + supplierID + ", " + price +
                         ", " + sales + ", " + total);
    }
  } catch (SQLException e) {
    JDBCTutorialUtilities.printSQLException(e);
  }
}
```

### Establishing Connections

First, establish a connection with the data source you want to use. A data source can be a DBMS, a legacy file system, or some other source of data with a corresponding JDBC driver.

This connection is represented by a Connection object. See Establishing a Connection for more information.

**Creating Statements**

A Statement is an interface that represents a SQL statement. You execute Statement objects, and they generate ResultSet objects, which is a table of data representing a database result set. You need a Connection object to create a Statement object.

For example, CoffeesTable.viewTable creates a Statement object with the following code:

stmt = con.createStatement();

There are three different kinds of statements:

- **Statement:** Used to implement simple SQL statements with no parameters.
- **Prepared Statement:** (Extends Statement.) Used for precompiling SQL statements that might contain input parameters. See Using Prepared Statements for more information.
- **Callable Statement: (Extends PreparedStatement.)** Used to execute stored procedures that may contain both input and output parameters. See Stored Procedures for more information.

**Executing Queries**

To execute a query, call an execute method from Statement such as the following:

- **execute:** Returns true if the first object that the query returns is a ResultSet object. Use this method if the query could return one or more ResultSet objects. Retrieve the ResultSet objects returned from the query by repeatedly calling Statement.getResultSet.
- **executeQuery:** Returns one ResultSet object.
- **executeUpdate:** Returns an integer representing the number of rows affected by the SQL statement. Use this method if you are using INSERT, DELETE, or UPDATE SQL statements.

For example, CoffeesTable.viewTable executed a Statement object with the following code:

ResultSet rs = stmt.executeQuery(query);

See Retrieving and Modifying Values from Result Sets for more information.

**Processing ResultSet Objects**

You access the data in a ResultSet object through a cursor. Note that this cursor is not a database cursor. This cursor is a pointer that points to one row of data in the ResultSet object. Initially, the cursor is positioned before the first row. You call various methods defined in the ResultSet object to move the cursor.

For example, CoffeesTable.viewTable repeatedly calls the method ResultSet.next to move the cursor forward by one row. Every time it calls next, the method outputs the data in the row where the cursor is currently positioned:

```
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
  String coffeeName = rs.getString("COF_NAME");
  int supplierID = rs.getInt("SUP_ID");
  float price = rs.getFloat("PRICE");
  int sales = rs.getInt("SALES");
  int total = rs.getInt("TOTAL");
  System.out.println(coffeeName + ", " + supplierID + ", " + price +
                     ", " + sales + ", " + total);
}
// ...
```

See Retrieving and Modifying Values from Result Sets for more information.

**Closing Connections**

When you are finished using a Connection, Statement, or ResultSet object, call its close method to immediately release the resources it's using.

Alternatively, use a try-with-resources statement to automatically close Connection, Statement, and ResultSet objects, regardless of whether an SQLException has been thrown. (JDBC throws an SQLException when it encounters an error during an interaction with a data source. An automatic resource statement consists of a try statement and one or more declared resources. For example, the CoffeesTable.viewTable method automatically closes its Statement object, as follows:

```
public static void viewTable(Connection con) throws SQLException {
  String query = "select COF_NAME, SUP_ID, PRICE, SALES, TOTAL from COFFEES";
  try (Statement stmt = con.createStatement()) {
    ResultSet rs = stmt.executeQuery(query);
    while (rs.next()) {
      String coffeeName = rs.getString("COF_NAME");
      int supplierID = rs.getInt("SUP_ID");
      float price = rs.getFloat("PRICE");
      int sales = rs.getInt("SALES");
```

```
        int total = rs.getInt("TOTAL");
        System.out.println(coffeeName + ", " + supplierID + ", " + price +
                           ", " + sales + ", " + total);
      }
    } catch (SQLException e) {
      JDBCTutorialUtilities.printSQLException(e);
    }
  }
```

The following statement is a try-with-resources statement, which declares one resource, stmt, that will be automatically closed when the try block terminates:

```
    try (Statement stmt = con.createStatement()) {
      // ...
    }
```