IO Stream

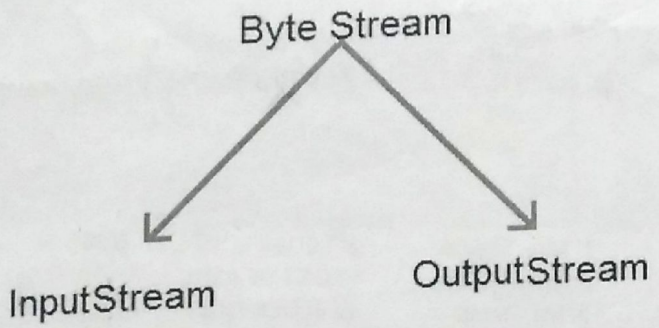
Java performs I/O through **Streams**. A Stream is linked to a physical layer by java I/O system to make input and output operation in java. In general, a stream means continuous flow of data. Streams are clean way to deal with input/output without having every part of your code understand the physical.

Java encapsulates Stream under **java.io** package. Java defines two types of streams. They are,

1. **Byte Stream :** It provides a convenient means for handling input and output of byte.
2. **Character Stream :** It provides a convenient means for handling input and output of characters. Character stream uses Unicode and therefore can be internationalized.

Byte Stream Classes

Byte stream is defined by using two abstract class at the top of hierarchy, they are InputStream and OutputStream.



These two abstract classes have several concrete classes that handle various devices such as disk files, network connection etc.

Some important Byte stream classes.

Stream class	Description
BufferedInputStream	Used for Buffered Input Stream.

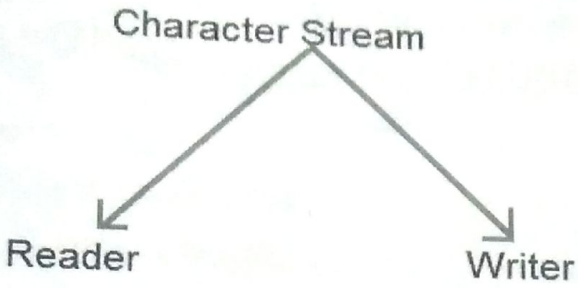
BufferedOutputStream	Used for Buffered Output Stream.
DataInputStream	Contains method for reading java standard datatype
DataOutputStream	An output stream that contain method for writing java standard data type
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that write to a file.
InputStream	Abstract class that describe stream input.
OutputStream	Abstract class that describe stream output.
PrintStream	Output Stream that contain `print()` and `println()` method

These classes define several key methods. Two most important are

1. `read()` : reads byte of data.
2. `write()` : Writes byte of data.

Character Stream Classes

Character stream is also defined by using two abstract class at the top of hierarchy, they are Reader and Writer.



These two abstract classes have several concrete classes that handle unicode character.

Some important Charcter stream classes.

Stream class	Description
BufferedReader	Handles buffered input stream.
BufferedWriter	Handles buffered output stream.
FileReader	Input stream that reads from file.
FileWriter	Output stream that writes to file.
InputStreamReader	Input stream that translate byte to character
OutputStreamReader	Output stream that translate character to byte.
PrintWriter	Output Stream that contain `print()` and `println()` method.
Reader	Abstract class that define character stream input

Writer	Abstract class that define character stream output

Reading Console Input

We use the object of BufferedReader class to take inputs from the keyboard.

Object of BufferedReader class

```
BufferedReader br = new BufferedReader(new
                        InputStreamReader (System.in) );
```

InputStreamReader is subclass of Reader class. It converts bytes to character.

Console inputs are read from this.

Reading Characters

`read()` method is used with BufferedReader object to read characters. As this function returns integer type value has we need to use typecasting to convert it into **char** type.

```
int read() throws IOException
```

Below is a simple example explaining character input.

```
class CharRead
{
 public static void main( String args[])
 {
  BufferedReader br = new Bufferedreader(new InputstreamReader(System.in));
  char c = (char)br.read();        //Reading character
 }
}
```

Reading Strings

To read string we have to use readLine() function with BufferedReader class's object.

```
String readLine() throws IOException
```

Program to take String input from Keyboard in Java

```
import java.io.*;
class MyInput
{
 public static void main(String[] args)
 {
  String text;
  InputStreamReader isr = new InputStreamReader(System.in);
  BufferedReader br = new BufferedReader(isr);
  text = br.readLine();            //Reading String
  System.out.println(text);
 }
}
```

Program to read from a file using BufferedReader class

```
import java. Io *;
class ReadTest
{
 public static void main(String[] args)
 {
  try
  {
   File fl = new File("d:/myfile.txt");
   BufferedReader br = new BufferedReader(new FileReader(fl)) ;
   String str;
```

```
  while ((str=br.readLine())!=null)
  {
   System.out.println(str);
  }
  br.close();
  fl.close();
 }
 catch (IOException  e)
 { e.printStackTrace(); }
}
}
```

Program to write to a File using FileWriter class

```
import java. Io *;
class WriteTest
{
 public static void main(String[] args)
 {
  try
  {
   File fl = new File("d:/myfile.txt");
   String str="Write this string to my file";
   FileWriter fw = new FileWriter(fl) ;
   fw.write(str);
   fw.close();
   fl.close();
  }
  catch (IOException  e)
  { e.printStackTrace(); }
 }
```

About File Handling in Java

Create BufferedReader from InputStreamReader and read line by line

```
import java.io.BufferedReader;
import java.io.InputStreamReader;

class ReadConsole {

  public static void main(String args[]) throws Exception {
    InputStreamReader isr = new InputStreamReader(System.in);

    BufferedReader br = new BufferedReader(isr);

    String s;
    while ((s = br.readLine()) != null) {
      System.out.println(s.length());
    }
    isr.close();
  }
}
```

One this page you can find a simple guide to reading and writing files in the Java programming language. The code examples here give you everything you need to read and write files right away, and if you're in a hurry, you can use them without needing to understanding in detail how they work.

File handling in Java is frankly a bit of a pig's ear, but it's not too

complicated once you understand a few basic ideas. The key things to remember are as follows.

You can read files using these classes:

- **FileReader** for text files in your system's default encoding (for example, files containing Western European characters on a Western European computer).
- **FileInputStream** for binary files and text files that contain 'weird' characters.

FileReader (for text files) should usually be wrapped in a **BufferedFileReader**. This saves up data so you can deal with it a line at a time or whatever instead of character by character (which usually isn't much use).

If you want to write files, basically all the same stuff applies, except you'll deal with classes named **FileWriter**with **BufferedFileWriter** for text files, or **FileOutputStream** for binary files.

Reading Ordinary Text Files in Java

If you want to read an ordinary text file in your system's default encoding (usually the case most of the time for most people), use **FileReader** and wrap it in a **BufferedReader**.

In the following program, we read a file called "temp.txt" and output the file line by line on the console.

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

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

        // The name of the file to open.
        String fileName = "temp.txt";

        // This will reference one line at a time
        String line = null;

        try {
            // FileReader reads text files in the default encoding.
            FileReader fileReader =
                new FileReader(fileName);

            // Always wrap FileReader in BufferedReader.
            BufferedReader bufferedReader =
                new BufferedReader(fileReader);

            while((line = bufferedReader.readLine()) != null) {
                System.out.println(line);
            }

            // Always close files.
            bufferedReader.close();
        }
        catch(FileNotFoundException ex) {
            System.out.println(
                "Unable to open file '" +
                fileName + "'");
        }
        catch(IOException ex) {
            System.out.println(
                "Error reading file '"
                + fileName + "'");
```

```
            // Or we could just do this:
            // ex.printStackTrace();
        }
    }
}
```

If "temp.txt" contains this:

```
I returned from the City about three o'clock on that
May afternoon pretty well disgusted with life.
I had been three months in the Old Country, and was
fed up with it.
If anyone had told me a year ago that I would have
been feeling like that I should have laughed at him;
but there was the fact.
The weather made me liverish,
the talk of the ordinary Englishman made me sick,
I couldn't get enough exercise, and the amusements
of London seemed as flat as soda-water that
has been standing in the sun.
'Richard Hannay,' I kept telling myself, 'you
have got into the wrong ditch, my friend, and
you had better climb out.'
```

The program outputs this:

```
I returned from the City about three o'clock on that
```

```
May afternoon pretty well disgusted with life.
I had been three months in the Old Country, and was
fed up with it.
If anyone had told me a year ago that I would have
been feeling like that I should have laughed at him;
but there was the fact.
The weather made me liverish,
the talk of the ordinary Englishman made me sick,
I couldn't get enough exercise, and the amusements
of London seemed as flat as soda-water that
has been standing in the sun.
'Richard Hannay,' I kept telling myself, 'you
have got into the wrong ditch, my friend, and
you had better climb out.'
```

Reading Binary Files in Java

If you want to read a binary file, or a text file containing 'weird' characters (ones that your system doesn't deal with by default), you need to use **FileInputStream** instead of FileReader. Instead of wrapping FileInputStream in a buffer, FileInputStream defines a method called **read()** that lets you fill a buffer with data, automatically reading just enough bytes to fill the buffer (or less if there aren't that many bytes left to read).

Here's a complete example.

```
import java.io.*;

public class Test {
    public static void main(String [] args) {

        // The name of the file to open.
        String fileName = "temp.txt";

        try {
            // Use this for reading the data.
            byte[] buffer = new byte[1000];

            FileInputStream inputStream =
                new FileInputStream(fileName);

            // read fills buffer with data and returns
            // the number of bytes read (which of course
            // may be less than the buffer size, but
            // it will never be more).
            int total = 0;
            int nRead = 0;
            while((nRead = inputStream.read(buffer)) != -1) {
                // Convert to String so we can display it.
                // Of course you wouldn't want to do this with
                // a 'real' binary file.
                System.out.println(new String(buffer));
                total += nRead;
            }

            // Always close files.
            inputStream.close();

            System.out.println("Read " + total + " bytes");
        }
```

```
        catch(FileNotFoundException ex) {
            System.out.println(
                "Unable to open file '" +
                fileName + "'");
        }
        catch(IOException ex) {
            System.out.println(
                "Error reading file '"
                + fileName + "'");
            // Or we could just do this:
            // ex.printStackTrace();
        }
    }
}
```

```
I returned from the City about three o'clock on that
May afternoon pretty well disgusted with life.
I had been three months in the Old Country, and was
fed up with it.
If anyone had told me a year ago that I would have
been feeling like that I should have laughed at him;
but there was the fact.
The weather made me liverish,
the talk of the ordinary Englishman made me sick,
I couldn't get enough exercise, and the amusements
of London seemed as flat as soda-water that
has been standing in the sun.
'Richard Hannay,' I kept telling myself, 'you
```

```
have got into the wrong ditch, my friend, and
you had better climb out.'

Read 649 bytes
```

Of course, if you had a 'real' binary file -- an image for instance -- you wouldn't want to convert it to a string and print it on the console as above.

Writing Text Files in Java

To write a text file in Java, use **FileWriter** instead of FileReader, and **BufferedOutputWriter** instead of BufferedOutputReader. Simple eh?

Here's an example program that creates a file called 'temp.txt' and writes some lines of text to it.

```
import java.io.*;

public class Test {
    public static void main(String [] args) {

        // The name of the file to open.
        String fileName = "temp.txt";

        try {
            // Assume default encoding.
            FileWriter fileWriter =
                new FileWriter(fileName);
```

```
            // Always wrap FileWriter in BufferedWriter.
            BufferedWriter bufferedWriter =
                new BufferedWriter(fileWriter);

            // Note that write() does not automatically
            // append a newline character.
            bufferedWriter.write("Hello there,");
            bufferedWriter.write(" here is some text.");
            bufferedWriter.newLine();
            bufferedWriter.write("We are writing");
            bufferedWriter.write(" the text to the file.");

            // Always close files.
            bufferedWriter.close();
        }
        catch(IOException ex) {
            System.out.println(
                "Error writing to file '"
                + fileName + "'");
            // Or we could just do this:
            // ex.printStackTrace();
        }
    }
}
```

The output file now looks like this (after running the program):

```
Hello there, here is some text.
We are writing the text to the file.
```

Writing Binary Files in Java

You can create and write to a binary file in Java using much the same techniques that we used to read binary files, except that we need **FileOutputStream** instead of **FileInputStream**.

In the following example we write out some text as binary data to the file. Usually of course, you'd probably want to write some proprietary file format or something.

```
import java.io.*;

public class Test {
    public static void main(String [] args) {

        // The name of the file to create.
        String fileName = "temp.txt";

        try {
            // Put some bytes in a buffer so we can
            // write them. Usually this would be
            // image data or something. Or it might
            // be unicode text.
            String bytes = "Hello theren";
            byte[] buffer = bytes.getBytes();

            FileOutputStream outputStream =
                new FileOutputStream(fileName);

            // write() writes as many bytes from the buffer
            // as the length of the buffer. You can also
            // use
```

```
            // write(buffer, offset, length)
            // if you want to write a specific number of
            // bytes, or only part of the buffer.
            outputStream.write(buffer);

            // Always close files.
            outputStream.close();

            System.out.println("Wrote " + buffer.length +
                " bytes");
        }
        catch(IOException ex) {
            System.out.println(
                "Error writing file '"
                + fileName + "'");
            // Or we could just do this:
            // ex.printStackTrace();
        }
    }
}
```

Lesson: All About Sockets

`URL`s and `URLConnection`s provide a relatively high-level mechanism for accessing resources on the Internet. Sometimes your programs require lower-level network communication, for example, when you want to write a client-server application.

In client-server applications, the server provides some service, such as processing database queries or sending out current stock prices. The client uses the service provided by the server, either displaying database query results to the user or making stock purchase recommendations to an investor. The communication that occurs between the client and the server must be reliable. That is, no data can be dropped and it must arrive on the client side in the same order in which the server sent it.

TCP provides a reliable, point-to-point communication channel that client-server applications on the Internet use to communicate with each other. To communicate over TCP, a client program and a server program establish a connection to one another. Each program binds a socket to its end of the connection. To communicate, the client and the server each reads from and writes to the socket bound to the connection.

What Is a Socket?

A socket is one end-point of a two-way communication link between two programs running on the network. Socket classes are used to represent the connection between a client program and a server program. The java.net package provides two classes--Socket and ServerSocket--that implement the client side of the connection and the server side of the connection, respectively.

Reading from and Writing to a Socket

This page contains a small example that illustrates how a client program can read from and write to a socket.

Writing a Client/Server Pair

The previous page showed an example of how to write a client program that interacts with an existing server via a Socket object. This page shows you how to write a program that implements the other side of the connection--a server program.

What Is a Socket?

Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request.

On the client-side: The client knows the hostname of the machine on which the server is running and the port number on which the server is listening. To make a connection request, the client tries to rendezvous with the server on the server's machine and port. The client also needs to identify itself to the server so it binds to a local port number that it will use during this connection. This is usually assigned by the system.

If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to the same local port and also has its remote endpoint set to the address and port of the client. It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.

On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server.

The client and server can now communicate by writing to or reading from their sockets.

Definition:

A *socket* is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to.

An endpoint is a combination of an IP address and a port number. Every TCP connection can be uniquely identified by its two endpoints. That way you can have multiple connections between your host and the server.

The `java.net` package in the Java platform provides a class, `Socket`, that implements one side of a two-way connection between your Java program and another program on the network. The `Socket` class sits on top of a platform-dependent implementation, hiding the details of any particular system from your Java program. By using the `java.net.Socket` class instead of relying on native code, your Java programs can communicate over the network in a platform-independent fashion.

Additionally, `java.net` includes the `ServerSocket` class, which implements a socket that servers can use to listen for and accept connections to clients. This lesson shows you how to use the `Socket` and `ServerSocket` classes.

If you are trying to connect to the Web, the `URL` class and related classes (`URLConnection`, `URLEncoder`) are probably more appropriate than the socket classes. In fact, URLs are a relatively high-level connection to the Web and use sockets as part of the underlying implementation.

Multi Threaded Server

How to create a multithreaded server by using ssock.accept() method of Socket class and MultiThreadServer(socketname) method of ServerSocket class.

```
import java.io.IOException;
import java.io.PrintStream;
import java.net.ServerSocket;
import java.net.Socket;

public class MultiThreadServer implements Runnable {
   Socket csocket;
   MultiThreadServer(Socket csocket) {
      this.csocket = csocket;
   }

   public static void main(String args[])
   throws Exception {
      ServerSocket ssock = new ServerSocket(1234);
      System.out.println("Listening");
      while (true) {
         Socket sock = ssock.accept();
         System.out.println("Connected");
         new Thread(new MultiThreadServer(sock)).start();
      }
   }
   public void run() {
      try {
         PrintStream pstream = new PrintStream
         (csocket.getOutputStream());
         for (int i = 100; i >= 0; i--) {
            pstream.println(i +
            " bottles of beer on the wall");
         }
         pstream.close();
         csocket.close();
```

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

Client Server Application

Each of these applications use the *client-server* paradigm, which is

1. One program, called the *server* blocks waiting for a client to connect to it
2. A client connects
3. The server and the client exchange information until they're done
4. The client and the server both close their connection

Points to be noted:

- Hosts have *ports*, numbered from 0-65535. Servers listen on a port. Some port numbers are reserved so you can't use them when you write your own server.
- Multiple clients can be communicating with a server on a given port. Each client connection is assigned a separate *socket* on that port.
- Client applications get a port and a socket on the client machine when they connect successfully with a server.

The four applications are

- A trivial date server and client, illustrating simple one-way communication. The server sends data to the client only.
- A capitalize server and client, illustrating two-way communication. Since the dialog between the client and server can comprise an unbounded number of messages back and forth, the server is *threaded* to service multiple clients efficiently.
- A two-player tic tac toe game, illustrating a server that needs to keep track of the state of a game, and inform each client of it, so they can each update their own displays.
- A multi-user chat application, in which a server must broadcast messages to all of its clients.

A Date Server and Client

The server

DateServer.java

```
import java.io.IOException;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Date;

/**
 * A TCP server that runs on port 9090.  When a client connects, it
 * sends the client the current date and time, then closes the
 * connection with that client.  Arguably just about the simplest
 * server you can write.
 */
public class DateServer {

    /**
     * Runs the server.
     */
    public static void main(String[] args) throws IOException {
        ServerSocket listener = new ServerSocket(9090);
        try {
            while (true) {
                Socket socket = listener.accept();
                try {
                    PrintWriter out =
                        new PrintWriter(socket.getOutputStream(), true);
                    out.println(new Date().toString());
                } finally {
                    socket.close();
                }
            }
        }
        finally {
            listener.close();
```

```
    }
  }
}
```

The client

DateClient.java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.Socket;

import javax.swing.JOptionPane;

/**
 * Trivial client for the date server.
 */
public class DateClient {

    /**
     * Runs the client as an application.  First it displays a dialog
     * box asking for the IP address or hostname of a host running
     * the date server, then connects to it and displays the date that
     * it serves.
     */
    public static void main(String[] args) throws IOException {
        String serverAddress = JOptionPane.showInputDialog(
            "Enter IP Address of a machine that is\n" +
            "running the date service on port 9090:");
        Socket s = new Socket(serverAddress, 9090);
        BufferedReader input =
            new BufferedReader(new InputStreamReader(s.getInputStream()));
        String answer = input.readLine();
        JOptionPane.showMessageDialog(null, answer);
        System.exit(0);
    }
}
```

Lesson: JDBC Introduction

The JDBC API is a Java API that can access any kind of tabular data, especially data stored in a Relational Database.

JDBC Product Components

JDBC includes four components:

1. **The JDBC API** — The JDBC™ API provides programmatic access to relational data from the Java™ programming language. Using the JDBC API, applications can execute SQL statements, retrieve results, and propagate changes back to an underlying data source. The JDBC API can also interact with multiple data sources in a distributed, heterogeneous environment.

   The JDBC API is part of the Java platform, which includes the *Java™ Standard Edition* (Java™ SE) and the *Java™ Enterprise Edition* (Java™ EE). The JDBC 4.0 API is divided into two packages: `java.sql` and `javax.sql`. Both packages are included in the Java SE and Java EE platforms.

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

   The Standard Extension packages `javax.naming` and `javax.sql` let you use a `DataSource` object registered with a *Java Naming and Directory Interface*™ (JNDI) naming service to establish a connection with a data source. You can use either connecting mechanism, but using a `DataSource` object is recommended whenever possible.

3. **JDBC Test Suite** — The JDBC driver test suite helps you to determine that JDBC drivers will run your program. These tests are not comprehensive or exhaustive, but they do exercise many of the important features in the JDBC API.
4. **JDBC-ODBC Bridge** — The Java Software bridge provides JDBC access via ODBC drivers. Note that you need to load ODBC binary code onto each client machine that uses this driver. As a result, the ODBC driver is most appropriate on a corporate network where client installations are not a major problem, or for application server code written in Java in a three-tier architecture.

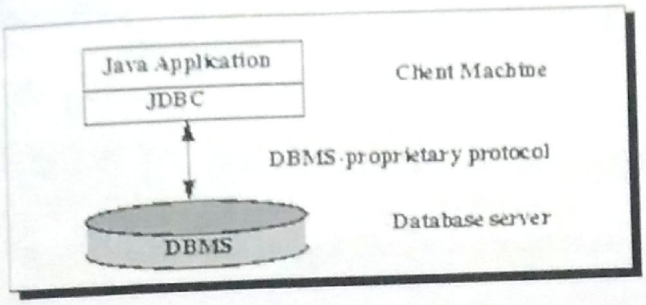
This Trail uses the first two of these these four JDBC components to connect to a database and then build a java program that uses SQL commands to communicate with a test Relational Database. The last two components are used in specialized environments to test web applications, or to communicate with ODBC-aware DBMSs.

JDBC Architecture

Two-tier and Three-tier Processing Models

The JDBC API supports both two-tier and three-tier processing models for database access.

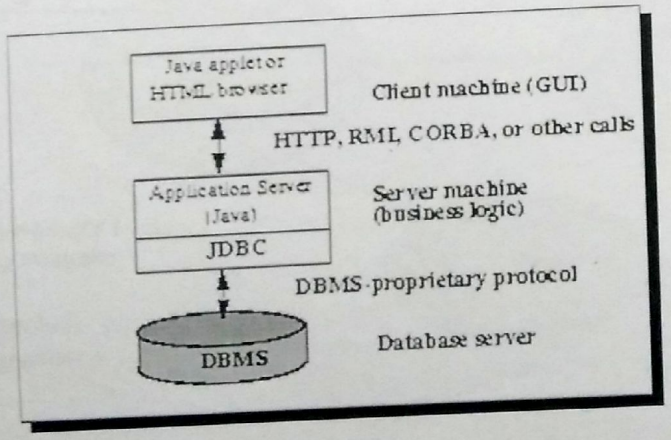
Figure 1: Two-tier Architecture for Data Access.



In the two-tier model, a Java applet or application talks directly to the data source. This requires a JDBC driver that can communicate with the particular data source being accessed. A user's commands are delivered to the database or other data source, and the results of those statements are sent back to the user. The data source may be located on another machine to which the user is connected via a network. This is referred to as a client/server configuration, with the user's machine as the client, and the machine housing the data source as the server. The network can be an intranet, which, for example, connects employees within a corporation, or it can be the Internet.

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

Figure 2: Three-tier Architecture for Data Access.



Until recently, the middle tier has often been written in languages such as C or C++, which offer fast performance. However, with the introduction of optimizing compilers that translate Java bytecode into efficient machine-specific code and technologies such as Enterprise JavaBeans™, the Java platform

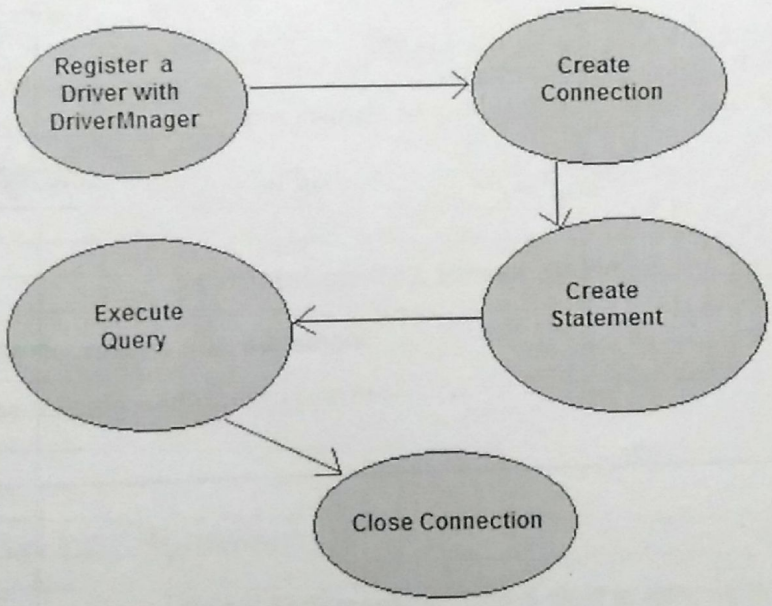
is fast becoming the standard platform for middle-tier development. This is a big plus, making it possible to take advantage of Java's robustness, multithreading, and security features.

With enterprises increasingly using the Java programming language for writing server code, the JDBC API is being used more and more in the middle tier of a three-tier architecture. Some of the features that make JDBC a server technology are its support for connection pooling, distributed transactions, and disconnected rowsets. The JDBC API is also what allows access to a data source from a Java middle tier.

Steps to connect a Java Application to Database

The following 5 steps are the basic steps involve in connecting a Java application with Database using JDBC.

1. Register the Driver
2. Create a Connection
3. Create SQL Statement
4. Execute SQL Statement
5. Closing the connection



Register the Driver

`Class.forName()` is used to load the driver class explicitly.

Example to register with JDBC-ODBC Driver

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

oracle, Access, Excel same

class.forname("com.imaginary.sql.msql.MsqlDriver") → msql

Create a Connection

`getConnection()` method of **DriverManager** class is used to create a connection.

Syntax

```
getConnection(String url)
getConnection(String url, String username, String password)
getConnection(String url, Properties info)
```

Example establish connection with Oracle Driver

```
Connection con = DriverManager.getConnection
("jdbc:oracle:thin:@localhost:1521:XE","username","password");
```

("jdbc:msql://dbserver.com/contacts", → msql);

("jdbc:odbc:Myaccess"); → Access

("jdbc:odbc:excelDB"); → excel

Create SQL Statement

`createStatement()` method is invoked on current **Connection** object to create a SQL Statement.

Syntax

```
public Statement createStatement() throws SQLException
```

Example to create a SQL statement

```
Statement s=con.createStatement();
```

Execute SQL Statement

`executeQuery()` method of **Statement** interface is used to execute SQL statements.

Syntax

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

Example to execute a SQL statement

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

Closing the connection

After executing SQL statement you need to close the connection and release the session. The `close()` method of **Connection** interface is used to close the connection.

Syntax

```
public void close() throws SQLException
```

Example of closing a connection

```
con.close();
```

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class Main {

  public static Connection getConnection() throws Exception {
    String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
    String url = "jdbc:odbc:excelDB";
    String username = "yourName";
    String password = "yourPass";
    Class.forName(driver);
    return DriverManager.getConnection(url, username, password);
  }

  public static void main(String args[]) throws Exception {
    Connection conn = null;
```

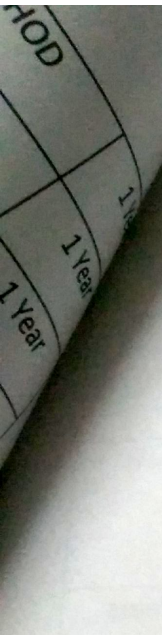
```
    Statement stmt = null;
    ResultSet rs = null;

    conn = getConnection();
    stmt = conn.createStatement();
    String excelQuery = "select * from [Sheet1$]";
    rs = stmt.executeQuery(excelQuery);

    while (rs.next()) {
      System.out.println(rs.getString("FirstName") + " " +
rs.getString("LastName"));
    }

    rs.close();
    stmt.close();
    conn.close();
  }
}
```

JDBC helps you to write Java applications that manage these three programming activities:

1. Connect to a data source, like a database
2. Send queries and update statements to the database
3. Retrieve and process the results received from the database in answer to your query

The following simple code fragment gives a simple example of these three steps:

```
public void connectToAndQueryDatabase(String username, String password) {

    Connection con = DriverManager.getConnection(
                     "jdbc:myDriver:myDatabase",
                     username,
                     password);

    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");

    while (rs.next()) {
        int x = rs.getInt("a");
        String s = rs.getString("b");
        float f = rs.getFloat("c");
    }
```



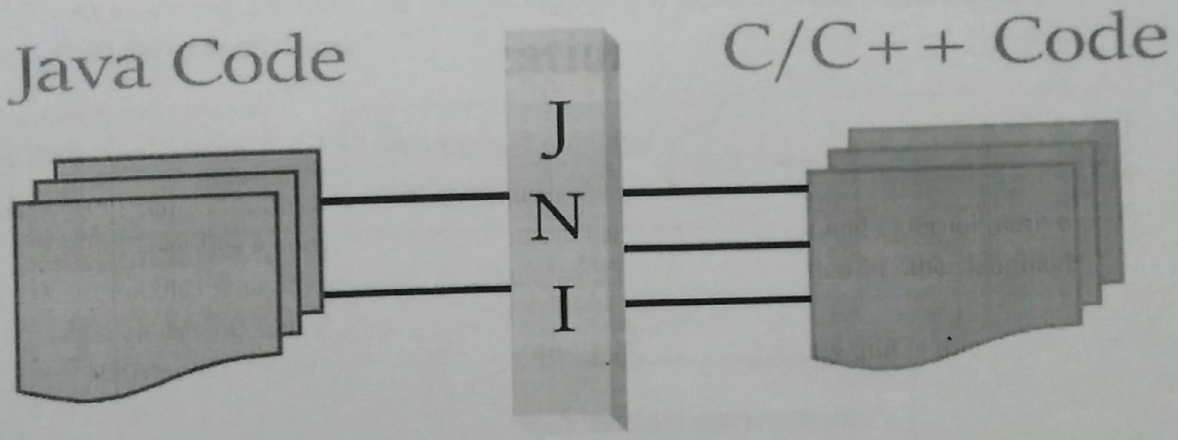
```
}
```

This short code fragment instantiates a `DriverManager` object to connect to a database driver and log into the database, instantiates a `Statement` object that carries your SQL language query to the database; instantiates a `ResultSet` object that retrieves the results of your query, and executes a simple `while` loop, which retrieves and displays those results. It's that simple.

JNI: Java Native Interface Introduction and "Hello World" application

Introduction, Purpose and features

1. JNI stands for **Java Native Interface**
2. JNI specifies a communication protocol between Java code and external, native code.
3. It enables your Java code to interface with native code written in other languages (such as C, C++)
4. Native code typically accesses the CPU and registers directly and is thus faster than interpreted code (like Java)
5. Java native methods are methods declared in your Java code (much like you declare an abstract method), but which are actually implemented in another programming language.



JNI – Java Native Interface

JNI allows Java programmers to

1. Leverage platform specific features outside a JVM. For example, you typically can't write a device driver in Java because you can't directly access the hardware
2. Leverage the improved speed possible with natively compiled code (such as C or assembler). For example, you might need an extremely fast math routine for a scientific application or game program.
3. Utilize existing code libraries in your Java programs. For example, there might be a really good file compression library written in C. Why try to rewrite it in Java when you can access it using JNI?

JNI Drawbacks

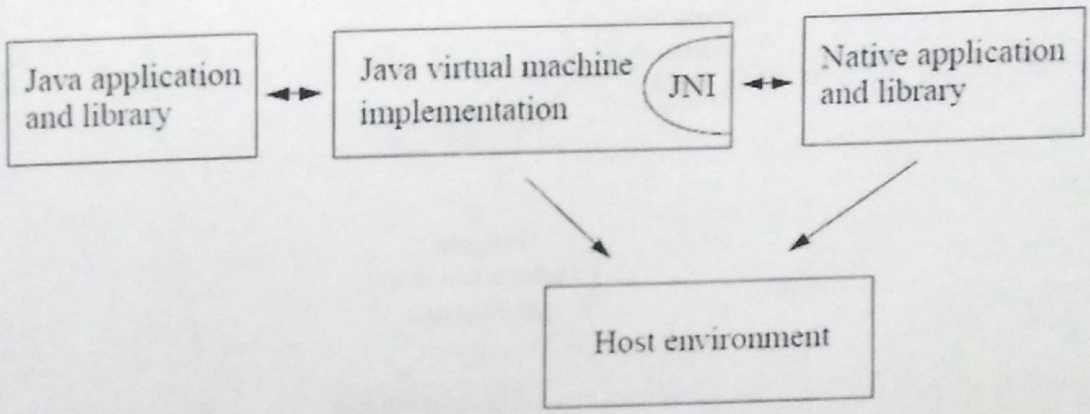
1. Your program is no longer platform independent
2. Your program is not as robust. If there is a null pointer exception in your native code, the JVM can't display a helpful message. It might even lock up.

JNI supports

1. Native methods can create and manipulate Java objects such as strings and arrays.
2. Native methods can manipulate and update Java objects passed into them (as parameters)
3. You can catch and throw exceptions from native methods and handle these exceptions in either the native method or your Java application
4. This almost seamless sharing of objects makes it very easy to incorporate native methods in your Java code

The Role of JNI

1. As a part of the Java virtual machine implementation, the JNI is a two-way interface that allows Java applications to invoke native code and vice versa.
2. The JNI is designed to handle situations where you need to combine Java applications with native code.
3. As a two-way interface, the JNI can support two types of native code: native libraries and native applications.



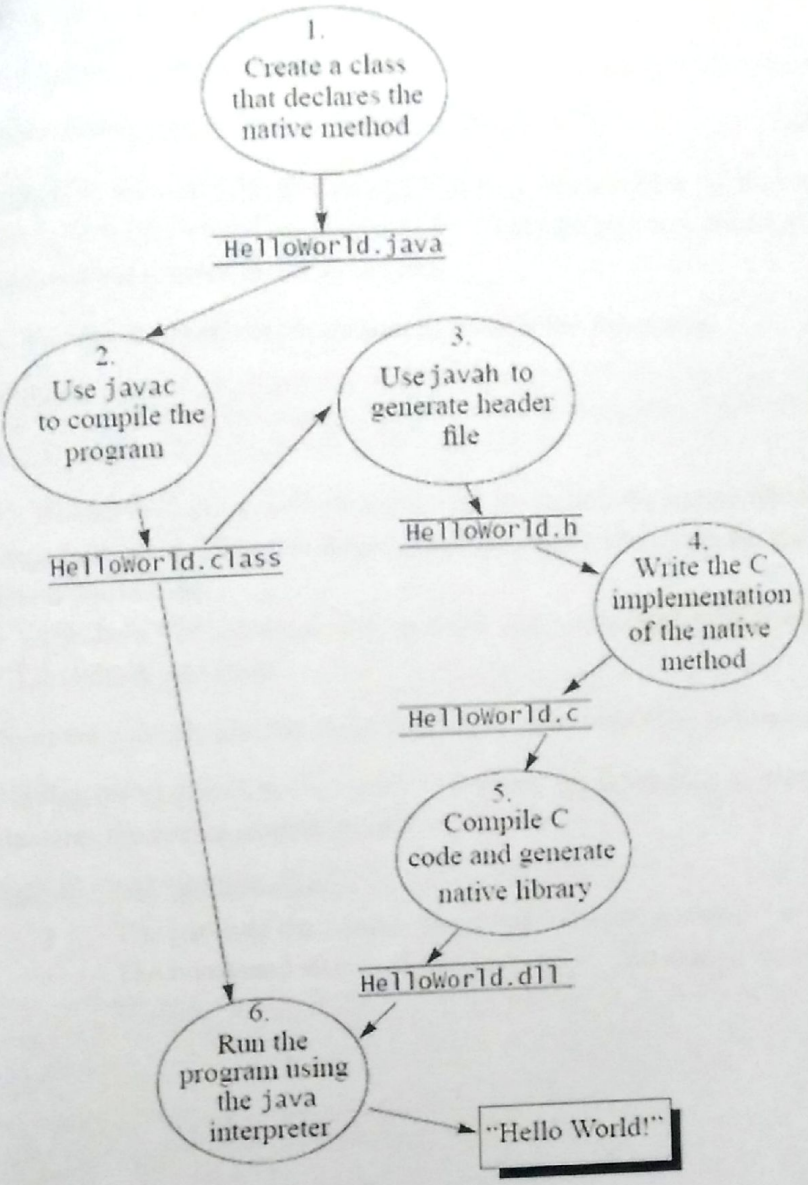
JNI "Hello World!" Application

Let's create the JNI "Hello World" application – a Java application that calls a C function via JNI to print "Hello World!".

The process consists of the following steps:

1. Create a class (HelloWorld.java) that declares the native method.
2. Use javac to compile the HelloWorld source file, resulting in the class file HelloWorld.class.
3. Use javah -jni to generate a C header file (HelloWorld.h) containing the function prototype for the native method implementation. The javah tool is provided with JDK or Java 2 SDK releases.
4. Write the C implementation (CLibHelloWorld.c) of the native method.
5. Compile the C implementation into a native library, creating HelloWorld.dll.

6. Run the HelloWorld program using the java runtime interpreter. Both the class file (HelloWorld.class) and the native library (HelloWorld.dll) are loaded at runtime.



The program defines a class named HelloWorld that contains a native method print().

1. Create a class (HelloWorld.java)

```
class HelloWorld {
        public native void print();  //native method
        static   //static initializer code
        {
                System.loadLibrary("CLibHelloWorld");
        }
        public static void main(String[] args)
        {
                HelloWorld hw = new HelloWorld();
        hw.print();
```

```
    }
}
```

There are two remarkable things here.

1. First one is the declaration of a native method using the keyword native.

```
public native void print();
```

This tells the compiler that print() will be accessed from an external library, and that it should not look for it in the Java source code. Accordingly, notice that there is no implementation of this method present in the Java code.

2. The second remarkable section of code is the following:

```
static {
        System.loadLibrary("CLibHelloWorld"); //**** Loads CLibHelloWorld.dll
}
```

- Before the native method print() can be called, the native library that implements print must be loaded. The code above loads the native library in the static initializer of the HelloWorld class.
- The Java VM automatically runs the static initializer before invoking any methods in the CLibHelloWorld class.

Next we compile the JNI HelloWorld java application as follows:

Having the application compiled we can use the javah tool to generate the header (h) file that declares the native method print().

```
javah -jni HelloWorld
```

- – The name of the header file is the class name with a ".h" appended to the end of it.
- – The command shown above generates a file named HelloWorld.h.

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include
/* Header for class HelloWorld */

#ifndef _Included_HelloWorld
#define _Included_HelloWorld
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     HelloWorld
 * Method:    print
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_HelloWorld_print
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

declares the native method print().

use the javah tool to generate the header (h) file that

```
javah -jni HelloWorld
```

- – The name of the header file is the class name with a ".h" appended to the end of it.
- – The command shown above generates a file named HelloWorld.h.

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include
/* Header for class HelloWorld */

#ifndef _Included_HelloWorld
#define _Included_HelloWorld
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     HelloWorld
 * Method:    print
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_HelloWorld_print
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

In the below line

```
JNIEXPORT void JNICALL Java_HelloWorld_print  (JNIEnv *, jobject);
```

- The method name is prefixed with Java_ and then the full name of the class to which it belongs, then the function name.
- This is to distinguish it from methods that belong to other classes and might have the same name.
- The first argument for every native method implementation is a JNIEnv interface pointer.
- The second argument is a reference to the HelloWorld object itself.

Writing the implementation:

Write the native C/C++ code which implements the method. Use the same signature that your header file uses. You might name your file something like "CLibHelloWorld.c".

```
#include "HelloWorld.h"
#include "jni.h"
#include  "stdio.h"

JNIEXPORT void JNICALL Java_HelloWorld_print(JNIEnv *env, jobject obj)
{
  printf("Hello world\n");
  return;
}
```

Here

- We use the **printf** function to display the string "Hello World!".
- Both arguments, the **JNIEnv** pointer and the reference to the object, are ignored.
- The C program includes three header files:
  - jni.h – provides information the native code needs to call JNI functions.
  - stdio.h – implements the printf() function.
  - HelloWorld.h – generated by using javah, it includes the C/C++ prototype for the Java_HelloWorld_print function.

Compile the C/C++ code into a library (a DLL if your code will run under Windows). – Click here to read step by step procedure to create DLL project using Visula Studio.

If you use C++ Builder to compile your library under Windows, make sure you create a DLL project and then add the C/C++ file to it (e.g. CLibHelloWorld.c). You'll need to add the following to your compiler's include path:

- \\javadir\\include
- \\javadir\\include\\win32

Be sure to name your project CLibHelloWorld so the DLL it creates will be named c_library.dll.

Run the Java class (main method). Be sure all the files and dll are in one folder.

```
java HelloWorld
```

You should see "Hello world" appear on the screen!

If you see a "java.lang.UnsatisfiedLinkError" error message, then Java couldn't find your shared library OR mismatch in native functions.

Serialization and Deserialization in Java

Serialization is a process of converting an object into a sequence of bytes which can be persisted to a disk or database or can be sent through streams. The reverse process of creating object from sequence of bytes is called **deserialization**.

A class must implement **Serializable** interface present in **java.io** package in order to serialize its object successfully. **Serializable** is a **marker interface** that adds serializable behaviour to the class implementing it.

Java provides **Serializable** API encapsulated under *java.io* package for serializing and deserializing objects which include,

- *java.io.serializable*
- *java.io.Externalizable*
- *ObjectInputStream*
- and *ObjectOutputStream* etc.

Marker interface

Marker Interface is a special interface in Java without any field and method. Marker interface is used to inform compiler that the class implementing it has some special behaviour or meaning. Some example of Marker interface are,

- java.io.Serializable
- java.lang.Cloneable
- java.rmi.Remote
- java.util.RandomAccess

All these interfaces does not have any method and field. They only add special behavior to the classes implementing them. However marker interfaces have been deprecated since Java 5, they were replaced by**Annotations**. Annotations are used in place of Marker Interface that play the exact same role as marker interfaces did before.

Signature of `writeObject()` and `readObject()`

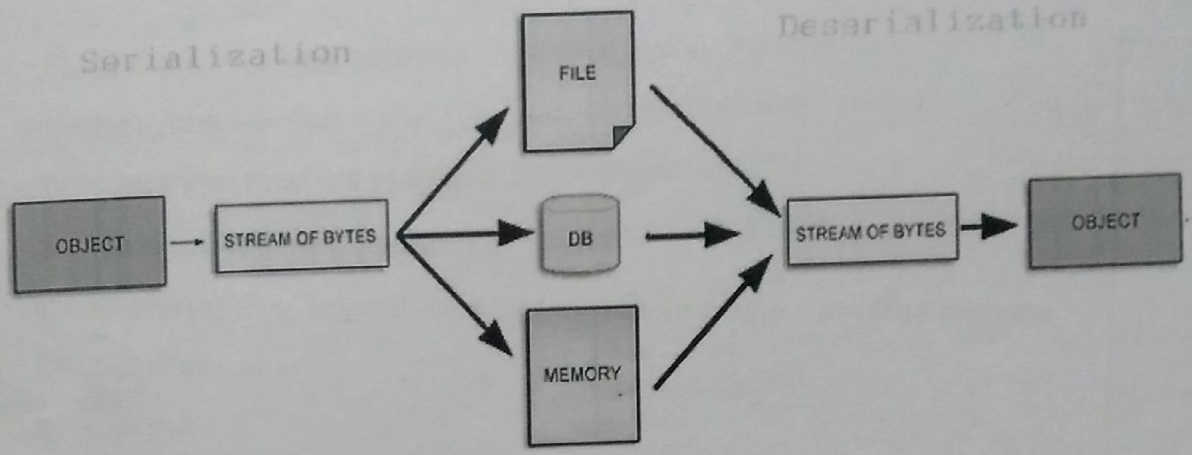
writeObject() method of *ObjectOutputStream* class serializes an object and send it to the output stream.

```
public final void writeObject(object x) throws IOException
```

readObject() method of *ObjectInputStream* class references object out of stream and deserialize it.

```
public final Object readObject() throws IOException,ClassNotFoundException
```

while serializing if you do not want any field to be part of object state then declare it either static or transient based on your need and it will not be included during java serialization process.



Serializing an Object

```
import java.io.*;
class studentinfo implements Serializable
{
 String name;
 int rid;
 static String contact;
 studentinfo(string n, int r, string c)
 {
  this.name = n;
  this.rid = r;
  this.contact = c;
 }
```

```
}

class Test
{
 public static void main(String[] args)
 {
 try
 {
  Studentinfo si = new studentinfo("Abhi", 104, "110044");
  FileOutputStream fos = new FileOutputStream("student.ser");
  Objectoutputstream oos = new ObjectOutputStream(fos);
  oos.writeObject(si);
  oos.close();
  fos.close();
  }
  catch (Exception e)
  { e. printStackTrace(); }
 }
}
```

Object of Studentinfo class is serialized using `writeObject()` method and written to **student.ser** file.

Deserialization of Object

```
import java.io * ;
class DeserializationTest
{
 public static void main(String[] args)
 {
  studentinfo si=null ;
  try
```

```
{
 FileInputStream fis = new FileInputStream("student.ser");
 ObjectOutputStream ois = new ObjectOutputStream(fis);
 si = (studentinfo)ois.readObject();
}
catch (Exception e)
 { e.printStackTrace(); }
System.out.println(si.name);
System.out. println(si.rid);
System.out.println(si.contact);
}
}
```

Output :

```
Abhi
104
null
```

Contact field is null because,it was marked as static and as we have discussed earlier static fields does not get serialized.

NOTE : Static members are never serialized because they are connected to class not object of class.

---

transient Keyword

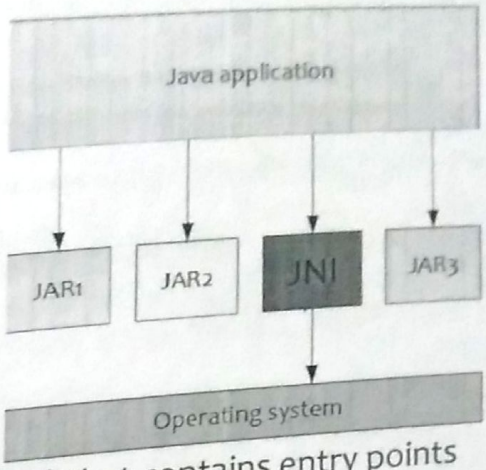
While serializing an object, if we don't want certain data member of the object to be serialized we can mention it transient. transient keyword will prevent that data member from being serialized.

```
class studentinfo implements Serializable
{
 String name;
 transient int rid;
 static String contact;
}
```

- Making a data member **transient** will prevent its serialization.
- In this example `rid` will not be serialized because it is **transient**, and `contact` will also remain unserialized because it is **static**.

In computing, the **Java Native Interface** (**JNI**) is a programming framework that enables Java code running in a Java Virtual Machine (JVM) to call and be called by[1] native applications (programs specific to a hardware and operating system platform) and libraries written in other languages such as C, C++ and assembly.

What is a Java Native Interface (JNI)?



JNI is a framework that connects Java via the Java Virtual Machine (JVM) to the underlying operating system or hardware through software that isn't (and cannot be reasonably) written in Java, but whose products or by-products are essential for consumption from a Java-based application (or other Java code).

A JNI is a shared-object library (on Windows, a DLL) that contains entry points reachable in Java via the JVM. If you do not understand these technologies, you may want to bone up on them, but it is not necessary to work through this tutorial and it may not be necessary for implementing your own JNI.

Your JNI can be written in most any language, most often C/C++ or even assembly, as long as it can observe certain calling conventions. On Windows, it is written as a DLL whereas on Linux and Unix it takes the final form of a shared-object library.

Toolstack and resources

Besides whatever file editing and source management software you like to use, all you need to develop a JNI on Linux is

- compiler like `gcc` or `g++` (plus `ld`, etc.)
- Java Developer's Kit (JDK)
- `make` (unless you like typing complicated commands over and over)
- Eclipse IDE (optional: if you like that sort of thing)
- Apache *log4j* (optional: very helpful if you're dumb like me)

16.1. JNI Overview

- An interface that allows Java to interact with code written in another language
- Motivation for JNI
  - Code reusability
    - Reuse existing/legacy code with Java (mostly C/C++)
  - Performance

- Native code used to be up to 20 times faster than Java, when running in interpreted mode
- Modern JIT compilers (HotSpot) make this a moot point
- Allow Java to tap into low level O/S, H/W routines
- JNI code is not portable!

Note

JNI can also be used to invoke Java code from within natively-written applications - such as those written in C/C++.
In fact, the java command-line utility is an example of one such application, that launches Java code in a Java Virtual Machine.

16.2. JNI Components

- `javah` - JDK tool that builds C-style header files from a given Java class that includes `native` methods
  - Adapts Java method signatures to native function prototypes
- `jni.h` - C/C++ header file included with the JDK that maps Java types to their native counterparts
  - `javah` automatically includes this file in the application header files

16.3. JNI Development (Java)

- Create a Java class with native method(s): `public native void sayHi(String who, int times);`
- Load the library which implements the method: `System.loadLibrary("HelloImpl");`
- Invoke the native method from Java

For example, our Java code could look like this:

```
package com.marakana.jniexamples;

public class Hello {
  public native void sayHi(String who, int times); //1

  static { System.loadLibrary("HelloImpl"); } //2

  public static void main (String[] args) {
    Hello hello = new Hello();
    hello.sayHi(args[0], Integer.parseInt(args[1])); //3
  }
}
```

❶ The method `sayHi` will be implemented in C/C++ in separate file(s), which will be compiled into a library.

❸

❷ The library filename will be called `libHelloImpl.so` (on Unix), `HelloImpl.dll` (on Windows) and `libHelloImpl.jnilib` (Mac OSX), but when loaded in Java, the library has to be loaded as `HelloImpl`.

16.4. JNI Development (C)

- We use the JDK `javah` utility to generate the header file `package_name_classname.h` with a function prototype for the `sayHi` method: `javac -d ./classes/ ./src/com/marakana/jniexamples/Hello.java` Then in the classes directory run: `javah -jni com.marakana.jniexamples.Hello` to generate the header file `com_marakana_jniexamples_Hello.h`
- We then create `com_marakana_jniexamples_Hello.c` to implement the `Java_com_marakana_jniexamples_Hello_sayHi` function

The file `com_marakana_jniexamples_Hello.h` looks like:

```
...
#include <jni.h>
```

```
...
JNIEXPORT void JNICALL Java_com_marakana_jniexamples_Hello_sayHi
  (JNIEnv *, jobject, jstring, jint);
...
```

The file Hello.c looks like:

```
#include <stdio.h>
#include "com_marakana_jniexamples_Hello.h"

JNIEXPORT void JNICALL Java_com_marakana_jniexamples_Hello_sayHi(JNIEnv *env, jobject obj, jstring
who, jint times) {
  jint i;
  jboolean iscopy;
  const char *name;
  name = (*env)->GetStringUTFChars(env, who, &iscopy);
  for (i = 0; i < times; i++) {
    printf("Hello %s\n", name);
  }
}
```

16.5. JNI Development (Compile)

- We are now ready to compile our program and run it
  - The compilation is system-dependent
- This will create `libHelloImpl.so`, `HelloImpl.dll`, `libHelloImpl.jnilib` (depending on the O/S)
- Set `LD_LIBRARY_PATH` to point to the directory where the compiled library is stored
- Run your Java application

For example, to compile `com_marakana_jniexamples_Hello.c` in the "classes" directory (if your `.h` file and `.c` file are there) on Linux do:

```
gcc -o libHelloImpl.so -lc -shared \
    -I/usr/local/jdk1.6.0_03/include \
    -I/usr/local/jdk1.6.0_03/include/linux com_marakana_jniexamples_Hello.c
```

On Mac OSX :

```
gcc -o libHelloImpl.jnilib -lc -shared \
    -I/System/Library/Frameworks/JavaVM.framework/Headers com_marakana_jniexamples_Hello.c
```

Then set the `LD_LIBRARY_PATH` to the current working directory:

```
export LD_LIBRARY_PATH=.
```

Finally, run your application in the directory where your compiled classes are stored ("classes" for example):

```
java com.marakana.jniexamples.Hello Student 5
Hello Student
Hello Student
Hello Student
Hello Student
Hello Student
```

Part 1: Introduction and "Hello World" application

Introduction, Purpose and features

1. JNI stands for **Java Native Interface**
2. JNI specifies a communication protocol between Java code and external, native code.
3. It enables your Java code to interface with native code written in other languages (such as C, C++)

RMI (Remote Method Invocation)

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

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

Understanding stub and skeleton

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

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

stub

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

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

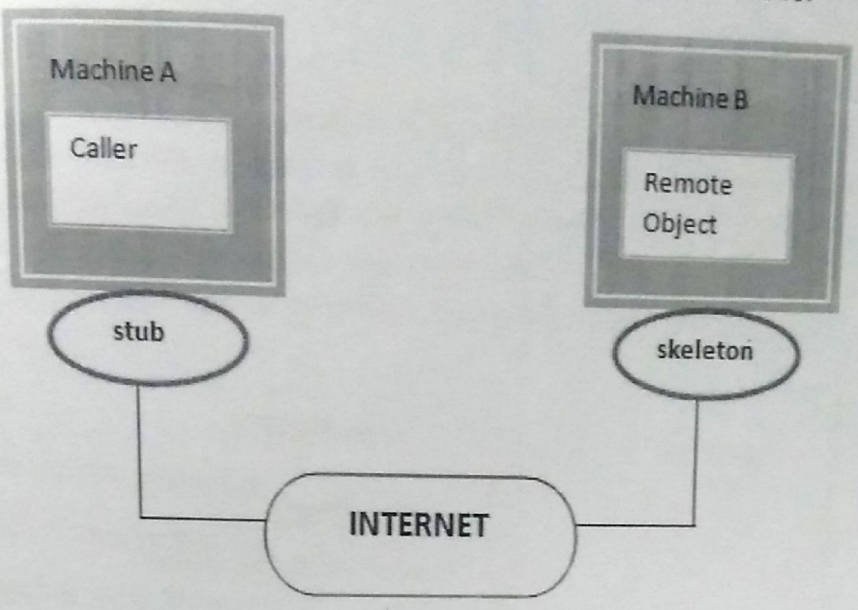
skeleton

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

1. It reads the parameter for the remote method

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

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



skeletons.

Understanding requirements for the distributed applications

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

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

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

Steps to write the RMI program

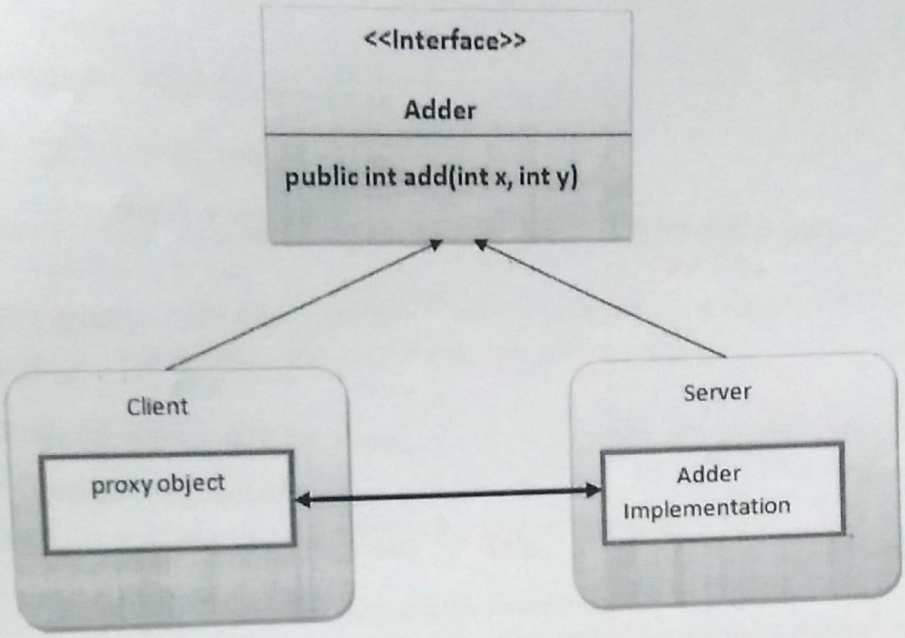
The is given the 6 steps to write the RMI program.

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.

1. **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.

2. **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.
3. **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.
4. **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.
5. **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.

```
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

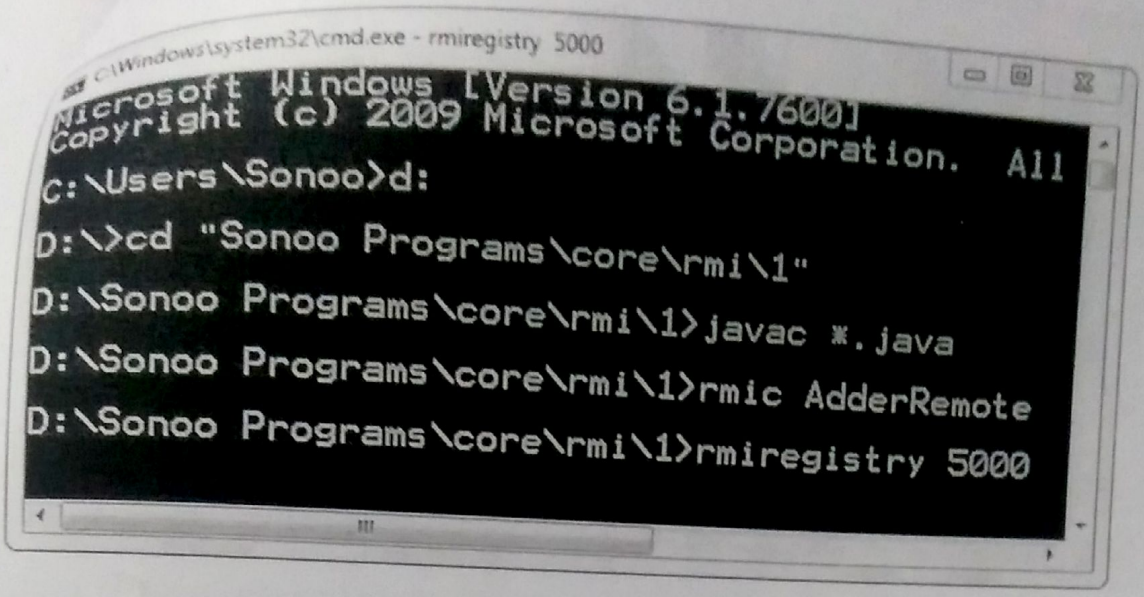
1. For running **this** rmi example,
2.
3. 1) compile all the java files

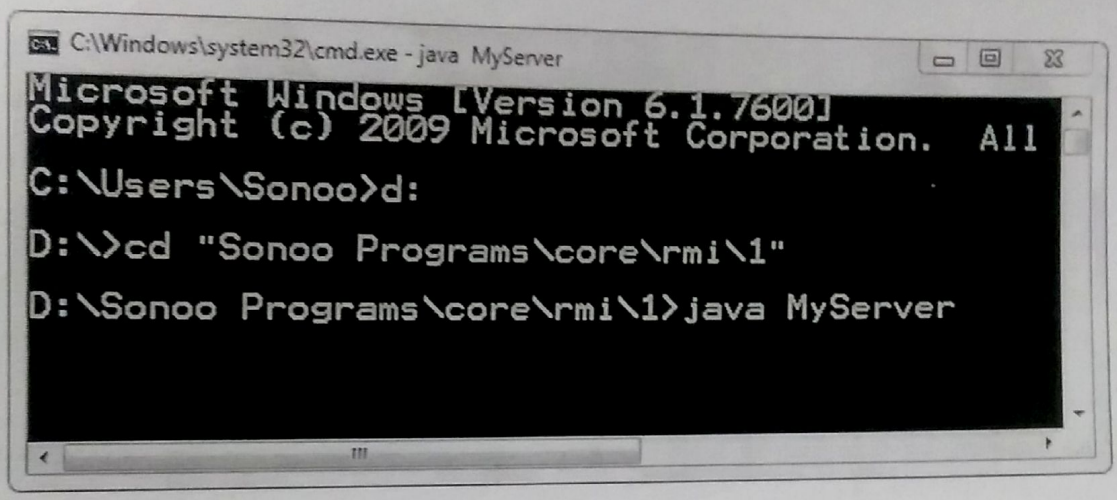
javac *.java

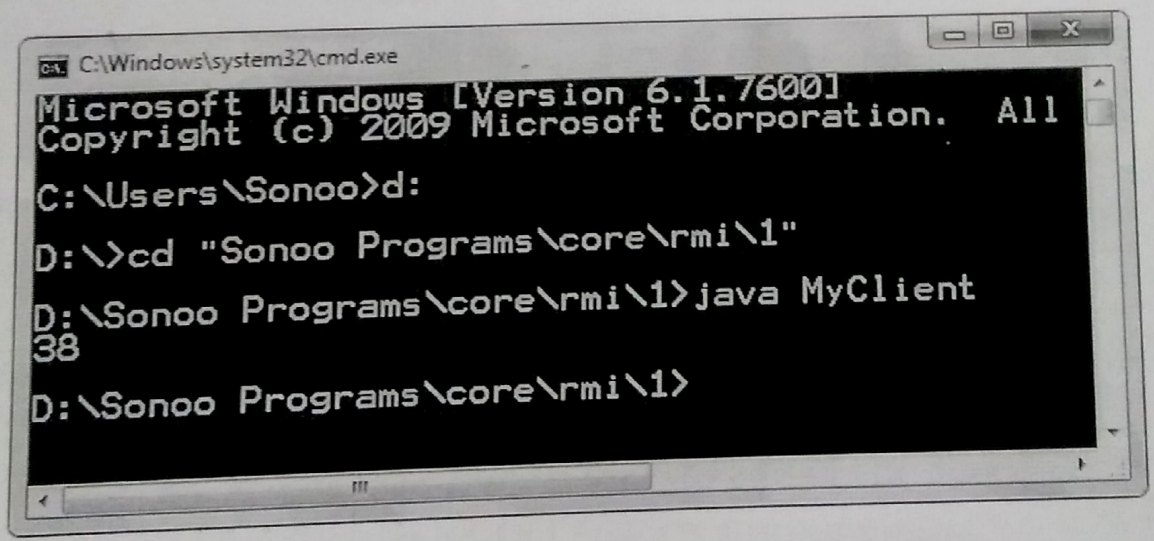
2)create stub and skeleton object by rmic tool

```
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







Meaningful example of RMI application with database

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

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

1) Create the table

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

CUSTOMER400

Table | Data | Indexes | Model | Constraints | Grants | Statistics | UI Defaults | Triggers | Dependencies | SQL

Query | Count Rows | Insert Row

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

row(s) 1 - 3 of 3

Download

2) Create Customer class and Remote interface

File: Customer.java

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

Note: Customer class must be Serializable.

File: Bank.java

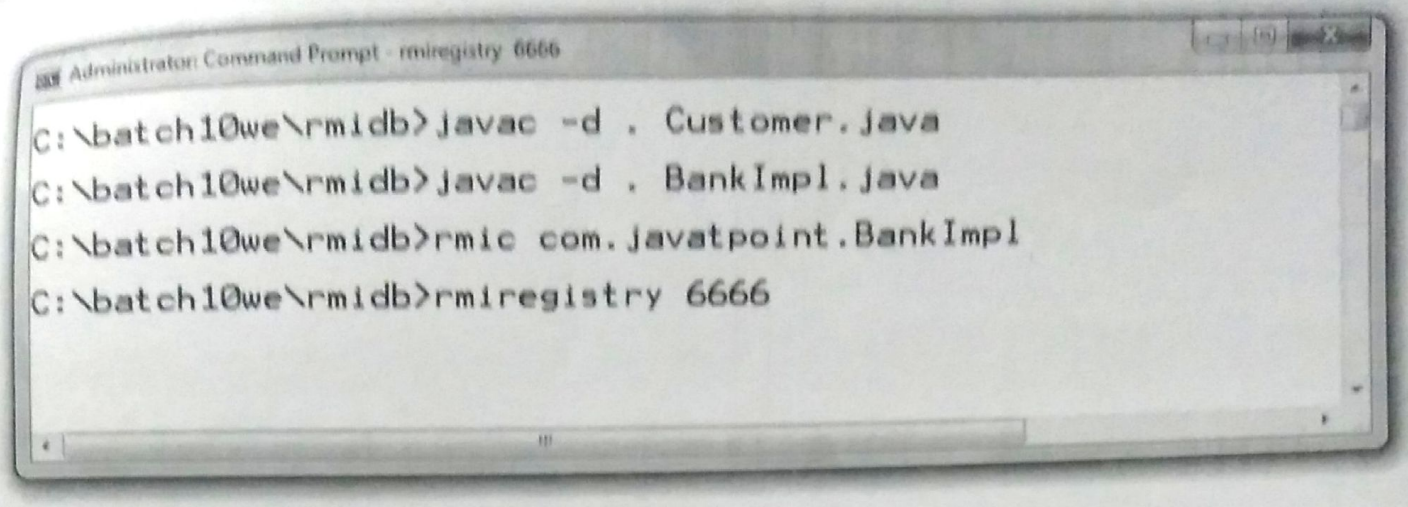
```
1. package com.javatpoint;
2. import java.rmi.*;
3. import java.util.*;
4. interface Bank extends Remote{
5. public List<Customer> getCustomers()throws RemoteException;
6. }
```

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

File: BankImpl.java

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

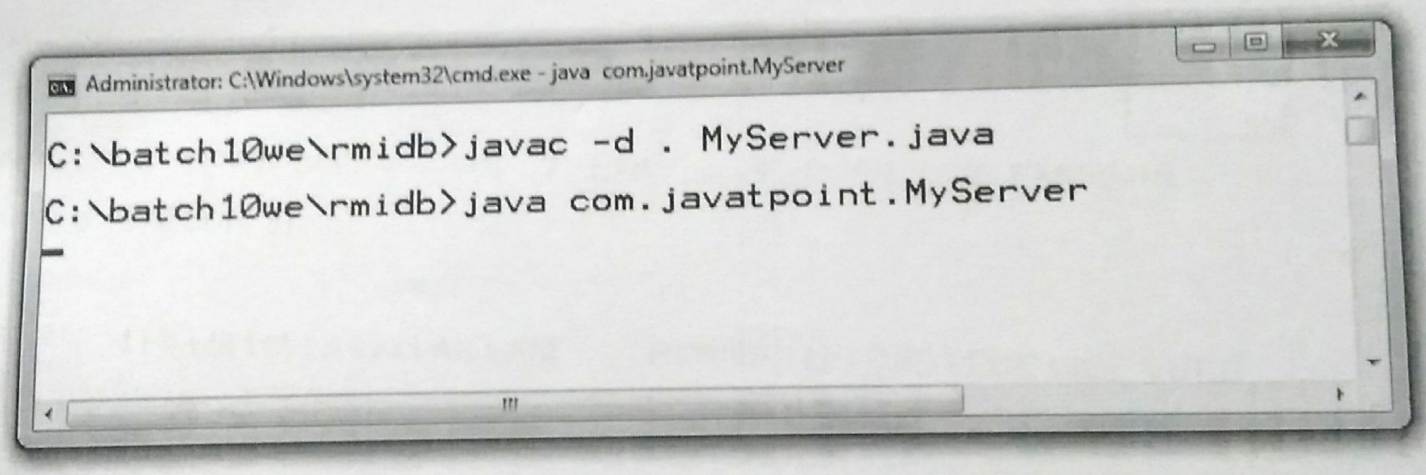
4) Compile the class rmic tool and start the registry service by rmiregistry tool



5) Create and run the Server

File: MyServer.java

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

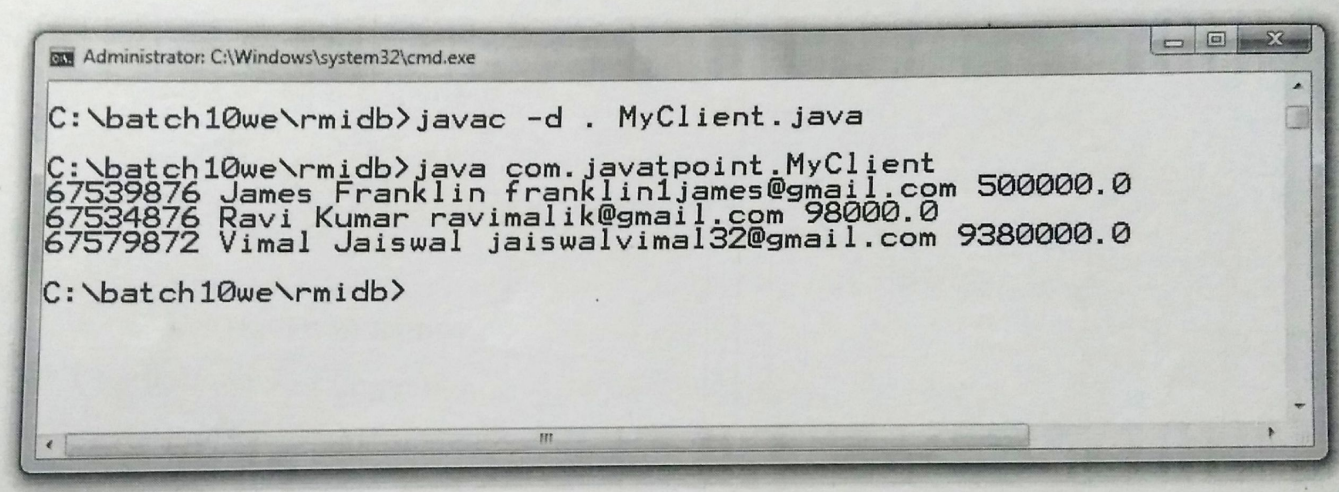


6) Create and run the Client

File: MyClient.java

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

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



download this example of rmi with database

Object Serialization :-

Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

Classes **ObjectInputStream** and **ObjectOutputStream** are high-level streams that contain the methods for serializing and deserializing an object.

The ObjectOutputStream class contains many write methods for writing various data types, but one method in particular stands out –

```
public final void writeObject(Object x) throws IOException
```

The above method serializes an Object and sends it to the output stream. Similarly, the ObjectInputStream class contains the following method for deserializing an object –

```
public final Object readObject() throws IOException, ClassNotFoundException
```

This method retrieves the next Object out of the stream and deserializes it. The return value is Object, so you will need to cast it to its appropriate data type.

To demonstrate how serialization works in Java, I am going to use the Employee class that we discussed early on in the book. Suppose that we have the following Employee class, which implements the Serializable interface –

Example

```
public class Employee implements java.io.Serializable {
   public String name;
   public String address;
```

```
   public transient int SSN;
   public int number;

   public void mailCheck() {
      System.out.println("Mailing a check to " + name + " " + address);
   }
}
```

Notice that for a class to be serialized successfully, two conditions must be met –

- The class must implement the java.io.Serializable interface.
- All of the fields in the class must be serializable. If a field is not serializable, it must be marked **transient**.

If you are curious to know if a Java Standard Class is serializable or not, check the documentation for the class. The test is simple: If the class implements java.io.Serializable, then it is serializable; otherwise, it's not.

Serializing an Object

The ObjectOutputStream class is used to serialize an Object. The following SerializeDemo program instantiates an Employee object and serializes it to a file.

When the program is done executing, a file named employee.ser is created. The program does not generate any output, but study the code and try to determine what the program is doing.

Note – When serializing an object to a file, the standard convention in Java is to give the file a **.ser** extension.

Example

```
import java.io.*;
public class SerializeDemo {

   public static void main(String [] args) {
      Employee e = new Employee();
      e.name = "Reyan Ali";
```

```
      e.address = "Phokka Kuan, Ambehta Peer";
      e.SSN = 11122333;
      e.number = 101;

      try {
         FileOutputStream fileOut =
         new FileOutputStream("/tmp/employee.ser");
         ObjectOutputStream out = new ObjectOutputStream(fileOut);
         out.writeObject(e);
         out.close();
         fileOut.close();
         System.out.printf("Serialized data is saved in /tmp/employee.ser");
      }catch(IOException i) {
         i.printStackTrace();
      }
   }
}
```

Deserializing an Object

The following DeserializeDemo program deserializes the Employee object created in the SerializeDemo program. Study the program and try to determine its output –

Example

```
import java.io.*;
public class DeserializeDemo {

   public static void main(String [] args) {
      Employee e = null;
      try {
         FileInputStream fileIn = new FileInputStream("/tmp/employee.ser");
         ObjectInputStream in = new ObjectInputStream(fileIn);
         e = (Employee) in.readObject();
         in.close();
         fileIn.close();
      }catch(IOException i) {
```

```
            i.printStackTrace();
            return;
        }catch(ClassNotFoundException c) {
            System.out.println("Employee class not found");
            c.printStackTrace();
            return;
        }

        System.out.println("Deserialized Employee...");
        System.out.println("Name: " + e.name);
        System.out.println("Address: " + e.address);
        System.out.println("SSN: " + e.SSN);
        System.out.println("Number: " + e.number);
    }
}
```

This will produce the following result –

Output

```
Deserialized Employee...
Name: Reyan Ali
Address:Phokka Kuan, Ambehta Peer
SSN: 0
Number:101
```

Here are following important points to be noted –

- The try/catch block tries to catch a ClassNotFoundException, which is declared by the readObject() method. For a JVM to be able to deserialize an object, it must be able to find the bytecode for the class. If the JVM can't find a class during the deserialization of an object, it throws a ClassNotFoundException.
- Notice that the return value of readObject() is cast to an Employee reference.
- The value of the SSN field was 11122333 when the object was serialized, but because the field is transient, this value was not sent to the output stream. The SSN field of the deserialized Employee object is 0.