Sockets:

The term *network programming* refers to writing programs that execute across multiple devices (computers), in which the devices are all connected to each other using a network.

The java.net package of the J2SE APIs contains a collection of classes and interfaces that provide the low-level communication details, allowing you to write programs that focus on solving the problem at hand.

The java.net package provides support for the two common network protocols:

- **TCP:** TCP stands for Transmission Control Protocol, which allows for reliable communication between two applications. TCP is typically used over the Internet Protocol, which is referred to as TCP/IP.
- **UDP:** UDP stands for User Datagram Protocol, a connection-less protocol that allows for packets of data to be transmitted between applications.

Socket Programming:

Sockets provide the communication mechanism between two computers using TCP. A client program creates a socket on its end of the communication and attempts to connect that socket to a server.

When the connection is made, the server creates a socket object on its end of the communication. The client and server can now communicate by writing to and reading from the socket.

The java.net.Socket class represents a socket, and the java.net.ServerSocket class provides a mechanism for the server program to listen for clients and establish connections with them.

The following steps occur when establishing a TCP connection between two computers using sockets:

- The server instantiates a ServerSocket object, denoting which port number communication is to occur on.
- The server invokes the accept() method of the ServerSocket class. This method waits until a client connects to the server on the given port.
- After the server is waiting, a client instantiates a Socket object, specifying the server name and port number to connect to.
- The constructor of the Socket class attempts to connect the client to the specified server and port number. If communication is established, the client now has a Socket object capable of communicating with the server.
- On the server side, the accept() method returns a reference to a new socket on the server that is connected to the client's socket.

After the connections are established, communication can occur using I/O streams. Each socket has both an OutputStream and an InputStream. The client's OutputStream is connected to the server's InputStream, and the client's InputStream is connected to the server's OutputStream.

ServerSocket Class Methods:

The **java.net.ServerSocket** class is used by server applications to obtain a port and listen for client requests

The ServerSocket class has four constructors:

SN	Methods with Description
1	**public ServerSocket(int port) throws IOException** Attempts to create a server socket bound to the specified port. An exception occurs if the port is already bound by another application.
2	**public ServerSocket(int port, int backlog) throws IOException** Similar to the previous constructor, the backlog parameter specifies how many incoming clients to store in a wait queue.
3	**public ServerSocket(int port, int backlog, InetAddress address) throws IOException** Similar to the previous constructor, the InetAddress parameter specifies the local IP address to bind to. The InetAddress is used for servers that may have multiple IP addresses, allowing the server to specify which of its IP addresses to accept client requests on
4	**public ServerSocket() throws IOException** Creates an unbound server socket. When using this constructor, use the bind() method when you are ready to bind the server socket

If the ServerSocket constructor does not throw an exception, it means that your application has successfully bound to the specified port and is ready for client requests.

Here are some of the common methods of the ServerSocket class:

SN	Methods with Description
1	**public int getLocalPort()** Returns the port that the server socket is listening on. This method is useful if you passed in 0 as the port number in a constructor and let the server find a port for you.
2	**public Socket accept() throws IOException** Waits for an incoming client. This method blocks until either a client connects to the server on the specified port or the socket times out, assuming that the time-out value has been set using the setSoTimeout() method. Otherwise, this method blocks indefinitely
3	**public void setSoTimeout(int timeout)** Sets the time-out value for how long the server socket waits for a client during the accept().
4	**public void bind(SocketAddress host, int backlog)** Binds the socket to the specified server and port in the SocketAddress object. Use this method if you instantiated the ServerSocket using the no-argument constructor.

Both the client and server have a Socket object, so these methods can be invoked by both the client and server.

SN	Methods with Description
1	**public void connect(SocketAddress host, int timeout) throws IOException** This method connects the socket to the specified host. This method is needed only when you instantiated the Socket using the no-argument constructor.
2	**public InetAddress getInetAddress()** This method returns the address of the other computer that this socket is connected to.
3	**public int getPort()** Returns the port the socket is bound to on the remote machine.
4	**public int getLocalPort()** Returns the port the socket is bound to on the local machine.
5	**public SocketAddress getRemoteSocketAddress()** Returns the address of the remote socket.
6	**public InputStream getInputStream() throws IOException** Returns the input stream of the socket. The input stream is connected to the output stream of the remote socket.
7	**public OutputStream getOutputStream() throws IOException** Returns the output stream of the socket. The output stream is connected to the input stream of the remote socket

8	**public void close() throws IOException** Closes the socket, which makes this Socket object no longer capable of connecting again to any server

Example:To get ip address, hostname and length of address of our system

```
import java .net.*;
public class Prog
{
public static void main(String ss[])
{
String str;
int l;
try
{
InetAddress i1=InetAddress.getLocalHost();
str=i1.getHostAddress();
String hostname=i1.getHostName();
l=str.length();
System.out.println("Ip of the system is : "+str);
System.out.println("Address Length is : "+l);
System.out.println("HostName is : "+hostname);
}
catch(Exception ex)
{
System.out.println("Exception caught is : "+ex.getMessage());
}
}
}
```

DEVELOPMENT OF CLIENT SERVER APPLICATION

Example:To establish a connection between a client and a server

Client.java

```
import.java.io.*;
import.java.net.*;
class Client
{
Public static void main(String args[])
{
Try
{Socket ob=new Socket();
ob.connect(new InetAddress("localhost",8181));
system.out.println("Connected to server");
}
Catch(Exception e)
{
System.out.println(e);
}
}
}
```

Server.java

```
import.java.io.*;
import.java.net.*;
class Server
{
Public static void main(String args[])
Throws IOException
{
Try
{
ServerSocket a=new ServerSocket();
a.bind(new InetSocketAddress("localhost",8181));
System.out.println("server is listening for connection");
Socket ab=a.accept();
System.out.println(“client connected”);
}
Catch(Exception e)
{
System.out.println(e);
}
}
}
```

DESIGN OF MULTITHREADED SERVER

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);
    }
  }
}
```

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:

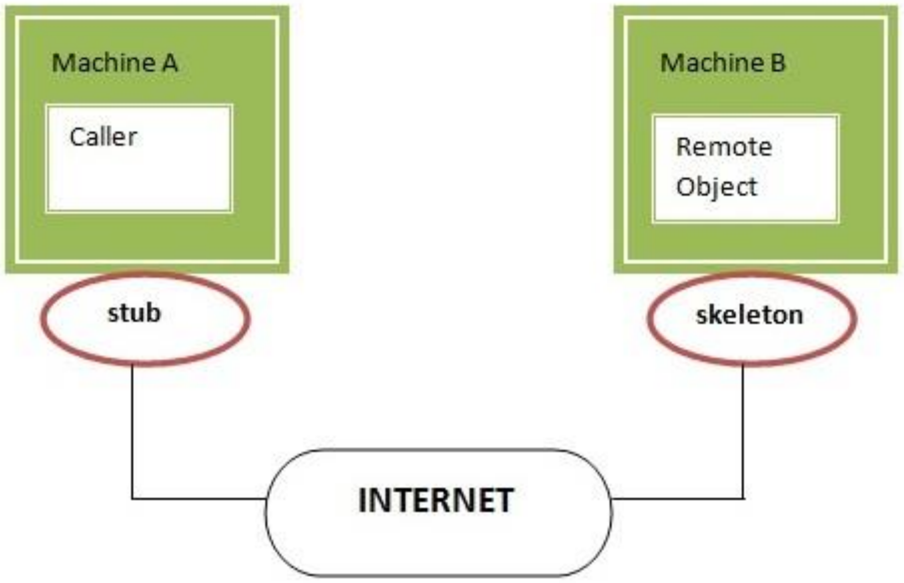
1. It initiates a connection with remote Virtual Machine (JVM),
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads (unmarshals) the return value or exception, and
5. It finally, returns the value to the caller.

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.

The application need to locate the remote method

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

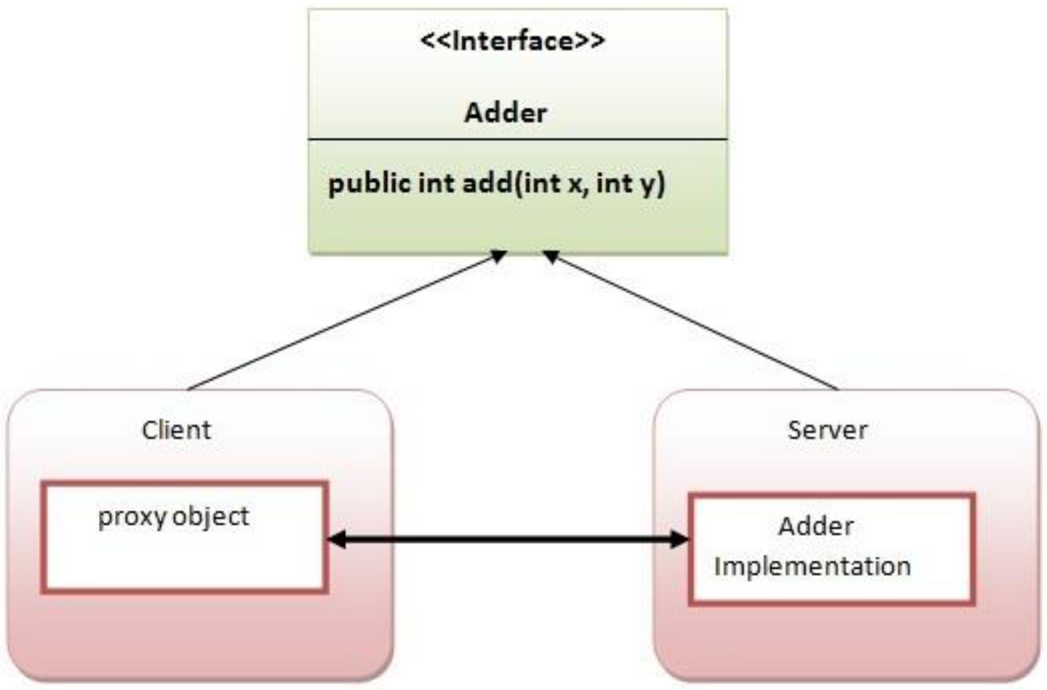
Steps to write the RMI program

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 rmi registry tool
5. Create and start the remote application
6. Create and start the client application

RMI 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.

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.

```java
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);}
}
        }
```

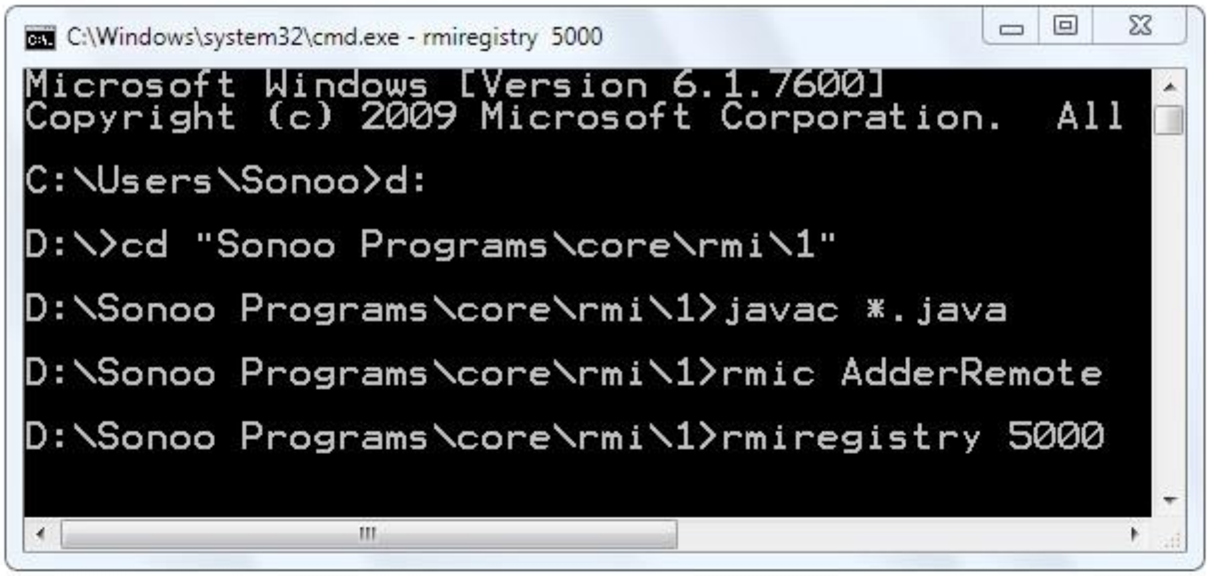
6) Create and run the client application

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.

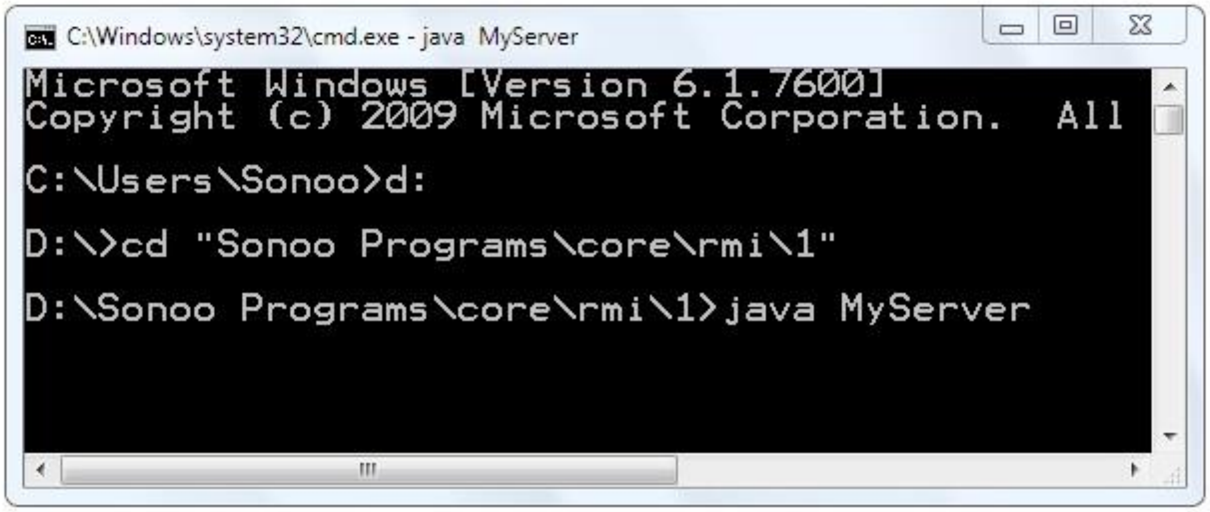
```java
import java.rmi.*;
public class MyClient{
public static void main(String args[]){
try{
Adder stub=(Adder)Naming.lookup("rmi://localhost:5000/sonoo");
System.out.println(stub.add(34,4));
}catch(Exception e){}
}
}
```

```
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
```

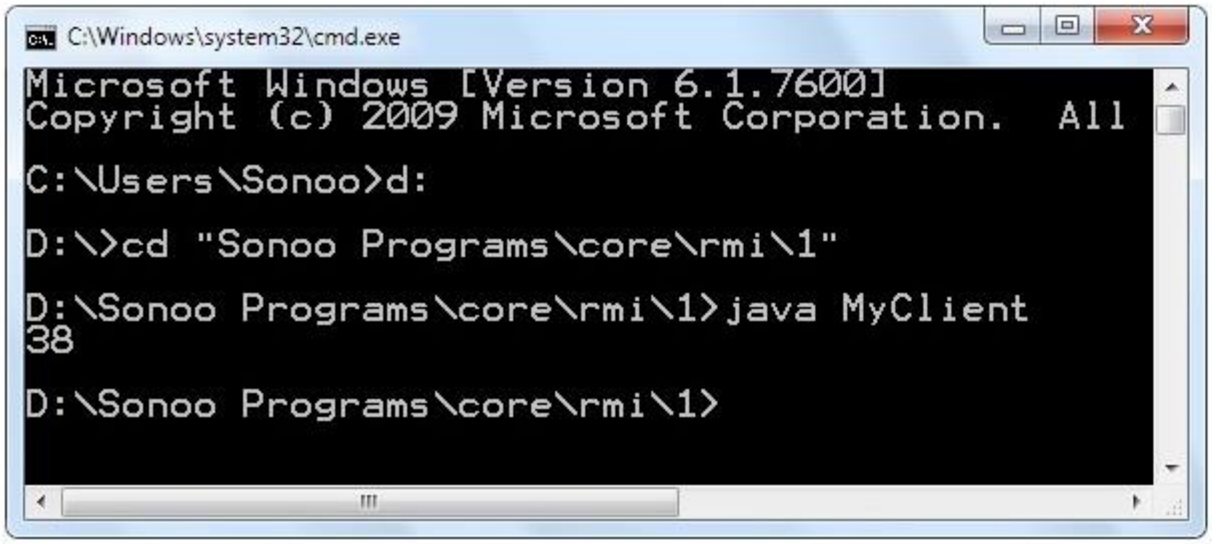
Output of this RMI examplejava native ino



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

C:\Users\Sonoo>d:

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

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

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

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



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

C:\Users\Sonoo>d:

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

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



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

C:\Users\Sonoo>d:

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

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

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

JAVA NATIVE INTERFACES

JNI is a mechanism that allows

• a Java program to call a function in a C or C++ program.

• a C or C++ program to call a method in a Java program.

Reasons for using Native Methods

• Get access to system features that can be handled more easily in C or C++. • Need access to legacy code that has been well tested.

• Need the performance that C provides and that Java does not have (yet)

Steps In Using JNI

1. Java code (write and compile)

• Declare native methods using native and no body. public native void nativeOne(); • Ensure that a shared library, to be created later, is loaded before the native method is called. System.loadLibrary("NativeLib");

Usually executed in a static initializer block in the class that calls the native method**(s).**

2. Create a C header file containing functionprototypes for the native methods. javah -jni NativeMethods where NativeMethods is the Java class containing the native methods.

3. Write C implementations of native methods using mangled names and extra parameters.

Native (C/C++) code function names are formed from the following pieces: 'Java_' Mangled fully qualified class name with periods becoming underscores '_'

For overloaded Java methods: Continue with two underscores '__' Mangled argument types

4. Compile C code and Create shared library

5. . Execute Java program

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

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`.

VECTOR

Vector implements a dynamic array. It is similar to ArrayList, but with two differences:

- Vector is synchronized.
- Vector contains many legacy methods that are not part of the collections framework.

Vector proves to be very useful if you don't know the size of the array in advance or you just need one that can change sizes over the lifetime of a program.

Below given are the list of constructors provided by the vector class.

SR.NO	Constructor and Description
1	**Vector( )** This constructor creates a default vector, which has an initial size of 10
2	**Vector(int size)** This constructor accepts an argument that equals to the required size, and creates a vector whose initial capacity is specified by size:
3	**Vector(int size, int incr)** This constructor creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward
4	**Vector(Collection c)**

	creates a vector that contains the elements of collection

STACK

Stack is a subclass of Vector that implements a standard last-in, first-out stack.Stack only defines the default constructor, which creates an empty stack. Stack includes all the methods defined by Vector, and adds several of its own.

```
Stack( )
```

Apart from the methods inherited from its parent class Vector, Stack defines following methods:

SN	Methods with Description
1	**boolean empty()** Tests if this stack is empty. Returns true if the stack is empty, and returns false if the stack contains elements.
2	**Object peek( )** Returns the element on the top of the stack, but does not remove it.
3	**Object pop( )** Returns the element on the top of the stack, removing it in the process.
4	**Object push(Object element)** Pushes element onto the stack. element is also returned.

5	**int search(Object element)** Searches for element in the stack. If found, its offset from the top of the stack is returned. Otherwise, .1 is returned.

HASHTABLE CLASSES

Java Hashtable class

A Hashtable is an array of list.Each list is known as a bucket.The position of bucket is identified by calling the hashcode() method.A Hashtable contains values based on the key. It implements the Map interface and extends Dictionary class.

1. It contains only unique elements.
2. It may have not have any null key or value.
3. It is synchronized.

Example of Hashtable:

```
import java.util.*;

class TestCollection16{

 public static void main(String args[]){

     Hashtable<Integer,String> hm=new Hashtable<Integer,String>();

    hm.put(100,"Amit");

  hm.put(102,"Ravi");

  hm.put(101,"Vijay");

  hm.put(103,"Rahul");

    for(Map.Entry m:hm.entrySet()){

   System.out.println(m.getKey()+" "+m.getValue());

  }  }}
```

Like HashMap, Hashtable stores key/value pairs in a hash table. When using a Hashtable, you specify an object that is used as a key, and the value that you want linked to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.

Below given is the list of constructors provided by the HashTable class.

Sr.No	Constructor and Description
1	**Hashtable( )** This is the default constructor of the hash table it instantiates the Hashtable class.
2	**Hashtable(int size)** This constructor accepts an integer parameter and creates a hash table that has an initial size specified by integer value size.
3	**Hashtable(int size, float fillRatio)** This creates a hash table that has an initial size specified by size and a fill ratio specified by fillRatio. This ratio must be between 0.0 and 1.0, and it determines how full the hash table can be before it is resized upward.
4	**Hashtable(int size, float fillRatio)** This constructor creates a hash table that is initialized with the elements in m. The capacity of the hash table is set to twice the number of elements in m. The default load factor of 0.75 is used.

ENUMERATIONS:

The Enumeration interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects.

This legacy interface has been superceded by Iterator. Although not deprecated, Enumeration is considered obsolete for new code. However, it is used by several methods defined by the legacy classes such as Vector and Properties, is used by several other API classes, and is currently in widespread use in application code.

The methods declared by Enumeration are summarized in the following table:

SN	Methods with Description
1	**boolean hasMoreElements( )** When implemented, it must return true while there are still more elements to extract, and false when all the elements have been enumerated.
2	**Object nextElement( )** This returns the next object in the enumeration as a generic Object reference.

Example:

```
import java.util.Vector;

import java.util.Enumeration;

public class EnumerationTester {

  public static void main(String args[]) {

    Enumeration days;

    Vector dayNames = new Vector();

    dayNames.add("Sunday");

    dayNames.add("Monday");
```

```
        dayNames.add("Tuesday");
        dayNames.add("Wednesday");
        dayNames.add("Thursday");
        dayNames.add("Friday");
        dayNames.add("Saturday");
        days = dayNames.elements();
        while (days.hasMoreElements()){
           System.out.println(days.nextElement());
        }
    }
}
```