Introduction to Distributed and Parallel Computing
CS-401

Dr. Sanjay Saxena
Visiting Faculty, CSE, IIIT Vadodara
Assistant Professor, CSE, IIIT Bhubaneswar
Post doc – University of Pennsylvania, USA
PhD – Indian Institute of Technology(BHU), Varanasi

Java RMI: Remote Method Invocation

- It is a mechanism that allows an object residing in one system (JVM) to access/invoke an object running on another JVM.
- RMI is used to build distributed applications; it provides remote communication between Java programs. It is provided in the package **java.rmi**.

Architecture of an RMI Application

In an RMI application, we write two programs, a **server program** (resides on the server) and a **client program** (resides on the client).

- Inside the server program, a remote object is created and reference of that object is made available for the client (using the registry).
- The client program requests the remote objects on the server and tries to invoke its methods.

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

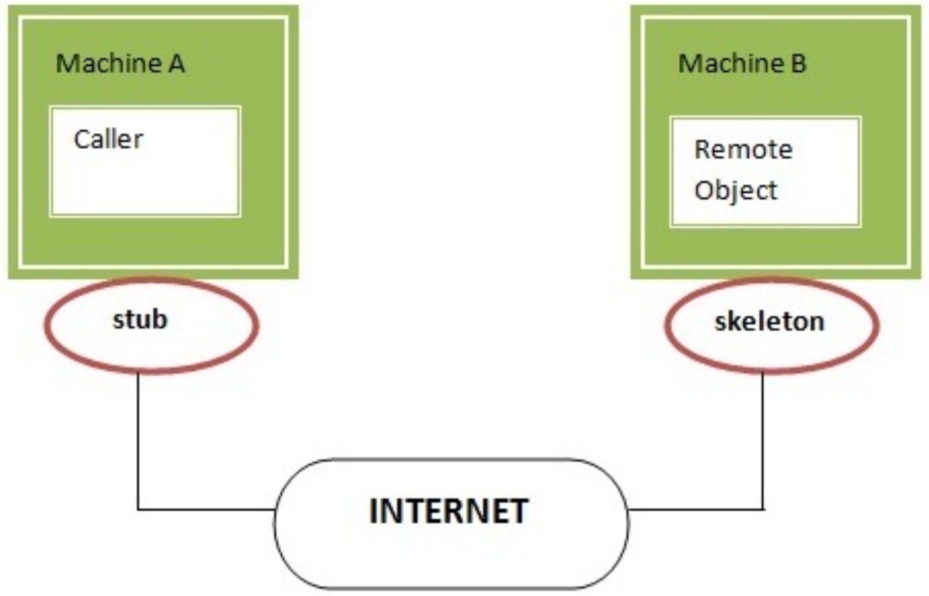
The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:

- It initiates a connection with remote Virtual Machine (JVM),
- It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
- It waits for the result
- It reads (unmarshals) the return value or exception, and
- 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:

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



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

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

Java RMI Example

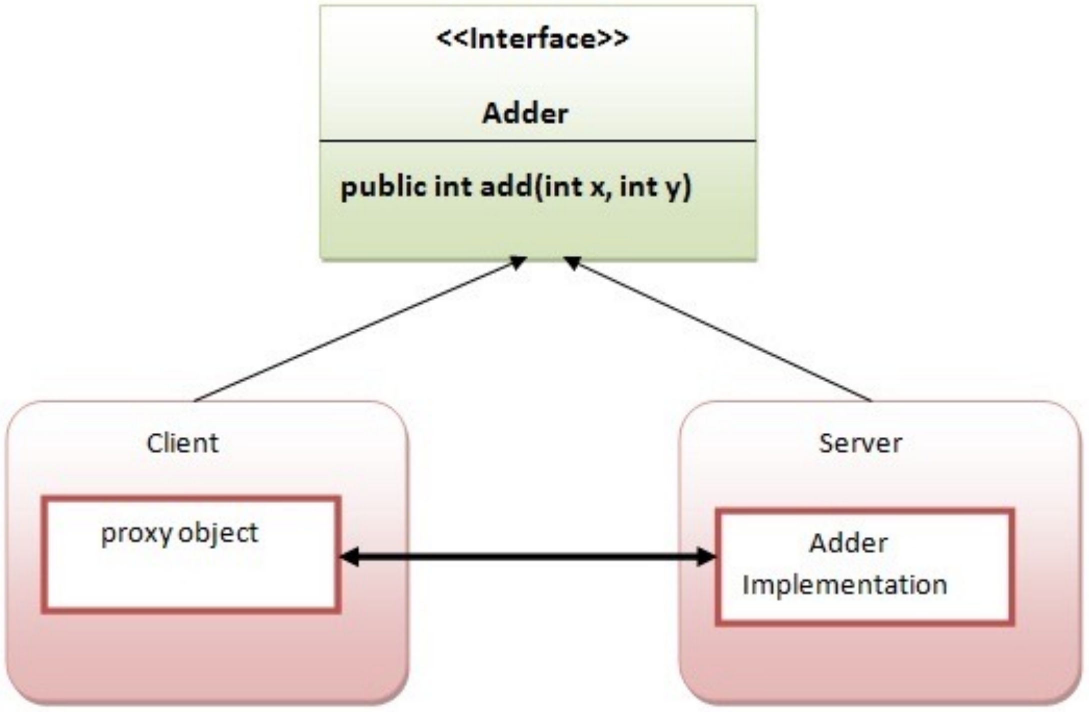
The is given the 6 steps to write the RMI program.

- ❖ Create the remote interface
- ❖ Provide the implementation of the remote interface
- ❖ Compile the implementation class and create the stub and skeleton objects using the rmic tool
- ❖ Start the registry service by rmiregistry tool
- ❖ Create and start the remote application
- ❖ Create and start the client application

RMI Example

In this example, we have followed all the 6 steps to create and run the rmi application.

The client application need only two files, remote interface and client application. In the rmi application, both client and server interacts with the remote interface. The client application invokes methods on the proxy object, RMI sends the request to the remote JVM. The return value is sent back to the proxy object and then to the client application.



1) create the remote interface

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

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

2) Provide the implementation of the remote interface

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

•Either extend the UnicastRemoteObject class,

•or use the exportObject() method of the UnicastRemoteObject class

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

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

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

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

```
rmic AdderRemote
```

4) Start the registry service by the rmiregistry tool

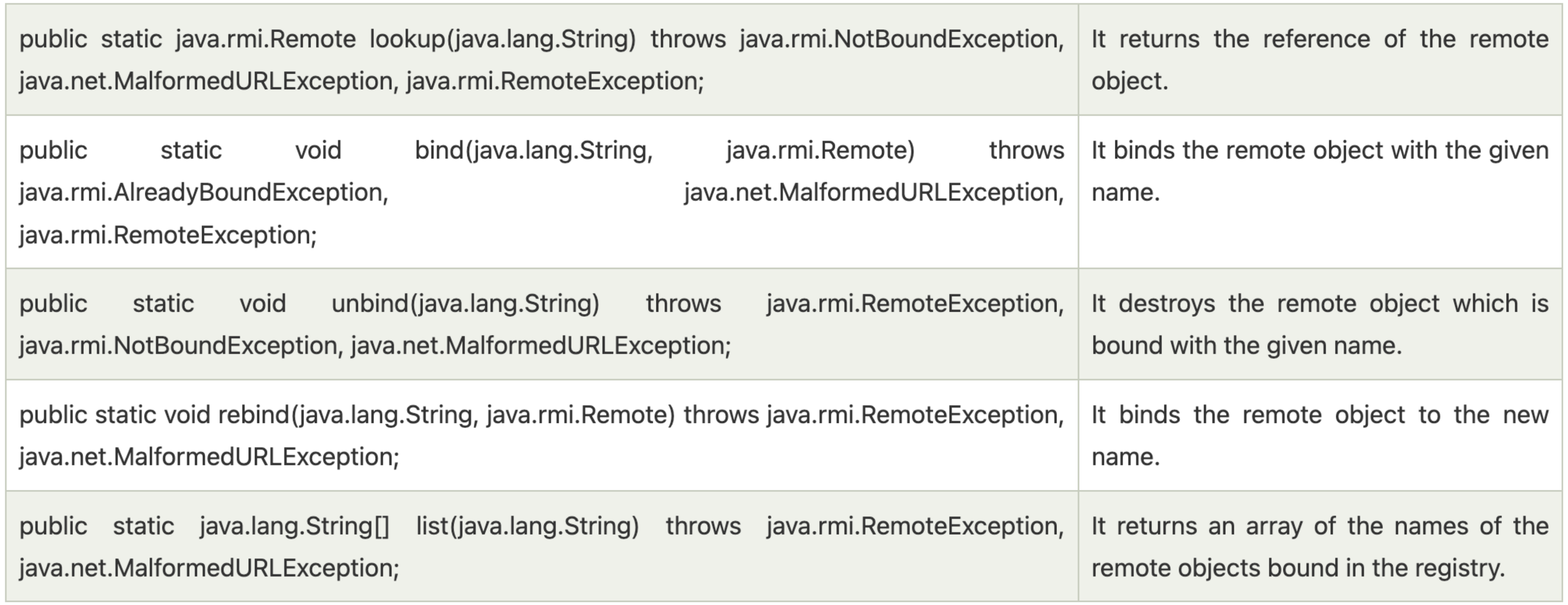
Now start the registry service by using the rmiregistry tool. If you don't specify the port number, it uses a default port number. In this example, we are using the port number 5000.

```
rmiregistry 5000
```

5) Create and run the server application

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

The Naming class provides 5 methods.



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

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

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

6) Create and run the client application

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

```
import java.rmi.*;
public class MyClient{
public static void main(String args[]){
try{
Adder   stub=(Adder)Naming.lookup("rmi://localhost:5000/sono
o");
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

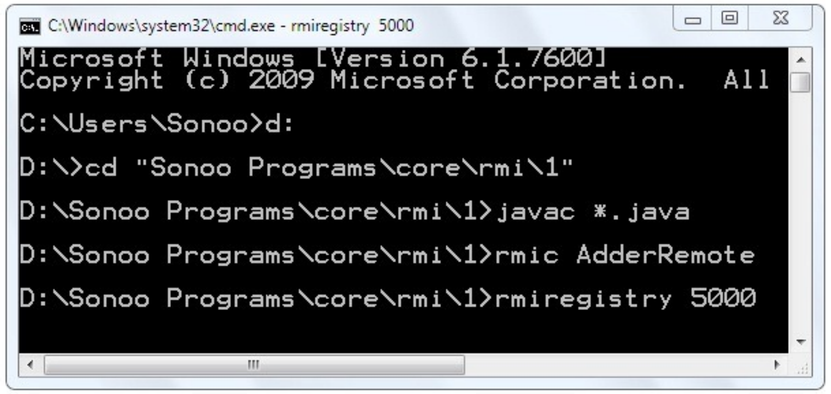
rmiregistry 5000

4)start the server in another command prompt

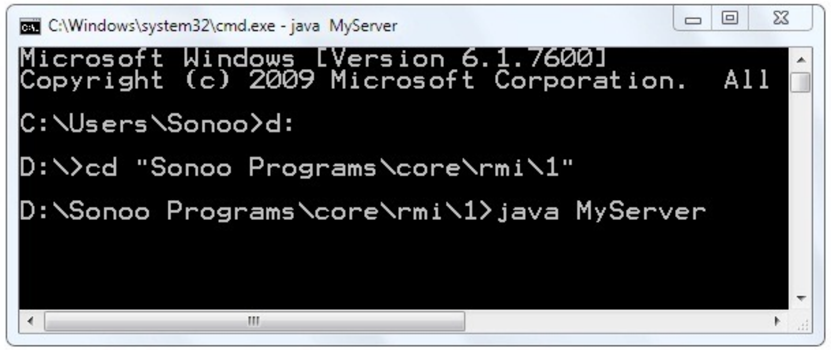
java MyServer

5)start the client application in another command prompt

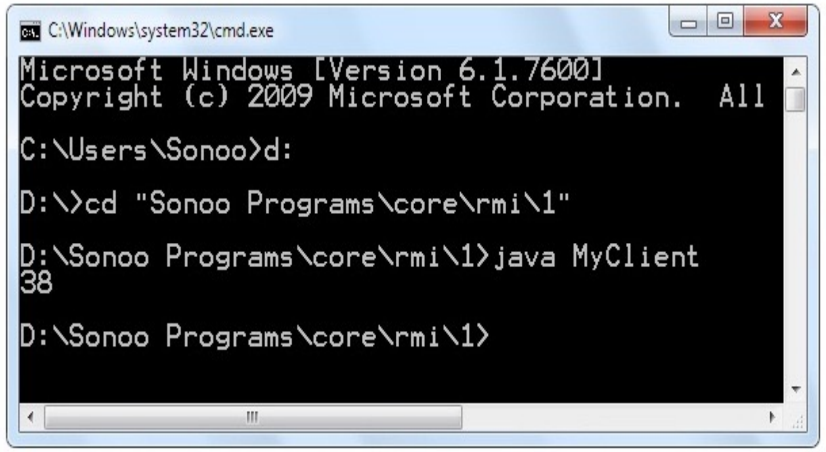
java MyClient



```
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>
```

Introduction to OpenMP

- OpenMP is a standard parallel programming API for shared memory environments, written in C, C++, or FORTRAN.
- It consists of a set of compiler directives with a “lightweight” syntax, library routines, and environment variables that influence run-time behavior.
- OpenMP is governed by OpenMP Architecture Review Board (or OpenMP ARB), and is defined by several hardware and software vendors.
- OpenMP behavior is directly dependent on the OpenMP implementation. Capabilities of this implementation can enable the programmer to separate the program into serial and parallel regions rather than just concurrently running threads, hides stack management, and provides synchronization of constructs.
- That being said OpenMP will not guarantee speedup, parallelize dependencies, or prevent data racing. Data racing, keeping track of dependencies, and working towards a speedup are all up to the programmer.

Why do we use OpenMP?

- OpenMP has received considerable attention in the past decade and is considered by many to be an ideal solution for parallel programming because it has unique advantages as a mainstream directive-based programming model.

- First of all, OpenMP provides a cross-platform, cross-compiler solution. It supports lots of platforms such as Linux, macOS, and Windows. Mainstream compilers including GCC, LLVM/Clang, Intel Fortran, and C/C++ compilers provide OpenMP good support.

- Secondly, using OpenMP can be very convenient and flexible to modify the number of threads. To solve the scalability problem of the number of CPU cores. In the multi-core era, the number of threads needs to change according to the number of CPU cores. OpenMP has irreplaceable advantages in this regard.

- Thirdly, using OpenMP to create threads is considered to be convenient and relatively easy because it does not require an entry function, the code within the same function can be decomposed into multiple threads for execution, and a for loop can be decomposed into multiple threads for execution.

OpenMP | Hello World program

1.Include the header file: We have to include the OpenMP header for our program along with the standard header files.

```
//OpenMP header

#include <omp.h>
```

2. Specify the parallel region: In OpenMP, we need to mention the region which we are going to make it as parallel using the keyword **pragma omp parallel**. The **pragma omp parallel** is used to fork additional threads to carry out the work enclosed in the parallel. **The original thread will be denoted as the master thread with thread ID 0.** Code for creating a parallel region would be,

```
#pragma omp parallel
{
//Parallel region code
}
```

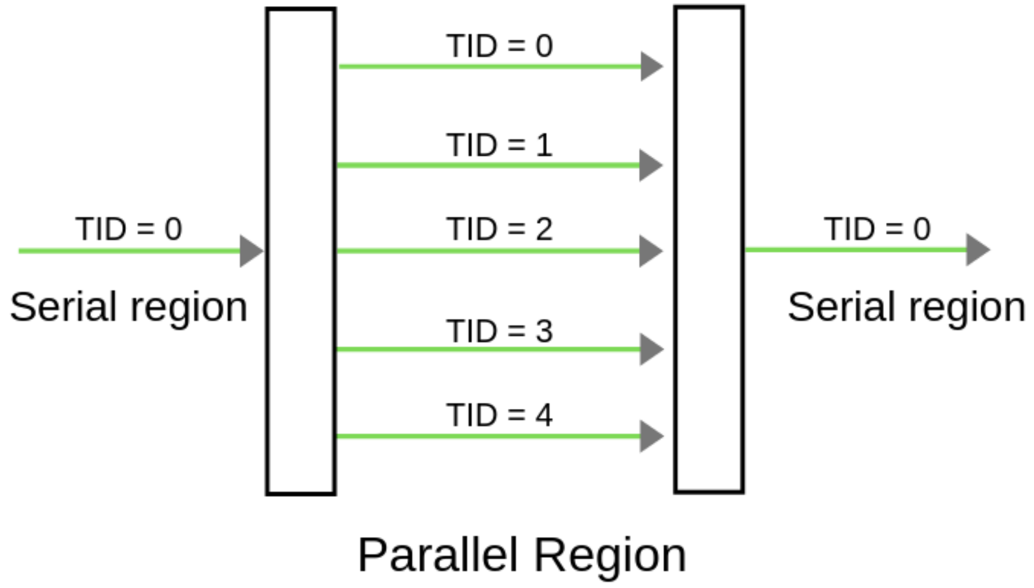
So

```
#pragma omp parallel
{
printf("Hello World... from thread = %d\n", omp_get_thread_num());
}
```

Set the number of threads:
we can set the number of threads to execute the program using the external variable.

```
export OMP_NUM_THREADS=5
```

Diagram of parallel region



As per the above figure, Once the compiler encounters the parallel regions code, the master thread(*thread which has thread id 0*) will fork into the specified number of threads. Here it will get forked into 5 threads because we will initialise the number of threads to be executed as 5, using the command export OMP_NUM_THREADS=5. Entire code within the parallel region will be executed by all threads concurrently. **Once the parallel region ended, all threads will get merged into the master thread.**

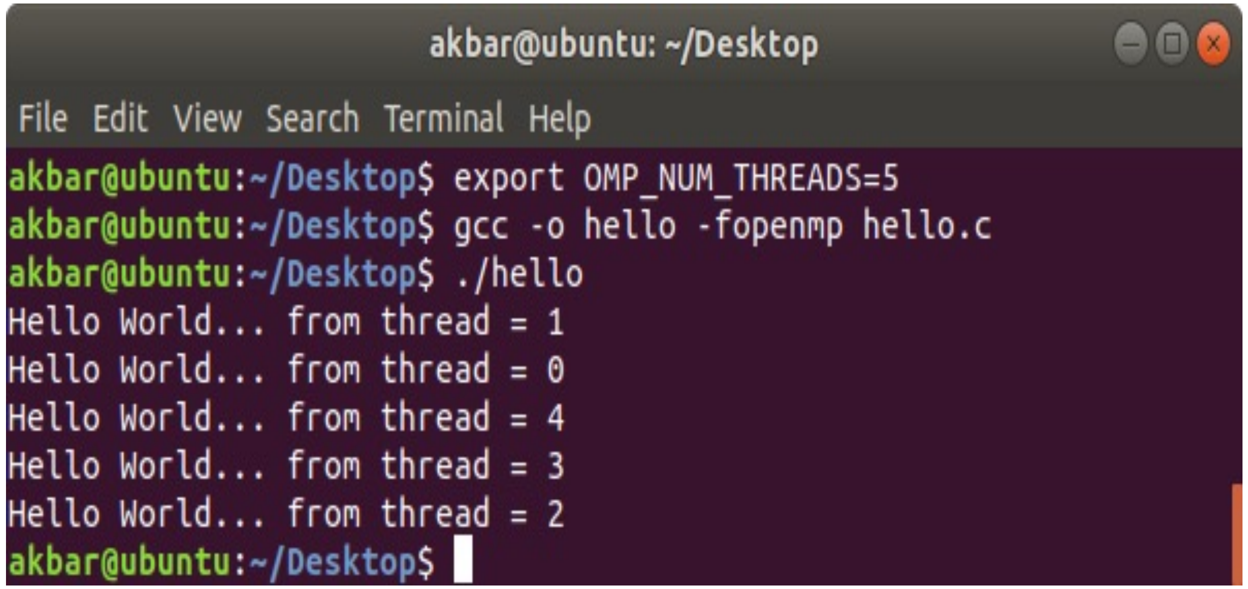
Compile:

```
gcc -o hello -fopenmp hello.c
```

Execute:

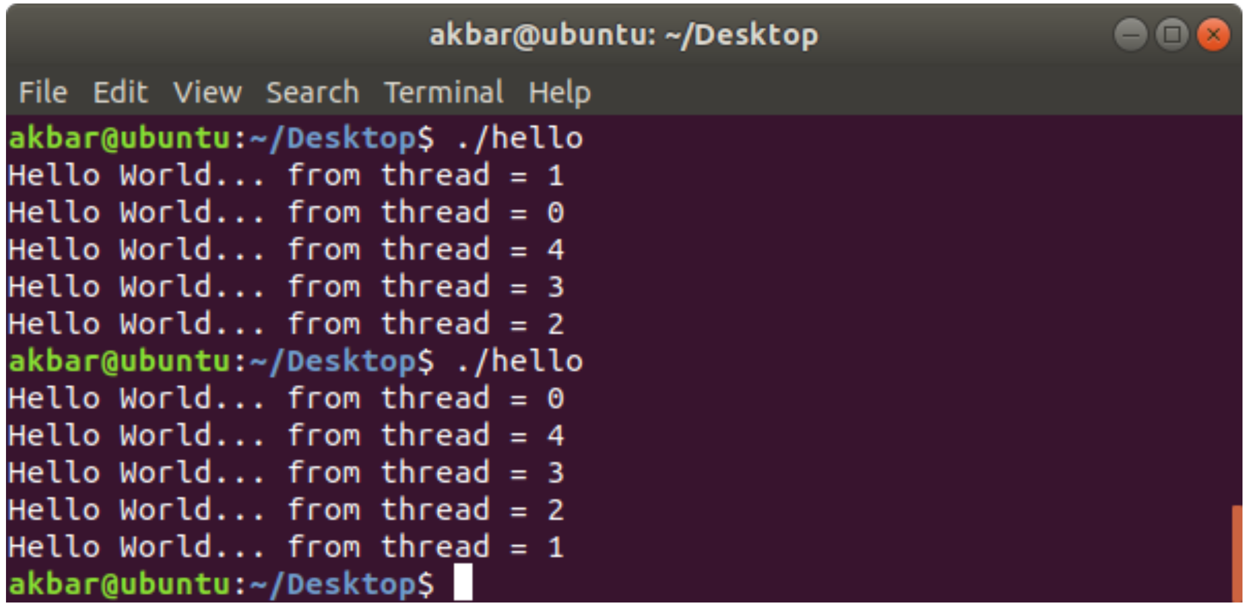
```
./hello
```

When run for 1st time:



```
akbar@ubuntu:~/Desktop$ export OMP_NUM_THREADS=5
akbar@ubuntu:~/Desktop$ gcc -o hello -fopenmp hello.c
akbar@ubuntu:~/Desktop$ ./hello
Hello World... from thread = 1
Hello World... from thread = 0
Hello World... from thread = 4
Hello World... from thread = 3
Hello World... from thread = 2
akbar@ubuntu:~/Desktop$
```

When run for multiple time: Order of execution of threads changes every time.



```
akbar@ubuntu:~/Desktop$ ./hello
Hello World... from thread = 1
Hello World... from thread = 0
Hello World... from thread = 4
Hello World... from thread = 3
Hello World... from thread = 2
akbar@ubuntu:~/Desktop$ ./hello
Hello World... from thread = 0
Hello World... from thread = 4
Hello World... from thread = 3
Hello World... from thread = 2
Hello World... from thread = 1
akbar@ubuntu:~/Desktop$
```

Thanks & Cheers!!

Small aim is a crime; have great aim.

Bharat-Ratan A. P. J. Abdul Kalam