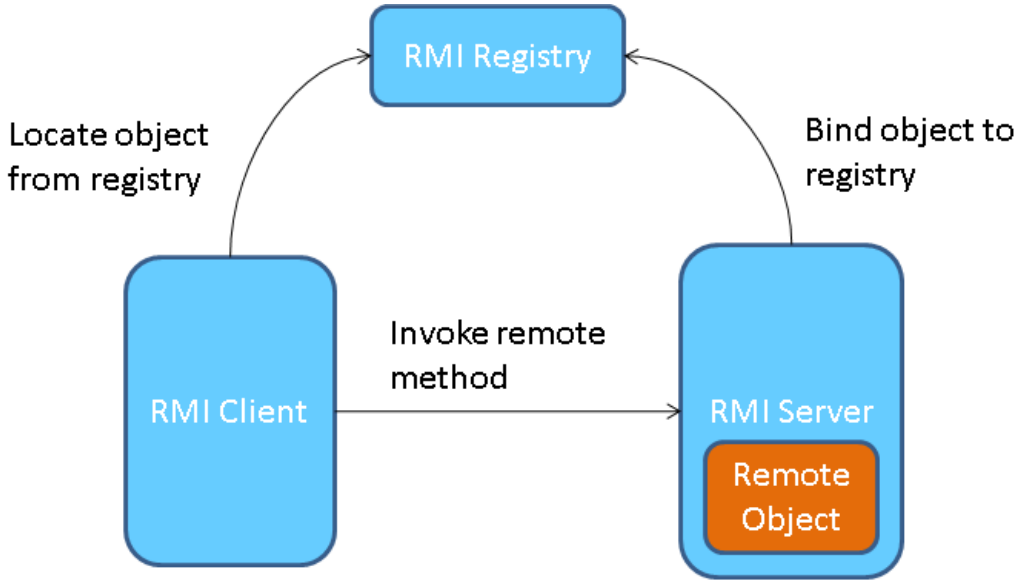
Remote Method Invocation (RMI) and File Handling

What is RMI?

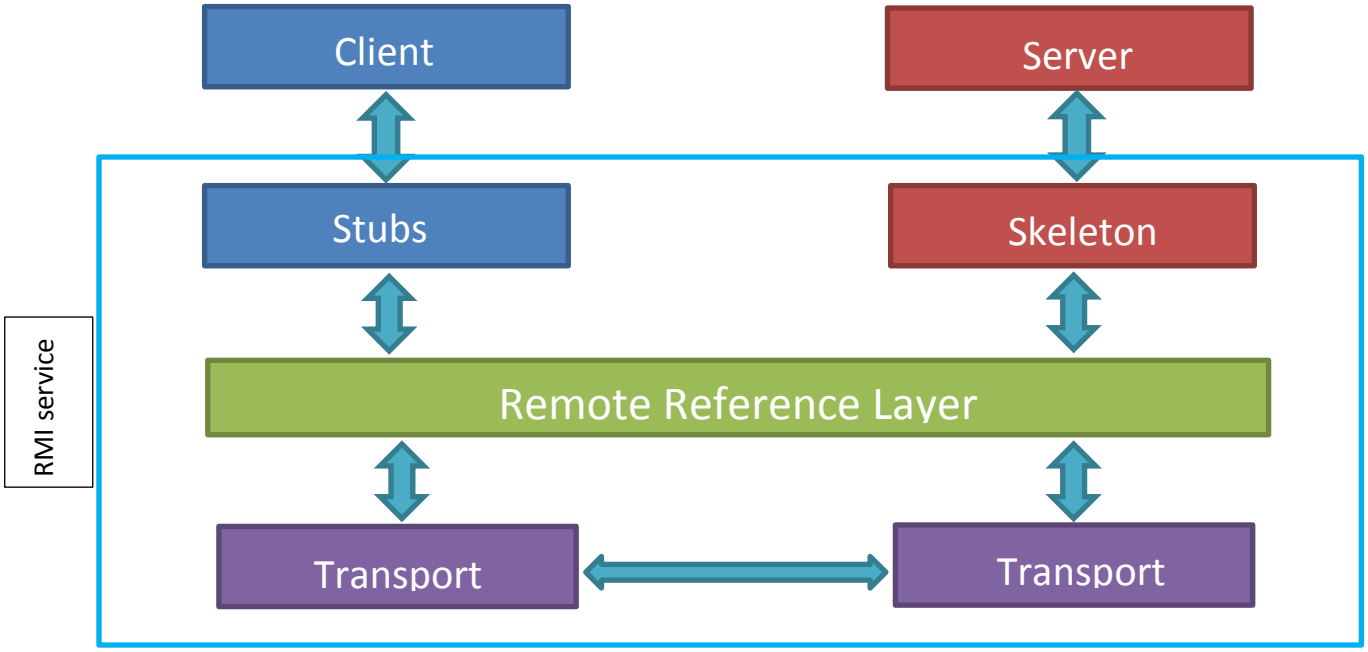
- The **RMI** (Remote Method Invocation) is an API that provides a mechanism to create distributed applications in java.
- The RMI allows an object to **invoke methods** on an object running in another JVM.

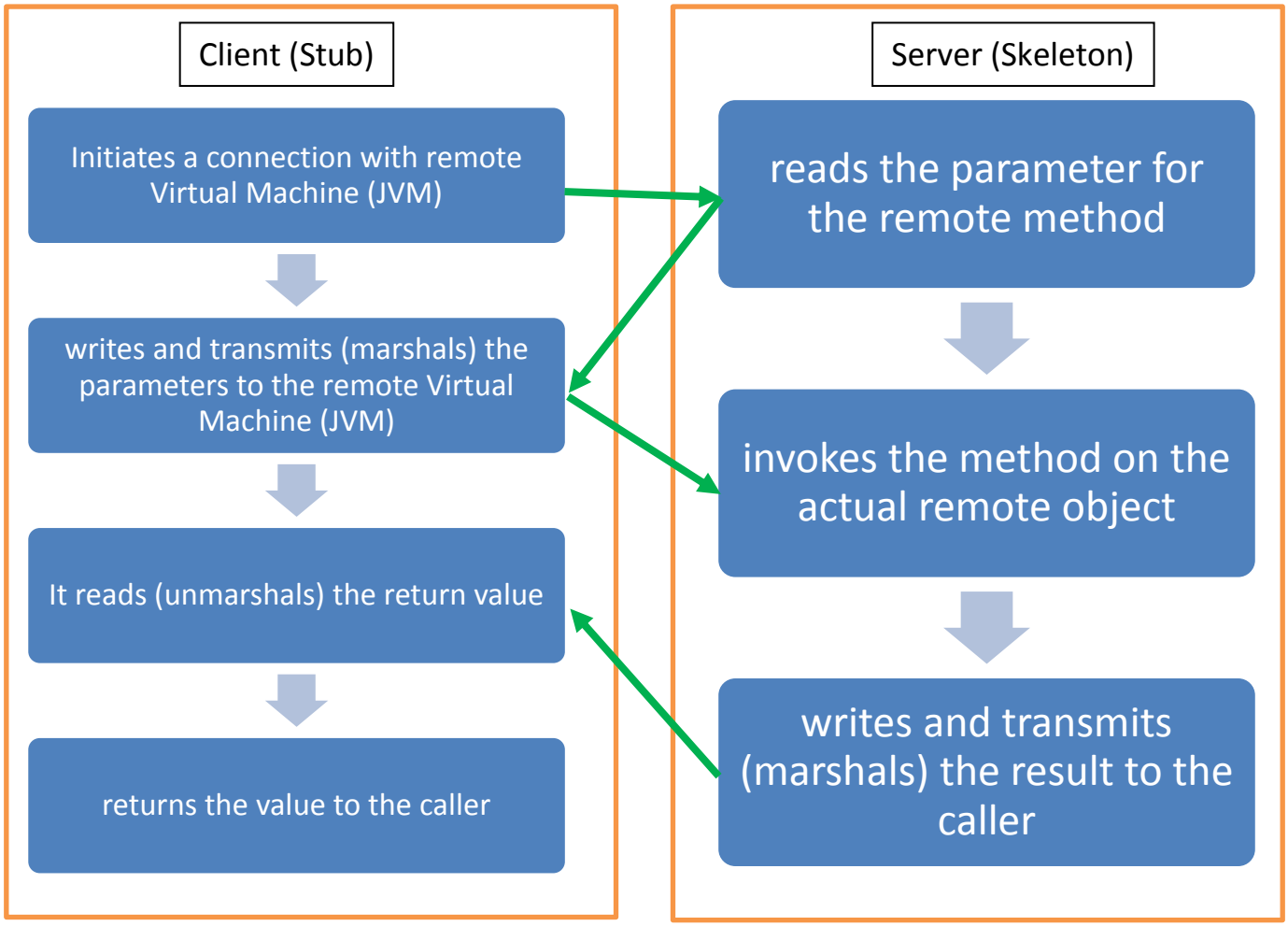


- Under RMI, the networking details required by explicit programming of streams and sockets disappear and the fact that an object is located remotely is almost transparent to the Java programmer.

The Basic RMI Process

- RMI uses **stub** and **skeleton** object for communication with the remote object.
- The **stub** is an object, acting as a gateway for the client side. All the outgoing requests are routed through it. It resides on the client side and represents the remote object.
- The **skeleton** is an object, acts as a gateway for the server-side object. All the incoming requests are routed through it.


Client
Server
Stubs
Skeleton
RMI service
Remote Reference Layer
Transport
Transport


Client (Stub)
Initiates a connection with remote Virtual Machine (JVM)
writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM)
It reads (unmarshals) the return value
returns the value to the caller
Server (Skeleton)
reads the parameter for the remote method
invokes the method on the actual remote object
writes and transmits (marshals) the result to the caller

Implementation Details

Step 1: define the remote interface

- The first thing to do is to create an interface that will provide the description of the methods that can be invoked by remote clients.

```
import java.rmi.*;
public interface Hello extends Remote
{
    public String getGreeting() throws RemoteException;

}
```

Step 2: Implementing the remote interface

- The next step is to implement the remote interface.
- To implement the remote interface, the class should extend to $UnicastRemoteObject$ class of $java.rmi\ package$.
- Also, a default constructor needs to be created to throw the $java.rmi.RemoteException$ from its parent constructor in class.

```
import java.rmi.*;
import java.rmi.server.*;

public class HelloImpl extends UnicastRemoteObject implements Hello
{
    public HelloImpl() throws RemoteException {
    }

    public String getGreeting() throws RemoteException {
        return ("Hello there!");
    }
}
```

Step 3: Create the server process.

- The server creates object(s) of the above implementation class and registers them with a naming service called the $registry$.
- It does this by using static method $rebind$ of class $Naming$.
- The $rebind$ method takes two arguments:
  - a $String$ that holds the name of the remote object as a URL with protocol rmi;
  - a reference to the remote object.
- The $rebind$ method establishes a connection between the object's name and its reference.

```
import java.rmi.*;

public class HelloServer {
    private static final String HOST = "localhost";
    public static void main(String[] args) throws Exception {

        HelloImpl temp = new HelloImpl();
        String rmiObjectName = "rmi://" + HOST + "/Hello";
        Naming.rebind(rmiObjectName, temp);
        System.out.println("Binding complete…\n");
    }
}
```

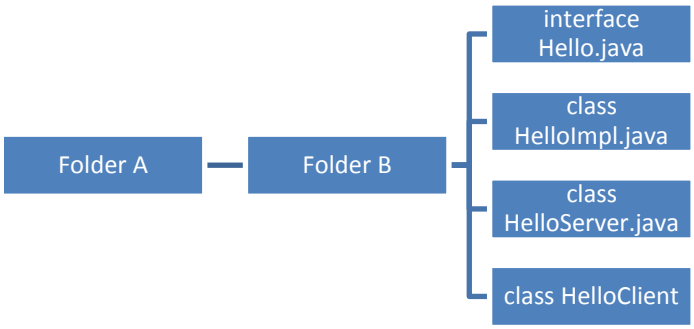
Step 4: Create the client process.

- The client obtains a reference to the remote object from the registry.
- It does this by using method $lookup$ of class $Naming$.
- The $lookup$ method takes same URL that the server did when binding the object reference to the object's name in the registry.

```java
import java.rmi.*;

public class HelloClient {
    private static final String HOST = "localhost";

    public static void main(String[] args) {
        try {
      Hello greeting = (Hello) Naming.lookup("rmi://" + HOST + "/Hello");
      System.out.println("Message received: " + greeting.getGreeting());

        } catch (ConnectException conEx) {
            System.out.println("Unable to connect to server!");
            System.exit(1);
        } catch (Exception ex) {
            ex.printStackTrace();
            System.exit(1);
        }
    }
}
```

How to run?

Suppose your files are organized as follows:



1. Open the folder containing the java files, which is Folder B.
2. Open terminal or CMD, compile the files using the command:

$$javac * .java$$

3. Go to the parent directory of the directory that contains the java files, which is Folder A.
4. Start the RMI service:

$$rmiregistry$$

5. Open another terminal window, and run the $HelloServer$ file:
$$java < package\ name >.HelloServer$$
6. Open another terminal window, and run the $HelloClient$ file:
$$java\ < package\ name >.HelloClient$$

Notice: Windows firewall or antiviruses may block the connections and the application raises exceptions.

Exercise

- Modify the application so the remote server provides methods for computing addition, subtraction, multiplication, and division of two integers.

File Handling

- Java provides two ways to handle files
  - Serial access
  - Random access

Serial Access Files

- Serial access is where data records are stored one after the other with no regard to the order.
  - each new item of data being added to the end of the file
- The internal structure of a serial file can be either **binary** or **text**.
- In java, we use the $File$ class for reading and writing files to the hard disk.
  - Class $File$ is contained within package $java.io$
- Examples:

$$File\ results = new\ File("c:\\data\\results.txt");$$

- To read a file, we need to wrap a $File$ object around a $Scanner$ object.
  - We use the $next, nextLine$, $nextInt$, $nextFloat$, … for reading.

$$Scanner\ input\ =\ new\ Scanner(new\ File("inFile.txt"));$$

- To write to a file, we wrap a $File$ object around a $PrintWriter$ object.
  - We use $print$ and $println$ for writing.

$$PrintWriter\ output$$
$$=\ new\ PrintWriter(new\ File("outFile.txt"));$$

- Example:

```
String file_name = "F:\\fci\\Network
programming\\Java Workspace\\src\\CH04\\data.txt";
File myfile = new File(file_name);
PrintWritear writer = new PrintWriter(myfile);
writer.println("Welcome");
writer.println(5);
writer.println(14/7);
writer.println(true);
writer.close();
Scanner reader = new Scanner(myfile);
String x1 = reader.next();
int x2 = reader.nextInt();
float x3 = reader.nextFloat();
boolean x4 = reader.nextBoolean();
System.out.println(x1);
System.out.println(x2);
System.out.println(x3);
System.out.println(x4);
```

Quiz: What if we change the text "$Weclome$" to "$Welcome\ to\ Network\ Programming$"?

- It is very important to close the file after writing to ensure that the file buffer has been emptied and all data written to the file.
- If we open an existing file and write again to it, the data of the file will be overwritten.

```
File file = new File("F:\\fci\\Network
programming\\Java Workspace\\src\\CH04\\data.txt");
PrintWriter writer = new PrintWriter(file);
writer.println("This is a new Text");
writer.close();
```

  - How to solve this problem?

- To append data to an existing file, we can use the $FileWriter$ object.

$$FileWriter(String < fileName >, boolean < append >)$$
$$FileWriter(File < fileName >, boolean < append >)$$

- Example

```
FileWriter file = new FileWriter("F:\\fci\\Network
programming"\\Java Workspace\\src\\CH04\\data.txt",
true);
file.write("This is another text!");
file.close();
```

Command Line Parameters

- To run a java program using cmd
  - Add java bin folder to the path
  - Compile the file using $javac\ filename.java$
  - Execute the file using $java\ filename.class$
- It is possible to supply values (command line parameters) in addition to the name of the program to be executed.
  - The parameters are stored in the array $args$.

- Example: an application that copies the data of one file (the first parameter) to a new file (the next parameter).

```
public static void main(String[] args) throws
FileNotFoundException {
    if (args.length < 2) {
        System.out.println(
                "You must supply TWO file names.");
        System.out.println("Syntax:");
        System.out.println(
                " java Copy <source> <destination>");
        return;
    }
    Scanner source = new Scanner(new File(args[0]));
    PrintWriter destination =
            new PrintWriter(new File(args[1]));
    String input;
    while (source.hasNext()) {
        input = source.nextLine();
        destination.println(input);
    }
    source.close();
    destination.close();
}
```

  - Run the terminal
  - Compile the file: $javac\ Copy.java$
  - Run the file: $java\ CH04.Copy\ CH04/data.txt\ CH04/new.txt$