

**import java.rmi.\*;**

**import java.util.ArrayList;**

**import java.util.Iterator;**

**import java.util.LinkedHashMap;**

**import java.util.List;**

**import java.util.Map;**

**import java.util.Map.Entry;**

**import java.util.Scanner;**

**import java.util.TreeMap;**

**public interface MyCalendar extends Remote**

**{**

**String EchoMessage() throws RemoteException;**

**public void setUserName(String name) throws RemoteException;**

**public String getUserName() throws RemoteException;**

**public boolean calendarExist(String userName) throws RemoteException;**

**public boolean createCalendar(String userName) throws RemoteException;**

**public boolean addEvent(String timeInterval, String eventDescription, String accessControl)throws RemoteException;**

**public void viewCalendar(String userName)throws RemoteException;**

**public List<String> deleteEvent(int eventNumber) throws RemoteException;**

**public EchoImpl createAnotherCalendar(String userName)throws RemoteException;**

**public boolean isOwner(String userName, int calendarNumber) throws RemoteException;**

**public void viewAllCalendarsHelper(Map<String,List<String>> map) throws RemoteException;**

**public void viewAnyCalendar(String userName, int index)throws RemoteException;**

**public List<String> modifyEvent(int eventNumber)throws RemoteException;**

**public void viewAllCalendars()throws RemoteException;**

**}**

**RMI: Creating Distributed Applications**

**General Steps**

1. Designing and implementing the components of distributed application.

2. Compiling sources.

3. Making classes network accessible.

4. Starting the application

1. Designing and implementing the components of distributed application.
   1. Defining the remote interfaces. A remote interface specifies the methods that can be invoked remotely by a client. Clients program to remote interfaces, not to the implementation classes of those interfaces. The design of such interfaces includes the determination of the types of objects that will be used as the parameters and return values for these methods. If any of these interfaces or classes do not yet exist, you need to define them as well.
   2. Implementing the remote objects. Remote objects must implement one or more remote interfaces. The remote object class may include implementations of other interfaces and methods that are available only locally. If any local classes are to be used for parameters or return values of any of these methods, they must be implemented as well.
   3. Implementing the clients. Clients that use remote objects can be implemented at any time after the remote interfaces are defined, including after the remote objects have been deployed.

**Designing and implementing the components of distributed application**

Separates the interfaces, remote object implementation, and client code into three packages:

* **SERVER**
  + INTERFACE
    - declaration – HandleTask and Task interfaces
    - extend Remote class
  + CLASSES
    - Implement remote objects
    - driver – ComputeEngine implementation class
* **CLIENT**
  + client – CalenderHandler client code and Calender task implementation

**Compiling sources &**

**Making classes network accessible**

**NOTE:** create source scripts for compiling and running the code

**Building the Interface**

First, we build the interface **JAR file** to be used for **server** and **client** development.

**Linux**:

cd /home/ainguane/src

javac compute/Compute.java compute/Task.java

jar cvf compute.jar compute/\*.class

Place the compute.jar in network-accessible location ~/ainguane/public\_html/classes

Now, the compute.jar file can be used to develop server and client applications.

**Building the Server Classes**

The ComputeEngine class depends on the Compute and Task interfaces, which are contained in the compute.jar JAR file. Therefore, you need the compute.jar file in your class path when you build the server classes.

**Linux**:

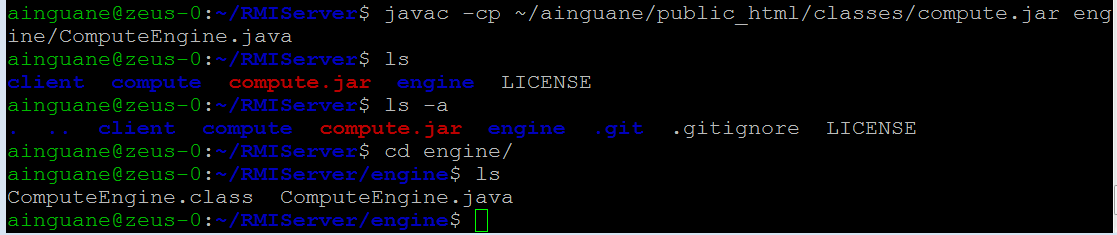
Location of interface jar file: /home/ainguane/public\_html/classes

cd ~/ainguane/src

javac -cp ~/ainguane/public\_html/classes/compute.jar

engine/ComputeEngine.java

**result in**



The stub class for ComputeEngine implements the Compute interface, which refers to the Task interface. So, the class definitions for those two interfaces need to be network-accessible for the stub to be received by other Java virtual machines such as the registry's Java virtual machine. The client Java virtual machine will already have these interfaces in its class path, so it does not actually need to download their definitions. The compute.jar file under the public\_html directory can serve this purpose.

Now, the compute engine is ready to deploy. You could do that now, or you could wait until after you have built the client.

**Building the Server Classes**

cd ~/ainguane/src

javac –cp ~/ainguane/public\_html/classes/compute.jar

client/ComputePi.java client/Pi.java

mkdir ~/ainguane/public\_html/classes/client

cp client/Pi.class ~/ainguane/public\_html/classes/client

Only the Pi class needs to be placed in the directory public\_html\classes\client because only the Pi class needs to be available for downloading to the compute engine's Java virtual machine. Now, you can run the server and then the client.

SECURITY : <https://docs.oracle.com/javase/tutorial/security/tour1/wstep1.html>

**client.policy**

grant codeBase "file:/home/ainguane/src/" {

permission java.security.AllPermission;

};

**server.policy**

grant codeBase "file:/ainguane/src/" {

permission java.security.AllPermission;

};

**Starting the application**

Before starting the compute engine, you need to start the RMI registry. The RMI registry is a simple server-side bootstrap naming facility that enables remote clients to obtain a reference to an initial remote object. It can be started with the rmiregistry command. Before you execute rmiregistry, you must make sure that the shell or window in which you will run rmiregistry either has no CLASSPATH environment variable set or has a CLASSPATH environment variable that does not include the path to any classes that you want downloaded to clients of your remote objects.

To start the registry on the server, execute the rmiregistry command. This command produces no output and is typically run in the background. For this example, the registry is started on the host mycomputer.

Linux:

rmiregistry &

By default, the registry runs on port 1099. To start the registry on a different port, specify the port number on the command line. Do not forget to unset your CLASSPATH environment variable.

Linux:

rmiregistry 2001 &

Once the registry is started, you can start the server. You need to make sure that both the compute.jar file and the remote object implementation class are in your class path. When you start the compute engine, you need to specify, using the java.rmi.server.codebase property, where the server's classes are network accessible. In this example, the server-side classes to be made available for downloading are the Compute and Task interfaces, which are available in the compute.jar file in the public\_html\classes directory of user ann. The compute engine server is started on the host mycomputer, the same host on which the registry was started.

Running on eclipse:

<http://www.ejbtutorial.com/java-rmi/a-step-by-step-implementation-tutorial-for-java-rmi>

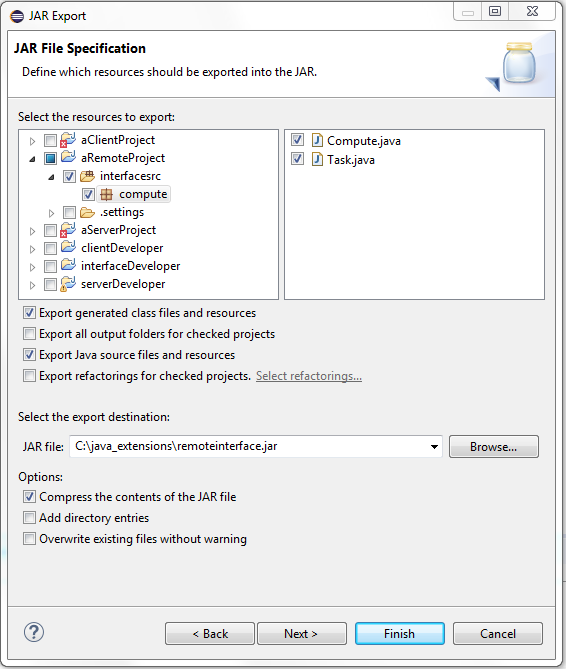
<https://www.youtube.com/watch?v=GURClZeR96E>

JAVA API

<https://docs.oracle.com/javase/tutorial/collections/implementations/queue.html>

CREATE A JAR file

<https://www.cs.utexas.edu/~scottm/cs307/handouts/Eclipse%20Help/jarInEclipse.htm>



**Q&A**

After you build either server-side or client-side classes with the javac compiler, if any of those classes *will need to be dynamically downloaded by other Java virtual machines*, you must ensure that their class files are placed in a network-accessible location.

Linux this location is /home/user/public\_html/classes because many web servers allow the accessing of a user's public\_html directory through an HTTP URL constructed as <http://host/~user/>.

If your web server does not support this convention, you could use a different location in the web server's hierarchy, or you could use a file URL instead. The file URLs take the form file:/home/user/public\_html/classes/ on Solaris OS or Linux

**Linux**:

Location: /home/ainguane/src/engine

Deploying the class files for **clients** to download in a subdirectory of public\_html directory

cd /home/ainguane/public\_html/classes on Solaris OS or Linux. This location is accessible through some web servers as http://*host*:*port*/~ainguane/classes/.