<http://docs.oracle.com/javase/tutorial/rmi/overview.html>

RMI application is sometimes referred to as a *distributed object application*.

Distributed object applications need to do the following:

* **Locate remote objects.** Applications can use various mechanisms to obtain references to remote objects. For example, an application can register its remote objects with RMI's simple naming facility, the RMI registry. Alternatively, an application can pass and return remote object references as part of other remote invocations.
* **Communicate with remote objects.** Details of communication between remote objects are handled by RMI. To the programmer, remote communication looks similar to regular Java method invocations.
* **Load class definitions for objects that are passed around.** Because RMI enables objects to be passed back and forth, it provides mechanisms for loading an object's class definitions as well as for transmitting an object's data.

The following illustration depicts an RMI distributed application that uses the RMI registry to obtain a reference to a remote object. The server calls the registry to associate (or bind) a name with a remote object. The client looks up the remote object by its name in the server's registry and then invokes a method on it. The illustration also shows that the RMI system uses an existing web server to load class definitions, from server to client and from client to server, for objects when needed.

**Advantages of Dynamic Code Loading**

One of the central and unique features of RMI is its ability to download the definition of an object's class if the class is not defined in the receiver's Java virtual machine. All of the types and behavior of an object, previously available only in a single Java virtual machine, can be transmitted to another, possibly remote, Java virtual machine. RMI passes objects by their actual classes, so the behavior of the objects is not changed when they are sent to another Java virtual machine.

an RMI application has the ability to download code dynamically, is often called a *behavior-based application*. Such applications usually require full agent-enabled infrastructures. With RMI, such applications are part of the basic mechanisms for distributed computing on the Java platform.

## Remote Interfaces, Objects, and Methods

a distributed application built by using Java RMI is made up of interfaces and classes. The interfaces declare methods. The classes implement the methods declared in the interfaces and, perhaps, declare additional methods as well. In a distributed application, some implementations might reside in some Java virtual machines but not others. Objects with methods that can be invoked across Java virtual machines are called *remote objects*.

An object becomes remote by implementing a *remote interface*, which has the following characteristics:

* A remote interface extends the interface java.rmi.Remote.
* Each method of the interface declares java.rmi.RemoteException in its throws clause, in addition to any application-specific exceptions.

RMI treats a remote object differently from a non-remote object when the object is passed from one Java virtual machine to another Java virtual machine. Rather than making a copy of the implementation object in the receiving Java virtual machine, RMI passes a remote *stub* for a remote object. The stub acts as the local representative, or proxy, for the remote object and basically is, to the client, the remote reference. The client invokes a method on the local stub, which is responsible for carrying out the method invocation on the remote object.

A stub for a remote object implements the same set of remote interfaces that the remote object implements. This property enables a stub to be cast to any of the interfaces that the remote object implements. However, *only* those methods defined in a remote interface are available to be called from the receiving Java virtual machine.

**Creating Distributed Applications by Using RMI**

Using RMI to develop a distributed application involves these general steps:

1. Designing and implementing the components of your distributed application.
2. Compiling sources.
3. Making classes network accessible.
4. Starting the application.

### Designing and Implementing the Application Components

First, determine your application architecture, including which components are local objects and which components are remotely accessible. This step includes:

* **Defining the remote interfaces.**
* **Implementing the remote objects.**
* **Implementing the clients.**

### Compiling Sources

As with any Java program, you use the javac compiler to compile the source files. The source files contain the declarations of the remote interfaces, their implementations, any other server classes, and the client classes.

**Note:** With versions prior to Java Platform, Standard Edition 5.0, an additional step was required to build stub classes, by using the rmic compiler. However, this step is no longer necessary.

### Making Classes Network Accessible

In this step, you make certain class definitions network accessible, such as the definitions for the remote interfaces and their associated types, and the definitions for classes that need to be downloaded to the clients or servers. Classes definitions are typically made network accessible through a web server.

### Starting the Application

Starting the application includes running the RMI remote object registry, the server, and the client.

## Passing Objects in RMI

Arguments to or return values from remote methods can be of almost any type, including local objects, remote objects, and primitive data types. More precisely, any entity of any type can be passed to or from a remote method as long as the entity is an instance of a type that is a primitive data type, a remote object, or a *serializable* object, which means that it implements the interface java.io.Serializable.

The rules governing how arguments and return values are passed are as follows:

* Remote objects are essentially passed by reference. A *remote object reference* is a stub, which is a client-side proxy that implements the complete set of remote interfaces that the remote object implements.
* Local objects are passed by copy, using object serialization. By default, all fields are copied except fields that are marked static or transient. Default serialization behavior can be overridden on a class-by-class basis.

Passing a remote object by reference means that any changes made to the state of the object by remote method invocations are reflected in the original remote object. When a remote object is passed, only those interfaces that are remote interfaces are available to the receiver. Any methods defined in the implementation class or defined in non-remote interfaces implemented by the class are not available to that receiver.

For example, if you were to pass a reference to an instance of the ComputeEngine class, the receiver would have access only to the compute engine'sexecuteTask method. That receiver would not see the ComputeEngine constructor, its main method, or its implementation of any methods ofjava.lang.Object.

In the parameters and return values of remote method invocations, objects that are not remote objects are passed by value. Thus, a copy of the object is created in the receiving Java virtual machine. Any changes to the object's state by the receiver are reflected only in the receiver's copy, not in the sender's original instance. Any changes to the object's state by the sender are reflected only in the sender's original instance, not in the receiver's copy.

## Implementing the Server's main Method

The main method is used to start the ComputeEngine and therefore needs to do the necessary initialization and housekeeping to prepare the server to accept calls from clients. This method is not a remote method, which means that it cannot be invoked from a different Java virtual machine. Because the main method is declared static, the method is not associated with an object at all but rather with the class ComputeEngine.

## Creating and Installing a Security Manager

The main method's first task is to create and install a security manager, which protects access to system resources from untrusted downloaded code running within the Java virtual machine. A security manager determines whether downloaded code has access to the local file system or can perform any other privileged operations.

If an RMI program does not install a security manager, RMI will not download classes (other than from the local class path) for objects received as arguments or return values of remote method invocations. This restriction ensures that the operations performed by downloaded code are subject to a security policy.

Here's the code that creates and installs a security manager:

if (System.getSecurityManager() == null) { System.setSecurityManager(new SecurityManager()); }

## Making the Remote Object Available to Clients

Next, the main method creates an instance of ComputeEngine and exports it to the RMI runtime with the following statements:

Compute engine = new ComputeEngine(); Compute stub = (Compute) UnicastRemoteObject.exportObject(engine, 0);

The static UnicastRemoteObject.exportObject method exports the supplied remote object so that it can receive invocations of its remote methods from remote clients. The second argument, an int, specifies which TCP port to use to listen for incoming remote invocation requests for the object. It is common to use the value zero, which specifies the use of an anonymous port. The actual port will then be chosen at runtime by RMI or the underlying operating system. However, a non-zero value can also be used to specify a specific port to use for listening. Once the exportObject invocation has returned successfully, the ComputeEngine remote object is ready to process incoming remote invocations.

The exportObject method returns a stub for the exported remote object. Note that the type of the variable stub must be Compute, not ComputeEngine, because the stub for a remote object only implements the remote interfaces that the exported remote object implements.

The exportObject method declares that it can throw a RemoteException, which is a checked exception type. The main method handles this exception with its try/catch block. If the exception were not handled in this way, RemoteException would have to be declared in the throws clause of the mainmethod. An attempt to export a remote object can throw a RemoteException if the necessary communication resources are not available, such as if the requested port is bound for some other purpose.

Before a client can invoke a method on a remote object, it must first obtain a reference to the remote object. Obtaining a reference can be done in the same way that any other object reference is obtained in a program, such as by getting the reference as part of the return value of a method or as part of a data structure that contains such a reference.

The system provides a particular type of remote object, the RMI registry, for finding references to other remote objects. The RMI registry is a simple remote object naming service that enables clients to obtain a reference to a remote object by name. The registry is typically only used to locate the first remote object that an RMI client needs to use. That first remote object might then provide support for finding other objects.

The java.rmi.registry.Registry remote interface is the API for binding (or registering) and looking up remote objects in the registry. Thejava.rmi.registry.LocateRegistry class provides static methods for synthesizing a remote reference to a registry at a particular network address (host and port). These methods create the remote reference object containing the specified network address without performing any remote communication.LocateRegistry also provides static methods for creating a new registry in the current Java virtual machine, although this example does not use those methods. Once a remote object is registered with an RMI registry on the local host, clients on any host can look up the remote object by name, obtain its reference, and then invoke remote methods on the object. The registry can be shared by all servers running on a host, or an individual server process can create and use its own registry.

The ComputeEngine class creates a name for the object with the following statement:

String name = "Compute";

The code then adds the name to the RMI registry running on the server. This step is done later with the following statements:

Registry registry = LocateRegistry.getRegistry(); registry.rebind(name, stub);

This rebind invocation makes a remote call to the RMI registry on the local host. Like any remote call, this call can result in a RemoteException being thrown, which is handled by the catch block at the end of the main method.

Note the following about the Registry.rebind invocation:

* The no-argument overload of LocateRegistry.getRegistry synthesizes a reference to a registry on the local host and on the default registry port, 1099. You must use an overload that has an int parameter if the registry is created on a port other than 1099.
* When a remote invocation on the registry is made, a stub for the remote object is passed instead of a copy of the remote object itself. Remote implementation objects, such as instances of ComputeEngine, never leave the Java virtual machine in which they were created. Thus, when a client performs a lookup in a server's remote object registry, a copy of the stub is returned. Remote objects in such cases are thus effectively passed by (remote) reference rather than by value.
* For security reasons, an application can only bind, unbind, or rebind remote object references with a registry running on the same host. This restriction prevents a remote client from removing or overwriting any of the entries in a server's registry. A lookup, however, can be requested from any host, local or remote.

Once the server has registered with the local RMI registry, it prints a message indicating that it is ready to start handling calls. Then, the main method completes. It is not necessary to have a thread wait to keep the server alive. As long as there is a reference to the ComputeEngine object in another Java virtual machine, local or remote, the ComputeEngine object will not be shut down or garbage collected. Because the program binds a reference to theComputeEngine in the registry, it is reachable from a remote client, the registry itself. The RMI system keeps the ComputeEngine's process running. TheComputeEngine is available to accept calls and won't be reclaimed until its binding is removed from the registry *and* no remote clients hold a remote reference to the ComputeEngine object.

The final piece of code in the ComputeEngine.main method handles any exception that might arise. The only checked exception type that could be thrown in the code is RemoteException, either by the UnicastRemoteObject.exportObject invocation or by the registry rebind invocation. In either case, the program cannot do much more than exit after printing an error message. In some distributed applications, recovering from the failure to make a remote invocation is possible. For example, the application could attempt to retry the operation or choose another server to continue the operation.