Java Remote Method Invocation (RMI). Java RMI is a mechanism that allows one to invoke a method on an object that exists in another address space. The other address space could be on the same machine or a different one. The RMI mechanism is basically an object-oriented RPC mechanism.

There are three processes that participate in supporting remote method invocation.

1. The *Client* is the process that is invoking a method on a remote object.
2. The *Server* is the process that owns the remote object. The remote object is an ordinary object in the address space of the server process.
3. The *Object Registry* is a name server that relates objects with names. Objects are *registered* with the Object Registry. Once an object has been registered, one can use the Object Registry to obtain access to a remote object using the name of the object.

There are two kinds of classes that can be used in Java RMI.

1. A *Remote* class is one whose instances can be used remotely. An object of such a class can be referenced in two different ways:
   1. Within the address space where the object was constructed, the object is an ordinary object which can be used like any other object.
   2. Within other address spaces, the object can be referenced using an *object handle*. While there are limitations on how one can use an object handle compared to an object, for the most part one can use object handles in the same way as an ordinary object.

For simplicity, an instance of a Remote class will be called a *remote object*.

1. A *Serializable* class is one whose instances can be copied from one address space to another. An instance of a Serializable class will be called a *serializable object*. In other words, a serializable object is one that can be marshaled. Note that this concept has no connection to the concept of serializability in database management systems.

If a serializable object is passed as a parameter (or return value) of a remote method invocation, then the value of the object will be copied from one address space to the other. By contrast if a remote object is passed as a parameter (or return value), then the object handle will be copied from one address space to the other.

One might naturally wonder what would happen if a class were both Remote and Serializable. While this might be possible in theory, it is a poor design to mix these two notions as it makes the design difficult to understand.

**Serializable Classes**

We now consider how to design Remote and Serializable classes. The easier of the two is a Serializable class. A class is Serializable if it implements the java.io.Serializable interface. Subclasses of a Serializable class are also Serializable. Many of the standard classes are Serializable, so a subclass of one of these is automatically also Serializable. Normally, any data within a Serializable class should also be Serializable. Although there are ways to include non-serializable objects within a serializable objects, it is awkward to do so. See the documentation of java.io.Serializable for more information about this.

Using a serializable object in a remote method invocation is straightforward. One simply passes the object using a parameter or as the return value. The type of the parameter or return value is the Serializable class. Note that both the Client and Server programs must have access to the definition of any Serializable class that is being used.

**Remote Classes and Interfaces**

Next consider how to define a Remote class. This is more difficult than defining a Serializable class. A Remote class has two parts: the interface and the class itself. The Remote interface must have the following properties:

1. The interface must be public.
2. The interface must extend the interface java.rmi.Remote.
3. Every method in the interface must declare that it throws java.rmi.RemoteException. Other exceptions may also be thrown.

The Remote class itself has the following properties:

1. It must implement a Remote interface.
2. It should extend the java.rmi.server.UnicastRemoteObject class. Objects of such a class exist in the address space of the server and can be invoked remotely. While there are other ways to define a Remote class, this is the simplest way to ensure that objects of a class can be used as remote objects. See the documentation of the java.rmi.server package for more information.
3. It can have methods that are not in its Remote interface. These can only be invoked locally.

Unlike the case of a Serializable class, it is not necessary for both the Client and the Server to have access to the definition of the Remote class. The Server requires the definition of both the Remote class and the Remote interface, but the Client only uses the Remote interface. Roughly speaking, the Remote interface represents the type of an object handle, while the Remote class represents the type of an object. If a remote object is being used remotely, its type must be declared to be the type of the Remote interface, not the type of the Remote class.

**Security:**

Note that both the Client and Server programs must have access to the definition of any Serializable class that is being used. If the Client and Server programs are on different machines, then class definitions of Serializable classes may have to be downloaded from one machine to the other. Such a download could violate system security. This problem is discussed in the [Security](http://www.eg.bucknell.edu/~cs379/DistributedSystems/rmi_tut.html#secure) section.