

UNIT 12

Remote Method Invocation (RMI)

(2 Hours) — Network Programming (CAC355)

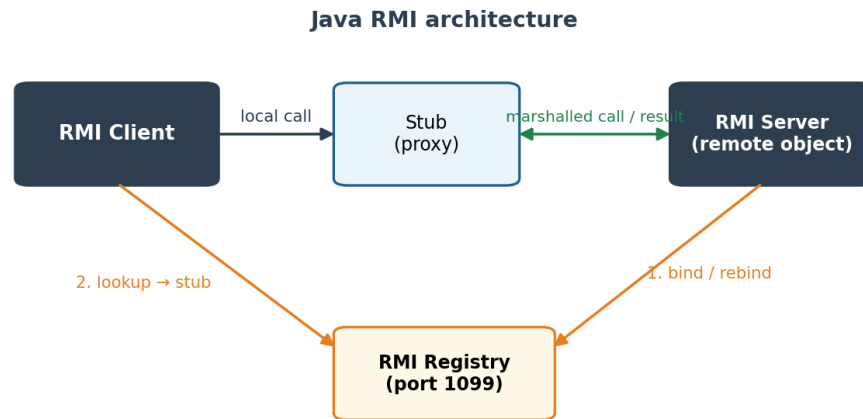
Please note

These notes may contain errors. They were generated automatically, so please verify important facts, API details and code against your textbook, the official Java documentation and your teacher's material before relying on them (especially for exams).

12.0 Introduction to RMI

Java RMI (Remote Method Invocation) lets an object running in one JVM call methods on an object in another JVM (possibly on another machine) almost as if it were local. It is Java's object-oriented alternative to RPC (Remote Procedure Call): the programmer works with interfaces and objects, and RMI takes care of the socket communication, **marshalling** (serialising) arguments and return values, and unmarshalling them at the other end.

Concept	Meaning
Remote interface	Interface extending <code>java.rmi.Remote</code> ; declares the methods that can be called remotely.
Remote object	Server-side object implementing the remote interface.
Stub	Client-side proxy object that forwards calls over the network (generated dynamically in modern Java).
RMI Registry	Simple naming service (default port 1099) that maps names to remote object stubs.
Marshalling	Converting parameters/results to a byte stream using Java serialization.



Client and server share the remote interface; the stub hides all network details

Figure 12.1: Server registers the object; client looks it up and calls it through the stub.

Parameter passing rules: primitive and Serializable objects are passed **by value** (copied); remote objects are passed **by reference** (as a stub). Therefore all parameter and return types must be primitives, **Serializable**, or remote interfaces.

12.1 Defining and Implementing the RMI Service Interface

12.1.1 Defining the Remote Interface

Rules for a remote interface:

- It must be **public** and extend **java.rmi.Remote** (a marker interface with no methods).
- Every method must declare **throws java.rmi.RemoteException** (or a superclass such as IOException or Exception), because any network call can fail.
- Parameters and return values must be serializable (or remote).

```
// Calculator.java
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Calculator extends Remote {
    int add(int a, int b) throws RemoteException;
    int multiply(int a, int b) throws RemoteException;
    String greet(String name) throws RemoteException;
}
```

Listing 12.1: The remote interface (shared by client and server)

12.1.2 Implementing the Service

The implementation class implements the interface and must be exported so it can receive calls. This is done either by extending **UnicastRemoteObject** (which exports automatically in its constructor, which then must declare throws **RemoteException**) or by calling **UnicastRemoteObject.exportObject(obj, port)** explicitly (port 0 = any free port).

```
// CalculatorImpl.java
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class CalculatorImpl extends UnicastRemoteObject implements Calculator {

    public CalculatorImpl() throws RemoteException {
        super(); // exports this object
    }
    public int add(int a, int b) throws RemoteException { return a + b; }
    public int multiply(int a, int b) throws RemoteException { return a * b; }
    public String greet(String name) throws RemoteException { return "Hello, " + name + "!"; }
}
```

Listing 12.2: The remote object implementation

*Note: Older Java versions required running the **rmic** compiler to generate stub classes. Since Java 5, stubs are generated dynamically, so you only need **javac** (**rmic** was removed in Java 15). Your textbook may still mention **rmic** and **skeletons**.*

12.2 Creating an RMI Server and Client

12.2.1 The Server

The server creates the remote object and **binds** it under a name in the RMI registry. The registry can be started either externally with the **rmiregistry** command, or inside the server with **LocateRegistry.createRegistry(1099)**.

```
// CalculatorServer.java
import java.rmi.Naming;
import java.rmi.registry.LocateRegistry;

public class CalculatorServer {
    public static void main(String[] args) {
        try {
            LocateRegistry.createRegistry(1099); // start registry in this JVM
            Calculator calc = new CalculatorImpl();
            Naming.rebind("rmi://localhost:1099/CalculatorService", calc);
            System.out.println("Calculator service is ready.");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Listing 12.3: RMI server

Method	Description
Naming.bind(name, obj)	Binds the name; throws AlreadyBoundException if the name is already used.
Naming.rebind(name, obj)	Binds, replacing any existing binding (most commonly used).
Naming.lookup(name)	Returns the stub for a name; throws NotBoundException if missing.
Naming.unbind(name)	Removes a binding.
Naming.list(url)	Array of names registered in the registry.

The name has the form **rmi://host:port/ServiceName** (rmi:// and port are optional; the default port is 1099). For security, bind(), rebind() and unbind() on a registry are only accepted from the *same host* as the registry.

12.2.2 The Client

The client looks up the stub by name, casts it to the remote interface and then calls methods exactly like on a local object, but must handle **RemoteException**.

```
// CalculatorClient.java
import java.rmi.Naming;

public class CalculatorClient {
    public static void main(String[] args) {
        try {
            String host = args.length > 0 ? args[0] : "localhost";
            Calculator calc = (Calculator) Naming.lookup("rmi://" + host + "/CalculatorService");

            System.out.println(calc.greet("Student"));
            System.out.println("5 + 3 = " + calc.add(5, 3));
            System.out.println("5 * 3 = " + calc.multiply(5, 3));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Listing 12.4: RMI client

Alternative with the registry API: *Registry r = LocateRegistry.getRegistry(host, 1099);*
Calculator c = (Calculator) r.lookup("CalculatorService");

Example 12.1 – Passing objects

A remote method is declared as `Result compute(Request r)`. What must be true of `Request` and `Result`?

Solution

Both must implement `java.io.Serializable` (they are copied to the other JVM) or be remote interfaces (passed as stubs). Otherwise a `MarshalException` (caused by `NotSerializableException`) is thrown at call time. Changes made to a serialized argument by the server are **not** visible to the client.

12.3 Running the RMI System

12.3.1 Steps

Step	Action	Command
1	Compile all classes (interface, implementation, server, client)	<code>javac *.java</code>
2	Start the registry (skip if the server calls <code>createRegistry</code>)	<code>rmiregistry 1099</code> (run in the directory containing the class files, or set <code>CLASSPATH</code>)
3	Start the server	<code>java CalculatorServer</code>
4	Run the client (same or another machine, needs <code>Calculator.class</code>)	<code>java CalculatorClient <server-host></code>

Expected client output:

```
Hello, Student!
5 + 3 = 8
5 * 3 = 15
```

12.3.2 Common Problems

Exception / symptom	Likely cause and fix
<code>java.rmi.ConnectException: Connection refused</code>	Registry or server not running, wrong host or port, or a firewall blocking port 1099 / the object's port.
<code>java.rmi.NotBoundException</code>	Name in <code>lookup()</code> does not match the name used in <code>bind()/rebind()</code> .
<code>java.rmi.UnmarshalException</code> / <code>ClassNotFoundException</code>	Client lacks the interface (or a <code>Serializable</code> parameter class) on its classpath.
<code>java.rmi.AccessException</code>	Trying to bind/unbind on a registry running on another host.
Client hangs or connects to 127.0.0.1 on a remote server	Server advertises a wrong IP; start with <code>-Djava.rmi.server.hostname=<server-ip></code> .
<code>NotSerializableException</code>	A parameter or result class does not implement <code>Serializable</code> .

Note: Modern RMI uses the security policy only if a SecurityManager is installed; the SecurityManager is deprecated for removal since Java 17, so recent code normally omits it. Older textbooks add `System.setSecurityManager(new RMISecurityManager())` and a policy file; follow your course's instructions if your lab environment requires it.

12.3.3 RMI Compared with Sockets

Aspect	Socket programming	RMI
Level	Low-level bytes, you design the protocol	High-level method calls on objects
Serialization	Manual	Automatic
Language	Any language can interoperate	Java to Java only (RMI-IIOP for CORBA)
Effort	More code for parsing and framing	Little network code
Performance	Can be tuned precisely	Some overhead from serialization

Key Takeaways

- ✓ RMI lets a client call methods on an object in another JVM through a stub; arguments are serialized (by value), remote objects are passed as stubs.
- ✓ A remote interface extends Remote and every method throws RemoteException.
- ✓ The implementation extends UnicastRemoteObject (or is exported with exportObject) and is registered with Naming.rebind() or Registry.rebind().
- ✓ The client uses Naming.lookup(), casts to the interface and handles RemoteException.
- ✓ Run order: compile → rmiregistry (or createRegistry) → server → client.
- ✓ Typical errors: connection refused, NotBoundException, ClassNotFoundException and NotSerializableException.

— End of Unit 12 —