Remote Method Invocation

RMI Basics

- RMI is the Java Distributed Object Model for facilitating communications among distributed objects.
- RMI is a higher-level API built on top of sockets.
- Socket-level programming allows you to pass data through sockets among computers.
- RMI enables you not only to pass data among objects on different systems, but also to invoke methods in a remote object.

RMI vs RPC

- RMI is like Remote Procedure Calls (RPC) in the sense that both RMI and RPC enable you to invoke methods, but there are some important differences.
  - With RPC, you call a standalone procedure.
  - With RMI, you invoke a method within a specific object. RMI can be viewed as object-oriented RPC.

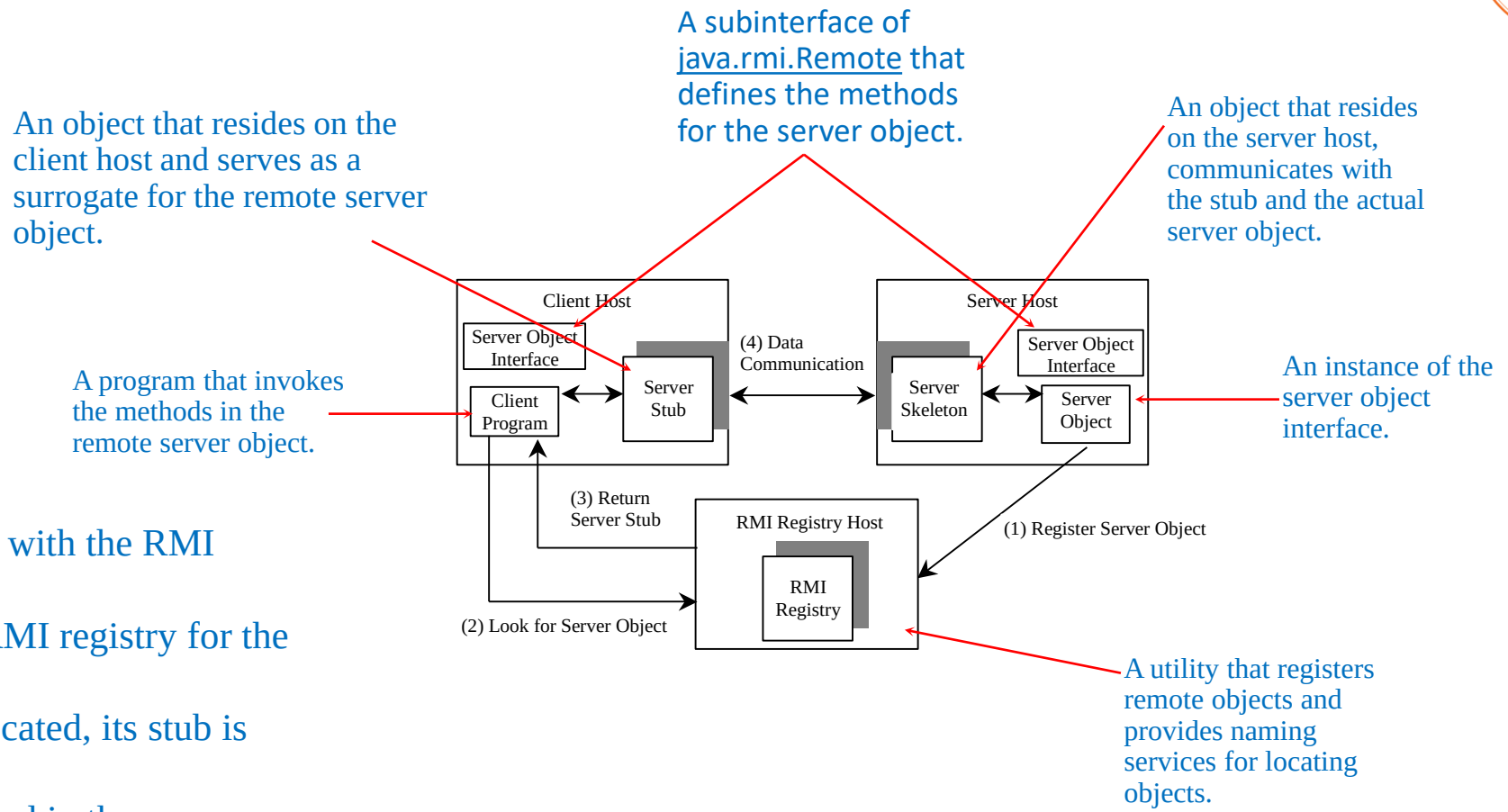
RMI vs Traditional Client/Server Approach

- A RMI component can act as both a client and a server, depending on the scenario in question.
- A RMI system can pass functionality from a server to a client and vice versa.
- A client/server system typically only passes data back and forth between server and client.

Key Components of the RMI Architecture

- Server object interface: A subinterface of java.rmi.Remote that defines the methods for the server object.
- Server class: A class that implements the remote object interface.
- Server object: An instance of the server class.
- RMI registry: A utility that registers remote objects and provides naming services for locating objects.
- Client program: A program that invokes the methods in the remote server object.
- Server stub: An object that resides on the client host and serves as a surrogate for the remote server object.
- Server skeleton: An object that resides on the server host and communicates with the stub and the actual server object.

How does RMI work?



RMI works as follows:

(1) A server object is registered with the RMI registry

(2) A client looks through the RMI registry for the remote object

(3) Once the remote object is located, its stub is returned in the client

(4) The remote object can be used in the same way as a local object. The communication between the client and the server is handled through the stub and skeleton.

Passing Parameters

- When a client invokes a remote method with parameters, passing parameters are handled under the cover by the stub and the skeleton. Let us consider three types of parameters:
  1. Primitive data type:A parameter of primitive type such as char, int, double, and boolean is passed by value like a local call.
  2. Local object type. A parameter of local object type such as java.lang.String is also passed by value.
  3. Remote object type

Local Object as Parameter

- This is completely different from passing object parameter in a local call.
- In a local call, an object parameter is passed by reference, which corresponds to the memory address of the object.
- In a remote call, there is no way to pass the object reference because the address on one machine is meaningless to a different Java VM.
- Any object can be used as a parameter in a remote call if the object is serializable.
- The stub serializes the object parameter and sends it in a stream across the network.
- The skeleton deserializes stream into an object.

Remote object type parameter passing

- Remote objects are passed differently from the local objects.
- When a client invokes a remote method with a parameter of some remote object type, the stub of the remote object is passed.
- The server receives the stub and manipulates the parameter through the stub.

How does a client locate the remote object?

- RMI registry provides the registry services for the server to register the object and for the client to locate the object.
- You can use several overloaded static getRegistry() methods in the LocateRegistry class to return a reference to a Registry.

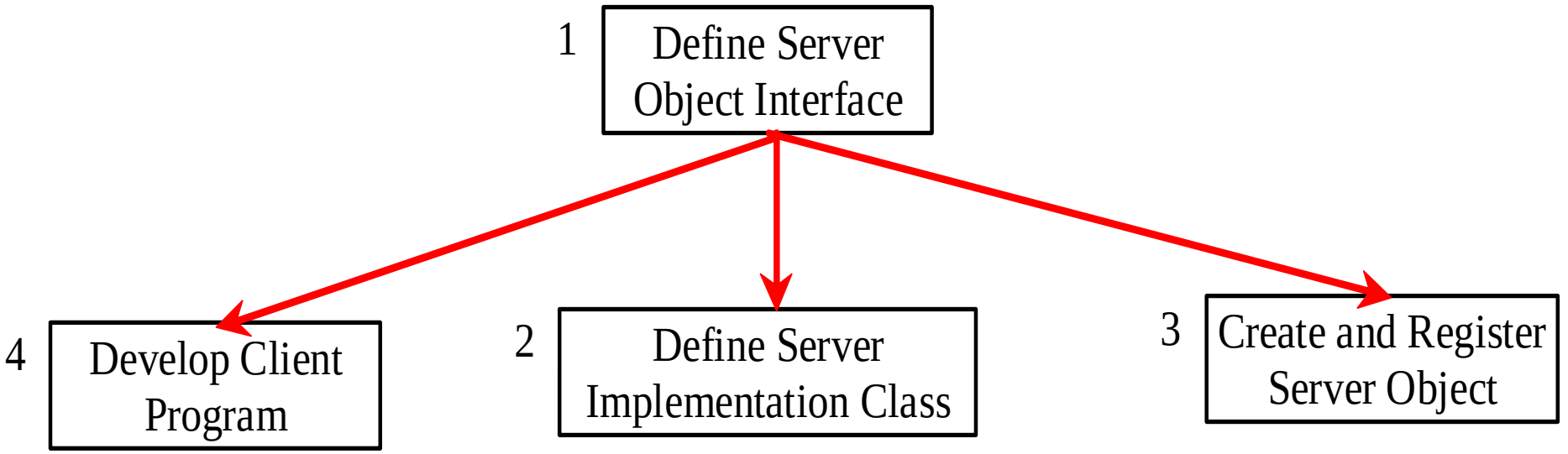
java.rmi.registry.LocateRegistry	
+getRegistry(): Registry	Returns a reference to the remote object Registry for the local host on the default registry port of 1099.
+getRegistry(port: int): Registry	Returns a reference to the remote object Registry for the local host on the specified port.
+getRegistry(host: String): Registry	Returns a reference to the remote object Registry on the specified host on the default registry port of 1099.
+getRegistry(host:String, port: int): Registry	Returns a reference to the remote object Registry on the specified host and port.

RMI Registry: Binding Objects

- Once a Registry is obtained, you can bind an object with a unique name in the registry using the bind or rebind method or locate an object using the lookup method.

java.rmi.registry.Registry	
+bind(name: String, obj: Remote): void	Binds the specified name with the remote object.
+rebind(name: String, obj: Remote): void	Binds the specified name with the remote object. Any existing binding for the name is replaced.
+unbind(name: String): void	Destroys the binding for the specified name that is associated with a remote object.
+list(name: String): String[]	Returns an array of the names bound in the registry.
+lookup(name: String): Remote	Returns a reference, a stub, for the remote object associated with the specified name.

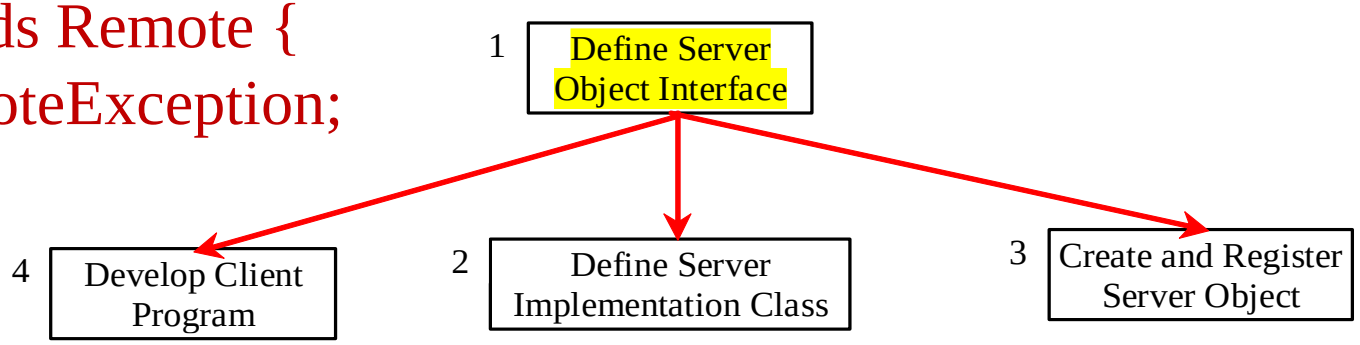
Developing RMI Applications



Step 1: Define Server Object Interface

Define a server object interface that serves as the contract between the server and its clients, as shown in the following outline:

```
public interface ServerInterface extends Remote {
  public void service1(...) throws RemoteException;
  // Other methods
}
```

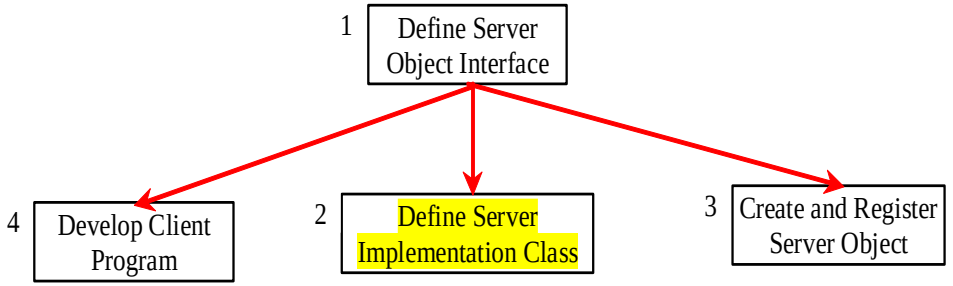


A server object interface must extend the java.rmi.Remote interface.

Step 2: Define Server Implementation Object

Define a class that implements the server object interface

```
public class ServerInterfaceImpl extends UnicastRemoteObject
  implements ServerInterface {
  public void service1(...) throws RemoteException {
    // Implement it
  }
  // Implement other methods
}
```

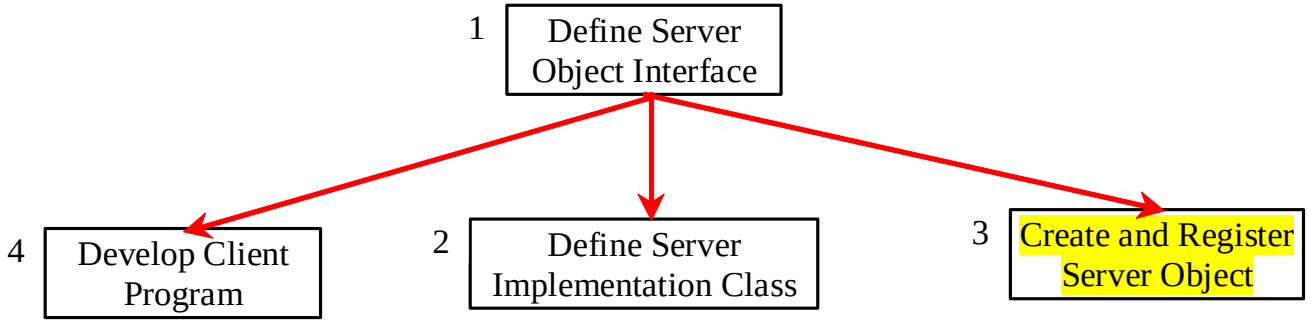


The server implementation class must extend the java.rmi.server.UnicastRemoteObject class.
The UnicastRemoteObject class provides support for point-to-point active object references using TCP streams.

Step 3: Create and Register Server Object

Create a server object from the server implementation class and register it with an RMI registry:

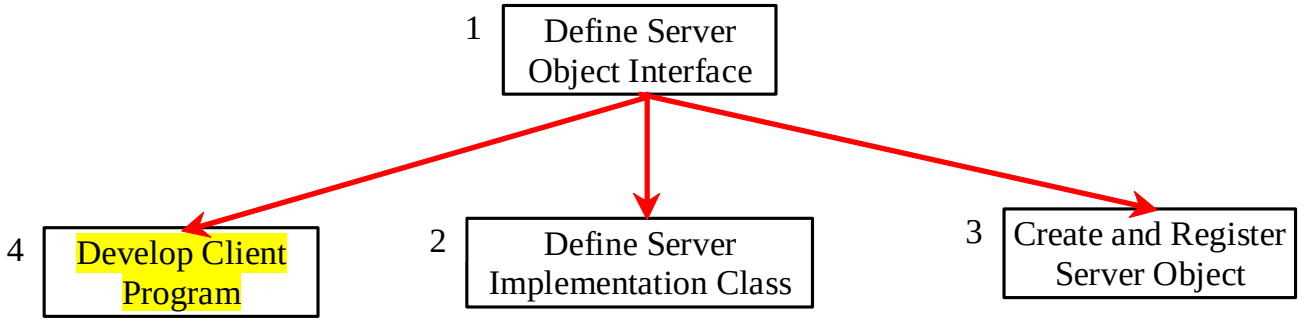
```
ServerInterface server = new ServerInterfaceImpl(...);
Registry registry = LocateRegistry.getRegistry();
registry.rebind("RemoteObjectName", server);
```



Step 4: Develop Client Program

Develop a client that locates a remote object and invokes its methods

```
Registry registry = LocateRegistry.getRegistry(host);
ServerInterface server = (ServerInterfaceImpl)
  registry.lookup("RemoteObjectName");
server.service1(...);
```



- java -Djava.security.manager -Djava.security.policy=path\to\my.policy FileName

RMI vs. Socket-Level Programming

- RMI enables you to program at a higher level of abstraction. It hides the details of socket server, socket, connection, and sending or receiving data. It even implements a multithreading server under the hood, whereas with socket-level programming you have to explicitly implement threads for handling multiple clients.
- RMI applications are scalable and easy to maintain. You can change the RMI server or move it to another machine without modifying the client program except for resetting the URL to locate the server. (To avoid resetting the URL, you can modify the client to pass the URL as a command-line parameter.) In socket-level programming, a client operation to send data requires a server operation to read it. The implementation of client and server at the socket-level is tightly synchronized.
- RMI clients can directly invoke the server method, whereas socket-level programming is limited to passing values. Socket-level programming is very primitive.

Consent

Data for this course is taken from several books, slides of already similar course taught in other universities and is the sole property of the respective owner. The copyright infringement is not intended, and this material is solely used for the academic purpose or as a teaching material