Chapter 40 Remote Method Invocation

Objectives

- ✦To explain how RMI works (§40.2).
- ✦To describe the process of developing RMI applications (§40.3).
- ✦To distinguish between RMI and socket-level programming (§40.4).
- ✦To develop three-tier applications using RMI (§40.5).
- ✦To use callbacks to develop interactive applications (§40.6).

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.

The Differences between RMI and RPC

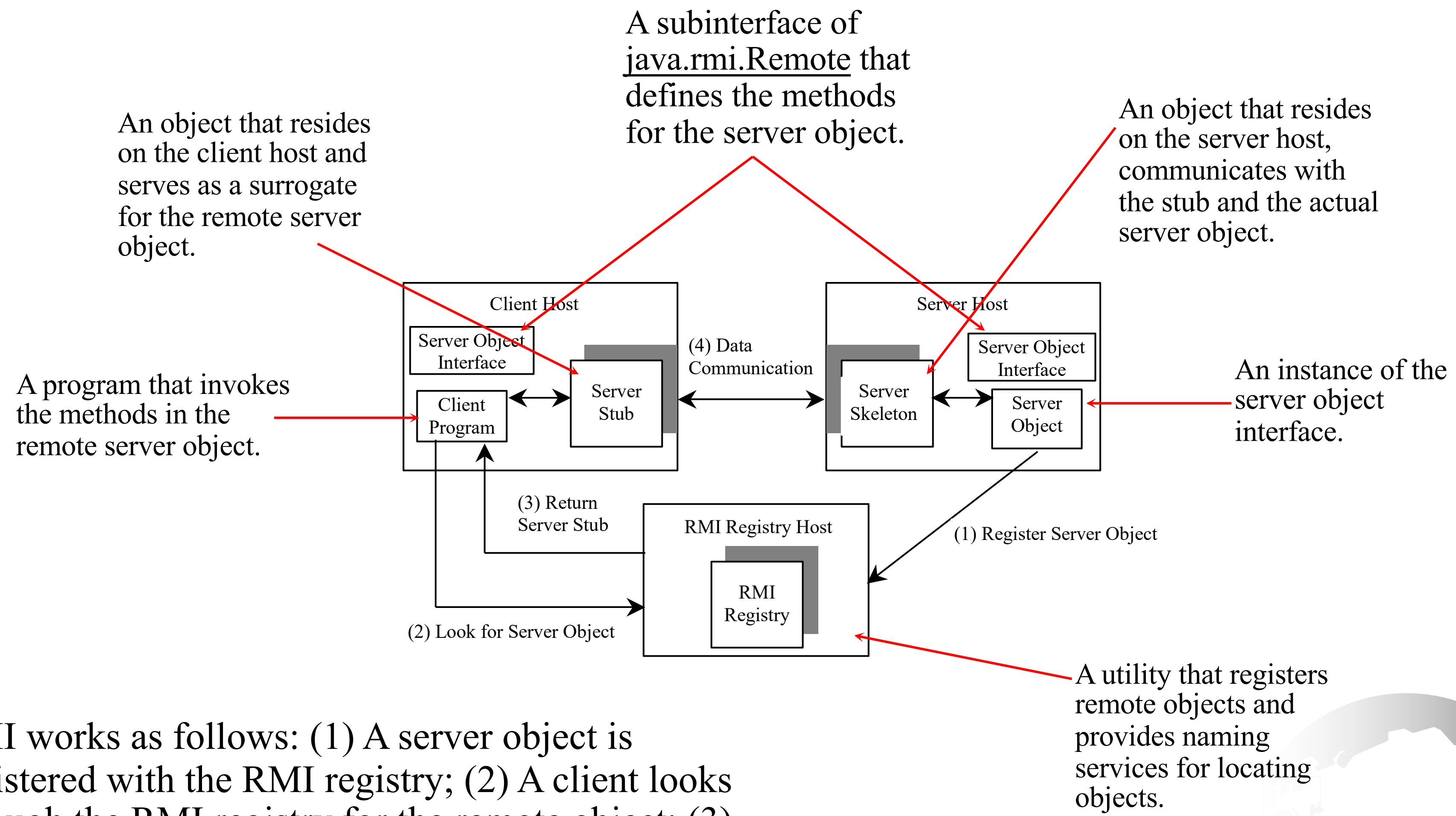
RMI is similar to Remote Procedure Calls (RPC) in the sense that both RMI and RPC enable you to invoke methods, but there are some important differences. With RPC, you call a standalone procedure. With RMI, you invoke a method within a specific object. RMI can be viewed as object-oriented RPC.

The Differences between RMI and 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 fourth between server and client.

How does RMI work?



A subinterface of java.rmi.Remote that defines the methods for the server object.
An object that resides on the client host and serves as a surrogate for the remote server object.
An object that resides on the server host, communicates with the stub and the actual server object.
A program that invokes the methods in the remote server object.
An instance of the server object interface.
Client Host
Server Object Interface
Client Program
Server Stub
(4) Data Communication
Server Host
Server Object Interface
Server Skeleton
Server Object
(3) Return Server Stub
RMI Registry Host
RMI Registry
(1) Register Server Object
(2) Look for Server Object
A utility that registers remote objects and provides naming services for locating objects.

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.

Passing Parameters, cont.

Local object type. A parameter of local object type such as java.lang.String is also passed by value. 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 as long as 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.

Passing Parameters, cont.

Remote object type. 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.

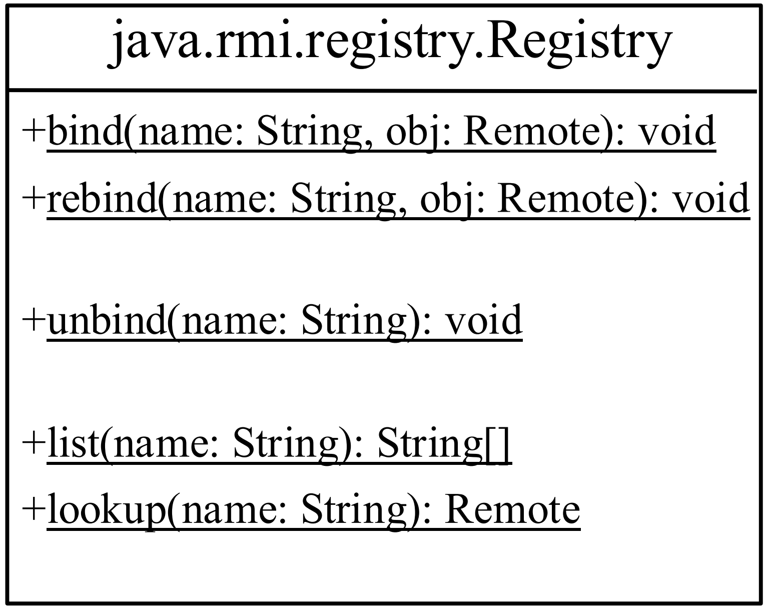
RMI Registry

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. 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.

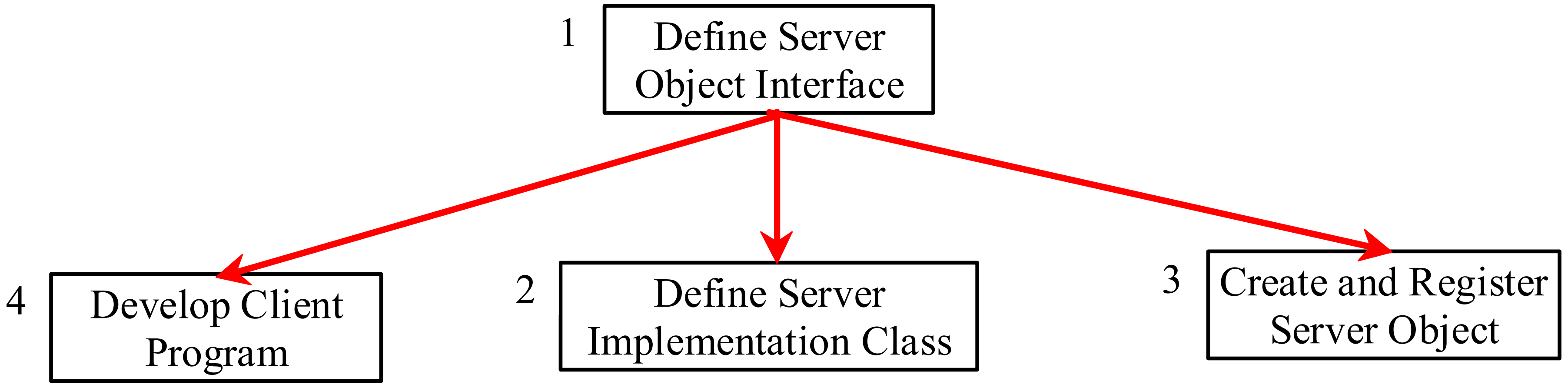
java.rmi.registry.LocateRegistry	
+getRegistry(): Registry	Returns a reference to the remote object Registry for the local host on the default registry port of 1099.
+getRegistry(port: int): Registry	Returns a reference to the remote object Registry for the local host on the specified port.
+getRegistry(host: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



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

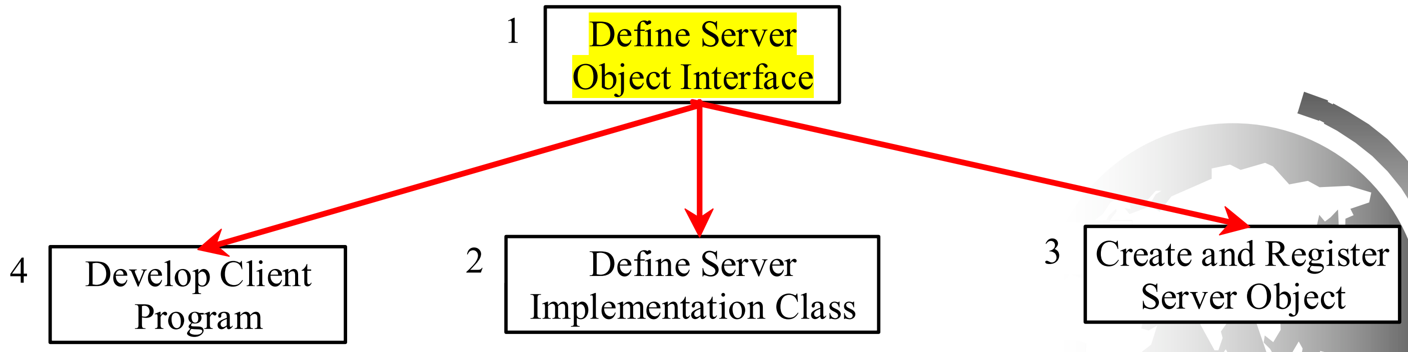

1
Define Server Object Interface
4
Develop Client Program
2
Define Server Implementation Class
3
Create and Register Server Object

Step 1: Define Server Object Interface

1. 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.



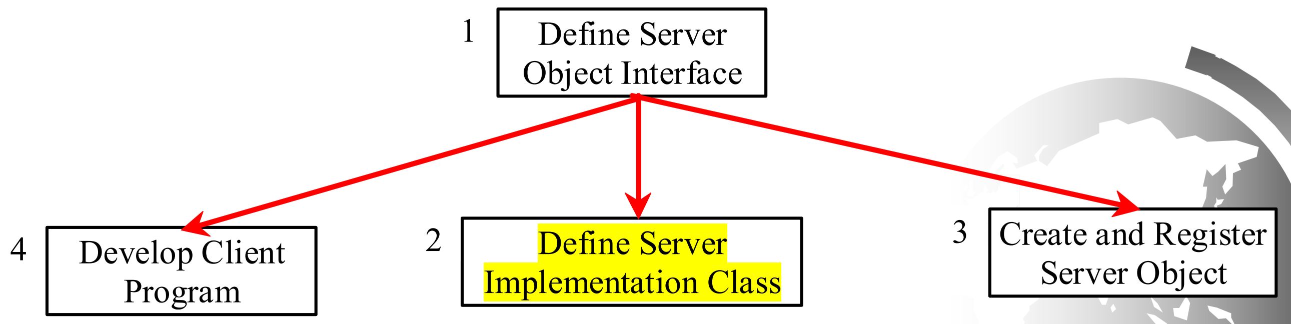
1 Define Server Object Interface
2 Define Server Implementation Class
3 Create and Register Server Object
4 Develop Client Program

Step 2: Define Server Implementation Object

2. Define a class that implements the server object interface, as shown in the following outline:

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

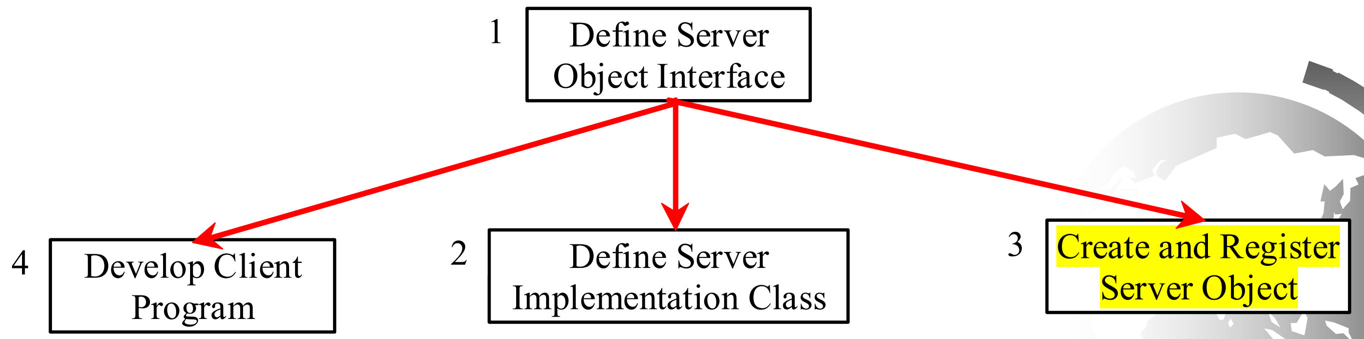
The server implementation class must extend the java.rmi.server.UnicastRemoteObject class. The UnicastRemoteObject class provides support for point-to-point active object references using TCP streams.


1
Define Server Object Interface
4
Develop Client Program
2
Define Server Implementation Class
3
Create and Register Server Object

Step 3: Create and Register Server Object

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

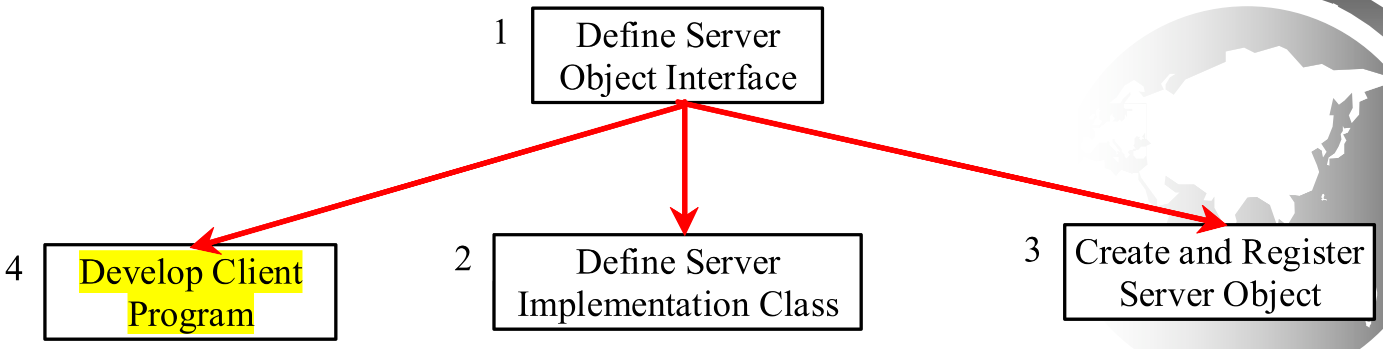
```
ServerInterface server = new ServerInterfaceImpl(...);
Registry registry = LocateRegistry.getRegistry();
registry.rebind("RemoteObjectName", server);
```


1
Define Server Object Interface
4
Develop Client Program
2
Define Server Implementation Class
3
Create and Register Server Object

Step 4: Develop Client Program

4. Develop a client that locates a remote object and invokes its methods, as shown in the following outline:

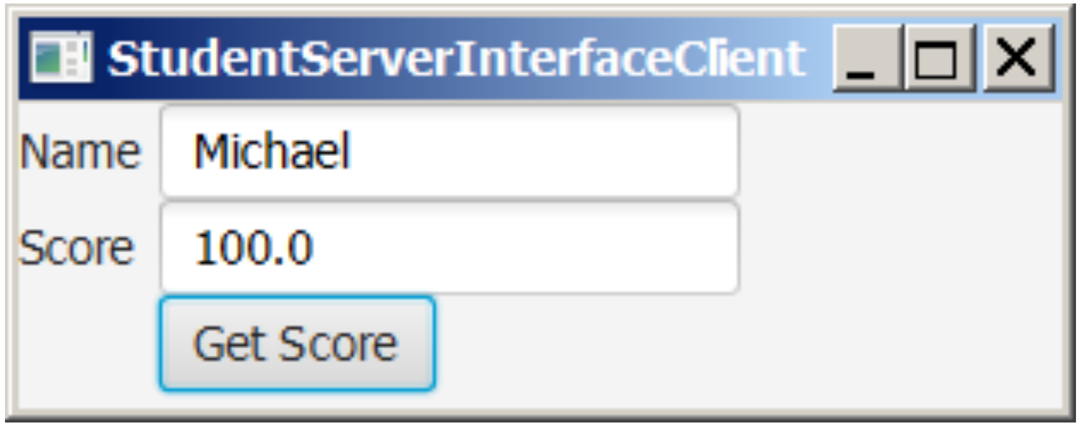
```
Registry registry = LocateRegistry.getRegistry(host);
ServerInterface server = (ServerInterfaceImpl)
  registry.lookup("RemoteObjectName");
server.service1(...);
```



1
Define Server Object Interface
4
Develop Client Program
2
Define Server Implementation Class
3
Create and Register Server Object

Example: Retrieving Student Scores from an RMI Server

- Problem: This example creates a client that retrieves student scores from an RMI server.



StudentServerInterfaceClient

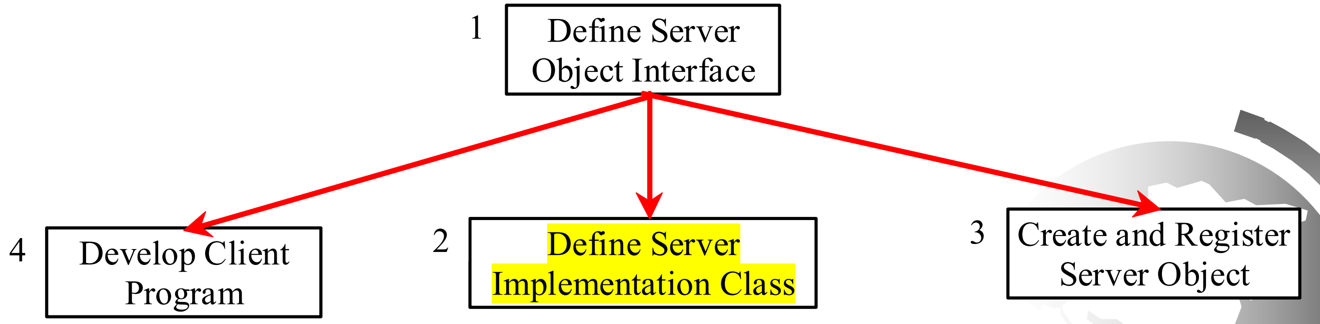
Name: Michael

Score: 100.0

Step 1: Define Server Object Interface

1. Create a server interface named StudentServerInterface. The interface tells the client how to invoke the server's findScore method to retrieve a student score.

StudentServerInterface

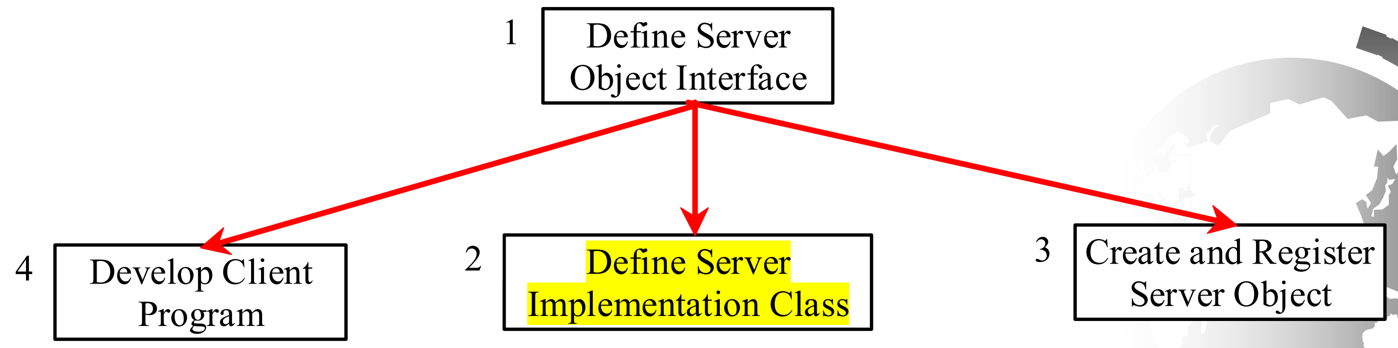


1 Define Server Object Interface
2 Define Server Implementation Class
3 Create and Register Server Object
4 Develop Client Program

Step 2: Define Server Implementation Object

2. Create server implementation named StudentServerInterfaceImpl that implements StudentServerInterface. The findScore method returns the score for a specified student. This method returns -1 if the score is not found.

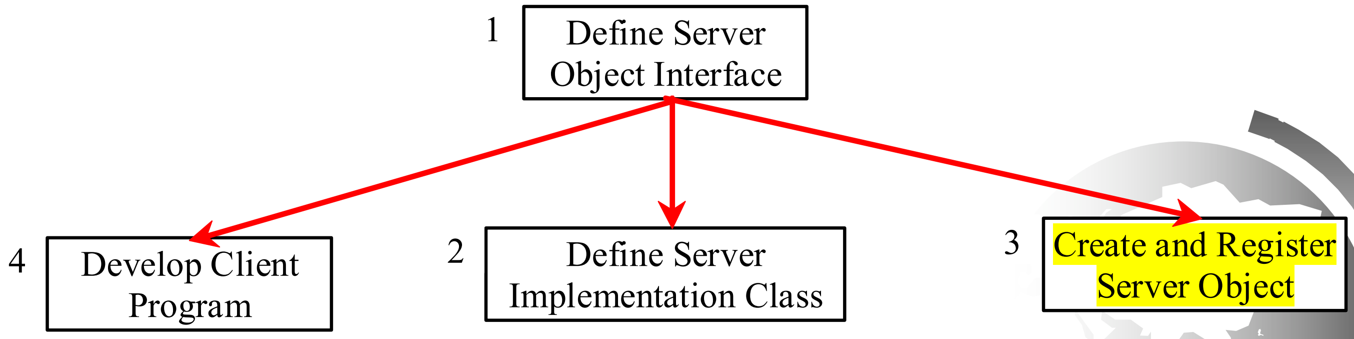
StudentServerInterfaceImpl


1
Define Server Object Interface
2
Define Server Implementation Class
3
Create and Register Server Object
4
Develop Client Program

Step 3: Create and Register Server Object

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

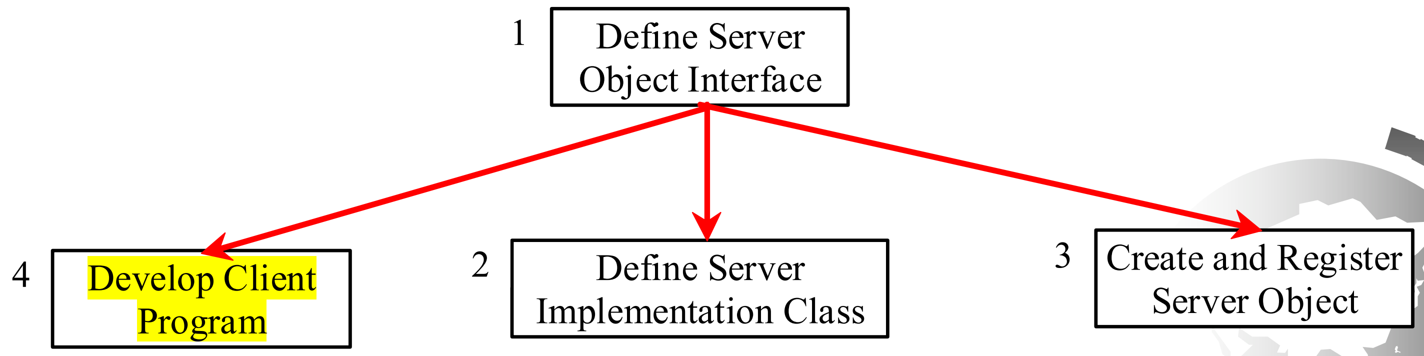
RegistrerWithRMIServer


1
Define Server Object Interface
4
Develop Client Program
2
Define Server Implementation Class
3
Create and Register Server Object

Step 4: Develop Client Program

4. Create a client as an applet named StudentServerInterfaceClient. The client locates the server object from the RMI registry, uses it to find the scores.

StudentServerInterfaceClient



1
Define Server Object Interface
4
Develop Client Program
2
Define Server Implementation Class
3
Create and Register Server Object

Run Example

1. Start RMI Registry by typing "**start rmiregistry**" at a DOS prompt from the book directory. By default, the port number 1099 is used by rmiregistry. To use a different port number, simply type the command "**start rmiregistry *portnumber***" at a DOS prompt.

2. Start RegisterWithRMIServer using the following command at C:\book directory:
C:\book>java RegisterWithRMIServer

3. Run StudentServerInterfaceClient as an application.

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. Avoid using it to develop client/server applications. As an analogy, socket-level programming is like programming in assembly language, while RMI programming is like programming in a high-level language.

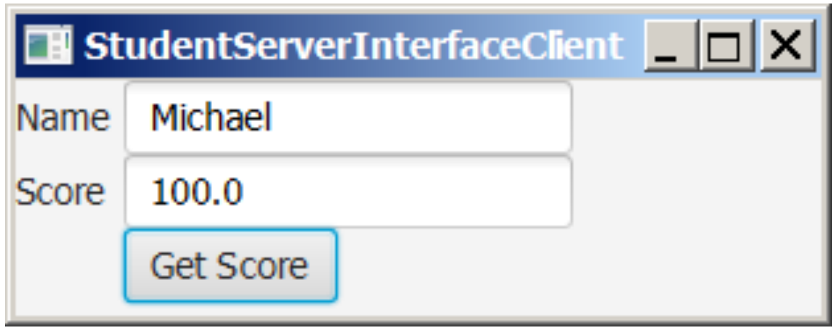
Developing Three-Tier Applications Using RMI

Three-tier applications have gained considerable attention in recent years, largely because of the demand for more scalable and load-balanced systems to replace traditional two-tier client/server database systems. A centralized database system not only handles data access but also processes the business rules on data. Thus, a centralized database is usually heavily loaded because it requires extensive data manipulation and processing. In some situations, data processing is handled by the client and business rules are stored on the client side. It is preferable to use a middle tier as a buffer between a client and the database. The middle tier can be used to apply business logic and rules, and to process data to reduce the load on the database.

A three-tier architecture does more than just reduce the processing load on the server. It also provides access to multiple network sites. This is especially useful to Java applets that need to access multiple databases on different servers, since an applet can only connect with the server from which it is downloaded.

Example: Retrieving Student Scores on a Database Using RMI

Problem: This example rewrites the preceding example to find scores stored in a database rather than a hash map. In addition, the system is capable of blocking a client from accessing a student who has not given the university permission to publish his/her score. An RMI component is developed to serve as a middle tier between client and database; it sends a search request to the database, processes the result, and returns an appropriate value to the client.



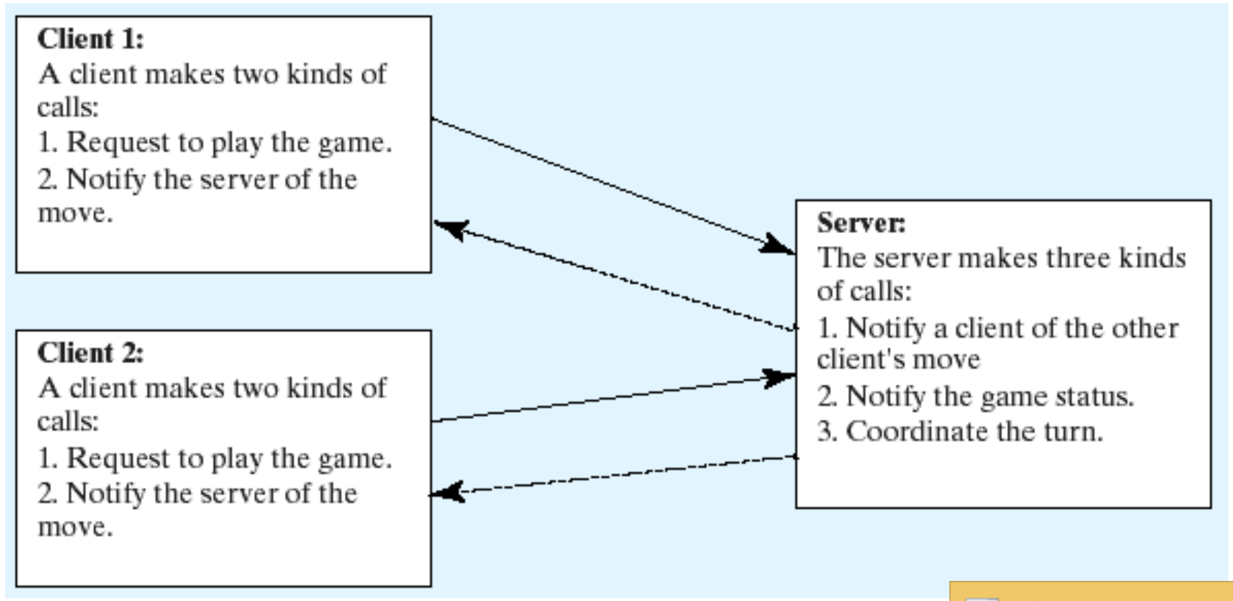
Name Michael
Score 100.0

RMI Call Backs

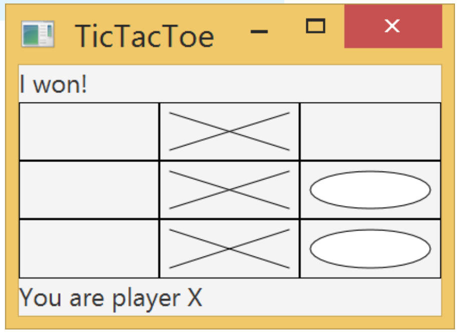
In a traditional client/server system, a client sends a request to a server, and the server processes the request and returns the result to the client. The server cannot invoke the methods on a client. One of the important benefits of RMI is that it supports *callbacks*, which enable the server to invoke the methods on the client. With the RMI callback feature, you can develop interactive distributed applications.

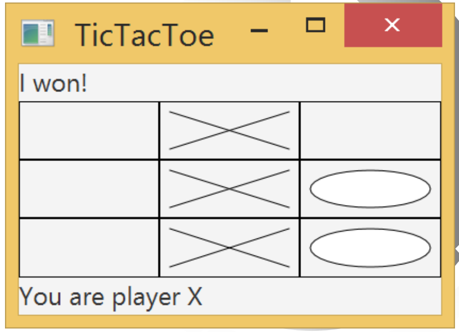
Example: Distributed TicTacToe Using RMI

Example 32.7, “Distributed TicTacToe Game,” was developed using stream socket programming. Write a new distributed TicTacToe game using the RMI.

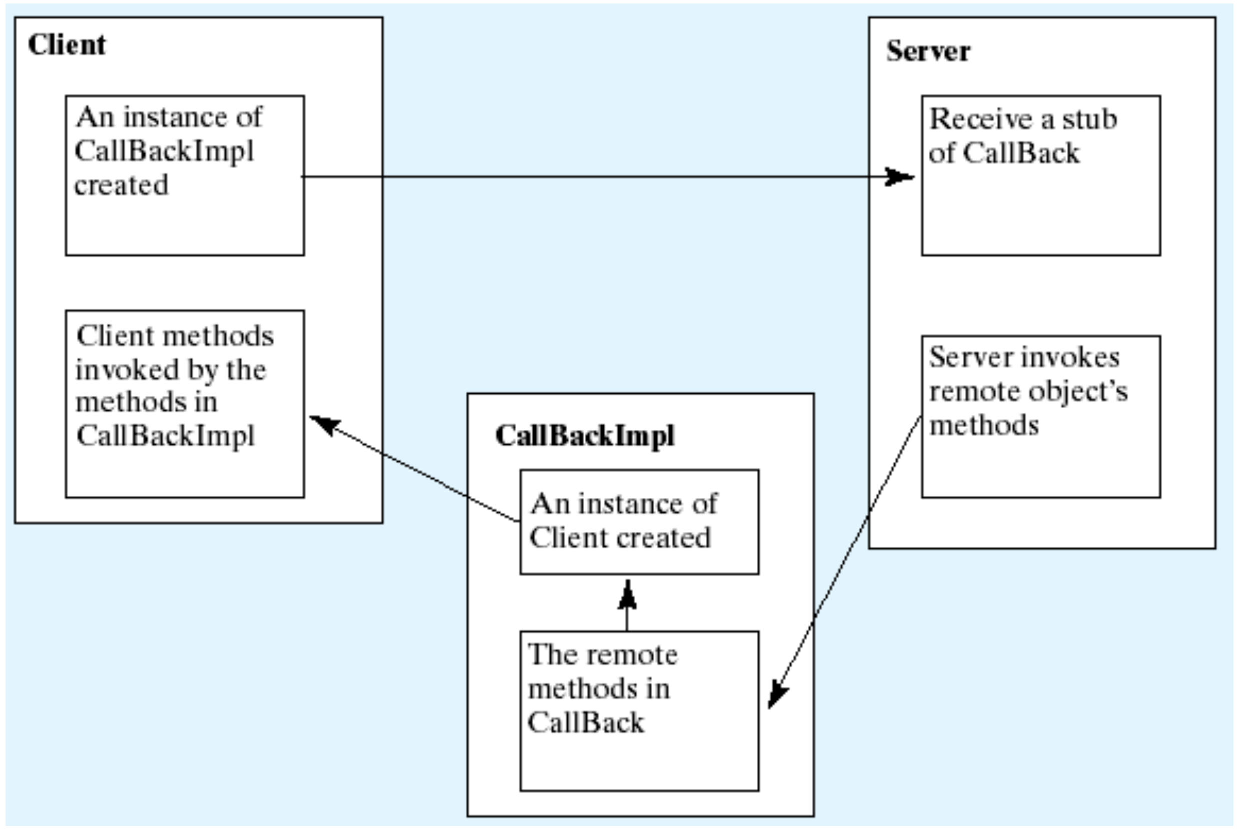


Client 1:
A client makes two kinds of calls:
1. Request to play the game.
2. Notify the server of the move.
Client 2:
A client makes two kinds of calls:
1. Request to play the game.
2. Notify the server of the move.
Server:
The server makes three kinds of calls:
1. Notify a client of the other client's move
2. Notify the game status.
3. Coordinate the turn.





Example: Distributed TicTacToe Using RMI


Client
An instance of CallBackImpl created
Client methods invoked by the methods in CallBackImpl
Server
Receive a stub of CallBack
Server invokes remote object's methods
CallBackImpl
An instance of Client created
The remote methods in CallBack

TicTacToeInterfce

CallBack

TicTacToeImpl

CallBackImpl

TicTacToeClientRMI