Exercise: Remote Method Invocation

1. The Exercise

This exercise involves building a simple networked In/Out board using Java's Remote Method Invocation facility.

You will gain familiarity with the RMI facility and will also make use of a number of the utility classes, as well as of the AWT.

2. The Components

You will need to develop four major sections to complete this exercise:

- InOutServerI interface
- InOutServer class
- Notifiable interface
- InOutClient class

The following sections define each of these.

InOutServerI Interface

An RMI server object is manipulated via one or more interfaces that define the methods it makes available to remote clients. In this exercise, the InOutServer defines an interface as follows (in the file *InOutServerI.java*):

```
import java.rmi.*;

interface InOutServerI extends Remote
  {
  String serverURL = "InOutServer";
  Character IN = new Character ('I'),
                OUT = new Character ('O');
  void createOrMark (String who, Character state) throws RemoteException;
  void remove (String who) throws RemoteException;
  void registerForNotification (Notifiable n) throws RemoteException;
  void unRegisterForNotification (Notifiable n) throws RemoteException;
  }
```

The use of most of these methods should be clear.

Note that all remote methods *must* be defined to throw a java.rmi.RemoteException, since Java assumes that network errors can potentially occur during the execution of any remote operation.

InOutServer Class

This is a simple Java application (file: *InOutServer.java*) that implements the above interface and binds a name to it.

Notifiable Interface

The notifiable interface (in the file *Notifiable.java*) is implemented by the client to define the callback protocol (recall that a general callback facility can only be achieved in Java via the use of interfaces…) that the server should use to notify whenever it's state changes:

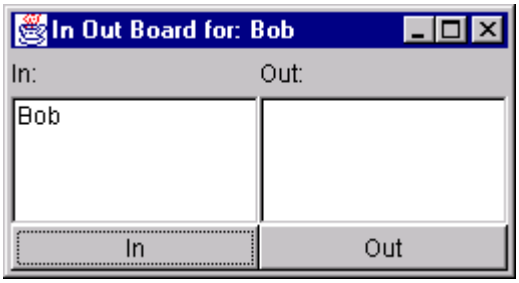
```
import java.rmi.*;
import java.util.*;

public interface Notifiable extends Remote
  {
  void notify (Hashtable resources) throws RemoteException;
  }
```

The resources parameter passed to the callback is a copy of the server's record: who is marked 'In' and who is marked 'Out'…it is a table of {who, current status} key/value pairs.

InOutClient Class

This is the only part of the system that the 'user' interacts with directly. The following picture shows an example of the client portion of the system:



This simple client is only allowed to mark its own user in or out: it obtains the user's name from the System property list.

The client utilises the server's remote interface and also implements the Notifiable interface as described earlier—this enables the server to call back to it the reflect any changes in its state.

To make the callback mechanism work correctly, the client (in the file *InOutClient.java*) will have to include a method invocation similar to:

```
try
    {
    UnicastRemoteObject.exportObject (this);
    }
catch (RemoteException re)
    {
    re.printStackTrace ();
    }
```

3. Additional Materials

You will need to create a policy file for the server. A sample file (call it policy.txt) is:

```
grant {
      // Allow everything for now
      permission java.security.AllPermission;
};
```

This file effectively turns all security checking off!

If you are using a simple DOS-based development environment, you may find it useful to use a batch file to automate your project. A sample file (called run.bat) is:

```
if "%OS%" == "Windows_NT" setlocal

set OCP=%CLASSPATH%
set CLASSPATH=

start /min rmiregistry -J-Djava.security.policy=file:./policy.txt

set CLASSPATH=.;%OCP%

javac *.java
rmic InOutServer
rmic InOutClient
start java -Djava.rmi.server.codebase=file:./
          -Djava.security.policy=file:./policy.txt InOutServer

echo *
echo * Please wait until a message appears in the server process's window
echo * indicating that it has been bound to the registry.
echo *
pause

java -Djava.security.policy=file:./policy.txt InOutClient

if "%OS%" == "Windows_NT" endlocal
```

4. Enhancing The Server (Optional)

The simple server you have written is started from the command line and can only be terminated via the OS's end task facility. In a production environment, this is typically not sufficient: many servers are also provided with a management interface that allows them to be "remote controlled."

Java RMI allows a server object to have more than one registered interface.

In this portion of the exercise you will add the following management interface to your server (file: *InOutServerManagement.java*):

```
import java.rmi.*;

interface InOutServerManagementI extends Remote
  {
  String managementURL = "InOutServerManagement";
  void stopService () throws RemoteException;
  void startService (String URL) throws RemoteException;
  void exitService () throws RemoteException;
  }
```

stopService () unbinds the server's 'public' name from the registry, startService () binds that name and so allows clients to use it remotely. exitService () causes the server to exit completely. You should change your server such that it waits for a startService () call before registering itself for clients to use.

You will also need to create a simple command-line Java application that can issue the appropriate management commands using the above interface.