

Activation of remote objects

The **Activatable** class

- Prior to the release of Java 2 SDK, an instance of a **UnicastRemoteObject** could be accessed from a server program that
 - created an instance of the remote object
 - ran all the time
- With the introduction of the class **java.rmi.activation.Activatable** and the RMI daemon, `rmid`, programs can be written to register information about remote object implementations that should be created and execute "on demand", rather than running all the time
- The RMI daemon, `rmid`, provides a Java virtual machine from which other JVM instances may be spawned

The Remote Interface

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

```
public interface MyRemoteInterface extends Remote {  
    public Object callMeRemotely() throws RemoteException;  
}
```

The Client

```
import java.rmi.*;

public class Client {
    public static void main(String args[]) {
        String server = "localhost";
        System.setSecurityManager(new RMISecurityManager());
        try {
            String location = "rmi://" + server +
                "/ActivatableImplementation";
            MyRemoteInterface mri =
                (MyRemoteInterface)Naming.lookup(location);
            String result = "failure";
            System.out.println("Making remote call to
                the server");
            result = (String)mri.callMeRemotely();
            System.out.println("Returned from
                remote call");
            System.out.println("Result: " + result);
        } catch (Exception e) {e.printStackTrace();}
    }
}
```

The Remote Interface Implementation

- There are four steps to create an implementation class
 - Make the appropriate imports in the implementation class
 - Extend your class from `java.rmi.activation.Activatable`
 - Declare a two-argument constructor in the implementation class
 - Implement the remote interface methods

The Remote Interface Implementation

```
import java.rmi.*;
import java.rmi.activation.*;

public class ActivatableImplementation extends Activatable
    implements MyRemoteInterface {

    public ActivatableImplementation(ActivationID id,
        MarshalledObject data) throws RemoteException {
        super(id, 0);
    }

    public Object callMeRemotely() throws RemoteException {
        return "Success";
    }
}
```

The “Setup” Class

- The job of the "setup" class is to create all the information necessary for the activatable class, without necessarily creating an instance of the remote object
- The setup class passes the information about the activatable class to rmid, registers a remote reference (an instance of the activatable class's stub class) and an identifier (name) with the rmiregistry, and then the setup class may exit

The “Setup” Class /2

- There are seven steps to create a setup class:
 - Make the appropriate imports
 - Install a security manager
 - Create an `ActivationGroup` instance
 - Create an `ActivationDesc` instance
 - Declare an instance of your remote interface and register with `rmid`
 - Bind the stub to a name in the `rmiregistry`
 - Quit the setup application

The “Setup” Class /3

- In this example, for simplicity, we use a policy file that gives global permission to anyone from anywhere
 - *Do not use this policy file in a production environment*
- In the setup application, the job of the activation group descriptor is to provide all the information that rmid will require to contact the appropriate existing JVM or spawn a new JVM for the activatable object

The “Setup” Class /4

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

public class Setup {
    public static void main(String[] args) throws Exception {
        System.setSecurityManager(new RMISecurityManager());

        Properties props = new Properties();
        props.put("java.security.policy",
            "/home/rmi/activation/policy");
        ActivationGroupDesc.CommandEnvironment ace = null;
        ActivationGroupDesc exampleGroup = new
            ActivationGroupDesc(props, ace);

        ActivationGroupID agi =
            ActivationGroup.getSystem().
                registerGroup(exampleGroup);
        [to be continued...]
```

The “Setup” Class /5

```
String location = "file:/home/rmi/activation/";
MarshaledObject data = null;
ActivationDesc desc = new ActivationDesc(agi,
    "ActivatableImplementation", location, data);
MyRemoteInterface mri =
    (MyRemoteInterface)Activatable.register(desc);
System.out.println("Got the stub for the
    ActivatableImplementation");
Naming.rebind("ActivatableImplementation", mri);
System.out.println("Exported
    ActivatableImplementation");
System.exit(0);
} //main
} //clas Setup
```

Compiling and Running the Code

- There are six steps to compile and run the code:
 - Compile the remote interface, implementation, client, and setup classes
 - Run `rmic` on the implementation class
 - Start the `rmiregistry`
 - Start the activation daemon, `rmid`
 - for Sun's implementation

```
    rmid -J-Dsun.rmi.activation.execPolicy=none
```
 - Run the setup program
 - Run the client

More Information about RMI /1

- Other activation features
 - Activation of an object that does not extend `java.rmi.activation.Activatable`
 - Activation of a `UnicastRemoteObject`

`http://java.sun.com/j2se/1.4.2/docs/guide/rmi/activation`

More Information about RMI /2

- Java security

<http://java.sun.com/products/jdk/1.2/doc/guide/security>

- Use of RMI with SSL

- Custom socket factories for RMI-based communication
- An application can export a remote object to use an RMI socket factory that creates SSL sockets, so it can use SSL socket communication instead of the default socket communication
- Java 2 SDK, v1.4 includes the Java Secure Socket Extension (JSSE) API which provides an implementation of SSL sockets

<http://java.sun.com/products/jsse/>

More Information about RMI /3

- The java.sun.com Web site
- William Grosso, *Java RMI*, O'Reilly, 2001