

CS 5523 Lecture 19: Design Issues for Java RMI

- *An example of Java remote invocation with parameters*
- *Callbacks*
- *Java serialization*
- *Java reflection*
- *RMI software*
- *Java persistent objects*

Figure 5.11
Java Remote interfaces *Shape* and *ShapeList*

```
import java.rmi.*;
import java.util.Vector;
public interface Shape extends Remote {
    int getVersion() throws RemoteException;
    GraphicalObject getAllState() throws RemoteException; 1
}
public interface ShapeList extends Remote {
    Shape newShape(GraphicalObject g) throws RemoteException; 2
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
}
```

Figure 5.13
Java class *ShapeListServer* with *main* method

```
import java.rmi.*;
public class ShapeListServer{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        try{
            ShapeList aShapeList = new ShapeListServant();           1
            Naming.rebind("Shape List", aShapeList );                2
            System.out.println("ShapeList server ready");
        }catch(Exception e) {
            System.out.println("ShapeList server main " + e.getMessage());}
        }
    }
```

Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn. 3
© Addison-Wesley Publishers 2000

Figure 5.14
Java class *ShapeListServant* implements interface *ShapeList*

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;
public class ShapeListServant extends UnicastRemoteObject implements ShapeList {
    private Vector theList; // contains the list of Shapes           1
    private int version;
    public ShapeListServant()throws RemoteException{...}
    public Shape newShape(GraphicalObject g) throws RemoteException {  2
        version++;
        Shape s = new ShapeServant(g, version);                       3
        theList.addElement(s);
        return s;
    }
    public Vector allShapes()throws RemoteException{...}
    public int getVersion() throws RemoteException { ... }
}
```

Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn. 3
© Addison-Wesley Publishers 2000

Figure 5.15
Java client of *ShapeList*

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class ShapeListClient{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        ShapeList aShapeList = null;
        try{
            aShapeList = (ShapeList) Naming.lookup("//bruno.ShapeList") ;           1
            Vector sList = aShapeList.allShapes();                                  2
        } catch(RemoteException e) {System.out.println(e.getMessage());}
        } catch(Exception e) {System.out.println("Client: " + e.getMessage());}
    }
}
```

Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn. 3
© Addison-Wesley Publishers 2000

ShapeListClient.java

```
package examples.RMIShape;
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
import java.awt.Rectangle;
import java.awt.Color;

public class ShapeListClient{

    public static void main(String args[]){
        String option = "Read";
        String shapeType = "Rectangle";
        if(args.length > 0) option = args[0]; // read or write
        if(args.length > 1) shapeType = args[1]; // specify Circle, Line etc
        System.out.println("option = " + option + "shape = " + shapeType);

        if(System.getSecurityManager() == null){
            System.setSecurityManager(new RMISecurityManager());
        } else System.out.println("Already has a security manager, so cant set RMI SM");
        ShapeList aShapeList = null;
    }
}
```

ShapeListClient.java (cont.)

```
try{
    aShapeList = (ShapeList) Naming.lookup("//Jean.torriano.net/ShapeList");
    System.out.println("Found server");
    Vector sList = aShapeList.allShapes();
    System.out.println("Got vector");
    if(option.equals("Read")){
        for(int i=0; i<sList.size(); i++){
            GraphicalObject g = ((Shape)sList.elementAt(i)).getAllState();
            g.print();
        }
    } else {
        GraphicalObject g = new GraphicalObject(shapeType,
                                                new Rectangle(50,50,300,400),
                                                Color.red, Color.blue, false);
        System.out.println("Created graphical object");
        aShapeList.newShape(g);
        System.out.println("Stored shape");
    }
} catch (RemoteException e) {System.out.println("allShapes: " + e.getMessage());}
} catch (Exception e) {System.out.println("Lookup: " + e.getMessage());}
} catch (RemoteException e) {System.out.println("allShapes: " + e.getMessage());}
} catch (Exception e) {System.out.println("Lookup: " + e.getMessage());}
}
```

GraphicalObject.java

```
package examples.RMIShape;
import java.awt.Rectangle;
import java.awt.Color;
import java.io.Serializable;

public class GraphicalObject implements Serializable{
    public String type;
    public Rectangle enclosing;
    public Color line;
    public Color fill;
    public boolean isFilled;

    public GraphicalObject() { }
    public GraphicalObject(String aType, Rectangle anEnclosing,
                           Color aLine,Color aFill, boolean anIsFilled) {
        type = aType;
        enclosing = anEnclosing;
        line = aLine;
        fill = aFill;
        isFilled = anIsFilled;
    }

    public void print(){
        System.out.print(type);
        System.out.print(enclosing.x + " , " + enclosing.y + " , " + enclosing.width
                          + " , " + enclosing.height);
    }
}
```

ShapeListServant.java

```
package examples.RMIShape;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;

public class ShapeListServant extends UnicastRemoteObject implements ShapeList{
    private Vector theList;
    private int version;

    public ShapeListServant()throws RemoteException{
        theList = new Vector();
        version = 0;
    }

    public Shape newShape(GraphicalObject g) throws RemoteException{
        version++;
        Shape s = new ShapeServant( g, version);
        theList.addElement(s);
        return s;
    }

    public Vector allShapes()throws RemoteException{ return theList; }

    public int getVersion() throws RemoteException{ return version;}
}
```

Callbacks:

- *Instead of client polling the server, the server calls a method in the client when it is updated.*
- *Callback refers to server's action in notifying the client*
- *Client creates a remote object that implements an interface for server to call.*
- *Server provides an operation for clients to "register" their callbacks.*
- *When an event occurs, the server calls the interested clients.'*

Callback pluses:

- *More efficient than polling*
- *More timely than polling*
- *Provides a way of server inquiring about client status*

Callback minuses:

- *May leave server with inconsistent state if client crashes or exits without notifying the server*
- *Requires the server to make a series of synchronous RMI's*

Leasing can overcome the first problem. We'll talk more about event notification to address the second problem.

Java object serialization:

- *flattens object(s) into compact form for disk storage or message transmission*
- *process doing deserialization has no knowledge of the object structure*

Example of a Java object:

```
public class Person implements Serializable {  
    private String name;  
    private String place;  
    private int year;  
    public Person (String aName, String aPlace, int aYear) {  
        name = aName;  
        place = aPlace;  
        year = aYear;  
    }  
    // methods  
}
```

Figure 4.9
Indication of Java serialized form

<i>Serialized values</i>				<i>Explanation</i>
Person	8-byte version number		h0	<i>class name, version number</i>
3	int year	java.lang.String name:	java.lang.String place:	<i>number, type and name of instance variables</i>
1934	5 Smith	6 London	h1	<i>values of instance variables</i>

The true serialized form contains additional type markers; h0 and h1 are handles

Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn. 3
© Addison-Wesley Publishers 2000

Java serialization details:

■ *Serialization file description:*

AC ED (magic number)

00 05 (version number of object serialization format)

■ *Object representation:*

73

class descriptor

object data

■ *Serial numbers:*

! *class descriptors and objects only appear once in the file*

! *they are assigned 4-byte serial numbers*

! *the next time a class or object is encountered, it is specified by the serial number rather than the class description.*

Java serialization details (cont):

■ *Class descriptor:*

72

2-byte length of class name

class name

8-byte fingerprint (based on first 8 bytes of HAS = Secure Hash Algorithm)

1-byte flag (classes that implement *Serializable* have a flag of 02)

2-byte count of data field descriptors

data field descriptors

78 (end marker)

superclass type or 70 if none

■ *If the same Class is used again in the file:*

71

4-byte serial number

Java serialization details (cont):

■ *Field descriptors:*

1-byte type code: (B = byte, C = char, D = double, ... [= array])

2-byte length of field name

field name

class name (if field is an object)

2-byte count of data field descriptors

data field descriptors

78 (end marker)

superclass type or 70 if none

■ *If the same Class is used again in the file:*

71

4-byte serial number

Java serialization details (cont):

■ *Array representation*

75
class descriptor
4-byte number of entries
entries

■ *Other data:*

00
data value

■ *Representation of unicode values uses Universal Transfer Format (UTF)*

Java serialization details (cont):

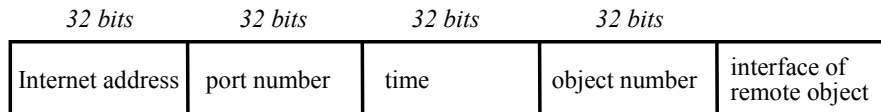
The definitive reference books for these topics are:

- *Core Java 2 Volume I: Fundamentals*
- *Core Java 2 Volume II: Advanced Features*

both books are by Cay S. Horstmann and Gary Cornell and published by Prentice Hall

Question: where is the information about class methods kept?

Figure 4.10
Representation of a remote object reference

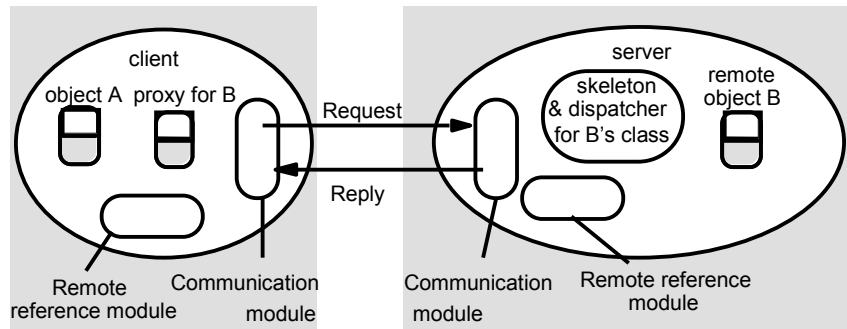


Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn. 3
© Addison-Wesley Publishers 2000

Java reflection:

- *Reflection is the ability to determine the properties of a class dynamically*
- *The Java package `java.lang.reflect` contains tools for analyzing classes at runtime*
- *Implements the following classes:*
 - *Method*
 - | *the objects can act like function pointers in C++*
 - | *have methods for returning the return type and types of parameters*
 - *Field – the objects give information about the field*
 - *Constructor – represents information about the constructor of the object*
 - *Class – has methods such as `getMethods()` that returns an array of the Method objects for the public methods of the class, etc.*

Figure 5.6
The role of proxy and skeleton in remote method invocation



Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn. 3
© Addison-Wesley Publishers 2000

RMI Software:

- *Proxy* – makes remote invocation “transparent” by handling marshalling, unmarshalling sending and receiving message
- *Dispatcher* – a server has a dispatcher and skeleton for each class representing a remote object.
- *Skeleton* – the class of each remote object has a skeleton that unmarshals arguments from request, invokes the method of the remote object, and marshals the results.

RMI servers and clients:

■ *Server:*

- *contains classes for dispatchers, skeletons and remote objects*
- *has initialization section for creating some remote objects*
- *registers some remote objects with the binder*

■ *Client:*

- *contains classes for proxies of all remote objects*
- *uses the binder to look up remote object references*
- *cannot create remote objects by directly calling constructors – calls factory methods instead*

Remote invocation and threads:

- *Servers are sometimes implemented so that remote invocation causes a new thread to be created to handle the call.*
- *A server with several remote objects might also allocate separate threads to handle each object.*

What are the advantages/disadvantages of each approach?

What other threading organizations could be used?

Activation of remote objects:

- *active objects* – available for invocation within a running process
- *passive objects* – can be made active (has state in marshalled form)
- *activation* – the process of creating an active object from a passive one:
 - ┆ creates a new instance of class
 - ┆ initializes it from the stored state
- *activator* –
 - ┆ records info about passive objects available for activation
 - ┆ starts named server processes and activates remote objects
 - ┆ records the location of servers that have already been activated

For next time:

- *Read CDK 4.3-4.4*
- *Read CDK 5.3*