COMP2207
Serialisation & Remote Reference Passing

Professor Timothy Norman

Department of Electronics and Computer Science, University of Southampton
T.J.Norman@soton.ac.uk

Academic Year 2017/18
How do we pass information between client and server?
Byte Streams for Method Parameters

- The stub needs to serialize input parameters to a remote method call.
- The dispatcher needs to deserialize input parameters in order to invoke the remote object.
- The reverse is required for any return parameters.
- So far, we have considered only java.lang.String.
- What about more complex, programmer-defined objects?
  - Need mechanisms to convert objects to/from byte streams.
Serializable Interface

- Java provides a simple means for object serialisation

```java
import java.io.Serializable;

/**
 * Note American spelling of Serializable
 */

public class MyClass implements Serializable {
    // Recommended to avoid unexpected InvalidClassExceptions
    private static final long serialVersionUID = 1L;
    // Other state and behaviour of the class
}
```

- If a Java class implements this interface, the compiler will generate bytecode to handle serialisation/deserialisation

- Assumptions:
  - We have a JVM at each end of the communication channel
  - The same version of the class is known at both ends
Java Streams

- Java streams are used for:
  - Scanning and formatting
  - I/O from the standard input
  - Sending/receiving values and objects via a network

- You’ve met them before
  **Byte streams** FileInputStream, FileOutputStream
  **Character streams** FileReader, FileWriter
  **Buffered streams** BufferedReader, BufferedWriter
  **Object streams** ObjectInputStream, ObjectOutputStream

- ...and you may have used some of the methods to read/write objects
  - readObject()
  - writeObject()

1. FileOutputStream myout = new FileOutputStream( "tmpfile.dat" );
2. ObjectOutputStream mystrm = new ObjectOutputStream( myout );
3. mystrm.writeObject( new MyClass() );
4. mystrm.flush();
Custom Serialisation

- Implementing the `java.io.Serializable` interface gives you default structure for the byte stream
- This can be customised, however...

```java
import java.io.Serializable;
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;

public class MyClass implements Serializable {
    private static final long serialVersionUID = 2L;

    private void writeObject( ObjectOutputStream out ) throws IOException {
        // Custom solution
    }

    private void readObject( ObjectInputStream in ) throws IOException, ClassNotFoundException {
        // Custom solution
    }
}
```
Mobile Code

We can use serialisation to create mobile code

- Client sends a message object to server that contains behaviour
  - e.g. execute DB query and filter results to generate summary
- Server calls a method on the received object to execute the behaviour
  - e.g. summary recorded in the object state
- The message object may then...
  - ...be sent back to the client, or
  - visit the next server
Task Example

- Let's define a simple, generic interface for tasks (NB: not Remote):

```java
public interface Task {
    public Object execute();
}
```

- ...and an implementation:

```java
public class ShoutTask implements Task, Serializable {
    private static final long serialVersionUID = 2L;
    private String _msg = null;

    public ShoutTask(String message) {
        this._msg = message;
    }

    public Object execute() {
        return _msg.toUpperCase();
    }
}
```
Another Task

- We may define any number of implementations of the Task interface

```java
public class LinesTask implements Task, Serializable {
    private static final long serialVersionUID = 2L;
    private String _msg = null;
    private int _n = 0;

    public LinesTask ( String message, int num )
    {
        this._msg = message;
        // We might want to check that num >= 1!
        this._n = num;
    }

    public Object execute()
    {
        String lines = new String( _msg );
        for( int i=1; i<_n; i++ )
        {
            lines += "\n" + _msg;
        }
        return lines;
    }
}
```
An Executor of Anonymous Tasks

- **Remote** interface:

```java
public interface ExecInterface extends Remote {
    public Object processTask(Task task) throws RemoteException;
}
```

- **...and its implementation:**

```java
public class ExecImpl implements ExecInterface {
    public Object processTask(Task task) throws RemoteException {
        return task.execute();
    }
}
```

- Any object that **implements** Task (and, of course, Serializable) can be sent to this remote object to have its `execute()` method run.

Tim Norman (ECS)
Exercise

- Download the code from
  https://secure.ecs.soton.ac.uk/notes/comp2207/tasks.zip

- Implement a task as follows:
  - The constructor expects a String and an int and stores these as member variables
  - The execute() method does this:
    - Convert the string to uppercase
    - Remove all non-alphabetic characters
    - Encode the message using a simple Caesar cipher determined by the int (% 26, of course); e.g. if the int is 3, an ‘A’ becomes a ‘D’ and a ‘Z’ becomes a ‘C’
    - Return the ciphertext
  - For example, we want be able to do this in the client, where stub is of type ExecInterface:

```
1 CeasarTask t = new CeasarTask( "1.Write a list of things to do; and 2.Do them!", 3 );
2 System.out.println( stub.processTask( t ) );
```

- ...it should return “ZULWHDOLVWRIWKLQJVWRGRDQGGRWKHP”
Java Parameter Passing

- Java uses references to objects
- ...but references to objects are passed by value in a method call

```java
public class Eg {
    class MyClass {
        public int _x;
        public MyClass( int x ) { this._x = x; }
    }
    public void swap( MyClass a, MyClass b ) {
        a._x = 100; MyClass tmp = a; a = b; b = tmp;
    }
    public void go() {
        MyClass one = new MyClass( 0 );
        MyClass two = new MyClass( 0 );
        System.out.println("one=" + one._x + ";\t\two=\" + two._x);
        swap( one, two );
        System.out.println("one=" + one._x + ";\t\two=\" + two._x);
    }
    public static void main( String[] args ) {
        (new Eg()).go();
    }
}
```

one=0; two=0
one=100; two=0
Reference or Value?

- With **serialisation** we’re passing a **copy** of the whole object…
- … **not** just a copy of the reference to the object…
- … but we **can** do that **if** the reference is to a **remote object**
Reference or Value?

- With serialization we’re passing a copy of the whole object...
- ...not just a copy of the reference to the object...
- ...but we can do that if the reference is to a remote object

```java
public interface AcctInterface extends Remote {
    public Float balance() throws RemoteException;
}
```

```java
public interface BankInterface extends Remote {
    public AcctInterface getAccount( int number ) throws RemoteException;
}
```
The Bank implementation passes references to remote account objects.
Both client and server have their own copy of a reference to the same account object.
Callbacks

- We can go further...
Callbacks

- We can go further...
  - Client creates a local remote object “callback”
  - Calls remote server object with reference to callback object
  - Call to remote server returns, but record of callback object reference is retained
  - When server has done work, method on callback object is called

- In RMI, client’s calling thread waits until remote method returns
- Callbacks allow some decoupling of client and server
Recall that we want to design systems that are robust to failures.

What if the behaviour of a remote object depends on past remote method calls as well as parameters passed?

If the server crashes, when restarted its behaviour may vary...
Recall that we want to design systems that are robust to failures.

What if the behaviour of a remote object depends on past remote method calls as well as parameters passed?

If the server crashes, when restarted its behaviour may vary...

... unless we build in state persistence.

The norm, therefore, is to store the state of important objects in databases.
Activation

- We also want to design systems that are long-lived.
- But, specific services may not be always required.
- If a service is used infrequently or in bursts of demand, do we want to have it running continually?
  - Consuming memory — can we use our computational resources more efficiently?
  - Consuming energy — can we be more green?
Activation

- We also want to design systems that are long-lived
- But, specific services may not be always required
- If a service is used infrequently or in bursts of demand, do we want to have it running continually?
  - Consuming memory — can we use our computational resources more efficiently?
  - Consuming energy — can we be more green?
- The RMI activation service
  - Generate remote objects, triggered by requests to references for them
  - Restart remote objects if they have shut down or have crashed
We want to create objects that have all the functionality for the system to activate them on demand:

- Extend `java.rmi.Activatable`
- Define how state is saved on shut down
- Define how state is recovered on restart
- Register remote object with the activation service — `rmid` (RMI Daemon)

Unlike `UnicastRemoteObjects`, a reference to an activatable remote object does not need to have a live object behind it

The reference is persistent
Summary

- We may pass more complex, **programmer-defined**, objects via serialisation
  - A copy of the **whole object** is passed from client to server or vice versa
- We may pass **references** to **remote objects** as parameters
  - Both client and server may interact with the **same object** via its remote interface
- Callbacks can help to **decouple** client and server
- **Persistence** and the **activation service** can enable us to build more **robust** and **scalable** systems