The purpose of this worksheet is to teach the fundamentals of distributed objects as implemented by Java’s Remote Method Invocation package.

Note: frequent reference is made to a machine called koestler. You will need to replace this name with the name of the machine you are using, or to localhost, if you are testing on a single machine.

1 RemoteThing

Java’s RMI package allows us to run a server, which manages objects on one machine and a client which can access those objects and make use of their methods from another machine.

This first example is as straightforward as distributed objects can be. We will create a remote object of type RemoteThing and a server for it. We will build a client which accesses the server to illustrate the simplest of distributed object concepts, that of invoking a method on an object on a different computer. This will serve to illustrate the minimum set of mechanisms required to configure the Java RMI environment. In particular, we will see that the machine which is running the server also needs to run the rmiregistry application, where the global names of remotely accessible objects will be published.

We begin by defining an interface which will be implemented by the object to be accessed remotely.

```java
import java.rmi.*;

public interface RemoteThingInterface extends Remote {
    public String accessThing()
        throws RemoteException;
}
```

The interface must extend the Remote interface which is part of the java.rmi package. Public methods of objects which implement this interface must throw RemoteException. This is an exception which will be handled by the client. You can’t avoid declaring it, because it will be thrown by the underlying RMI implementation when something is wrong (e.g. your server isn’t running).

Next we define the object to be distributed.
import java.rmi.*;
import java.rmi.server.*;

public class RemoteThing extends UnicastRemoteObject
    implements RemoteThingInterface {

    public RemoteThing() throws RemoteException { super(); }

    public String accessThing() throws RemoteException{
        return "Hello Remote World";
    }
}

The RemoteThing class extends the UnicastRemoteObject class, which is part of the java.rmi.server package. This means that we intend that every instance of this class, that is every RemoteThing will be unique (i.e. not replicated) and so messages to it can be directed to this single copy. The public methods of this class must throw RemoteException for the same reason as before, this exception may be thrown by the underlying implementation and must be handled by the client. The constructor must call the constructor of its superclass. The public method accessThing is the heart of our application. It implements the specification given in RemoteThingInterface, which is to return a string to tell us of its existence.

Next we need a server, which will create an instance of the object to be accessed remotely and register it under a global name that the client can use to find it.

import java.rmi.*;
import java.rmi.server.*;

public class RemoteThingServer {

    public static void main(String[] args){
        try{
            RemoteThing remoteThing=new RemoteThing();
            Naming.rebind("remoteThing",remoteThing);
            System.out.println("remoteThing ready");
        } catch(Exception e){}
    }
}

This is an application that creates a single instance of RemoteThing, which it calls remoteThing. Next, it registers the object with a Naming Service. The class Naming is obtained from the java.rmi package. It has methods for binding local object references to global names (which we do here) and for looking them up remotely (which we will do in the client). The method rebind registers the
remoteThing object under the global name "remoteThing", although we could have used any string we wished to refer to it.

When the server is run, assuming a Naming Service is also running on the same machine (we will use rmiregistry), a binding between the global name "remoteThing" and the reference to remoteThing, which is local to the server, will be recorded. We use rebind so that any previous binding to this global name is overwritten.

Finally, we can write the client which will access this remotely accessible object.

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

public class RemoteThingClient {

    public static void main(String[] args){
        try{
            RemoteThingInterface remoteThing=
            (RemoteThingInterface)
            Naming.lookup("rmi://localhost/remoteThing");
            System.out.println("remoteThing found ");
            System.out.println(remoteThing.accessThing());
        } catch(Exception e){System.out.println(e);}
    }
}
```

Again, this is an application. It declares a variable of type RemoteThingInterface to hold the reference to the remote object. It then uses Naming.lookup to try to obtain a reference to an object bound to the name "rmi://localhost/remoteThing". This is in fact a URL which refers to the remote object stored by our server under the name "remoteThing". The URL includes the protocol (rmi) and the machine (localhost) on which the object has been deployed. We use localhost for testing. Should we have deployed the remote object elsewhere, we would need to put that machine name in the URL (e.g. "rmi://koestler.ecs.soton.ac.uk/remoteThing"). If the lookup is successful, an Object is returned which we must cast to the type we believe that object to be (if we’re wrong, we’ll get an exception). This client, when it is run, will either print errors if we haven’t set everything up right, or will print a message from the remote object.

Now we have the component parts of our distributed application, we must compile them in the right order.

To run these applications, we must proceed as follows. First we must run rmiregistry (part of the JDK distribution) on the machine on which we intend to run the server. Then we can start, in its own window, RemoteThingServer. This should tell us that it has successfully registered remoteThing with rmiregistry and that it is available for use. In a third window, run RemoteThingClient. Assuming everything has compiled correctly and been installed correctly, we should get a cheery message from the remote object.
Note: An alternative way of starting rmiregistry is to use:

```
LocateRegistry.createRegistry(port)
```

This will create a Registry instance that accepts requests on the specified port. (This is useful when a particular server running on a host needs to use its own instance of the registry.) A client can then obtain a remote reference to the Registry instance by providing a host name (String) and a port:

```
LocateRegistry.getRegistry(host, port)
```

**Exercise 1.** Compile and run the code exactly as presented here, on a single workstation. Now, deploy RemoteThingServer to another machine and run it there. Be careful to start rmiregistry on that machine first. Deploy the server by deploying all the class files bar the client. Don’t forget to change the machine name in the client. Experiment with the deployed files on the client and the server to see which are needed by each at run time.

**Exercise 2.** Add the following code to the accessThing method, in place of the existing return statement.

```java
String whoami="unknown";
try{whoami=InetAddress.getLocalHost().getHostName();}
catch(Exception e){}
return "Hello from "+whoami;
```

Don’t forget to include import java.net.*; Now you can run the server on different machines and it will tell you which machine it is on.

**Exercise 3.** Add a method to RemoteThing which allows you to give each instance a differentiating characteristic. For example, give the instance a String which is used to identify it. Now, extend RemoteThingServer to create and register two different instances of RemoteThing and extend the RemoteThing-Client to access them. Thus show that the two remote instances are separate objects.

**Exercise 4.** Now modify RemoteThing so that as well as being able to set its unique differentiating characteristic, you can retrieve that remotely too. Create two instances in the server and access them both from the client. Now, by calling these remote methods in the following way (for example):

```java
Thing1_identity=remoteThing1.getIdentity();
Thing2_identity=remoteThing2.getIdentity();
remoteThing1.setIdentity(Thing2_identity);
remoteThing2.setIdentity(Thing1_identity);
```
the client can be made to switch their identities on each run. If we now put a
loop into the client, so that it does this repeatedly, and if we run two copies of
the client accessing the same server, we should be able to observe interference
between the clients. Eventually, both remote objects will end up with the same
differentiating characteristic. You can accelerate this experiment by making the
client sleep (for a random time) between updating the two objects.

Exercise 5. Devise a way to prevent the interference seen in Exercise 4.

Exercise 6. Start again from the simple server which has just a single instance
of RemoteThing. Modify the definition of RemoteThing so that it takes time
to execute and prints messages on entry and exit. Thus demonstrate that two
clients can be executing the same remote method at the same time. Show that,
by making the method synchronized, this interference can be excluded. What
does this execution tell you about how threads are allocated in the server?

2 MovableThing

The purpose of this example is to illustrate the relationship between objects
which are accessed using remote references and those which are copied from ma-
chine to machine. In the first example on this sheet, the only object transferred
between machines was a String, the result of the query remoteThing.accessThing().
This is a rather trivial example of the fact that Java will copy objects between
machines, where these objects are not explicitly remote objects. In the case of
remote objects of course, all that is copied between machines is a remote refer-
ce. We will construct a situation where movable objects refer to both remote
and movable objects, and see what happens when we move the movable objects
between machines.

We start by defining a MovableThing. This is any class which satisfies the
constraints that its objects are serializable. An object is serializable if it is a
simple type (number, string, etc) or an array, or an object all of whose fields
are serializable. A remote reference is serializable. Java will serialize a graph of
mutually referring objects and reconstitute that graph structure automatically.
You will see a little of that behaviour here.

```java
import java.net.*;
import java.io.*;
import java.rmi.RemoteException;

public class MovableThing implements Serializable {
    String myName;
    MovableThing mt;
    RemoteThingInterface rt;
}
```
public synchronized void setName(String myName){
    this.myName=myName;
}

public synchronized String getName(){
    String whereAmInow="location_unknown";
    try{whereAmInow=InetAddress.getLocalHost().getHostName();}
    catch (Exception e){}
    return this.myName+("+whereAmInow+");
}

public synchronized void setMT(MovableThing mt){
    this.mt=mt;
}

public synchronized MovableThing getMT(){
    return this.mt;
}

public synchronized void setRT(RemoteThingInterface rt){
    this.rt=rt;
}

public synchronized RemoteThingInterface getRT(){
    return this.rt;
}

public String describe() throws RemoteException{
    return getName() + "[" + ((mt==null)?"null":mt.describe()) + "," + ((rt==null)?"null":rt.describe()) + "]";
}

This is straightforward, if a little lengthy. We have given MovableThing an identifier which we refer to as myName. MovableThing will also refer to two other objects, one of type RemoteThing (or, more correctly RemoteThingInterface) and the other of type MovableThing. This will allow us to build simple graphs such as that which will be created by the following statements (taken from the client we will eventually build).

RemoteThingInterface rt2;
MovableThing mt1, mt2; ...

mt1.setRT(rt2);
mt1.setMT(mt2);
Thus mt1 is the root of a simple tree which has immediate descendents rt2 and mt2. You will see that when we move mt1 from one machine to another, (rt2 being a remote reference) only the reference to rt2 is moved, whereas mt1 and mt2, being local objects are copied in their entirety. Indeed, just moving mt1 causes mt2 to be moved as part of the serialization.

There are other methods of MovableThing which are of interest. When we call getName, the name which is returned will be decorated with the machine name of the machine which is supporting the object. When we copy the object from one machine to another, this annotation will change.

The describe method is used to ask an object to construct a string describing its contents. This will be needed when we have succeeded in building a graph structure which spans more than one machine. However, describe assumes that the graph contains no cycles, so take care.

Here is the interface for a new RemoteThing, which has a similar structure to MovableThing, the only difference being that RemoteThings refer only to MovableThings. That is sufficient for our experiment.

```java
import java.rmi.*;

class RemoteThing extends UnicastRemoteObject implements RemoteThingInterface{
  private String myName; private MovableThing mt;
  public RemoteThing() throws RemoteException { super(); }
}
```

The behaviour of the methods in this interface is best understood simply by studying its implementation.
public synchronized void setName(String myName)
    throws RemoteException{
    this.myName=myName;
}

public synchronized String getName()
    throws RemoteException{
    String whereAmI="location_unknown";
    try(whereAmI=InetAddress.getLocalHost().getHostName());
    catch(Exception e){}
    return this.myName+"("+whereAmI+"\n");
}

public synchronized void setMT(MovableThing mt)
    throws RemoteException{
    this.mt=mt;
}

public synchronized MovableThing getMT()
    throws RemoteException{
    return this.mt;
}

public String describe()
    throws RemoteException{
    return getName() + "[" + ((mt==null)?"null":mt.describe()) + "]";
}

As we promised, the RemoteThing is very similar to MovableThing. It can be
named and we can find out what that name is along with an indication of which
machine is supporting it. We can also ask it to describe its contents.

You may wonder why I haven’t derived these two very similar classes from a
superclass which combines their common behaviour. Well, I have left that as an
exercise. When you have done it, you will see that (elegant though the solution
may be) it obscures some of the behaviour which we are trying to illustrate.

The server for this distributed application is simplicity itself. Apart from
having a new name, it is identical to one we made earlier.

import java.rmi.*;
import java.rmi.server.*;

public class MovableThingServer {

    public static void main(String[] args){


try{
    RemoteThing remoteThing1=new RemoteThing();
    Naming.rebind("remoteThing1",remoteThing1);
    System.out.println("remoteThing1 ready");
    RemoteThing remoteThing2=new RemoteThing();
    Naming.rebind("remoteThing2",remoteThing2);
    System.out.println("remoteThing2 ready");
}catch(Exception e){System.out.println(e);}

We create and publish two remote objects. We will manipulate them from the client.

import java.rmi.*;
import java.rmi.server.*;

public class MovableThingClient {

    public static void main(String[] args){
        try{
            RemoteThingInterface rt1=(RemoteThingInterface)
                Naming.lookup("rmi://koestler.ecs.soton.ac.uk/remoteThing1");
            System.out.println("remoteThing1 found ");
            rt1.setName("rt1");

            RemoteThingInterface rt2=(RemoteThingInterface)
                Naming.lookup("rmi://koestler.ecs.soton.ac.uk/remoteThing2");
            System.out.println("remoteThing2 found ");
            rt2.setName("rt2");

            MovableThing mt1 = new MovableThing();
            mt1.setName("mt1");

            MovableThing mt2 = new MovableThing();
            mt2.setName("mt2");

            mt1.setRT(rt2);
            mt1.setMT(mt2);

            rt1.setMT(mt1); // <-- This is where the work is done

            System.out.println(mt1.describe());
            System.out.println(mt2.describe());
            System.out.println(rt1.describe());
            System.out.println(rt2.describe());

        }catch(Exception e){System.out.println(e);}
    }
}

execute \{\text{catch(Exception e)\{System.out.println(e);\}}\}
}

In the client we have assumed that the server is running on koestler.ecs.soton.ac.uk. The client obtains references to the two RemoteThings running in the server and then sets up two local MovableThings. Setting the fields of mt1 builds a local graph (just two nodes, try drawing it), but with one of those nodes (mt2) referring to a remote object (rt2).

Then we call rt1.setMT(mt1). This is where all the interesting things happen. Because mt1 is not a remote object, Java copies it to the server. It copies it by serializing it at the client and deserializing it at the server. Note, this means the server must have access to the MovableThing.class file. By inspecting the output from the client, you will see that there are small but significant differences between the structures in the client and server machines (assuming these machines are different).

**Exercise 7.** Start by compiling and running the applications exactly as they are here. Don’t forget that you need rmiregistry running on the server. After you have persuaded yourself that you understand the output from the client, try experimenting with more complex graph structures. Either, avoid cycles in the graph or modify describe to be able to avoid the problems they cause. For example, describe could print out only to a certain depth.

**Exercise 8.** Now, deploy the server on two machines, ideally running the client on a third machine. Modify the client to refer to remote objects in both servers and then copy the graph to both. Observe that the references to objects on different machines are global and can be freely moved between machines without any problems.