# Unit 4B. Java Remote Method Invocation: Further Look

**Outcomes Summary.** You will learn

- how to setup event notification in Java RMI
- how to setup and use a remote factory, why this is useful
- about the lifetime of remote objects and how to manage it
- what errors can occur during RMI and how they can be handled

**Further Reading:** Sun RMI Specs + Grosso 2001 Java RMI ch17,16.

In this unit we will see a slightly more advanced example Java RMI client-server system. The functionality of the system is still quite basic — it is a re-implementation of the duck searching server and client from Practical 3. Nevertheless, we will be careful to decompose the server sub-system using several general patterns that make enterprise-level systems more maintainable and scalable. Thus we will learn more about the design of distributed tiered systems as well as about how to use Java RMI.

## 4B.1 Notification

In an OO model it is common that an object A makes it possible for other objects to subscribe to be notified of certain changes in its state. We saw an example of this in the chat server to which clients subscribed for new messages. Whenever a thread executed the server's `sendMessage` method, the thread would be instructed to call the `newMessage` method for each subscribed client.

This idea works similarly in distributed and non-distributed setting. When the clients and server are in different JVMs, both the `subscribe` and `newMessage` calls are remote. We will now see another example of notification, which illustrates how to use a *dedicated listener object* to deal with the notification calls instead of adding the listening responsibility to an object that already has a number of other responsibilities, as happened to the chat client object.

### 4B.1.1 Player movement example

In the practical, when the client program created a new player, it called a server method `createPlayer` and was given a remote reference to the player. Then when it wanted to move the player, it had to call a server's method `movePlayer` and pass the remote reference to it. The server then called the `move` method of the player and also checked whether the player caught the duck or not.

It is desirable that clients are allowed to call the `move` method of the player directly — it is more convenient for clients and gives better cohesion in the design. Nevertheless, the server must be notified of the movement to allow it to update the image on screen and evaluate the move (see Figure 4B.1 on the left).

It is not very wise to make the server object listen to the movement notifications of all players. The notification would need to identify which player moved and the handler would need to look it up. It would be similar to the dreaded deprecated practice of making a Swing GUI frame listen to all the events generated by all buttons and other widgets within it using a complicated handler with very many cases inside. Like in the GUI programming model, there is a better solution — create a dedicated handler object for each component (ie for each widget in a GUI, for each player in our server — see Figure 4B.1 on the right).

The usual trick to define such listeners as anonymous inner classes would not always work here. For example, the following attempt:

```
1   private void subscribeToPlayer(final PlayerInterface player)
2       throws RemoteException
3   {
4       player.subscribe
5       (
6           new PositionListenerInterface()
7           {
8               public String newPosition(int x, int y)
9                   throws RemoteException
10              {
11                  return playerMoved(player);
12              }
13          }
14      );
15  }
```

would work only as long as the player is in the same JVM as the listener. The anonymous class defined on lines 6–12 implements `PositionListenerInterface` but does not extend `UnicastRemoteObject` and thus its instance (ie the listener) is not a remote object and the remote player cannot connect to it. A solution that works for remote notification is shown in Figure 4B.1 — it involves creating a named inner class for the listeners.

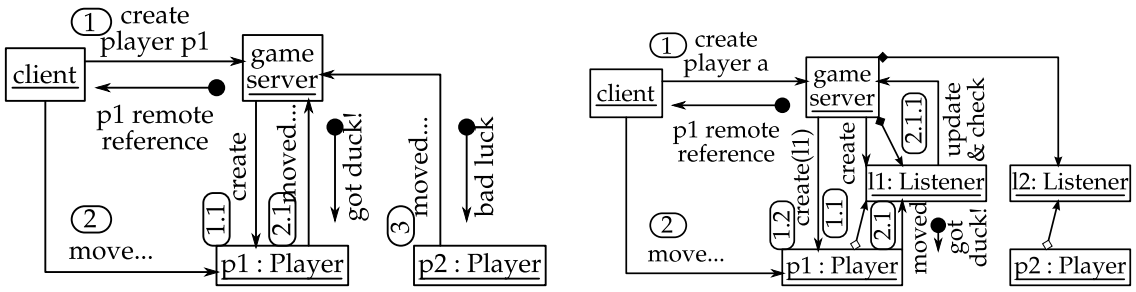
Notice that the inner class `PlayerListener` cannot be moved to a separate file because it is a true inner class — it contains a call to the server's instance method `playerMoved`. Thus `PlayerListener` is not only one class but a family of almost identical classes. The classes differ in which server instance their method `newPosition` calls.

## 4B.2 Factories

A factory object is usually responsible for creating and managing instances of another class. It can also manage a naming scheme for instances it created so that users of the factory can refer to these instances by some kind of convenient names, usually strings.

Why do we need factories? Can we not simply use class constructors and the usual local or remote references as names? Most of the time yes, but sometimes a factory can provide benefits, eg it:

- allows the sharing of instances through their *factory names*;




```java
public interface PositionListenerInterface extends Remote
{
    /**
     * A method by which a movement listener is notified
     * about a new position.
     * @return a report on the consequences of the new position
     * @param x new horizontal coordinate
     * @param y new vertical coordinate
     * @throws RemoteException
     */
    public String newPosition(int x, int y) throws RemoteException;
}
```

somewhere on the server:

```java
    private void subscribeToPlayer(final PlayerInterface player)
        throws RemoteException
    {
        player.subscribe(new PlayerListener(player));
    }

    private class PlayerListener // inner class
        extends UnicastRemoteObject
        implements PositionListenerInterface
    {
        private static final long serialVersionUID = 7552567727669253086L;

        private PlayerInterface player;

        protected PlayerListener(PlayerInterface player)
            throws RemoteException
        {
            super();
            this.player = player;
        }

        public String newPosition(int x, int y)
            throws RemoteException
        {
            return playerMoved(player); // method of outer class
        }
    }
```

Figure 4B.1: Notification of player's movement — using dedicated listeners.

```
1  public class PlayerFactory
2      implements PlayerFactoryInterface
3  {
4      private Map<String,Player> players;
5      private int xMin, yMin, xMax, yMax;
6      private Random generator;
7
8      public PlayerFactory()
9      {
10         super();
11         generator = new Random();
12         players = new HashMap<String, Player>();
13     }
14
15     public void newBounds(int xMin, int yMin, int xMax, int yMax)
16     {
17         this.xMin = xMin;
18         ...etc...
19     }
20
21     public synchronized PlayerInterface getPlayer(String name)
22     {
23         Player p = players.get(name);
24         // if it has not been found, create it:
25         if (p == null)
26         {
27             p = createPlayer(name);
28             players.put(name, p);
29             relocatePlayer(name);
30         }
31         return p;
32     }
33
34     public void relocatePlayer(String name)
35     {
36         Player p = players.get(name);
37         // find a vacant space on the board:
38         int x, y;
39         do
40         {
41             x = xMin + generator.nextInt(xMax - xMin + 1);
42             y = yMin + generator.nextInt(yMax - yMin + 1);
43         }
44         while (squareOccupied(x, y));
45
46         p.setX(x); p.setY(y);
47     }
48
49     // check to see if a square on the grid is already occupied
50     private boolean squareOccupied(int x, int y)
51     {
52         cycle through players checking their location
53     }
54
55     private Player createPlayer(String name)
56     {
57         return new Player(...parameters...);
58     }
59 }
```

Figure 4B.2: A simple player factory.

- allows the construction of an instance to be aware of *previously created instances*;
- can improve efficiency by *reusing instances* that would otherwise be garbage collected;
- allows the complete *encapsulation* of the class code, hiding it from all its users.

Some of these benefits will be clear from the following example.

Figure 4B.2 shows a simple factory for players that could be used by the duck-searching server. It allows one to obtain a player with a given name. If that player already exists, it returns that player, otherwise it creates it.

This shows how a factory can serve as our own registry of shared objects. (If sharing is not desirable, the factory can still be useful but it would be changed so that `getPlayer` never reuses players unless they are on an "disused" list. Such recycling can still increase the speed of `getPlayer`.)

In a DS context, the factory itself can be made an RMI remote object. This can entirely hide the `Player` class from the classes that use players. Thus not only the client but also the server will have no knowledge of the internal structure of `Player` objects, giving looser coupling. The operation of a remote player factory is illustrated in Figure 4B.3.

## 4B.3 Lifetime of remote objects

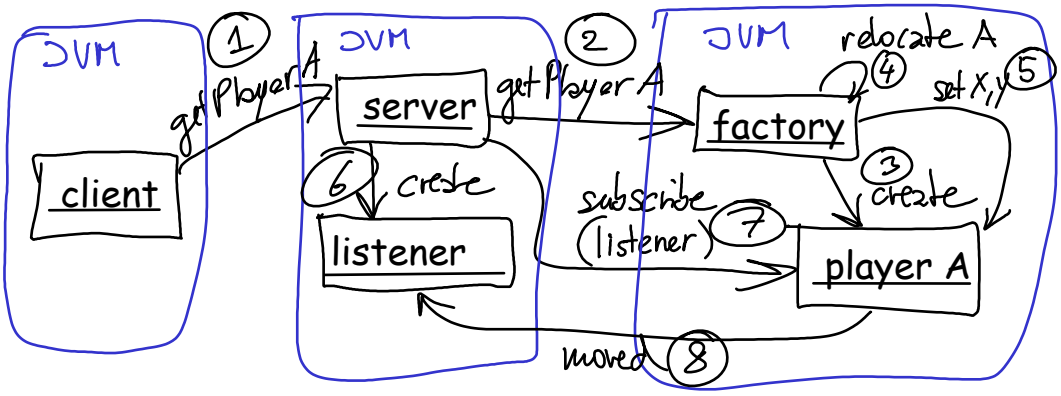
Java programmers can write code to create new objects but they do not worry about the rest of the object's lifetime — they can assume the object will be there as long as they need it. The only time they are sure the objects disappear is when the JVM shuts down. With remote objects, the situation becomes a bit more complicated.

### 4B.3.1 Distributed garbage collection

To save memory, the JVM garbage collects all objects that have become clearly completely inaccessible by any of the running threads. This happens eg when the object was assigned to a field and later the field was overwritten with another value; or when it is assigned to a local variable inside some method and then the method is finished and the variable forgotten by the executing thread.

The garbage collection mechanism has been extended for remote objects to take account of any remote references. When a JVM (call it JVM A) obtains a remote reference (either as a parameter of an incoming RMI call or as a return value of its RMI call, which includes using the registry), JVM A requests a *lease* for the remote object from its hosting JVM (call it JVM B). Therefore, JVM B will not garbage collect the object even if there are no uses for it inside JVM B. When JVM A garbage collects the remote reference, it notifies JVM B that its lease is no longer required.

As the name suggests, a lease is not permanent. It has to be renewed at regular intervals. When JVM A does not renew the lease, after some time (around 15 minutes by



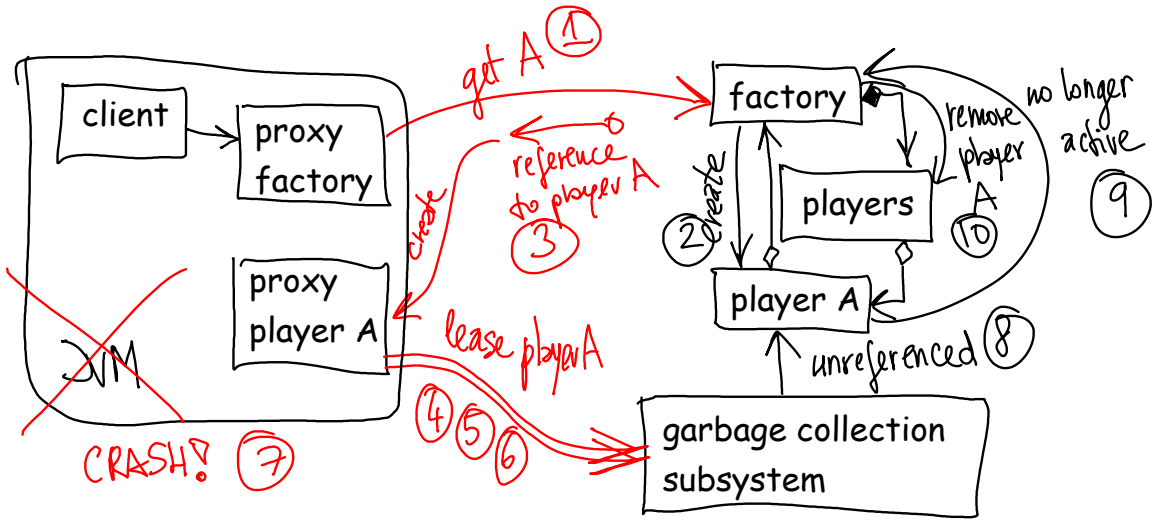

```
1  public interface PlayerFactoryInterface extends Remote
2  {
3      PlayerInterface getPlayer(String name) throws RemoteException;
4
5      void relocatePlayer(String name) throws RemoteException;
6
7      void newBounds(int xMin, int yMin, int xMax, int yMax)
8          throws RemoteException;
9
10     void newColourLimit(int colLimit) throws RemoteException;
11 }
```

```
1  public interface PlayerInterface extends Remote
2  {
3      public static final int numberOfFrames = 4;
4
5      String move(Direction d) throws RemoteException;
6
7      public int getXPos() throws RemoteException;
8      public int getYPos() throws RemoteException;
9      boolean atLocation(int x, int y) throws RemoteException;
10
11     public void subscribe(PositionListenerInterface listener)
12         throws RemoteException;
13
14     public int getStep() throws RemoteException;
15     public Direction getDirection() throws RemoteException;
16     public int getColour() throws RemoteException;
17     public String getName() throws RemoteException;
18 }
```

```
1  public interface PositionListenerInterface extends Remote
2  {
3      public String newPosition(int x, int y) throws RemoteException;
4  }
```

```
1  public enum Direction implements Serializable
2  {
3      LEFT, RIGHT, UP, DOWN
4  }
```

Figure 4B.3: Player creation using player factory and all player-related code given to the server.




```
public class Player
    extends UnicastRemoteObject
    implements PlayerInterface, Unreferenced
{
    ...other fields...

    private PlayerFactory factory;

    ...other methods...

    public void unreferenced()
    {
        try
        {
            factory.playerNoLongerActive(name);
        }
        catch (RemoteException e)
        {
            e.printStackTrace();
        }
    }
}
```

```
public class PlayerFactory
    implements PlayerFactoryInterface
{

    ...fields and other methods...

    public void playerNoLongerActive(String name) throws RemoteException
    {
        players.remove(name);
        System.out.printf("playerNoLongerActive(%s): ok\n", name);
    }
}
```

Figure 4B.4: Implementation of automatic deactivation of players in a player factory.

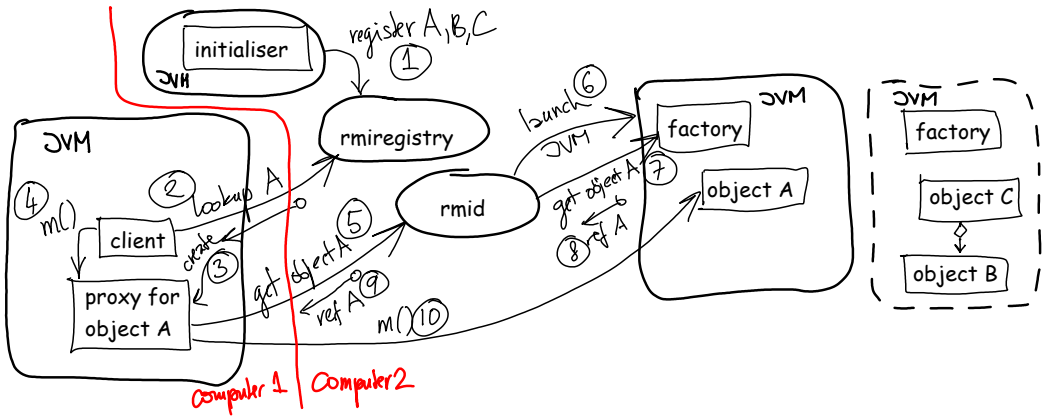



Figure 4B.5: How activation works

default) JVM B will conclude that JVM A no longer needs the object. Thus JVM B will be able to garbage collect the object even when its lease holders crash or lose network connection.

Java RMI offers a mechanism by which the programmer of JVM B can influence what happens when an local object that had remote references loses all leases and becomes free of remote references. If the object implements the interface `java.rmi.server.Unreferenced`, it has to have a method `unreferenced` and this method will be automatically called by one of the JVM's threads when the object loses all remote references. Via this method, it is possible to arrange that the object loses all its local references so that it can be garbage collected.

This is particularly useful for a remote factory, such as the one we have in Figure 4B.2. When the factory has given access to one of its players to remote users, it may wish to know when the users are done with it. Then it could, for example, remove the player from its internal index of players or move it on a special list of inactive players ready for recycling. Figure 4B.4 shows extracts of code implementing this idea.

### 4B.3.2 Activation framework and persistence

A remote object can disappear due to a failure of the hosting JVM or it may seem to disappear due to a network failure. Moreover, the two failures may become indistinguishable for other JVMs. One way to try to alleviate the problem with failing hosts is to make their crucial remote objects persistent so that their lifetime extends the lifetime of the JVM. (Java 2 enterprise edition (J2EE) includes a persistence mechanism that is fairly transparent to the user — nevertheless, we will not study it here.)

Persistence is particularly useful in conjunction with the Java RMI activation framework. In this framework, remote objects may be registered with rmiregistry before they are actually created. Moreover, the JVM hosting these objects may not exist. The JVM and the remote objects within it are launched only when another JVM requires them. Obviously there has to be another program that administers such launch of new JVMs and remote objects. The program is called the RMI daemon (rmid).

Figure 4B.5 illustrates a typical activation sequence. First, a name is registered in rmiregistry that is intended to correspond to an activatable remote object. When someone looks this name up, they are given a proxy for the object-to-be. This proxy is a bit more sophisticated then the proxies we talked about up until now — it does not hold a remote reference to its object, it will contact rmid to obtain the reference when required. When the proxy asks rmid for the reference, rmid selects an appropriate JVM for this object (and perhaps creates it) and uses a generic factory on the JVM to create the remote object.

## 4B.4 Remote exceptions

A remote method invocation could throw 2 kinds of exceptions:

- exceptions unrelated to RMI, ie exceptions thrown by the method on the remote JVM, remotely propagated to the RMI caller;
- RMI exceptions, ie exceptions caused by:
  - network configuration errors, eg:
    `java.rmi.ConnectException` (eg computer refused connection)
  - network failures, eg:
    `java.rmi.ConnectIOException` (eg timeout during connect)
    `java.rmi.MarshalException` (eg timeout during data exchange)
  - remote JVM crashes, updates, eg:
    `java.rmi.UnknownHostException` (eg computer renamed)
    `java.rmi.NoSuchObjectException` (eg restart, no persistence)
    `java.rmi.StubNotFoundException` (eg object no longer remote)

# 4B.5 Listings of a player factory

## 4B.5.1 Player factory interface

```
1  public interface PlayerFactoryInterface extends Remote
2  {
3      PlayerInterface getPlayer(String name) throws RemoteException;
4
5      void relocatePlayer(String name) throws RemoteException;
6
7      void newBounds(int xMin, int yMin, int xMax, int yMax)
8          throws RemoteException;
9
10     void newColourLimit(int colLimit) throws RemoteException;
11 }
```

## 4B.5.2 Player interface

```
1  public interface PlayerInterface extends Remote
2  {
3      public static final int numberOfFrames = 4;
4
5      public int getXPos() throws RemoteException;
6      public int getYPos() throws RemoteException;
7      boolean atLocation(int x, int y) throws RemoteException;
8      String move(Direction d) throws RemoteException;
9
10     public void subscribe(PositionListenerInterface listener)
11         throws RemoteException;
12
13
14     public int getStep() throws RemoteException;
15     public Direction getDirection() throws RemoteException;
16     public int getColour() throws RemoteException;
17     public String getName() throws RemoteException;
18     public Object getHost() throws RemoteException;
19 }
```

### 4B.5.3 Player factory class

```
1  public class PlayerFactory
2      extends UnicastRemoteObject
3      implements PlayerFactoryInterface
4  {
5
6      private static final long serialVersionUID =
7          -8403831721783321585L;
8
9      private Map<String,Player> players;
10     private int xMin, yMin, xMax, yMax;
11     private int numOfColours;
12
13     private Random generator;
14
15     public PlayerFactory()
16         throws RemoteException
17     {
18         super();
19         generator = new Random();
20         players = new HashMap<String, Player>();
21     }
22
23     public void newBounds(int xMin, int yMin, int xMax, int yMax)
24     {
25         this.xMin = xMin; this.yMin = yMin;
26         this.xMax = xMax; this.yMax = yMax;
27     }
28
29     public void newColourLimit(int colLimit)
30     {
31         this.numOfColours = colLimit;
32     }
33
34     public synchronized PlayerInterface getPlayer(String name)
35         throws RemoteException
36     {
37         Player p = players.get(name);
38
39         // if it has not been found, create it:
40         if (p == null)
41         {
42             p = createPlayer(name);
43             players.put(name, p);
44             relocatePlayer(name);
45         }
46
47         return p;
48     }
49
50     public void relocatePlayer(String name)
51         throws RemoteException
52     {
53         Player p = players.get(name);
54
55         if (p != null)
56         {
57             int x, y;
```

```
58
59              // find a vacant space on the board:
60              do
61              {
62                  x = xMin + generator.nextInt(xMax - xMin + 1);
63                  y = yMin + generator.nextInt(yMax - yMin + 1);
64              }
65              while (squareOccupied(x, y));
66
67              p.setX(x);
68              p.setY(y);
69          }
70      }
71
72      public void playerNoLongerActive(String name) throws RemoteException
73      {
74          players.remove(name);
75      }
76
77      // check to see if a square on the grid is already occupied
78      private boolean squareOccupied(int x, int y)
79          throws RemoteException
80      {
81          // cycle through players checking their location
82          for (Player p : players.values())
83          {
84              if (p.atLocation(x, y)) { return true; }
85          }
86
87          return false;
88      }
89
90      private Player createPlayer(String name)
91          throws RemoteException
92      {
93          Player newPlayer =
94              new Player
95                  (
96                          xMin, yMin,
97                          generator.nextInt(1000) % numOfColours,
98                          name,
99                          xMin, yMin, xMax, yMax,
100                         this
101                 );
102         return newPlayer;
103     }
104
105     public static void main(String[] args)
106         throws RemoteException, MalformedURLException
107     {
108         System.setSecurityManager(new RMISecurityManager());
109
110         PlayerFactory factory = new PlayerFactory();
111
112         Naming.rebind("PlayerFactory", factory);
113     }
114 }
```

### 4B.5.4 Playing field panel class

```
1  public class pnlPeople extends JPanel
2  {
3
4      private static final long serialVersionUID =
5         -6991217044763959938L;
6
7      private final int xGridSize = 30;
8      private final int yGridSize = 15;
9      private final int squareSize = 20;
10
11     private final Color backgroundColour = new Color(32, 96, 32);
12     private final Color[] colourList =
13         { Color.blue, Color.cyan, Color.orange, Color.red, Color.pink,
14                 Color.yellow, Color.pink, Color.gray, Color.magenta };
15
16     private BufferedImage img;
17     private Graphics gArea;
18     private MediaTracker mediaTracker;
19     private Image imgDuck;
20     private static Image[][] icons;
21
22     private static final String playerFactoryURI =
23         "rmi://localhost/PlayerFactory";
24     private PlayerFactoryInterface playerFactory;
25
26     private Map<String,PlayerInterface> players;
27     private PlayerInterface duck;
28     private int position;
29
30     public pnlPeople()
31         throws RemoteException, MalformedURLException, NotBoundException
32     {
33         super();
34         initialize();
35     }
36
37     private synchronized void initialize()
38         throws RemoteException, MalformedURLException, NotBoundException
39     {
40         mediaTracker = new MediaTracker(this);
41
42         // create an image that is the same size as field of play
43         img = new BufferedImage
44                     (xGridSize * squareSize, yGridSize * squareSize,
45                      BufferedImage.TYPE_INT_RGB);
46         gArea = img.getGraphics();
47
48         // create structures to keep track of players:
49         players = new HashMap<String,PlayerInterface>();
50         loadPlayerIcons();
51
52         playerFactory =
53             (PlayerFactoryInterface) Naming.lookup(playerFactoryURI);
54         playerFactory.newBounds(0, 0, xGridSize - 1, yGridSize - 1);
55         playerFactory.newColourLimit(colourList.length);
56
57         // Load duck icon:
```

```
58          imgDuck = Toolkit.getDefaultToolkit().getImage("images/duck.png");
59          mediaTracker.addImage(imgDuck, 0);
60          // create the "duck" player:
61          duck = playerFactory.getPlayer("duck");
62          placeDuck();
63
64          // initialise the position of the next person who finds the duck
65          position = 1;
66
67          try
68          {
69              mediaTracker.waitForID(0);
70          }
71          catch (InterruptedException e)
72          {
73              e.printStackTrace();
74          }
75
76          updateImage();
77      }
78
79      /**
80       * Loads the icons for the players
81       */
82      private void loadPlayerIcons()
83      {
84        body left out
85      }
86
87      private void subscribeToPlayer(final PlayerInterface player)
88          throws RemoteException
89      {
90          player.subscribe(new PlayerListener(player));
91      }
92
93      private class PlayerListener
94          extends UnicastRemoteObject
95          implements PositionListenerInterface
96      {
97          private static final long serialVersionUID = 7552567727669253086L;
98
99          private PlayerInterface player;
100
101         protected PlayerListener(PlayerInterface player)
102             throws RemoteException
103         {
104             super();
105             this.player = player;
106         }
107
108         public String newPosition(int x, int y)
109             throws RemoteException
110         {
111             return playerMoved(player);
112         }
113     }
114
115
116
```

```
117     private void placeDuck() throws RemoteException
118     {
119         playerFactory.relocatePlayer(duck.getName());
120     }
121
122     private synchronized void updateImage()
123         throws RemoteException
124     {
125         body left out
126     }
127
128     private void drawPlayer(PlayerInterface p)
129         throws RemoteException
130     {
131         int xPlot = p.getXPos() * squareSize;
132         int yPlot = p.getYPos() * squareSize;
133         gArea.drawImage
134             (
135                     icons[p.getDirection().ordinal()][p.getStep()],
136                     xPlot, yPlot, null
137             );
138
139         // if the Player is on the top line of grid, display number below
140         if (yPlot == 0)
141         {
142             yPlot = yPlot + 30;
143         }
144
145         // print the Player number
146         gArea.setColor(colourList[p.getColour()]);
147         gArea.drawString(p.getName(), xPlot, yPlot);
148     }
149
150     public void paintComponent(Graphics g)
151     {
152         g.drawImage(img, 0, 0, null);
153     }
154
155
156     private String playerMoved(PlayerInterface player)
157         throws RemoteException
158     {
159         String result;
160
161         synchronized (this)
162         {
163             if (player.atLocation(duck.getXPos(), duck.getYPos()))
164             {
165                 placeDuck();
166                 result =
167                    "Got the duck. Congratulations! You are placed: "
168                    + position;
169                 position++;
170             }
171             else
172             {
173                 result = "Unlucky! Try again!";
174             }
175         }
```

```
176
177         updateImage();
178
179         return result;
180     }
181
182     public synchronized PlayerInterface getPlayer(String name)
183         throws RemoteException
184     {
185         PlayerInterface p = playerFactory.getPlayer(name);
186
187         if ( players.get(name) == null ) // new player?
188         {
189             // new player must be registered for drawing:
190             players.put(name, p);
191             updateImage();
192             // ensure the big brother will be notified about all
193             // movements of this player:
194             subscribeToPlayer(p);
195         }
196
197         return p;
198     }
199
200     public synchronized void hidePlayer(PlayerInterface p)
201         throws RemoteException
202     {
203         players.remove(p.getName());
204         updateImage();
205     }
206 }
```