Departamento de Ingeniería Telemática
Escuela Superior de Ingenieros
Universidad de Sevilla

PRÁCTICA 6.

RMI: Paso de clases definidas por el usuario y tipos complejos

1. OBJETIVO

El objetivo es mostrar el paso de objetos como parámetro y como valores devueltos en los métodos remotos, tanto de clases definidas por el usuario como de tipos más complejos.

2. DESCRIPCIÓN

Tanto los parámetros de un método RMI como el resultado devuelto por el mismo pueden ser objetos de cualquier tipo, siempre que sean serializables (la mayoría de los objetos lo son, excepto aquellos que por su propia esencia estén vinculados con recursos locales y no tenga sentido transferirlos a otra máquina como, por ejemplo, un descriptor de fichero, un thread o un socket). Por tanto, en un método RMI se pueden usar objetos de clases definidas por el usuario y objetos de clases complejas, como puede ser un contenedor (ejemplo una lista): el proceso de serialización se encarga de empaquetar toda la información vinculada con ese objeto, de manera que luego se pueda recuperar en el mismo estado.

2.1 Paso de clases definidas por el usuario y tipos complejos: Servicio de banco

En esta sección, vamos a extender el servicio bancario visto en la práctica anterior de manera que utilice una clase definida por el usuario, así como una clase compleja, como puede ser una lista de cuentas bancarias.

En esta nueva versión, una cuenta bancaria va a quedar identificada por el nombre y el número de identificación del titular, en lugar de sólo por el nombre. Esta doble identificación va a quedar englobada en una nueva clase llamada Titular que representa al titular de una cuenta:

```
import java.io.*;

class Titular implements Serializable {
     private String nombre;
     private String iD;
     Titular(String n, String i) {
         nombre = n;
         iD = i;
     }
     public String obtenerNombre() {
         return nombre;
     }
     public String obtenerID() {
```

```
        return iD;
     }
     public String toString() {
        return nombre + " | " + iD;
     }
}
```

Titular.java

Como se puede observar, se trata de una clase trivial pero que presenta una característica importante: dado que se van a usar objetos de esta clase como parámetros y valores de retorno de métodos RMI, es necesario especificar que esta clase implemente la interfaz `Serializable`, declarando así el programador que esa clase puede *serializarse*. En esta nueva versión, la interfaz que declara una cuenta queda como sigue:

```
import java.rmi.*;

interface Cuenta extends Remote {

    Titular obtenerTitular() throws RemoteException;
    float obtenerSaldo() throws RemoteException;
    float operacion(float valor) throws RemoteException;

}
```

Cuenta.java

Y su implementación dada en el fichero `CuentaImpl.java`:

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

class CuentaImpl extends UnicastRemoteObject implements Cuenta {

    private Titular tit;
    private float saldo = 0;

    CuentaImpl(Titular t) throws RemoteException {
        tit = t;
    }

    public Titular obtenerTitular() throws RemoteException {
        return tit;
    }

    public float obtenerSaldo() throws RemoteException {
        return saldo;
    }

    public float operacion(float valor) throws RemoteException {
        saldo += valor;
        return saldo;
    }
}
```

CuentaImpl.java

Por otro lado, se ha modificado el servicio bancario para que almacene las cuentas creadas en un contenedor (concretamente, en una lista enlazada como en el ejemplo del servicio de chat) y ofrezca un nuevo método remoto que devuelta toda la lista de cuentas.

A continuación, se muestra la interfaz remota correspondiente a esta *fábrica* de cuentas:

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

interface Banco extends Remote {

    Cuenta crearCuenta(Titular t) throws RemoteException;
    List<Cuenta> obtenerCuentas() throws RemoteException;

}
```

Banco.java

Y la clase que implementa esta interfaz :

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

class BancoImpl extends UnicastRemoteObject implements Banco {

    List<Cuenta> l;

    BancoImpl() throws RemoteException {
        l = new LinkedList<Cuenta>();
    }

    public Cuenta crearCuenta(Titular t) throws RemoteException {
        Cuenta c = new CuentaImpl(t);
        l.add(c);
        return c;
    }

    public List<Cuenta> obtenerCuentas() throws RemoteException {
       return l;
    }
}
```

BancoImpl.java

Nótese cómo el nuevo método retorna directamente una lista. El proceso de *serialización* en el que se apoya RMI reconstruye automáticamente esa lista de referencias remotas en la JVM del cliente.

El servidor y el cliente son similares a los ejemplos anteriores y su código se muestra a continuación:

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

```
class ServidorBanco  {
    static public void main (String args[]) {
       if (args.length!=1) {
            System.err.println("Uso: ServidorBanco numPuertoRegistro");
            return;
        }
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        try {
            BancoImpl srv = new BancoImpl();
            Naming.rebind("rmi://localhost:" + args[0] + "/Banco", srv);
        }
        catch (RemoteException e) {
            System.err.println("Error de comunicacion: " + e.toString());
            System.exit(1);
        }
        catch (Exception e) {
            System.err.println("Excepcion en ServidorBanco:");
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

ServidorBanco.java

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

class ClienteBanco {
    static public void main (String args[]) {
        if (args.length!=4) {
            System.err.println("Uso: ClienteBanco hostregistro
numPuertoRegistro nombreTitular IDTitular");
            return;
        }

       if (System.getSecurityManager() == null)
            System.setSecurityManager(new SecurityManager());

        try {
            Banco srv = (Banco) Naming.lookup("//" + args[0] + ":" + args[1]
+ "/Banco");
            Titular tit = new Titular(args[2], args[3]);
            Cuenta c = srv.crearCuenta(tit);
            c.operacion(30);

            List <Cuenta> l;
            l = srv.obtenerCuentas();
            for (Cuenta i: l) {
                Titular t = i.obtenerTitular();
                System.out.println(t + ": " +i.obtenerSaldo());
            }

            c.operacion(-5);
```

```
            l = srv.obtenerCuentas();
            for (Cuenta i: l)
                System.out.println(i.obtenerTitular() + ": "
+i.obtenerSaldo());
        }
        catch (RemoteException e) {
            System.err.println("Error de comunicacion: " + e.toString());
        }
        catch (Exception e) {
            System.err.println("Excepcion en ClienteBanco:");
            e.printStackTrace();
        }
    }
}
```

ClienteBanco.java

En este punto se considera conveniente incidir en la diferencia que existe entre cuando se usan en un método RMI referencias a objetos remotos y cuando se utilizan referencias a objetos convencionales.

Vamos a fijarnos en ese extracto previo en la variable `c` y en la variable `t`. Ambas almacenan una referencia a un objeto retornado por un método RMI (`crearCuenta` y `obtenerTitular`, respectivamente). Sin embargo, hay una diferencia muy importante:

- La variable `c` guarda una referencia a un objeto remoto que implementa la interfaz `Cuenta`. Todas las llamadas a métodos que se hagan usando esa referencia (por ejemplo, `c.operacion(30)`) se convierten en operaciones remotas que, en caso de que provoquen algún tipo de actualización, repercutirá directamente sobre el objeto en el servidor.
- La variable `t` guarda una referencia a un objeto local que es una copia del objeto en el servidor. Todas las llamadas a métodos que se hagan utilizando esa referencia (nótese que la sentencia `println(t...` invoca automáticamente al método `toString` del objeto) serán operaciones locales y, por tanto, en caso de que causen algún tipo de actualización, no afectará al objeto en el servidor.

Como comentario final, tenga en cuenta que el fichero `class` que define el titular de una cuenta (fichero `Titular.class`) tiene que estar presente en las máquinas donde se generarán y ejecutarán los programas cliente y servidor.

⇒ 1. Descargue el fichero de código correspondiente a esta práctica. Compile y ejecute este ejemplo.

⇒ 2. Partiendo de la fábrica de logs desarrollada en la práctica 5 correspondiente a RMI, modificarla para que mantenga información de los objetos ServicioLog que ha creado. El cliente para la creación del log le debe proporcionar el nombre del fichero de log y un nombre que identifique la aplicación. Para almacenar esta información se utilizará la clase Log. En la creación el cliente debe pasar un objeto del tipo Log. La fábrica de logs se debe ampliar para que permita obtener la lista de todos los logs creados.

A continuación se dan las interfaces de FabricaLog y ServicioLog. El Servicio de log mantiene la fecha de creación del objeto servicio que es proporcionado en la creación del servicio por la fábrica.

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

interface FabricaLog extends Remote {
   ServicioLog crearLog(Log log) throws RemoteException;
   List<ServicioLog> obtenerLogs() throws RemoteException;
}
```

FabricaLog.java

```
import java.rmi.*;
import java.util.Date;

interface ServicioLog extends Remote {
      void log (String m) throws RemoteException;
      Log obtenerLog() throws RemoteException;
      Date obtenerFecha() throws RemoteException;
}
```

ServicioLog.java

La clase Log será la siguiente:

```
import java.io.*;

class Log implements Serializable {
     private String nombreFichero;
     private String idApp;
     Log(String nomFich, String id) {
         nombreFichero = nomFich;
         idApp = id;
     }
     public String obtenerNombreFichero() {
         return nombreFichero;
     }
     public String obtenerIdApp() {
         return idApp;
     }
     public String toString() {
         return nombreFichero + " | " + idApp;
     }
}
```

Log.java

⇒ 3. En la tarea habilitada para la práctica, entregue un fichero comprimido (.zip) con nombre su login que sea el directorio de nombre su login comprimido (que contiene el directorio p6RMIcodigo con todo su trabajo).