→ Java RMI - Remote Method Invocation

→ Callbacks

Vimos, na folha prática anterior, um exemplo muito simples de uma aplicação cliente/ servidor em que o cliente acede à referência remota de um objecto servidor e invoca métodos remotos sobre esse objecto.

Pode acontecer que o servidor precise, por sua vez, de invocar um ou mais métodos no cliente, tornando-se o cliente num servidor e o servidor num cliente. Quando o servidor invoca um método do cliente diz-se que ocorre um "callback". Por exemplo, se o cliente quiser ser automaticamente informado de alterações que ocorram no servidor, o servidor terá que aceder a um método do cliente.

Implementar um callback:

A) O cliente terá que implementar uma interface remota

. Definir a interface remota com os métodos que poderão ser acedidos remotamente

B) O cliente tem que estar disponível como servidor

. Isto é, ser subclasse da classe java.rmi.server.UnicastRemoteObject
(equivalente a exportar o cliente como um objecto remoto)

C) Passar referência remota do cliente para o processo servidor

. Assim, o servidor poderá usar essa referência para fazer invocações no cliente.

1 – Estude o exemplo que se segue, construa a classe cliente, e implemente-o.

Interface do servidor

```
public interface Hello_S_I extends java.rmi.Remote {

  public void printOnServer(String s) throws java.rmi.RemoteException;
  public void subscribe (String s, Hello_C_I cliente) throws java.rmi.RemoteException;
}
```

Interface do cliente

```
public interface Hello_C_I extends java.rmi.Remote {
  public void printOnClient (String s)throws java.rmi.RemoteException;
}
```

Servidor

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

public class HelloServer  extends   java.rmi.server.UnicastRemoteObject
                                implements Hello_S_I {
    private static Hello_C_I client;

  public HelloServer()  throws java.rmi.RemoteException{
    super();
  }
  //Método remoto
  public void printOnServer(String s) throws java.rmi.RemoteException{
   System.out.println( " SERVER : " +s );
  }
  //Método remoto
  public void subscribe (String name, Hello_C_I c) throws java.rmi.RemoteException{
    System.out.println("Subscribing " + name );
    client = c;
  }
    //Método local
  public static String lerString (){
    String s = "";
    try{
```

```
    BufferedReader in = new BufferedReader ( new InputStreamReader (System.in), 1);
    s= in.readLine();
   }
   catch (IOException e){
    System.out.println( e.getMessage());
   }
   return s;
}

 public static void main (String [] args){
   String s;
   System.setSecurityManager(new SecurityManager());
   try {
    HelloServer h = new HelloServer();
    Naming.rebind ("Hello",h);
    while (true){
      System.out.println("Mensagem para o cliente:");
      s= lerString();
      client.printOnClient(s);
    }
   }
   catch (RemoteException r){
      System.out.println("Exception in server"+r.getMessage());
   }
   catch (java.net.MalformedURLException u){
      System.out.println("Exception in server - URL" );
   }
  }
}
```

Exercício 1: Suponha um servidor que atribui um número sequencial a cada cliente, e que, após cada 10 clientes, sorteia um número (gerado aleatoriamente) de entre os números atribuídos aos últimos 10 clientes. O número do vencedor deverá ser comunicado a todos os últimos 10 clientes.

→ Factories

Até agora, vimos exemplos em que os objectos remotos são instanciados ou no main do servidor ou no construtor de servidor. Uma interface remota não pode incluir construtores, e portanto um cliente não pode instanciar directamente um objecto remoto.
Se quisermos instanciar objectos a pedido de um cliente, teremos que definir métodos numa interface remota com esse objectivo.
Um método que instancie um objecto remoto denomina-se por "factory method". Um "factory object" é um objecto com métodos "factory". (Na prática são métodos comuns.)

Exemplo:

Suponhamos que queremos um cliente para criar objectos do tipo Cidade em que cada objecto funciona como um servidor com informação sobre uma dada cidade.

1 - Para isso, começamos por definir uma interface remota, CidadeFactory com um único método que terá como parâmetro o nome do objecto Cidade a criar, e que devolve como resultado uma referência remota para um objecto do tipo Cidade:

```
public interface CidadeFactory extends java.rmi.Remote {

   public Cidade getServidorCidade (String nomeCidade)

                                                  throws java.rmi.RemoteException;
}
```

2 - Para podermos aceder aos objectos criados, dinamicamente através da factory, precisamos de uma interface remota para os objectos do tipo cidade. Suponhamos, para já apenas um método que nos dá a população da cidade.

```
public interface Cidade extends java.rmi.Remote {

      public int getPopulacao()  throws java.rmi.RemoteException;
}
```

3 – Necessitamos agora de implementar a interface Cidade

```
public class CidadeImpl  extends java.rmi.server.UnicastRemoteObject
                                                    implements Cidade {
  private String nomeCidade;
  int populacao = 20000;

public CidadeImpl() throws java.rmi.RemoteException {
    super();
  }
public CidadeImpl(String nomeCidade) throws java.rmi.RemoteException {
    super();
    this.nomeCidade = nomeCidade;
  }
public int getPopulacao()  throws java.rmi.RemoteException {
   return populacao;
  }
}
```

- Para já, todos os objectos do tipo cidade têm uma população de 20000 habitantes, no final poderá completar a classe de forma a poder actualizar os dados da cidade.

4 – A implementação da interface CidadeFactory irá instanciar um objecto do tipo CidadeImpl e devolver ao cliente a referência para esse objecto. Simultaneamente esta classe contém o main do servidor.

```
public class CidadeFactoryImpl extends java.rmi.server.UnicastRemoteObject
                                    implements CidadeFactory, java.io.Serializable {{
  public CidadeFactoryImpl() throws java.rmi.RemoteException {
    super();
  }
 public Cidade getServidorCidade (String nomeCidade)
                                        throws java.rmi.RemoteException{
    CidadeImpl ServidorCidade = new CidadeImpl(nomeCidade);
```

```
  return (Cidade) ServidorCidade;
 }

 public static void main (String arg[]){
  System.setSecurityManager(new SecurityManager());
  try {
   CidadeFactory factory = new CidadeFactoryImpl ();
   java.rmi.Naming.rebind("CidadeFactory", factory) ;
   System.out.println( "CidadeFactory registada");
  }
  catch (Exception e ){
   System.out.println( e.getMessage());
  }
}
}
```

5 – Acedendo ao servidor anterior, os processos clientes poderão criar objectos remotos do tipo Cidade, e invocar métodos nesses objectos.

Processo Cliente:

```
class CidadeCliente {
 public static void main (String args[]){
 Remote cidades = null;
 Cidade Covilha = null, CasteloBranco=null, Guarda=null;
  try {
   cidades =  Naming.lookup("//127.0.0.1/CidadeFactory");
 }
 catch (Exception e){
  System.out.println( e.getMessage());
 }
// criar um servidor para cada cidade
```

```
 try{
   Covilha = ((CidadeFactory)cidades).getServidorCidade("Covilha");
   CasteloBranco = ((CidadeFactory)cidades).getServidorCidade("CasteloBranco");
   Guarda = ((CidadeFactory)cidades).getServidorCidade("Guarda");
 }
 catch (Exception e){
   System.out.println( e.getMessage());
 }
//invocar métodos nesses objectos
 try {
    int i = Covilha.getPopulacao();
    System.out.println( "Covilhã tem " + i + " habitantes");
    i = CasteloBranco.getPopulacao();
    System.out.println( "Castelo Branco tem " + i + " habitantes");
    i = Guarda.getPopulacao();
    System.out.println( "Guarda tem " + i + " habitantes");
     }
 catch (Exception e){
  System.out.println(  e.getMessage());
 }
}
```

} Depois de estudar e implementar este exemplo em máquinas diferentes, defina outros métodos para a classe Cidade.