

→ **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 objeto servidor e invoca métodos remotos sobre esse objeto.

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 objeto 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 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 {
        // Exercício:
        // - Lançar o registry
        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" );
}
}
}
```

Cliente

```
import java.rmi.*;
public class HelloClient extends java.rmi.server.UnicastRemoteObject implements Hello_C_I {
public HelloClient() throws RemoteException {
    super();
}
//Método remoto
public void printOnClient (String s)throws java.rmi.RemoteException{
    System.out.println ("Message from server: " + s);
}
public static void main (String [] args){
    // System.setSecurityManager(new SecurityManager());
    try {
        Hello_S_I h= (Hello_S_I)Naming.lookup ("Hello");
        HelloClient c = new HelloClient();
        h.subscribe( "Nome da máquina cliente ...", (Hello_C_I)c);
    }
    catch (Exception r){
        System.out.println ( "Exception in client" +r.getMessage());
    }
}}
```

Exercício 2: 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. Implemente em Java RMI.

→ **Factories**

Até agora, vimos exemplos em que os objetos 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 diretamente um objeto remoto.

Se quisermos instanciar objetos a pedido de um cliente, teremos que definir métodos numa interface remota com esse objetivo.

Um método que instancie um objeto remoto denomina-se por “factory method”. Um “factory object” é um objeto com métodos “factory”. (Na prática são métodos comuns.)

Exemplo:

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

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

```
public interface CidadeFactory extends java.rmi.Remote {
    public Cidade getServidorCidade (String nomeCidade)
                                                throws java.rmi.RemoteException;
}
```

B - Para podermos aceder aos objetos criados, dinamicamente através da factory, precisamos de uma interface remota para os objetos 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;
}
```

C – Precisamos 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 objetos do tipo cidade têm uma população de 20000 habitantes, no final poderá completar a classe de forma a poder atualizar os dados da cidade.

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

```
public class CidadeFactoryImpl extends java.rmi.server.UnicastRemoteObject
                                implements CidadeFactory {
    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());
    }
}
```

-> lançar o registry

```
try {
    CidadeFactory factory = new CidadeFactoryImpl ();
```

```
    java.rmi.Naming.rebind("CidadeFactory", factory) ;
    System.out.println( "CidadeFactory registada");
}
catch (Exception e ){
    System.out.println( e.getMessage());
} } }
```

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

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());
}
}
}
```

3 – Depois de estudar e implementar o exemplo anterior, defina outros métodos para a classe Cidade.

4 – Suponha que a associação académica da UBI decidiu abrir uma campanha de angariação de fundos para a construção de uma pista de desportos radicais.

Para dar suporte à gestão da campanha deve construir uma aplicação cliente / servidor através da qual quem quiser aderir, se pode inscrever e indicar qual o seu donativo. O donativo será depois feito através de um depósito numa conta bancária aberta para a campanha. O processo cliente deve poder escolher de entre as seguintes opções:

1 – Donativo; 2 – Consultar total; 3 – Consultar doadores; 4 – Sair

A opção “Donativo” deve enviar para o servidor, o valor doado e o nome do doador.

A opção “Consultar total” deve obter como resposta o valor total doado até ao momento.

A opção “Consultar doadores” deve obter como resposta a lista com os nomes dos doadores. (Se um mesmo doador fizer mais de uma doação, deverá aparecer apenas uma vez na lista, não é necessário guardar qual foi o donativo de cada doador, apenas o total acumulado pelo conjunto dos doadores).

Para implementar a aplicação usando **Java RMI**, construa e teste,

- a) A interface remota;
- b) O servidor;
- c) A classe que implementa o objeto remoto;
- d) O cliente.

5 – Suponha que na aplicação do exercício 4, se pretende implementar um callback para comunicar ao processo cliente que foi o doador nº 100. Modifique a aplicação anterior para implementar essa funcionalidade.