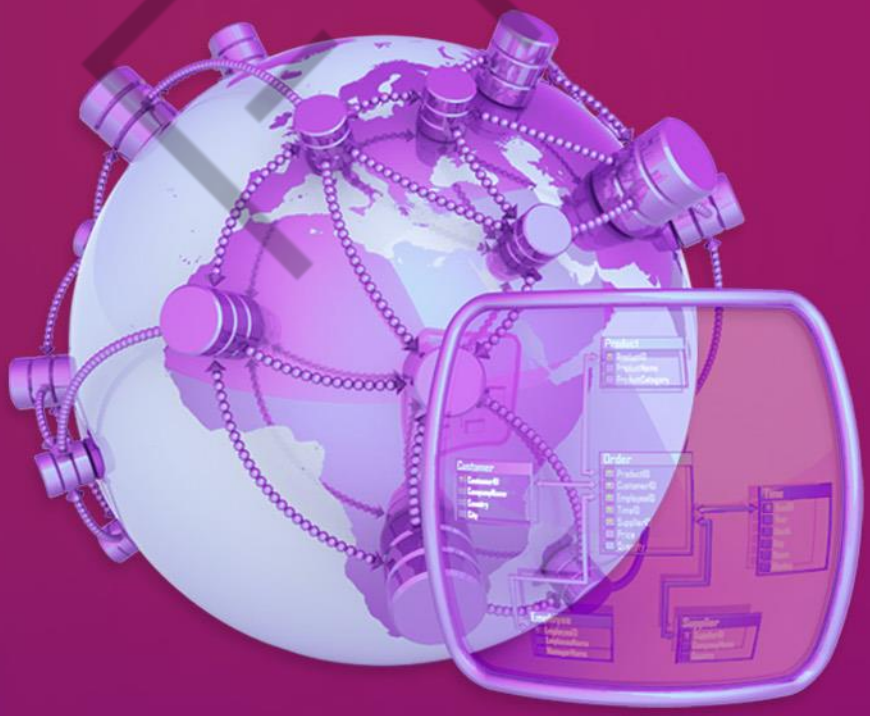
FRAMEWORKS JAVA, .NET & WEBSERVICES

JAVA EM TODA A **PARTE**



4A

LISTA DE FIGURAS

Figura 1 – Exemplo de guardar e recuperar dados 5
Figura 2 – Exemplo do computador tentando se comunicar com o celular 14
Figura 3 – Arquitetura de camadas RMI 15

LISTA DE CÓDIGOS-FONTE

Código-fonte 1 – Serializando 2 objetos String 7
Código-fonte 2 – Classe Logradouro.....9
Código-fonte 3 – Classe que tenta serializar uma instância de “Logradouro” 10
Código-fonte 4 – Classe Logradouro implementando “Serializable” 10
Código-fonte 5 – Classe que desserializa um texto 12
Código-fonte 6 – Classe que desserializa um “Logradouro” 13
Código-fonte 7 – Interface remota “TipoEstabelecimentoService” 16
Código-fonte 8 – Implementação da interface remota “TipoEstabelecimentoService” 17
Código-fonte 9 – Classe que ficará “escutando” as solicitações RMI.....19
Código-fonte 10 – Cliente RMI para o “TipoEstabelecimentoService” 21
Código-fonte 11 – Saída da “TipoEstabelecimentoCliente” informando um código existente.....22
Código-fonte 12 – Saída da “TipoEstabelecimentoCliente” informando um código inexistente 22
Código-fonte 13 – Classe que usa *Socket* para pesquisa remota em “TipoEstabelecimentoDAO”.....24
Código-fonte 14 – Saída recebida pelo cliente telnet do *Socket* 25
Código-fonte 15 – Saída no cliente telnet do *Socket* após consultas e pedido de saída 25

SUMÁRIO

1 JAVA EM TODA A PARTE.....5
1.1 Introdução 5
1.2 Serialização.....6
1.3 Serializando uma String 7
1.4 Só objetos Serializable e primitivos podem ser serializados 8

2 DESSERIALIZAÇÃO.....11
2.1 Desserialização de um texto 11
2.2 Desserialização de um objeto do tipo Logradouro 12

3 RMI (REMOTE METHOD INVOCATION) 14
3.1 Introdução 14
3.2 Implementando um Servidor RMI.....15
3.2.1 Criando um Servidor RMI que realiza uma consulta no TipoEstabelecimentoDAO 16
3.2.2 Implementando um Cliente RMI 20
3.2.3 Fazendo a comunicação entre Servidor e Cliente RMI 21
3.3 *Sockets*.....22
3.3.1 *Socket* Java que usa um DAO 23
3.3.2 Acessando o *Socket* via telnet 24
3.3.3 Por que usar *Socket* se posso usar RMI? 26

REFERÊNCIAS.....27

1 JAVA EM TODA A PARTE

Por enquanto, aprendemos a criar sistemas em Java que ficam todos instalados em um único lugar ou servidor. Mas quando precisarmos distribuir os objetos que devem trafegar de um sistema para o outro? Veremos aqui a tecnologia que permite essa transferência, o RMI, e os túneis para trafegá-los, os *Sockets*!

1.1 Introdução

Imagine que você tenha necessidade de que um sistema “lembre” de algumas informações mesmo após ser encerrado e executado novamente. Que recurso você usaria para essa tarefa? Imagino que pensou em banco de dados, afinal, estudamos ORM com JPA/*Hibernate* recentemente. De fato, seria uma opção, mas nem sempre banco de dados é a melhor solução quando precisamos guardar e recuperar dados.

Alguns exemplos: se você precisa guardar poucas informações, como as preferências (configurações) de um sistema ou se você precisa de alguns dados dos usuários do sistema; se você precisa enviar dados de seu sistema feitos em Java para outros também feitos em Java. Para situações como essas, a plataforma Java oferece o recurso de serialização e desserialização de objetos, que estudaremos a seguir.

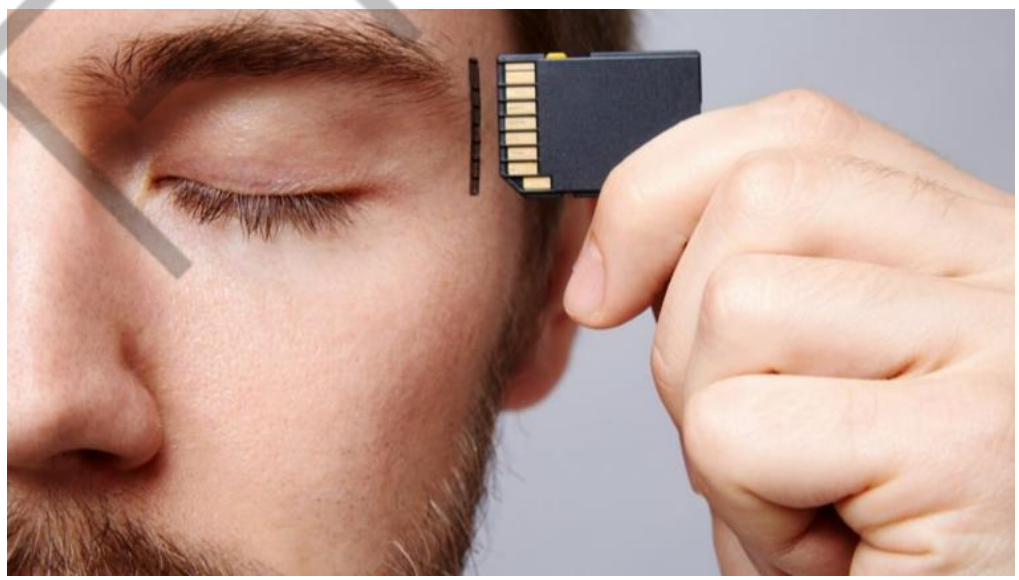


Figura 1 – Exemplo de guardar e recuperar dados
Fonte: Shutterstock (2018)

1.2 Serialização

Em Java, a serialização é o processo de converter um objeto em uma sequência de bytes que podem ser facilmente transmitidos e armazenados. Isso é útil quando você precisa enviar um objeto através da rede ou salvá-lo em um arquivo, por exemplo.

Para ser serializável em Java, um objeto deve implementar a interface Serializable. Essa interface não possui nenhum método a ser implementado, mas serve para sinalizar ao compilador que o objeto pode ser serializado.

Durante o processo de serialização, o objeto é convertido em uma sequência de bytes que representam o seu estado atual. Esse estado inclui os valores de seus atributos e quaisquer referências a outros objetos que o objeto possa ter.

Depois de ser serializado, o objeto pode ser transmitido pela rede ou salvo em um arquivo. Quando necessário, ele pode ser desserializado, ou seja, transformado de volta em um objeto, com todos os seus valores e referências originais.

A serialização em Java é uma técnica poderosa e útil, mas deve ser usada com cuidado. É importante lembrar que, ao serializar um objeto, você está incluindo todo o seu estado atual e quaisquer referências a outros objetos. Se você não tomar cuidado, pode acabar serializando um objeto que inclui referências a objetos que não devem ser serializados ou que não estão disponíveis no momento em que o objeto é desserializado. Isso pode causar erros e comportamentos inesperados no seu programa.

Em resumo, a serialização em Java é um processo que permite transformar um objeto em uma sequência de bytes que podem ser facilmente transmitidos e armazenados. Para ser serializável, um objeto deve implementar a interface Serializable e tomar cuidado com as referências a outros objetos incluídas no estado serializado.

Quando temos situações nas quais precisamos guardar informações de um sistema e bancos de dados não são a melhor opção ou nem sequer são cogitados, podemos usar o recurso de serialização de objetos do Java. Ele permite guardar ou enviar, por uma rede, objetos criados num programa Java de tal forma que qualquer outro programa também escrito em Java consiga recuperá-los. Esse é um recurso

nativo da linguagem Java, ou seja, não é necessário importar nenhuma biblioteca no projeto.

1.3 Serializando uma String

No nosso primeiro exemplo de serialização, vamos serializar um texto simples, ou seja, um objeto do tipo *String* do Java. Veja como isso pode ser feito no código-fonte abaixo.

```
// pacote

import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class SerializadorStrings
{
    public static void main( String[] args ) throws IOException
    {
      FileOutputStream fileStream = new
FileOutputStream("texto.ser");

      String texto = "Não sou dono do mundo, mas sou filho do
dono!";

      ObjectOutputStream os = new
ObjectOutputStream(fileStream);
      os.writeObject(texto);

      os.close();
      fileStream.close();
    }
}
```

Código-fonte 1 – Serializando 2 objetos String
Fonte: Elaborado pelo autor (2018)

O código faz uso da serialização em Java para salvar uma String em um arquivo.

A classe FileOutputStream é usada para criar um fluxo de saída que permitirá escrever os bytes do arquivo no disco. Em seguida, a String é definida e o objeto ObjectOutputStream é usado para escrevê-la no arquivo.

Por fim, o ObjectOutputStream e o FileOutputStream são fechados para liberar recursos.

Lembrando que, para esse código funcionar corretamente, a classe String já implementa a interface Serializable do Java, permitindo que seja serializada sem a necessidade de qualquer adaptação.

No Código-fonte nós também criamos um método ***main()*** apenas para poder executar a classe e já verificar o resultado. Notou que indicamos, na assinatura do método, que ele pode lançar uma exceção do tipo IO*Exception*?

Essa ação foi necessária porque, quando usamos os recursos de serialização do Java, muitos de seus métodos e construtores possuem a indicação dessa exceção que é do tipo checada. Ou seja, é obriga a usar blocos *try-catch-finally* ou repassar o tratamento anunciando a exceção no método criado, que foi o que fizemos.

Repare que importarmos duas classes, todas do pacote java.io, da biblioteca-padrão do Java. São elas: *FileOutputStream* e *ObjectOutputStream*. A *FileOutputStream* é usada para criar um objeto que indica o arquivo no qual um ou mais objetos serão serializados e, no exemplo, exibe o arquivo "*texto.ser*". A *ObjectOutputStream* é utilizada para criar um objeto que escreverá um ou mais objetos que desejamos serializar e, no exemplo, solicitamos que escrevesse um texto para serialização.

Após serializarmos o texto, solicitamos o fechamento de seu *ObjectOutputStream* e, em seguida, do *FileOutputStream*. Ambos os fechamentos foram realizados com o método ***close*()**. É após o fechamento do *FileOutputStream* que o arquivo *texto.ser* é criado em disco.

Após a execução dessa classe, um arquivo chamado *texto.ser* será criado no diretório do projeto. Se tentar abrir esse arquivo com um editor de texto, verá que é inelegível, afinal, trata-se de um arquivo binário, que só um outro programa escrito em Java é capaz de entender, fazendo a desserialização, que veremos daqui a pouco.

1.4 Só objetos Serializable e primitivos podem ser serializados

No exemplo do Código-fonte "Serializando dois objetos String" vimos como serializar um texto. Essa serialização foi possível porque a classe String do Java

implementa a interface java.io.Serializable do pacote-padrão do Java. Imagine que precisemos serializar uma classe chamada Logradouro, que pode ser vista no código-fonte a seguir:

```
public class Logradouro {
    private long id;
    private String nome;
    private TipoLogradouro tipo;
    private String cep;
         // construtor, getters e setters

}
```

Código-fonte 2 – Classe Logradouro
Fonte: Elaborado pelo autor (2021)

Para facilitar nossa serialização, vamos criar TipoLogradouro como um enum. Veja o código a seguir:

```
public enum TipoLogradouro {
    RUA,AVENIDA,TRAVESSA,PRACA,QUADRA,ALAMEDA
}
```

Código-fonte 3 – TipoLogradouro será criado como um enum
Fonte: Elaborado pelo autor (2021)

Se criamos uma outra classe para serializar a Logradouro, ela seria como o código-fonte abaixo:

```
// pacote

import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class SerializadorLogradouro
{
    public static void main( String[] args ) throws IOException
    {
      FileOutputStream        fileStream        =        new
FileOutputStream("logradouro.ser");

          Logradouro logradouro = new Logradouro(1, "Rua Ze
Locao", TipoLogradouro.RUA,"12345-000");
```

```
            ObjectOutputStream          os          =          new
ObjectOutputStream(fileStream);
            os.writeObject(logradouro);

            os.close();
            fileStream.close();
        }
}
```

Código-fonte 3 – Classe que tenta serializar uma instância de “Logradouro”
Fonte: Elaborado pelo autor (2018)

Ao executar a classe do Código-fonte “Classe que tenta serializar uma instância de Logradouro”, tomaremos uma exceção do tipo java.io.NotSerializableException porque nossa classe Logradouro não implementa a java.io.Serializable. Podemos providenciá-la, alterando o Código-fonte “Classe Logradouro”. Vamos apenas mudar a definição da classe, como pode ser visto no código-fonte a seguir:

```
// pacote
import java.io.Serializable;

public class Logradouro implements Serializable {
     // atributos, construtor, getters e setters
}
```

Código-fonte 4 – Classe Logradouro implementando “Serializable”
Fonte: Elaborado pelo autor (2018)

Após essa alteração, a classe SerializadorLogradouro executaria sem erro algum. Porque, além de Logradouro implementar Serializable, seus atributos implementaram-na (String) ou são tipos primitivos (long) ou ainda enums, que, na linguagem Java, são serializáveis por natureza.

E, se um dos atributos de Logradouro não for *Serializable* nem primitivo? Nesse caso, uma instância sua também não poderá ser serializada.

2 DESSERIALIZAÇÃO

Desserialização é o processo de converter uma sequência de bytes em um objeto Java. Essa sequência de bytes é geralmente criada durante o processo de serialização, que converte um objeto Java em uma sequência de bytes.

Para desserializar um objeto é necessário usar um objeto do tipo ObjectInputStream, que lê a sequência de bytes e cria um novo objeto Java com base nos dados lidos. Durante a desserialização, todos os valores dos atributos e referências aos objetos são recuperados e o objeto é restaurado em seu estado original antes de ser serializado.

A desserialização é útil em situações em que você precisa transmitir objetos Java por meio da rede ou armazená-los em arquivos. É importante lembrar que o objeto que está sendo desserializado deve ter a mesma classe e mesmos atributos que o objeto original que foi serializado. Se houver alguma diferença, a desserialização pode falhar ou gerar resultados inesperados.

Quando fazemos o processo contrário da serialização, ou seja, quando lemos um arquivo de objetos serializados Java e o convertemos em objetos Java, estamos fazendo a desserialização. A seguir, veremos como desserializar o texto e a instância de Logradouro serializados anteriormente.

2.1 Desserialização de um texto

É possível recuperar o texto serializado no *texto.ser* como podemos ver no código-fonte abaixo:

```
// pacote
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class DesserializadorStrings
{
    public static void main( String[] args ) throws
IOException, ClassNotFoundException
    {
        FileInputStream fileStream = new
FileInputStream("frases.ser");
```

```
        ObjectInputStream os = new
ObjectInputStream(fileStream);

        String textoRecuperado = (String) os.readObject();
        System.out.println("Texto recuperado: "
                           + textoRecuperado);

        os.close();
        fileStream.close();
    }
}
```

Código-fonte 5 – Classe que desserializa um texto
Fonte: Elaborado pelo autor (2018)

Note que aqui as classes usadas foram *FileInputStream* e *ObjectInputStream*. A primeira exibe o arquivo no qual acreditamos estar o arquivo com objetos serialiazados java. A segunda serve para recuperar o conteúdo do objeto serializado. O método **readObject()** anuncia uma *ClassNotFoundException,* por isso essa exceção está na assinatura do método *main().* Quando o usamos, devemos indicar o *cast* para o tipo de objetos que acreditamos estar serializando (no caso, String).

Após desserializarmos o texto, solicitamos sua exibição na saída-padrão para confirmar que o programa funcionou. Depois que já lemos e desserializamos o que queríamos, fechamos os objetos das classes *FileInputStream* e *ObjectInputStream* para que o arquivo *texto.ser* seja liberado para o sistema operacional. Sem fecharmos esses objetos, o sistema operacional poderia não conseguir ler ou alterar o arquivo.

2.2 Desserialização de um objeto do tipo Logradouro

É possível recuperar também o Logradouro que serializamos anteriormente. Podemos ver como no código-fonte a seguir:

```
// pacote
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class DesserializadorLogradouro
{
    public static void main( String[] args ) throws
IOException, ClassNotFoundException
    {
```

```
        FileInputStream        fileStream        =        new
FileInputStream("logradouro.ser");
        ObjectInputStream          os          =          new
ObjectInputStream(fileStream);

        Logradouro logradouroRecuperado =
                                  (Logradouro) os.readObject();
        System.out.println("Logradouro recuperado: "
                     + logradouroRecuperado.getNome());

        os.close();
        fileStream.close();
    }
}
```

Código-fonte 6 – Classe que desserializa um “Logradouro”
Fonte: Elaborado pelo autor (2018)

Perceba que aqui fizemos basicamente o mesmo que foi feito no Código-fonte “Classe que desserializa um texto”. A única diferença é que fizemos o *cast* para a classe Logradouro ao invocar o método *readObject().* Após desserializar o Logradouro, podemos invocar qualquer método dele, como fizemos no Código-fonte “Classe que desserializa um Logradouro’”.

3 RMI (REMOTE METHOD INVOCATION)

Em dias em que a Internet está em quase todo o lugar e nos quais se fala muito de IoT (Internet das Coisas), a comunicação entre sistemas é de grande importância. Com Java, umas das formas de integrar sistemas que estão sendo executados em computadores diferentes na mesma rede é com o uso do RMI (*Remote Method Invocation*, ou “Invocação de Método Remoto”, em tradução livre).

3.1 Introdução

Imagine que você precise fazer com que dois computadores (notebooks, celulares etc.) comuniquem-se remotamente. Exemplos desse tipo de situação são: pedir para um elevador descer quando está para entrar no edifício ou pedir para o ar ar-condicionado do carro ligar quando você começa a se preparar para sair do escritório. Uma das maneiras de conseguir essa conexão é fazendo com que um programa escrito em Java solicite para outro programa em outro computador executar um determinado método. Na plataforma Java, essa conexão pode ser feita com o uso do RMI.

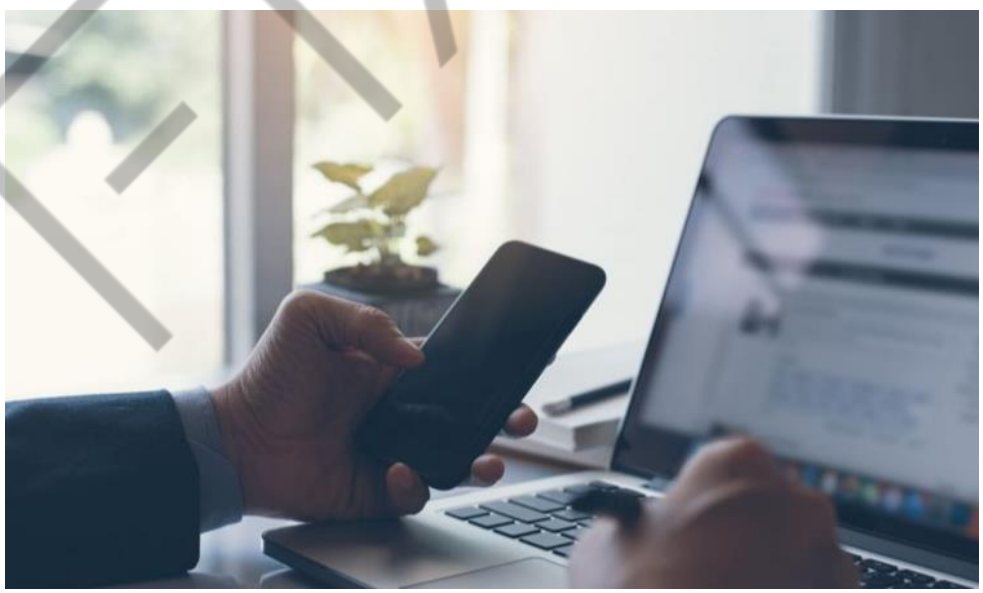


Figura 2 – Exemplo do computador tentando se comunicar com o celular
Fonte: Shutterstock (2018)

O RMI do Java foi baseado em uma tecnologia anterior conhecida como RPC (*Remote Procedure Calls*, ou “Chamadas de Procedimentos Remotos”, em tradução

livre), criada na década de 1980. As chamadas são enviadas de um programa Java (o "*Cliente*") para outro (o "*Servidor*") remotamente, por meio de uma rede. Um programa pode funcionar como *Cliente* e *Servidor* ao mesmo tempo, sem problemas. Nesse caso, ele está habilitado para enviar e receber chamadas remotas. A arquitetura do RMI do Java é dividida em três camadas:

Stub/Skeleton contêm as interfaces que os objetos da aplicação usam para interagir entre si. O *Stub* fica no processo *Cliente* e o *Skeleton,* no processo *Servidor*. Eles são os componentes que abstraem a complexidade de conexão com rede, envio, recebimento e tradução dos dados trocados via rede.

Referência remota é responsável pela criação e gerenciamento das referências para objetos remotos.

Camada de transporte implementa e abstrai o protocolo responsável pelas solicitações enviadas e recebidas entre os objetos remotos pela rede.

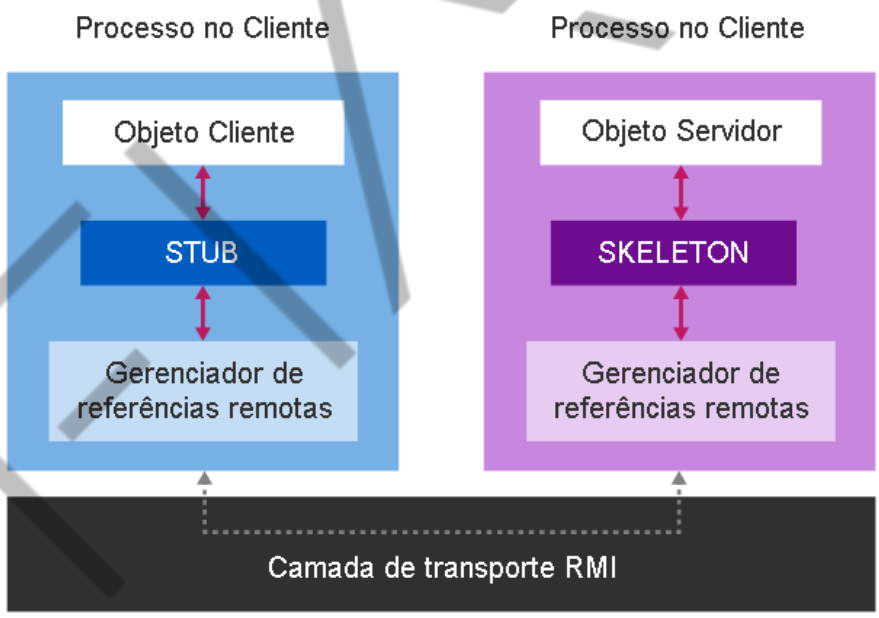


Figura 3 – Arquitetura de camadas RMI
Fonte: Elaborado pelo autor (2018)

3.2 Implementando um Servidor RMI

Um Servidor RMI é um programa Java capaz de receber pedidos para a invocação de um método enviados por um programa em outro computador. Após

receber o pedido, ele executa o método solicitado e devolve o resultado dessa execução.

O primeiro passo para a criação de um Servidor RMI é criar uma **interface** que estenda *java.RMI.Remote*. O segundo passo é criar uma classe que crie e registre o *Skeleton* e que inicie o serviço de escuta de solicitações remotas.

Como exemplo de implementação de Servidor RMI, vamos criar um servidor que receba um pedido de consulta ao TipoEstabelecimentoDAO, conforme vimos no capítulo *Generic DAO*. Ele receberá um código de tipo de estabelecimento e devolverá o nome do Tipo de Estabelecimento encontrado na busca ou *null* se não for encontrado. Você pode colocar os códigos deste capítulo no mesmo projeto do TipoEstabelecimentoDAO ou criar um novo projeto e pôr o anterior como dependência, o que achar mais fácil.

3.2.1 Criando um Servidor RMI que realiza uma consulta no TipoEstabelecimentoDAO

Conforme já explicado, o primeiro passo é criar a interface do serviço RMI. Vamos criar a *TipoEstabelecimentoService,* conforme consta no código-fonte abaixo:

```
package fiapon.digital.capitulo7.rmi.server;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface TipoEstabelecimentoService extends Remote {

     public String pesquisar(Integer idPesquisa) throws
RemoteException;

}
```

Código-fonte 7 – Interface remota “TipoEstabelecimentoService”
Fonte: Elaborado pelo autor (2018)

A *interface* que criamos estende a *java.rmi.Remote*. Isso é necessário para que possamos criar um servidor RMI. Outro detalhe é que os métodos públicos que queremos expor em nosso serviço devem anunciar uma *java.rmi.RemoteExpection* em sua assinatura. Fizemos isso no método **pesquisar()**. Vamos agora criar a

implementação dessa interface, a TipoEstabelecimentoServiceImpl, que está no código-fonte a seguir.

Importante: caso o Cliente que faça chamadas a esse Servidor seja um outro projeto, apenas a interface deve ser disponibilizada para ele, nunca a implementação.

```
package br.com.fiap.smartcities.service;

import java.rmi.RemoteException;
import javax.persistence.EntityManager;
import javax.persistence.Persistence;
import br.com.fiap.smartcities.dao.TipoEstabelecimentoDAO;
import br.com.fiap.smartcities.domain.TipoEstabelecimento;

public class TipoEstabelecimentoServiceImpl implements
TipoEstabelecimentoService {

     public TipoEstabelecimentoServiceImpl() {
          super();
     }

     public String pesquisar(Integer idPesquisa) throws
RemoteException {

          EntityManager em =
Persistence.createEntityManagerFactory("smartcities-
orm").createEntityManager();

          TipoEstabelecimentoDAO dao = new
TipoEstabelecimentoDAO(em);
          TipoEstabelecimento tipoEstabelecimento =
dao.buscar(idPesquisa);

          if (tipoEstabelecimento != null) {
               return tipoEstabelecimento.getNome();
          } else {
               return null;
          }

     }


}
```

Código-fonte 8 – Implementação da interface remota “TipoEstabelecimentoService”
Fonte: Elaborado pelo autor (2018)

Repare que criamos um construtor na *TipoEstabelecimentoServiceImpl*. Ação obrigatória para que seja uma implementação de serviço remoto válida. Também é obrigatório que o construtor anuncie uma *java.rmi.RemoteException*.

Dentro do método pesquisar(), nós iniciamos um *EntityManager* e o usamos como argumento ao criar um objeto do tipo TipoEstabelecimentoDAO.

Usamos o TipoEstabelecimentoDAO para tentar recuperar um objeto TipoEstabelecimento. Se for encontrado, o método retorna o seu nome. Caso contrário, o método retorna *null*.

Agora precisamos criar uma classe que crie e registre o *Skeleton* e que inicie o serviço de escuta de solicitações remotas. Vamos chamá-la de TipoEstabelecimentoServidor. Veja essa classe no código-fonte abaixo:

```
package br.com.fiap.smartcities.rmi;

import java.rmi.AlreadyBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

import
br.com.fiap.smartcities.service.TipoEstabelecimentoService;
import
br.com.fiap.smartcities.service.TipoEstabelecimentoServiceImp
l;

public class TipoEstabelecimentoServidor {

     public static void main(String[] args) {

          try {

               TipoEstabelecimentoService   service   =   new
TipoEstabelecimentoServiceImpl();
               TipoEstabelecimentoService      skeleton      =
(TipoEstabelecimentoService)
UnicastRemoteObject.exportObject(service, 0);

               Registry               registry              =
LocateRegistry.createRegistry(7777);
```

```
                registry.bind("TipoEstabelecimentoService",
skeleton);

                System.out.println("TipoEstabelecimentoService
registrado e pronto para aceitar solicitações.");

          } catch (RemoteException | AlreadyBoundException e)
{
                System.out.println("Não foi possível iniciar o
Servidor RMI!");
                e.printStackTrace();
          }

     }

}
```

Código-fonte 9 – Classe que ficará “escutando” as solicitações RMI
Fonte: Elaborado pelo autor (2018)

Criamos um método main() na classe para que ela possa ser executada. Nele, instanciamos um objeto do tipo TipoEstabelecimentoService com a classe que o implementa (TipoEstabelecimentoServiceImpl). Logo em seguida, pedimos que o Java gere seu *Skeleton* por meio do método **exportObject()** da classe ***java.rmi.server.UnicastRemoteObject***. O valor 0 (zero) que usamos nesse método indica que a porta TCP usada será configurada via registro (*bind*) de serviço.

A seguir, começamos o processo de registro do Servidor. Criamos um objeto do tipo *java.rmi.registry.Registry* invocando o método **createRegistry()** da classe *java.rmi.registry.LocateRegistry*. O argumento que passamos nesse método indica a porta TCP na qual o Servidor ficará esperando pelo recebimento de chamadas de clientes RMI. O último passo foi registrar o *Skeleton*, invocando o método **bind()** no objeto do tipo *java.rmi.registry.Registry*, passando como argumento o *Skeleton*.

Com esse código, iniciamos um Servidor RMI, que escuta na porta TCP 7777. Esse valor de porta pode ser qualquer um, embora seja recomendável valores a partir de 1024. Valores menores que esse costumam pedir permissão de superusuário em sistemas operacionais baseados em Unix. A maior porta permitida no TCP é a 65535.

A seguir, veremos como implementar um Cliente RMI capaz de se comunicar com esse Servidor que criamos.

3.2.2 Implementando um Cliente RMI

Um Cliente RMI é um programa Java capaz de enviar solicitações a um Servidor RMI e trabalhar com a resposta recebida. Para criarmos um Cliente capaz de se comunicar com o Servidor que acabamos de criar, precisamos apenas da interface TipoEstabelecimentoServidor.

Podemos implementar esse Cliente no mesmo projeto do Servidor ou em outro. A sugestão é criar um novo projeto, lembrando que ele precisa apenas da interface TipoEstabelecimentoServidor no mesmo pacote (declaração *package*) do projeto do Servidor. Vejamos como seria o Cliente no código-fonte a seguir, na classe que chamamos de TipoEstabelecimentoCliente.

```
package br.com.fiap.smartcities.rmi;

import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.util.Scanner;

import
br.com.fiap.smartcities.service.TipoEstabelecimentoService;

public class TipoEstabelecimentoCliente {

     public static void main(String[] args) {

          try {

               Registry               registry               =
LocateRegistry.getRegistry("127.0.0.1", 7777);
               TipoEstabelecimentoService        stub        =
(TipoEstabelecimentoService) registry

     .lookup("TipoEstabelecimentoService");

               System.out.print("Digite o código de Tipo de
estabelecimento: ");

               int        codigoPesquisa        =        new
Scanner(System.in).nextInt();
```

```
                String                  resultado                  =
stub.pesquisar(codigoPesquisa);
                if (resultado != null) {
                     System.out.println("Registro encontrado: "
+ resultado);
                } else {
                     System.out.println("Tipo                  de
estabelecimento não encontrado");
                }

           } catch (RemoteException | NotBoundException e) {

                System.out.println("Erro ao tentar chamada para
o servidor");
                e.printStackTrace();

           }

      }

}
```

Código-fonte 10 – Cliente RMI para o “TipoEstabelecimentoService”
Fonte: Elaborado pelo autor (2018)

No Cliente, usamos as classes *java.rmi.registry.LocateRegistry* e *java.rmi.registry.Registry* para registrar o servidor ao qual queremos ter acesso. No método **getRegistry(),** passamos como argumentos o endereço IP e a porta em que está localizado o servidor RMI com o qual queremos nos comunicar.

Em seguida, solicitamos a criação do Stub usando o método ***lookup*()** do *java.rmi.registry.Registry*. Como esse método sempre retorna um *Object*, temos que fazer o *cast* para a interface do Servidor, no caso, a TipoEstabelecimento*Service*.

Por fim, basta invocar o método **pesquisar()** da interface TipoEstabelecimento*Service* com o código que queremos realizar a consulta. No exemplo, solicitamos o código via *scanner* (entrada-padrão do Java).

3.2.3 Fazendo a comunicação entre Servidor e Cliente RMI

Com o Servidor e o Cliente criados, agora, vamos testar a comunicação entre eles. O primeiro passo é iniciar o Servidor. Para isso, basta executar a classe

TipoEstabelecimentoServidor. Após iniciada, a saída gerada será apenas a frase “TipoEstabelecimentoService registrado e pronto para aceitar solicitações”.

Com o Servidor pronto, iniciaremos o Cliente, ou seja, a classe TipoEstabelecimentoCliente. Para tornar o exemplo mais interessante, tente executar o Cliente num computador diferente do Servidor, porém na mesma rede. Vamos supor que iniciamos essa classe e indicamos um código de Tipo de Estabelecimento que sabemos que existe. Nesse caso, teríamos uma saída como a do código-fonte abaixo:

```
Digite o código de Tipo de estabelecimento: 10

Tipo de estabelecimento encontrado: Parque
```

Código-fonte 11 – Saída da “TipoEstabelecimentoCliente” informando um código existente
Fonte: Elaborado pelo autor (2018)

Caso informássemos um código inexistente, a saída seria como a do código-fonte a seguir:

```
Digite o código de Tipo de estabelecimento: 50

Tipo de estabelecimento não encontrado
```

Código-fonte 12 – Saída da “TipoEstabelecimentoCliente” informando um código inexistente
Fonte: Elaborado pelo autor (2018)

Acabamos de ver como criar um Cliente e um Servidor RMI básico que mostra o “caminho das pedras” para o desenvolvimento de qualquer projeto usando RMI. A seguir, veremos como criar *Sockets* Java, utilizados de forma transparente pelo RMI.

A implementação dos recursos de RMI (Service, Cliente e Servidor) pode ser baixada em:

Git: https://github.com/FIAP/smartcities-orm/tree/07-rmi

Zip: https://github.com/FIAP/smartcities-orm/archive/refs/heads/07-rmi.zip

3.3 *Sockets*

Sockets em Java são uma maneira de permitir que programas enviem e recebam dados pela rede. Em outras palavras, um socket é um objeto que permite

que um programa se comunique com outro programa, mesmo que eles estejam em computadores diferentes.

Imagine que você queira enviar uma mensagem para um amigo que está em outra cidade. Você pode usar o celular para fazer isso, digitando a mensagem e pressionando o botão "enviar". Em termos de programação, um socket é como o seu celular - é o meio pelo qual você envia e recebe dados pela rede.

Em Java, você pode criar um socket usando a classe Socket ou ServerSocket. Um Socket é usado para se conectar a outro programa na rede, enquanto um ServerSocket é usado para esperar por conexões de outros programas.

Depois de criar um Socket ou ServerSocket, você pode usar métodos como getInputStream() e getOutputStream() para enviar e receber dados através do socket.

Resumindo, sockets em Java são uma maneira de permitir que programas se comuniquem através da rede, permitindo que dados sejam enviados e recebidos de forma eficiente e confiável.

Acabamos de aprender o que é e como funciona o RMI do Java. É possível criar ainda um outro jeito de comunicar sistemas em computadores diferentes, que são os *Sockets*, tecnologia de mais baixo nível que o RMI. Com ela, apenas ocupamos uma porta TCP no servidor *Socket*. A seguir, iremos criar uma pequena aplicação *Socket* Java.

3.3.1 *Socket* Java que usa um DAO

Como exemplo de *Socket*, criaremos uma classe que também usará o TipoEstabelecimentoDAO. Ela solicitará um código e retornará o nome do registro encontrado ou uma mensagem dizendo que não encontrou registro. Essa classe pode ser criada no mesmo projeto em que criou o Servidor RMI. Vejamos o código-fonte abaixo:

```
$ telnet 127.0.0.1 7777
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Tecle <Enter> para iniciar
```

```
Pesquisa por código de Tipo de Estabelecimento (0 para sair):
1
Realizando pesquisa, aguarde... Registro encontrado: Escola

Pesquisa por código de Tipo de Estabelecimento (0 para sair):
2
Realizando pesquisa, aguarde... Nenhum registro encontrado!

Pesquisa por código de Tipo de Estabelecimento (0 para sair):
10
Realizando pesquisa, aguarde... Registro encontrado: Parque

Pesquisa por código de Tipo de Estabelecimento (0 para sair):
0

Connection closed by foreign host.
```

Código-fonte 13 – Classe que usa *Socket* para pesquisa remota em "TipoEstabelecimentoDAO"
Fonte: Elaborado pelo autor (2021)

Na classe do Código-fonte "Classe que usa *Socket* para pesquisa remota em 'TipoEstabelecimentoDAO'" iniciamos um objeto do tipo java.net.ServerSocket, indicando no construtor a porta que será ocupada pelo *Socket* (no caso, a 7777). O construtor dessa classe anuncia uma *IOException*, por isso, repassamos para o método **main()**. Em seguida, criamos um objeto do tipo **j***ava.net.Socket***,** o Cliente. É nesse momento que o *Socket* passa a funcionar e ficar disponível na rede.

O objeto Saída, do tipo *java.io.PrintWriter,* irá enviar, pela rede, mensagens de texto para o cliente do *Socket*. Note que, sempre que quisermos que as mensagens sejam enviadas ao cliente, invocamos o método **flush()**. Para receber o que o cliente envia, criamos um objeto do tipo *Scanner* a partir do objeto Cliente.

Ao ser executada, essa classe apenas imprime na saída-padrão "*Porta 7777 aberta!*".

3.3.2 Acessando o *Socket* via telnet

Podemos testar o *Socket* que criamos anteriormente apenas usando um comando do sistema operacional chamado telnet. Ele existe no Windows, Linux e MacOS. No Windows, ele deve ser executado no *prompt de comando* e, no Linux e MacOS, num *terminal bash*. Sugerimos que, se possível, execute os comandos a

seguir de um computador diferente, porém, na mesma rede do computador no qual iniciou o *Socket*.

O comando que você deve executar é telnet 127.0.0.1 7777. No lugar de 127.0.0.1, use o IP do computador em que está o *Socket*, caso execute esse comando de outro computador na rede. Ao executar esse comando, você verá um texto como o do código a seguir:

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Tecle <Enter> para iniciar
```

Código-fonte 14 – Saída recebida pelo cliente telnet do *Socket*
Fonte: Elaborado pelo autor (2018)

A partir daí, basta ir digitando um código para pesquisa ou 0 (zero) para desfazer a conexão. Existe uma sequência de consultas e um pedido de saída no código-fonte abaixo:

```
$ telnet 127.0.0.1 7777
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Tecle <Enter> para iniciar

Pesquisa por código de Tipo de Estabelecimento (0 para sair):
1
Realizando pesquisa, aguarde... Registro encontrado: Escola

Pesquisa por código de Tipo de Estabelecimento (0 para sair):
2
Realizando pesquisa, aguarde... Nenhum registro encontrado!

Pesquisa por código de Tipo de Estabelecimento (0 para sair):
10
Realizando pesquisa, aguarde... Registro encontrado: Parque

Pesquisa por código de Tipo de Estabelecimento (0 para sair):
0

Connection closed by foreign host.
```

Código-fonte 15 – Saída no cliente telnet do *Socket* após consultas e pedido de saída
Fonte: Elaborado pelo autor (2018)

3.3.3 Por que usar *Socket* se posso usar RMI?

Talvez você tenha se perguntado: Por que usar *Sockets* se posso usar RMI, que abstrai mais a comunicação e facilita a programação pelo compartilhamento de interfaces Java? Simples, *Sockets*, como dito no início deste tópico, são mais baixo nível que RMI, pois só ocupam uma porta diretamente no protocolo TCP. Assim, é possível usar *Sockets* para disponibilizar o acesso a um projeto Java e a dispositivos, como *Arduino*, e *Raspberry Pi*, muito usados na prototipação e implementação de projetos IoT.

Seguem os links para download do projeto completo deste capítulo:

Git: https://github.com/FIAP/smartcities-orm/tree/08-socket

Zip: https://github.com/FIAP/smartcities-orm/archive/refs/heads/08-socket.zip

REFERÊNCIAS

CALVERT, Kenneth L.; DANAHOO, Michael J. **TCP/IP sockets in Java:** practical guide for programmers. Burlington (EUA): Morgan Kaufmann, 2011.

MOLINARI, W. **Desconstruindo a Web.** São Paulo: Casa do Código, 2016.

ORACLE. **Lesson:** all about sockets. [s.d]. Disponível em: <https://docs.oracle.com/javase/tutorial/networking/sockets/index.html>. Acesso em: 13 jan. 2021.

ORACLE. **Getting started using Java RMI**. [s.d.]. Disponível em: <https://docs.oracle.com/javase/7/docs/technotes/guides/rmi/hello/hello-world.html>. Acesso em: 13 jan. 2021.