4

Objetos Distribuídos em Java

Helder da Rocha (helder@acm.org)

Objetivo

- O objetivo deste módulo é fornecer os pré-requisitos de computação distribuída necessários à eficiente utilização da tecnologia EJB
  - Background mínimo sobre Java RMI e Java IDL (implementação de CORBA em Java)
  - Desenvolvimento com Java RMI-IIOP (tecnologia usada na comunicação entre EJBs). Consiste do modelo de programação Java RMI com protocolo CORBA (IIOP)
  - Descrição das diferentes maneiras como parâmetros são passados em ambientes distribuídos
- Para mais detalhamento sobre este assunto, consulte o seminário J433 - Objetos Distribuídos

RMI e RPC

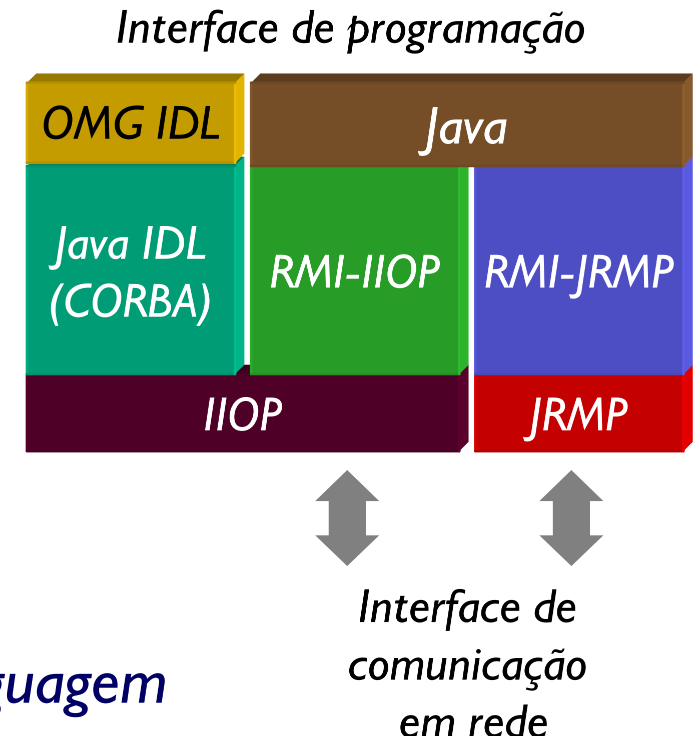
- RMI e RPC são técnicas usadas para isolar, dos clientes e servidores, os detalhes da comunicação em rede
  - Utilizam protocolos padrão, stubs e interfaces
  - Lidam com diferentes representação de dados
- RPC: Remote Procedure Call
  - Chamada procedural de um processo em uma máquina para um processo em outra máquina.
  - Permitem que os procedimentos tradicionais permanecam em múltiplas máquinas, porém consigam se comunicar
- RMI: Remote Method Invocation
  - É RPC em versão orientada a objetos
  - Permite possível chamar métodos em objetos remotos
  - Beneficia-se de características do paradigma OO: herança, encapsulamento, polimorfismo

Dificuldades em RPC

- Para dar a impressão de comunicação local, implementações de RPC precisam lidar com várias dificuldades, entre elas
  - Marshalling e unmarshalling (transformação dos dados em um formato independente de máquina)
  - Diferenças na forma de representação de dados entre máquinas
- Implementações RMI tem ainda que decidir como lidar com particularidades do modelo OO:
  - Como implementar e controlar referências dinâmicas remotas (herança e polimorfismo)
  - Como garantir a não duplicação dos dados e a integridade do encapsulamento?
  - Como implementar a coleta de lixo distribuída?
  - Como implementar a passagem de parâmetros que são objetos (passar cópia rasa, cópia completa, referência remota)?
- Padrões diversos. Como garantir a interoperabilidade?

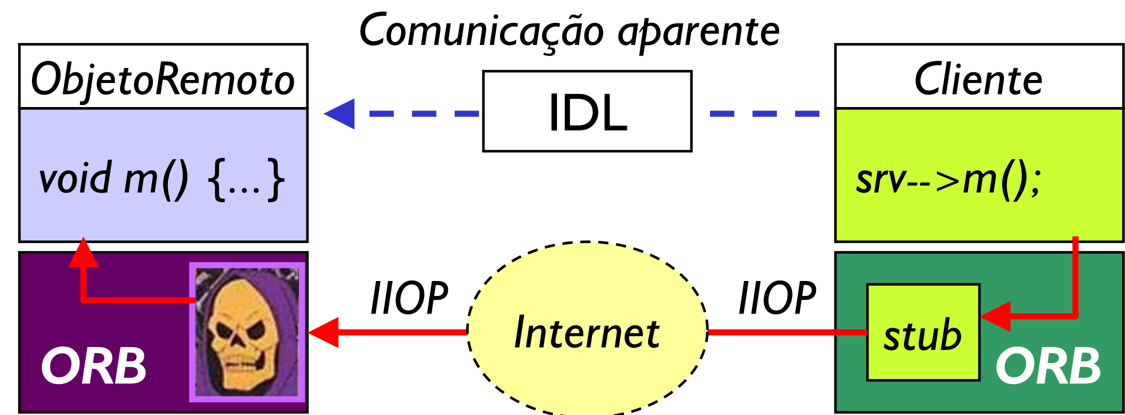
Objetos distribuídos em Java

- Java RMI sobre JRMP
  - Protocolo Java nativo (Java Remote Method Protocol) - Java-to-Java
  - Serviço de nomes não-hierárquico e centralizado
  - Modelo de programação Java: interfaces
- Java IDL: mapeamento OMG IDL-Java
  - Protocolo OMG IIOP (Internet Inter-ORB Protocol) independente de plataforma/linguagem
  - Serviço de nomes (COS Naming) hierárquico, distribuído e transparente quanto à localização dos objetos
  - Modelo de programação CORBA: OMG IDL (language-neutral)
- Java RMI sobre IIOP
  - Protocolo OMG IIOP, COS Naming, transparência de localidade, etc.
  - Modelo de programação Java com geração automática de OMG IDL



Comunicação CORBA: ORB

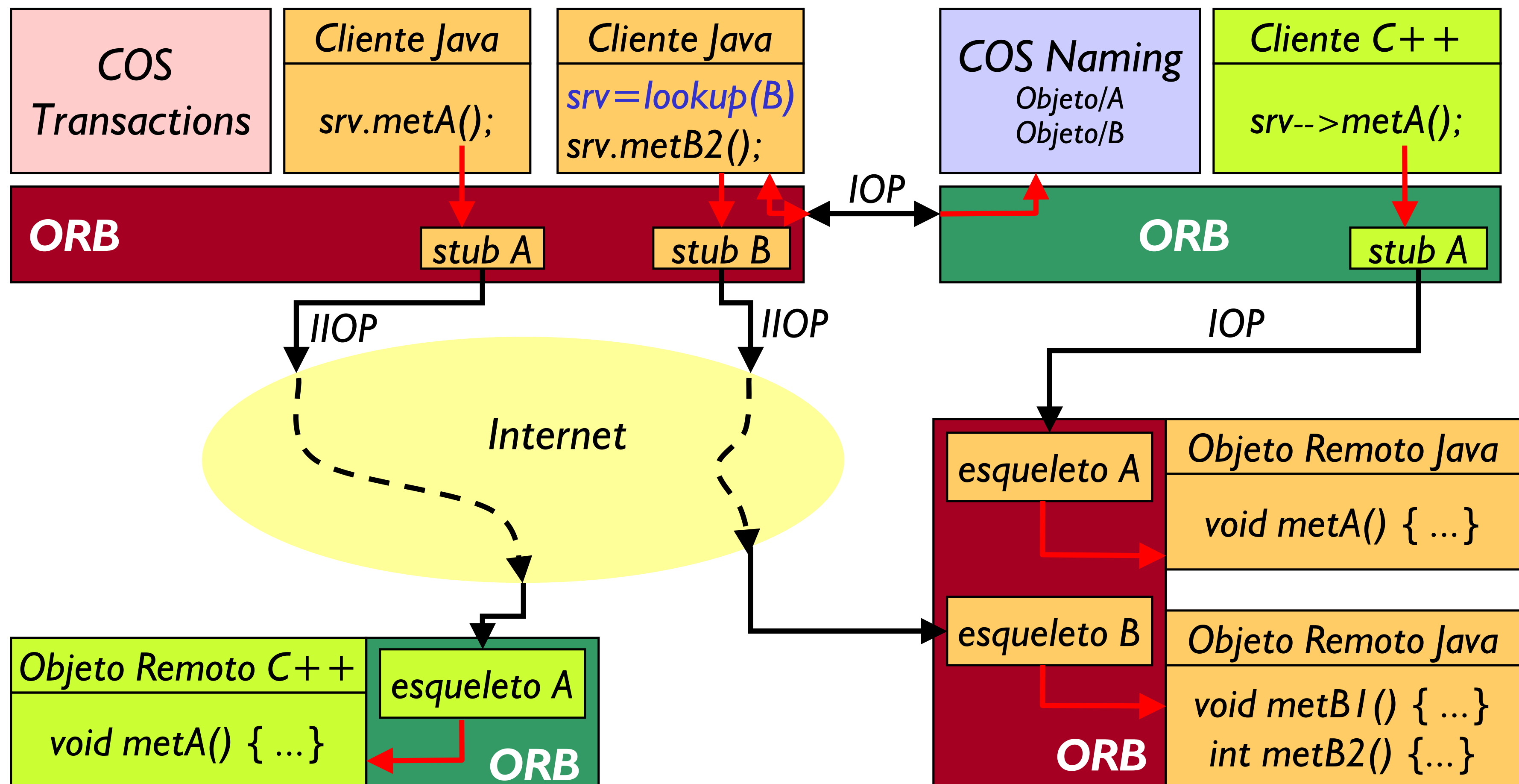
- ***IDL***
  - *Abstração da interface do objeto remoto*
  - *Usada para gerar stubs e esqueletos*



- ***Stub*** *(lado-cliente)*
  - *Transforma os parâmetros em formato independente de máquina (marshalling) e envia requisições para o objeto remoto através do ORB passando o nome do método e os dados transformados*
- ***ORB****: barramento comum*
  - *ORB do cliente passa dados via IIOP para ORB do servidor*
- ***Esqueleto*** *(lado-servidor)*
  - *Recebe a requisição com o nome do método, decodifica (unmarshals) os parâmetros e os utiliza para chama o método no objeto remoto*
  - *Transforma (marshals) os dados retornados e devolve-os para o ORB*

Comunicação CORBA: IIOP e serviços

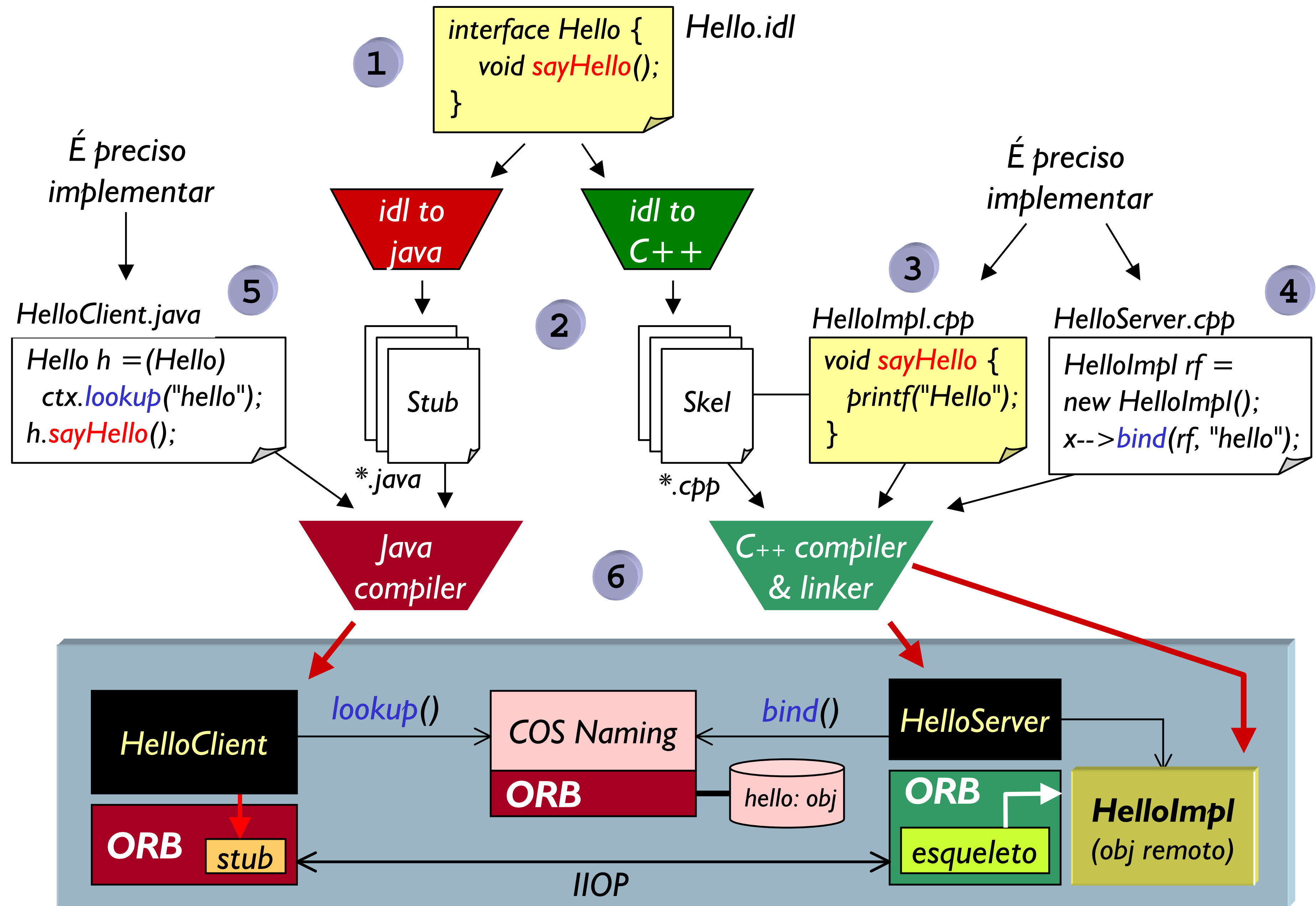
- *Com vários ORBs se comunicando via IIOP, clientes tem acesso a objetos remotos e serviços em qualquer lugar*
  - *ORB encarrega-se de achar o objeto (location transparency)*



Como criar uma aplicação CORBA

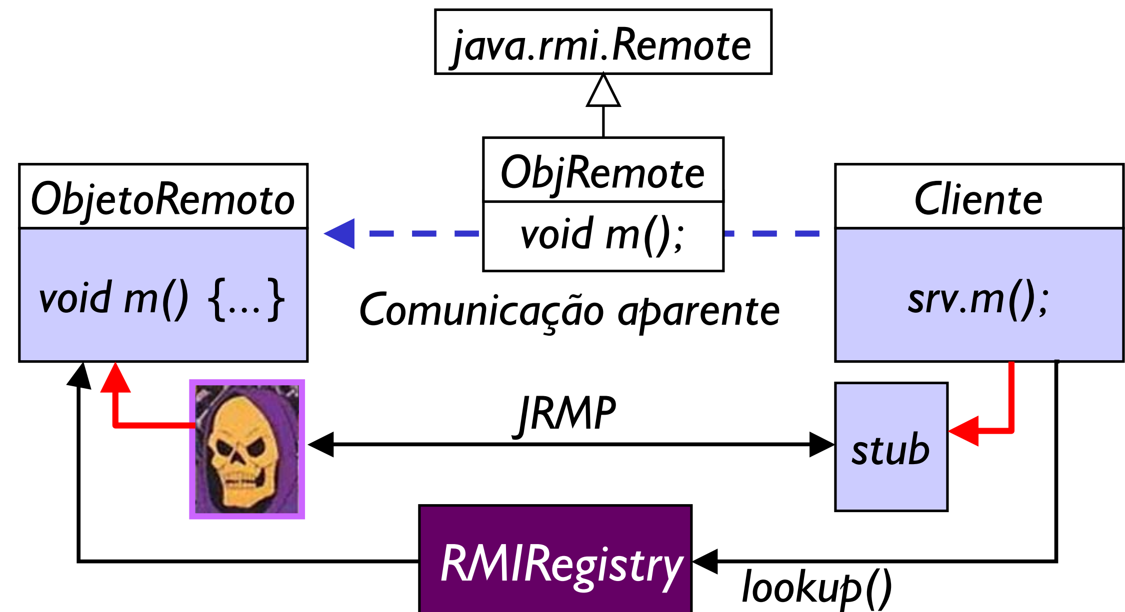
1. Criar uma interface para cada objeto remoto em IDL
2. Compilar a IDL para gerar código de stubs e skeletons
   - Usando ferramenta do ORB (em JavaIDL: idlj)
3. Implementar os objetos remotos
   - Modelo de herança: objeto remoto herda do esqueleto gerado
   - Modelo de delegação (mais flexível): classe "Tie" implementa esqueleto e delega chamadas à implementação
4. Implementar a aplicação servidora
   - Registrar os objetos remotos no sistema de nomes (e usa, opcionalmente, outros serviços)
5. Implementar o cliente
   - Obter o contexto do serviço de nomes (COS Naming)
   - Obtém o objeto remoto através de seu nome
   - Converter objeto para tipo de sua interface: xxxHelper.narrow(obj)
6. Compilar e gerar os executáveis

Do IDL a uma aplicação distribuída



Comunicação RMI

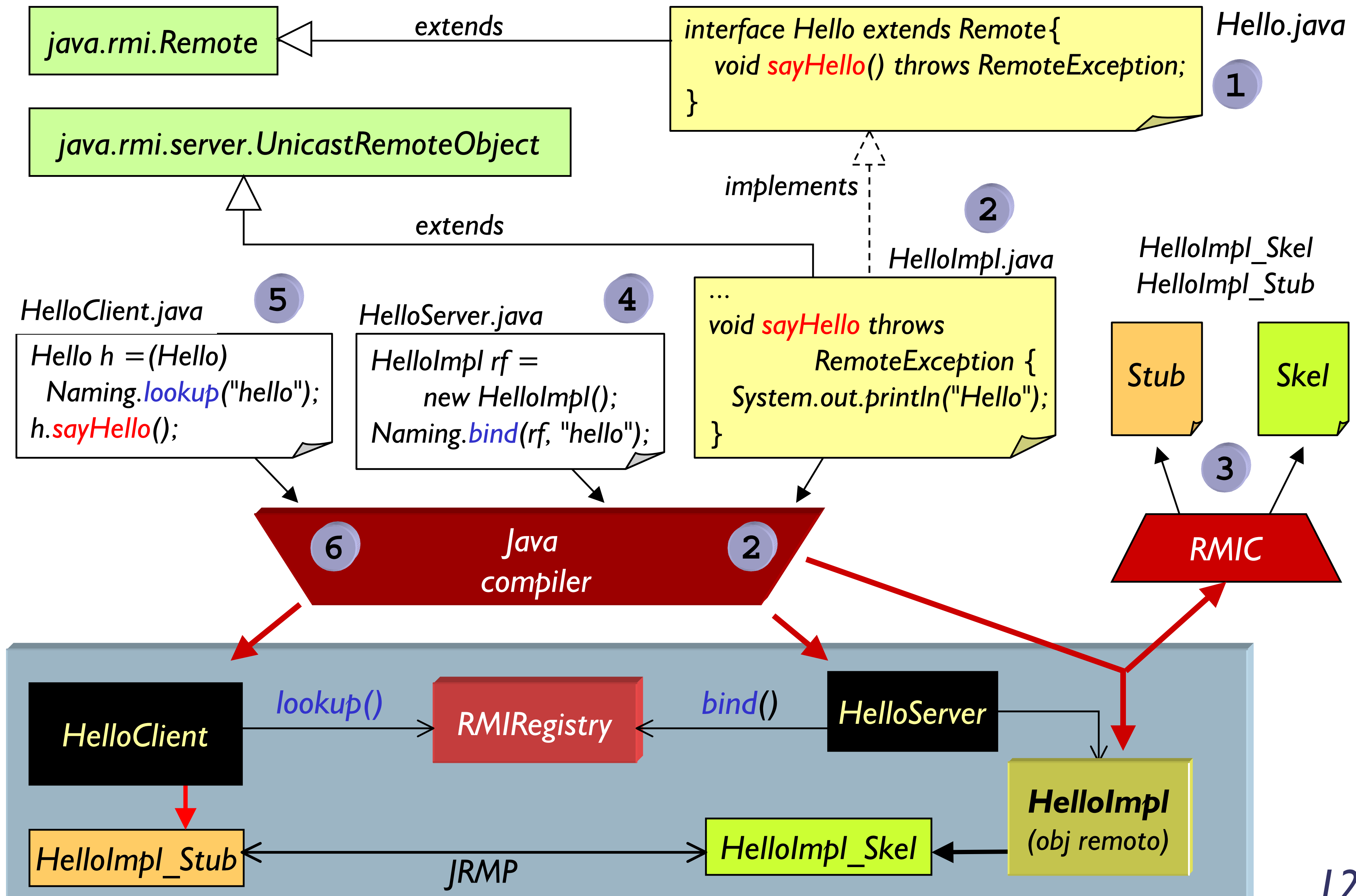
- **Interface Remote**
  - Abstração da interface do objeto remoto
  - Usada para gerar stubs e esqueletos
- **Stub** (lado-cliente)
  - Transforma parâmetros (serializados) e envia requisição pela rede (sockets)
- **Esqueleto** (lado-servidor)
  - Recebe a requisição, desserializa os parâmetros e chama o método no objeto remoto. Depois serializa objetos retornados.
- **RMIRegistry**
  - Servidor registra o objeto usando java.rmi.Naming.bind()
  - Cliente recupera o objeto usando java.rmi.Naming.lookup()



Como criar uma aplicação RMI

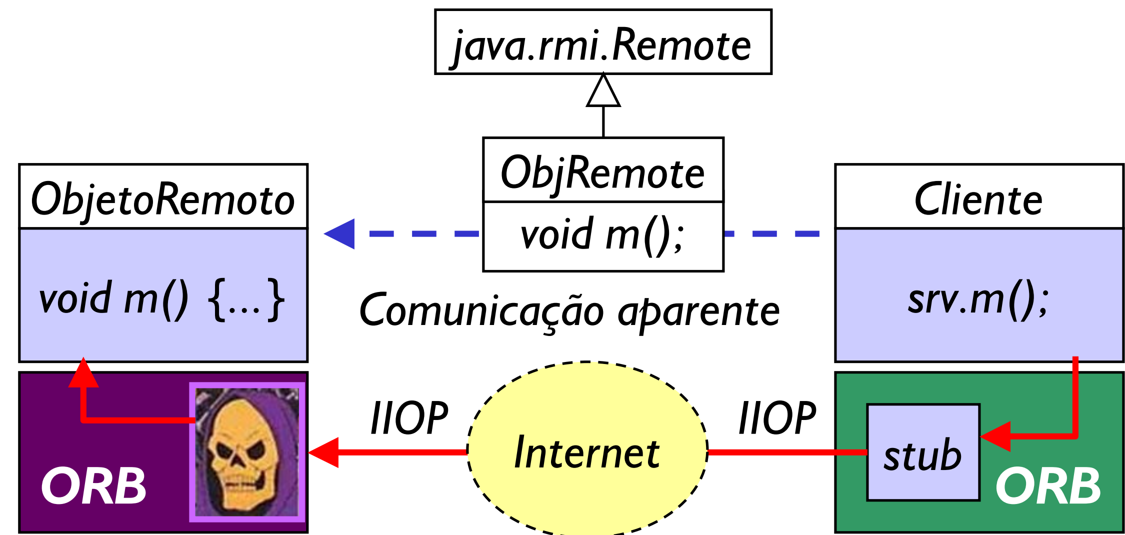
1. Criar subinterface de java.rmi.Remote para cada objeto remoto
   - Interface deve declarar todos os métodos visíveis remotamente
   - Todos os métodos devem declarar java.rmi.RemoteException
2. Implementar e compilar os objetos remotos
   - Criar classes que implementem a interface criada em (1) e estendam java.rmi.server.UnicastRemoteObject (ou Activatable)
   - Todos os métodos (inclusive construtor) provocam RemoteException
3. Gerar os stubs e skeletons
   - Rodar a ferramenta rmic sobre a classe de cada objeto remoto
4. Implementar a aplicação servidora
   - Registrar os objetos remotos no RMIRegistry usando Naming.bind()
   - Definir um SecurityManager (obrigatório p/ download de cód. remoto)
5. Implementar o cliente
   - Definir um SecurityManager e conectar-se ao codebase do servidor
   - Recuperar os objetos remotos usando (Tipo)Naming.lookup()
6. Compilar servidor e cliente

Construção de aplicação RMI



Comunicação RMI-IIOP

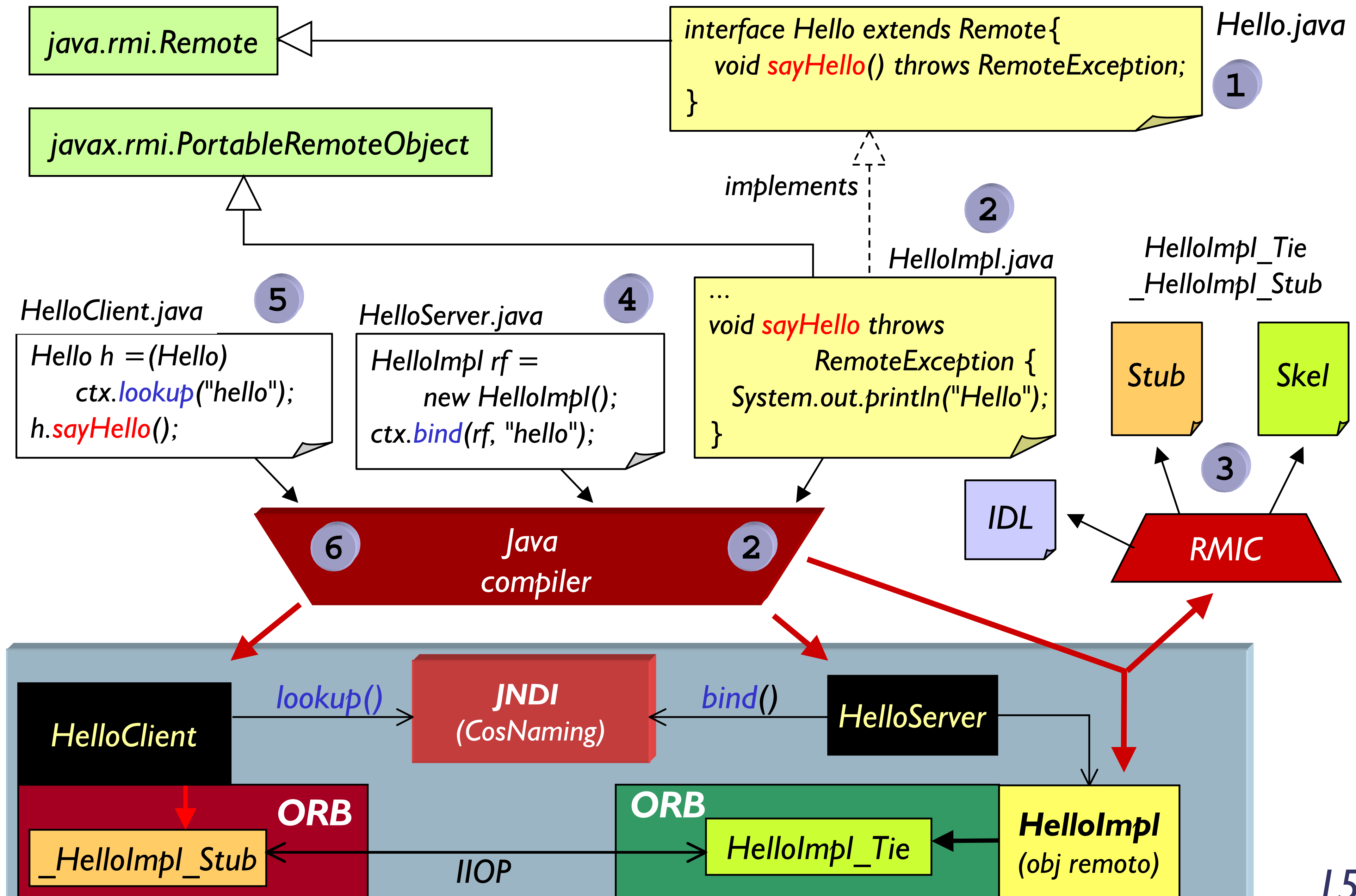
- **Interface Remote**
  - Abstração da interface do objeto remoto
  - Usada para gerar stubs e esqueletos
- **Stub** (lado-cliente)
  - Transforma parâmetros (serializados) em formato independente de máquina e envia requisição pela rede através do ORB
- **ORB**: barramento comum
  - ORB do cliente passa dados via IIOP para ORB do servidor
- **Esqueleto** (lado-servidor)
  - Recebe a requisição do ORB e desserializa os parâmetros
  - Chama o método do objeto remoto
  - Transforma os dados retornados e devolve-os para o ORB



Como criar uma aplicação RMI-IIOP

1. Criar subinterface de java.rmi.Remote para cada objeto remoto
   - Interface deve declarar todos os métodos visíveis remotamente
   - Todos os métodos devem declarar java.rmi.RemoteException
2. Implementar e compilar os objetos remotos
   - Criar classes que implementem a interface criada em (1) e estendam javax.rmi.PortableRemoteObject
   - Todos os métodos (inclusive construtor) provocam RemoteException
3. Gerar os stubs e skeletons (e opcionalmente, os IDLs)
   - Rodar a ferramenta rmic -iiop sobre a classe de cada objeto remoto
4. Implementar a aplicação servidora
   - Registrar os objetos remotos no COSNaming usando JNDI
   - Definir um SecurityManager
5. Implementar o cliente
   - Definir um SecurityManager e conectar-se ao codebase do servidor
   - Recuperar objetos usando JNDI e PortableRemoteObject.narrow()
6. Compilar

Construção de aplicação RMI-IIOP



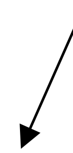
Interface Remote

- *Declara os métodos que serão visíveis remotamente*
  - *Todos devem declarar provocar RemoteException*
- *Indêntica à interface usada em RMI sobre JRMP*

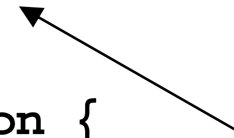
```
package hello.rmiop;

import java.rmi.*;

public interface Hello extends Remote {

    public String sayHello()
                  throws RemoteException ;
    public void sayThis(String toSay)
                  throws RemoteException;

}
```

Implementação do objeto remoto

Classe base para todos os objetos remotos RMI-IIOP



```
package hello.rmiop;

import java.rmi.*;

public class HelloImpl
                    extends javax.rmi.PortableRemoteObject
                    implements Hello {

    private String message = "Hello, World!";
    public HelloImpl() throws RemoteException { }

    public String sayHello()
                    throws RemoteException {
        return message;
    }
    public void sayThis(String toSay)
                    throws RemoteException {
        message = toSay;
    }
}
```



Construtor declara que pode provocar RemoteException!

Geração de Stubs e Skeletons

- *Tendo-se a implementação de um objeto remoto, pode-se gerar os stubs e esqueletos*

```
> rmic -iiop hello.rmiop.HelloImpl
```

- *Resultados*
  - *_Hello_Stub.class*
  - *_HelloImpl_Tie.class - este é o esqueleto!*
- *Para gerar, opcionalmente, uma (ou mais) interface IDL compatível use a opção -idl*

```
> rmic -iiop -idl hello.rmiop.HelloImpl
```

- *Resultado*

Hello.idl

```
#include "orb.idl"
module hello {
  module rmiop {
    interface Hello {
      ::CORBA::WStringValue sayHello( );
      void sayThis(in ::CORBA::WStringValue arg0 );
    };
  };
};
```

Tipos definidos em orb.idl (equivalem a wstring)

Servidor e rmi.policy

```
package hello.rmiop;
import java.rmi.*;
import javax.naming.*;

public class HelloServer {
   public static void main (String[] args) {
      if (System.getSecurityManager() == null)
         System.setSecurityManager(new RMISecurityManager());
      try {
         Hello hello = new HelloImpl();
         Context initCtx = new InitialContext(System.getProperties());
         initCtx.rebind("hellormiop", hello);
         System.out.println("Remote object bound to 'hellormiop'.");
      } catch (Exception e) {
         if (e instanceof RuntimeException)
            throw (RuntimeException)e;
         System.out.println("" + e);
      }
   }
}
```

SecurityManager viabiliza download de código

Informações de segurança e serviço de nomes e contexto foram inicial passadas na linha de comando

Associando o objeto com um nome no serviço de nomes

Arquivo de politicas de segurança para uso pelo SecurityManager

rmi.policy

```
grant {
    permission java.net.SocketPermission "*:1024-65535", "connect, accept, resolve";
    permission java.net.SocketPermission "*:80", "connect, accept, resolve";
    permission java.util.PropertyPermission "*", "read, write";
};
```

Cliente

```
package hello.rmiop;

import java.rmi.*;
import javax.naming.*;

public class HelloClient {
  public static void main (String[] args)  {
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
    }
    try {
      Context initCtx = new InitialContext(System.getProperties());
      Object obj = initCtx.lookup("hellormiop");
      Hello hello = (Hello)
         javax.rmi.PortableRemoteObject.narrow(obj,
                                               hello.Hello.class);
      System.out.println(hello.sayHello());
      hello.sayThis("Goodbye, Helder!");
      System.out.println(hello.sayHello());

    } catch (Exception e) {
      if (e instanceof RuntimeException)
         throw (RuntimeException)e;
      System.out.println("" + e);
    }
  }
}
```

Obtenção do objeto com associado a"hellormiop" no contexto inicial do serviço de nomes

Não basta fazer cast! O objeto retornado é um objeto CORBA (e não Java) cuja raiz não é java.lang.Object mas org.omg.CORBA.Object narrow transforma a referência no tipo correto

Objetos serializáveis

- Objetos (parâmetros, tipos de retorno) enviados pela rede via RMI ou RMI-IIOP precisam ser serializáveis
  - **Objeto serializado**: objeto convertido em uma representação binária (tipo BLOB) reversível que preserva informações da classe e estado dos seus dados
  - Serialização representa em único BLOB todo o estado do objeto e de todas as suas dependências, recursivamente
  - Usado como formato "instantâneo" de dados
  - Usado por RMI para passar parâmetros pela rede
- Como criar um objeto serializável (da forma default*)
  - Acrescentar "implements java.io.Serializable" na declaração da classe e marcar os campos não serializáveis com o modificador transient
  - Garantir que todos os campos da classe sejam (1) valores primitivos, (2) objetos serializáveis ou (3) campos declarados com transient

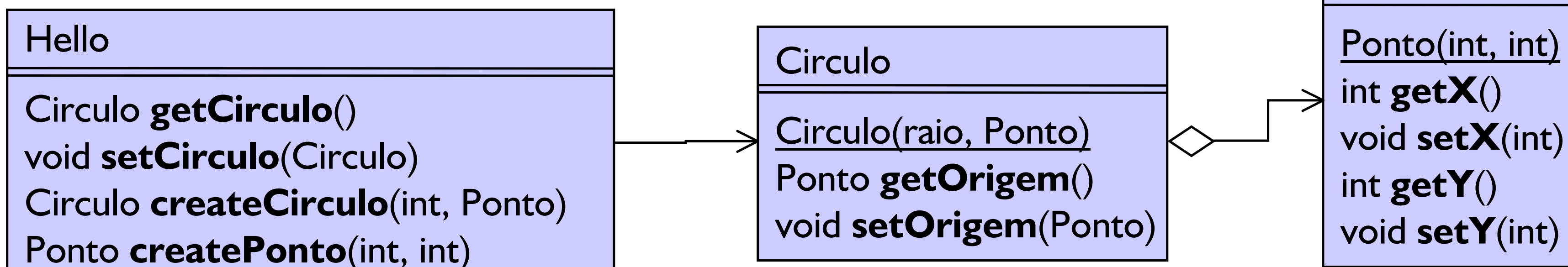
* É possível implementar a serialização de forma personalizada implementando métodos writeObject(), readObject(), writeReplace() e readResolve(). Veja especificação.

Passagem de parâmetros em objetos remotos

- Na chamada de um método remoto, todos os parâmetros são copiados de uma máquina para outra
  - Métodos recebem cópias serializadas dos objetos em passados argumentos
  - Métodos que retornam devolvem uma cópia do objeto
- Diferente da programação local em Java!
  - Quando se passa um objeto como parâmetro de um método a referência é copiada mas o objeto não
  - Quando um método retorna um objeto ele retorna uma referência àquele objeto
- Questões
  - O que acontece quando você chama um método de um objeto retornado por um método remoto?

Passagem de parâmetros local

- *Suponha o seguinte relacionamento**



- *Localmente, com uma referência do tipo Hello, pode-se*
  - *(a) Obter um Circulo*

```
Circulo c1 = hello.getCirculo();
```

  - *(b) Trocar o Circulo por outro*

Este método chama new Circulo(raio, Ponto)

```
hello.setCirculo(hello.createCirculo(50, new Ponto(30, 40));
```

  - *(c) Mudar o objeto Ponto que pertence ao Circulo criado em (b)*

```
Circulo c2 = hello.getCirculo();
c2.setOrigem(hello.createPonto(300, 400));
```

  - *(d) Mudar o valor da coordenada x do ponto criado em (c)*

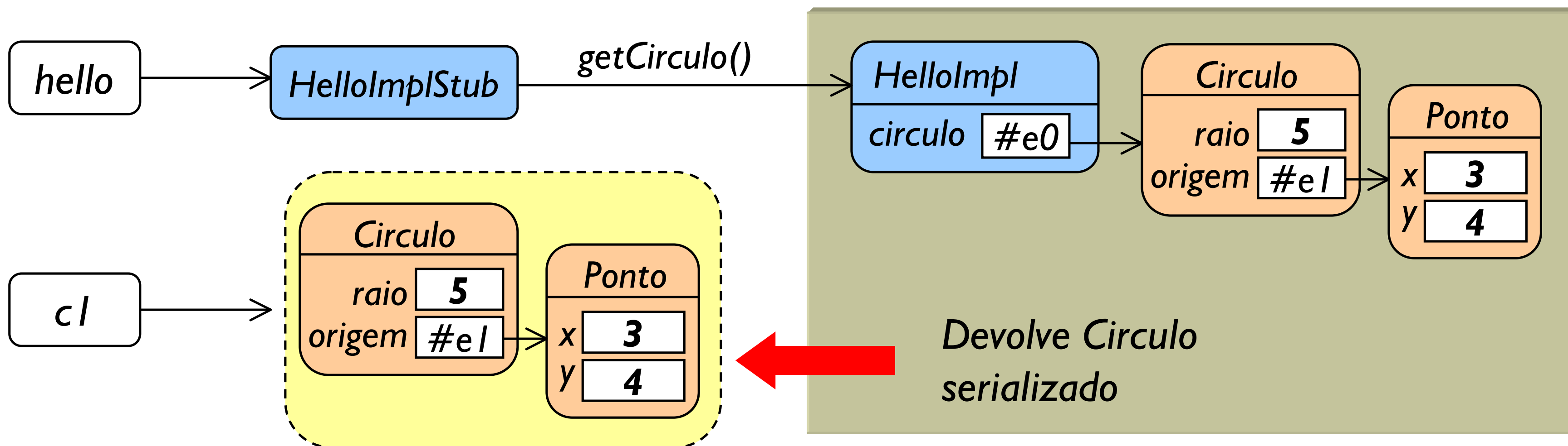
```
c2.getOrigem().setX(3000);
```

... e remotamente?

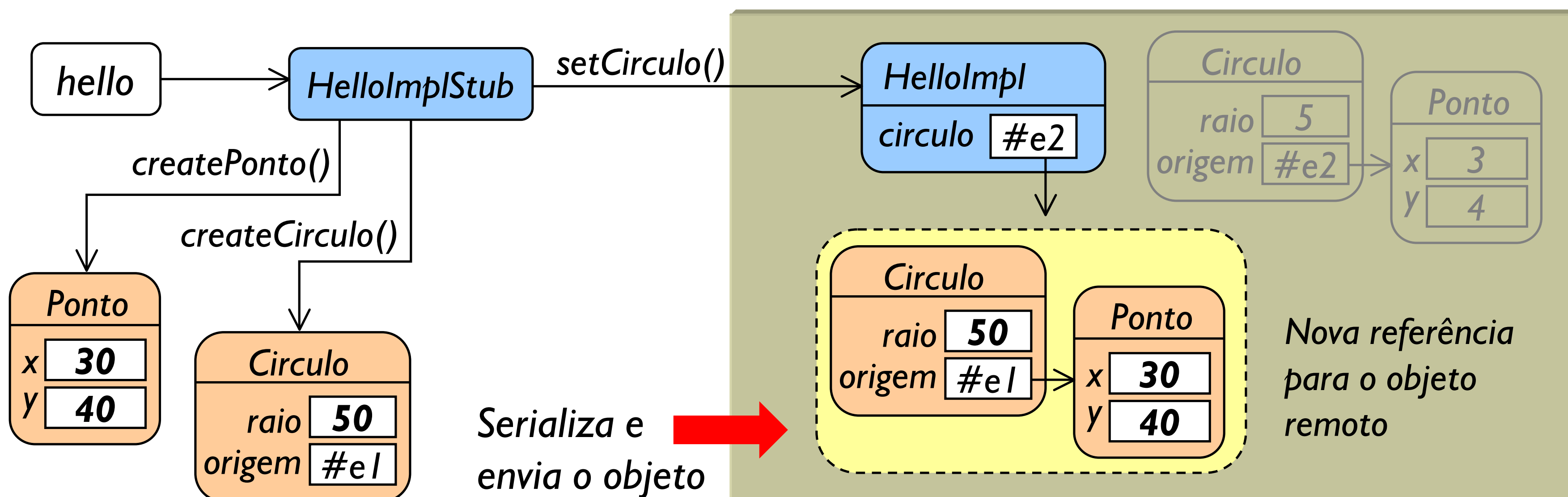
* *veja cap03/src*

Passagem de parâmetros em RMI

a `Circulo c1 = hello.getCirculo();` *// obtém cópia serializada do círculo remoto*

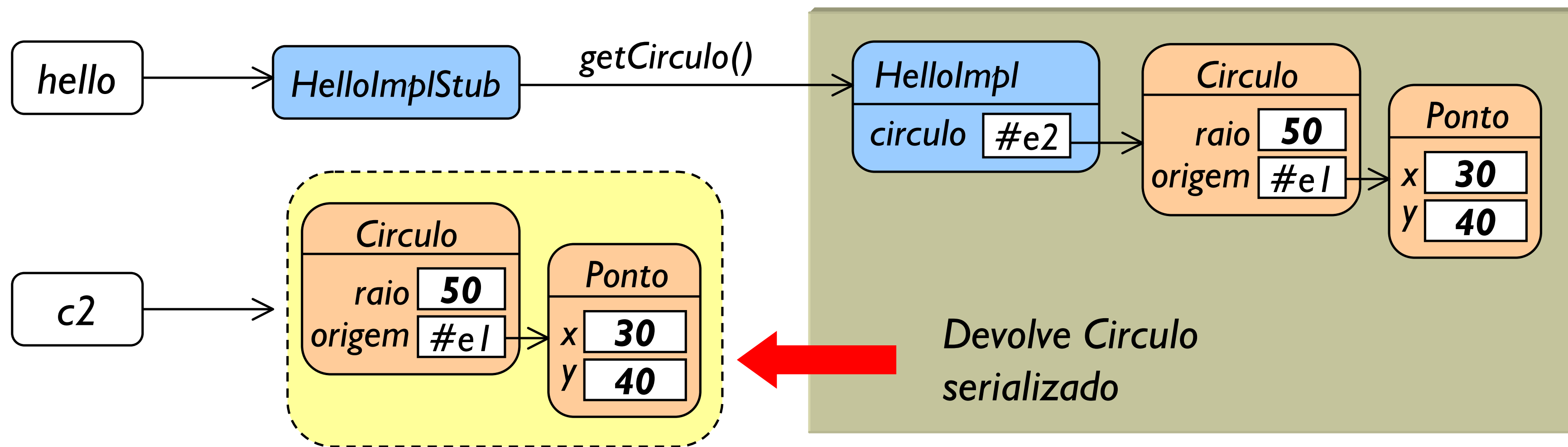


b `hello.setCirculo(hello.createCirculo(50, hello.createPonto(30, 40)));`



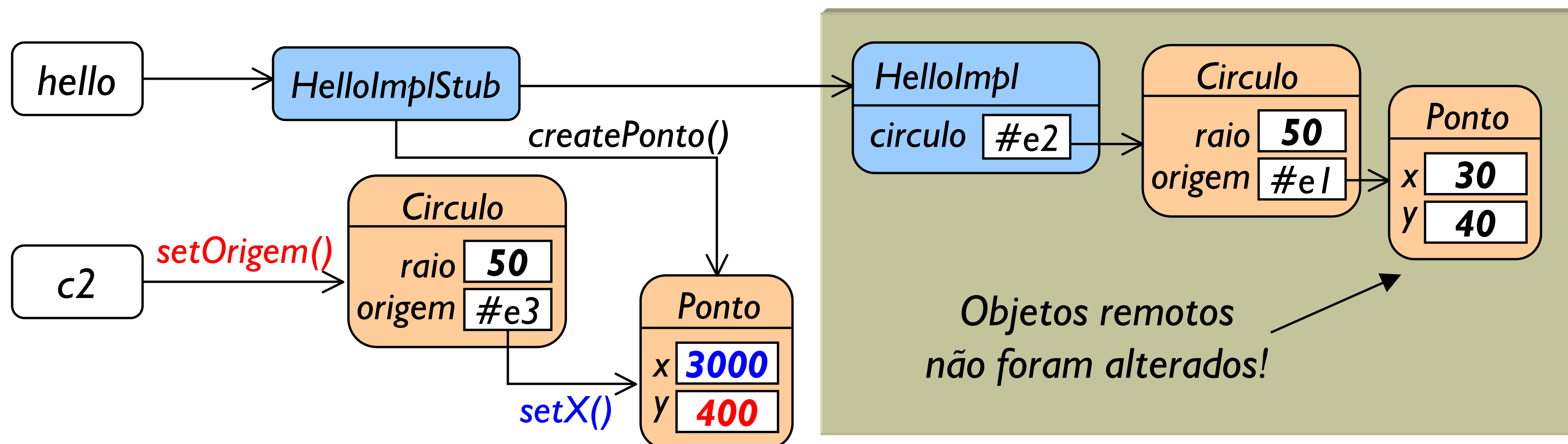
Conseqüências da passagem por valor

a `Circulo c2 = hello.getCirculo();` *// obtém cópia serializada do círculo remoto*



c `c2.setOrigem(hello.createPonto(300, 400));` *// Circulo **local**; Ponto **local**!*

d `c2.getOrigem().setX(3000);` *// altera objeto Ponto **local!***

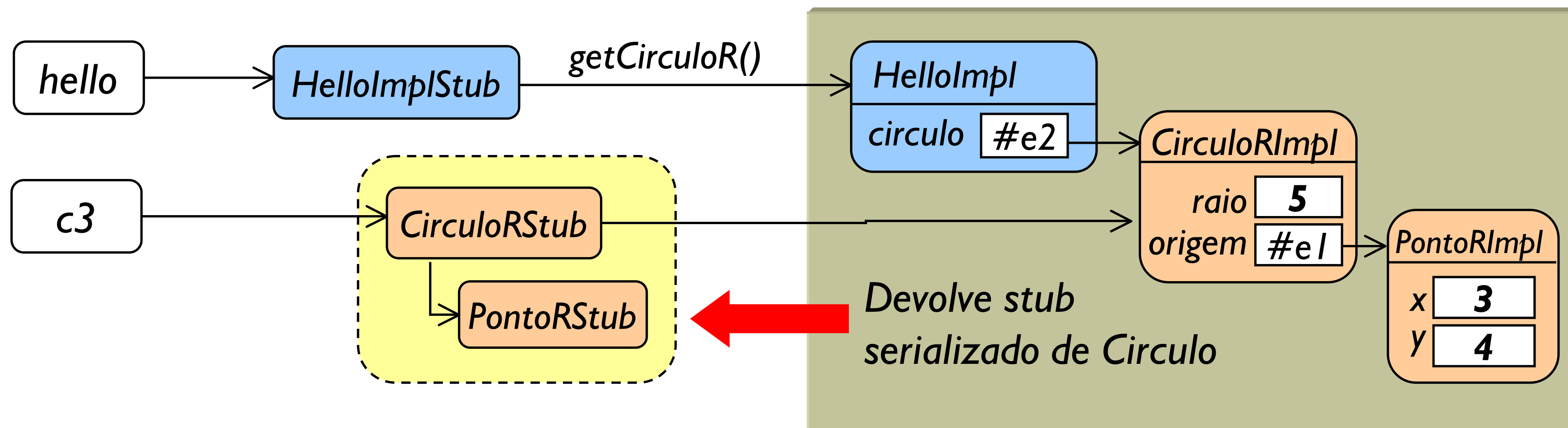


Passagem de parâmetros por referência

- Aplicações RMI sempre passam valores através da rede
  - A referência local de um objeto só tem utilidade local
- Desvantagens
  - Objetos cuja interface é formada pelos seus componentes (alteração nos componentes não é refletida no objeto remoto)
  - Objetos grandes (demora para transferir)
- Solução: passagem por referência
- RMI simula passagem por referência através de stubs
  - Em vez de devolver o objeto serializado, é devolvido um stub que aponta para objeto remoto (objeto devolvido, portanto, precisa ser "objeto remoto", ou seja, implementar java.rmi.Remote)
  - Se cliente altera objeto recebido, stub altera objeto remoto
- Como implementar?
  - Basta que parâmetro ou tipo de retorno seja objeto remoto
  - Objetos remotos são sempre retornados como stubs (referências)

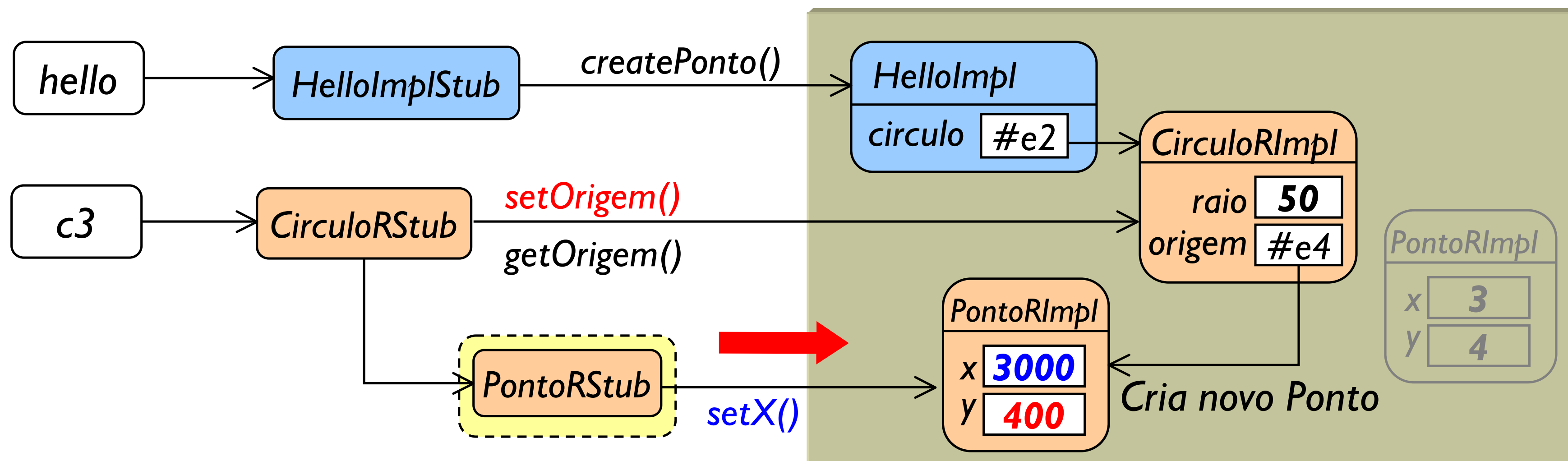
Passagem por referência

a `CirculoR c3 = hello.getCirculoR();` // obtém referência (stub) para círculo remoto



c `c2.setOrigem(hello.createPonto(300, 400));` // CirculoR remoto; PontoR remoto

d `c2.getOrigem().setX(3000);` // altera PontoR remoto (que acaba de ser criado)



Passagem de parâmetros: resumo

- *Quando você passa um objeto como parâmetro de um método, ou quando um método retorna um objeto...*
  - *... em programação Java local*
    - *Referência local (**número** que contém endereço de memória) do objeto é passada*
  - *... em Java RMI (RMI-JRMP ou RMI-IIOP)*
    - *Se tipo de retorno ou parâmetro for objeto remoto (implementa java.rmi.Remote), stub (**objeto** que contém referência remota) é serializado e passado;*
    - *Se parâmetro ou tipo de retorno for objeto serializável mas não remoto (não implementa java.rmi.Remote), uma copia serializada do objeto é passada*

Veja demonstração: Rode, a partir do diretório cap04, `ant build`*, depois, em janelas separadas,* `ant orbd`*,* `ant runrmiopserver` *e* `ant runrmiopclient`

Conclusão

- *Java oferece três opções nativas para implementar sistemas de objetos distribuídos*
  - *Java RMI sobre Java RMP*
  - *Java RMI sobre IIOP*
  - *Java IDL*
- *Enterprise JavaBeans utilizam RMI sobre IIOP, logo*
  - *EJBs comportam-se como objetos remotos RMI*
  - *EJBs podem se comunicar com objetos CORBA*
- *Objetos em aplicações EJB são passados*
  - *Por referência local, em EJBs com interface local*
  - *Por referência remota (stub RMI-IIOP), em EJBs com interface java.rmi.Remote*
  - *Por valor, em value objects (data-transfer objects)*

Exemplos

- Há dois exemplos disponíveis com RMI (cap04/exemplos)
  - Rode ant -projecthelp e leia os README.txt
- hellormi: usa RMI sobre JRMP
  - Gera dois JARs. Um com o cliente e o segundo com o servidor. Ambos abrem interfaces gráficas. A partir deles é possível mandar mensagens entre máquinas.
  - Inicie antes o RMI Registry
- od: demonstra Java IDL e RMI (IIOP e JRMP)
  - Mostra o exemplo com passagem de objetos serializados e passagem de referências (stubs para objetos remotos)
  - Inicie o ORBD para as aplicações IIOP (Corba, RMI-IIOP) e RMI Registry para as aplicações JRMP (RMI, RMI-JNDI)

Execução do exemplo OD

- *Usando Ant; no subdiretório cap04/exemplos, monte a aplicação (que é similar à mostrada nos slides) com*
  ```
  > ant buildrmiop
  ```
- *Depois, em janelas diferentes, inicie os servidores e cliente digitando*
  ```
  > ant orbd
  > ant runrmiopserver
  > ant runrmiopclient
  ```
- *Se não quiser usar o Ant, inicie o ORB em linha de comando com*
  ```
  > orbd -ORBInitialPort 1900 -ORBInitialHost localhost &
  ```
  rode o servidor usando
  ```
  > java hello.rmiop.HelloServer
   -Djava.rmi.server.codebase=file:///aulaj2ee/cap03/build/
   -Djava.security.policy=../lib/rmi.policy
   -Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
   -Djava.naming.provider.url=iiop://localhost:1900
  Remote object bound to 'hellormiop'.
  ```
  e o cliente com
  ```
  > java hello.rmiop.HelloClient <mesmas propriedades -D do servidor>
  Hello, World!
  Goodbye!
  ```

Exercício

- *1. Utilize os arquivos fornecidos e transforme Produto e ProdutoFactory em objetos remotos (cap04/exercicios)*
  - *a) Altere as interfaces Produto e ProdutoFactory para que sejam interfaces Remote*
  - *b) Transforme as classes *Impl.java para que sejam objetos remotos RMI-IIOP*
  - *c) Gere os stubs e skeletons (use o alvo build do ant, que está pronto, se desejar)*
  - *d) Complete a classe ProdutoServer para que faça o bind do objeto remoto ProdutoFactory*
  - *e) Complete a classe ProdutoClient para que faça o lookup de ProdutoFactory, utilize-o para criar alguns Produtos remotos e chamar alguns métodos (siga o roteiro descrito nos comentários)*
  - *e) Rode o ORBD, em uma janela, depois o servidor e o cliente em janelas separadas.*

Exercício extra (opcional)

- 2. Alterar a implementação de objeto remoto do exercício anterior para utilizar banco de dados
  - a) Inicie o JBoss/HSQLDB e rode ant create-table no buildfile fornecido para criar a tabela no banco (consulte o script SQL usado para criar a tabela no arquivo sql/create-table.sql)
  - b) Utilize a classe ProdutoDAO fornecida e o value object ProdutoVO (retornado por um dos métodos do DAO)
  - c) Altere ProdutoImpl para que use o DAO (Data Access Object) e o value object (se desejar, utilize as classes semi-prontas)
- 3. Registre o objeto remoto no JBoss
  - a) Altere o arquivo jndi.properties, localizado no diretório lib/ para que utilize as classes e pacotes do driver JNDI do JBoss (estão comentados)
  - b) Configure o build.xml para que utilize os JARs do cliente JBoss, necessários para JNDI, em $JBOSS_HOME/client/lib

Fontes

[1] *Sun Microsystems. Java IDL Documentation. http://java.sun.com/j2se/1.4/docs/guide/idl/*
Ponto de partida para tutoriais e especificações sobre Java IDL (parte da documentação do J2SDK 1.4)

[2] *Sun Microsystems. RMI-IIOP Tutorial. http://java.sun.com/j2se/1.4/docs/guide/rmi-iiop/*
Ponto de partida para tutoriais e especificações sobre Java RMI-IIOP (parte da documentação do J2SDK 1.4)

[3] *Ed Roman et al. Mastering EJB 2, Appendixes A and B: RMI-IIOP, JNDI and Java IDL*
http://www.theserverside.com/books/masteringEJB/index.jsp
Contém um breve e objetivo tutorial sobre os três assuntos

[4] *Jim Farley. Java Distributed Computing. O'Reilly and Associates, 1998. Esta é a primeira edição (procure a segunda). Compara várias estratégias de computação distribuída em Java.*

[5] *Helder da Rocha, Análise Comparativa de Dempenho entre Tecnologias Distribuídas Java. UFPB, 1999, Tese de Mestrado.*

[6] *Qusay H. Mahmoud Distributed Programming with CORBA (Manning)*
http://developer.java.sun.com/developer/Books/corba/ch11.pdf
Breve e objetivo tutorial CORBA

[7] *Qusay H. Mahmoud Distributed Java Programming with RMI and CORBA*
http://developer.java.sun.com/developer/technicalArticles/RMI/rmi_corba/ Sun, Maio 2002.
Artigo que compara RMI com CORBA e desenvolve uma aplicação que transfere arquivos remotos em RMI e CORBA

Curso J530: Enterprise JavaBeans

Revisão 2.0 - Junho de 2003

© 2001-2003, Helder da Rocha

(helder@acm.org)