RMI-IIOP e Enterprise JavaBeans

Helder da Rocha
www.argonavis.com.br

Objetos Distribuídos

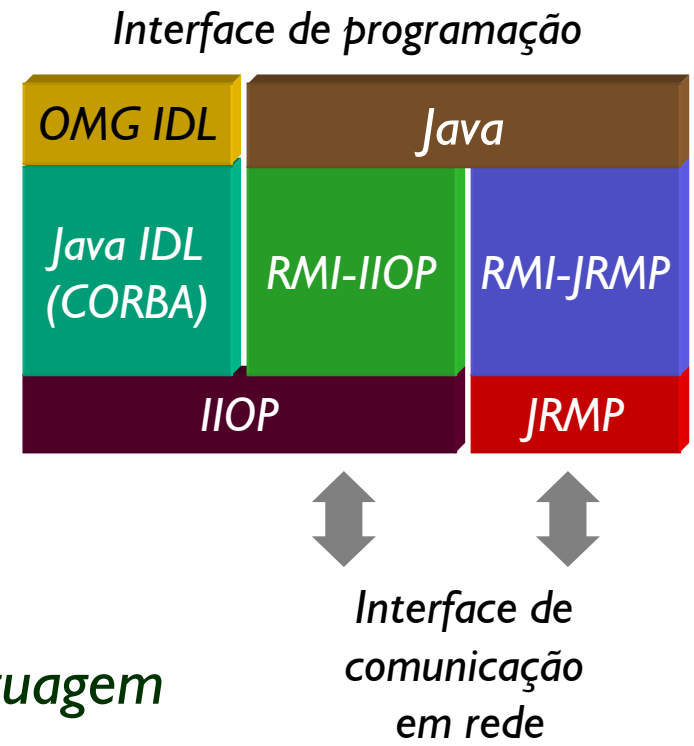
- A comunicação em rede pode ser realizada...
  - Usando soquetes e portas, lidando com todas as dificuldades da comunicação em rede
  - Usando RPC para fazer chamadas remotas como se fossem locais, abstraindo detalhes da comunicação em rede
- RPC: Remote Procedure Call
  - Permite que um processo chame outro em outra máquina.
  - Procedimentos são distribuídos em múltiplas máquinas
- RMI: Remote Method Invocation
  - Versão orientada a objetos de RPC
  - Permite chamar métodos em objetos remotos
  - Objetos podem ser distribuídos em múltiplas máquinas
  - Implementações RMI: Microsoft DCOM, CORBA, Java RMI

Características de Objetos Distribuídos

- As soluções que implementam objetos distribuídos geralmente tem uma estrutura que consiste de
  - Serviço de registro de objetos: serviço que mapeia um nome a um objeto para que possa ser localizados pelas aplicações que querem usar seus serviços
  - Interface de comunicação: documento ou classe que declara a interface (métodos, parâmetros, tipos de retorno, etc.) que podem ser chamados no objeto. É o principal instrumento de comunicação.
  - Infraestrutura de comunicação: barramento que utiliza um protocolo comum, stubs que encapsulam código de rede do cliente e ties que encapsulam servidores.
  - Formato de dados comum usado na comunicação

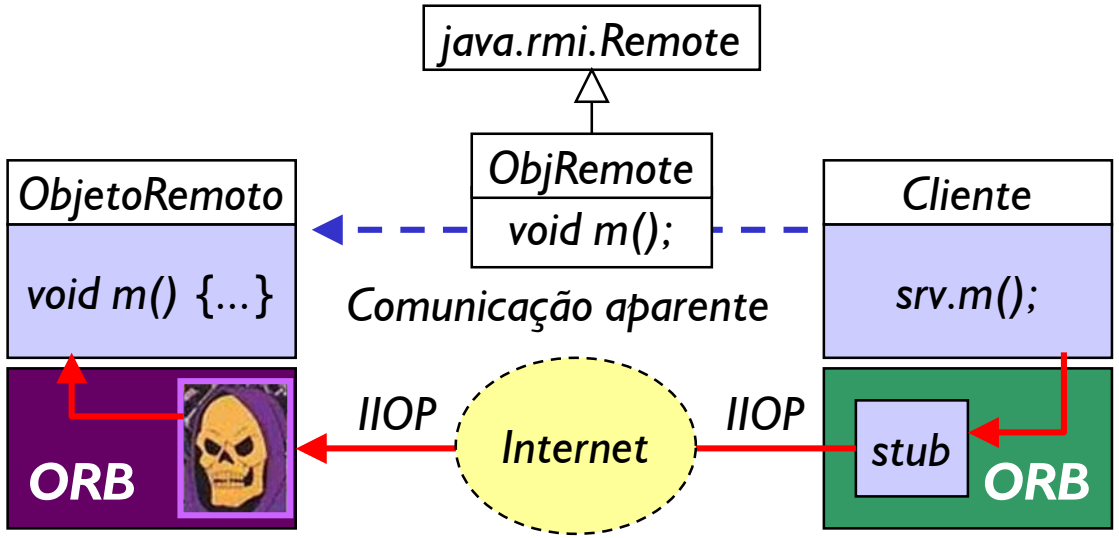
Objetos distribuídos em Java

- *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 RMI-IIOP

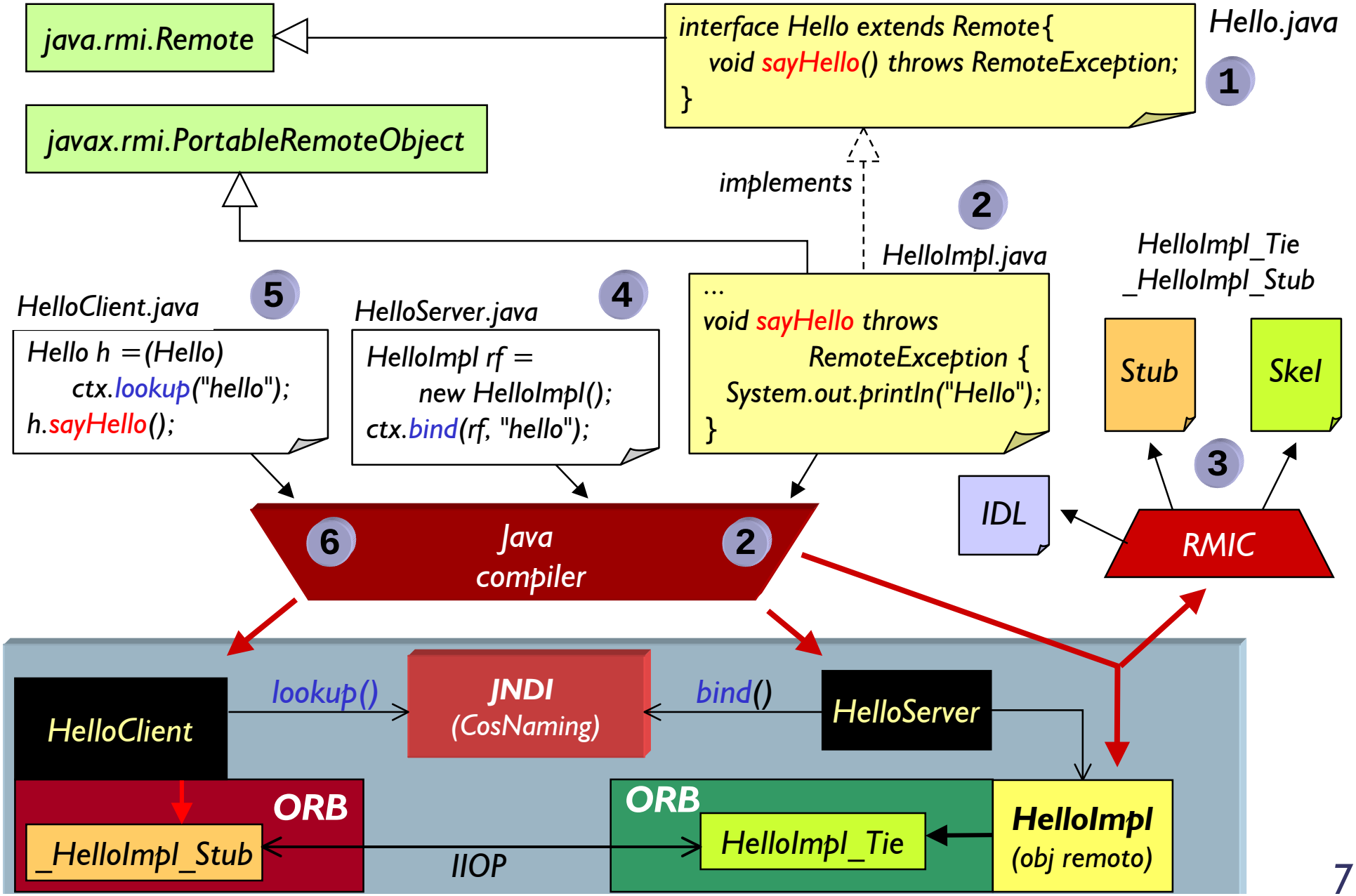
- **Interface Remote**
  - Abstração da interface do objeto remoto
  - Usada para gerar stubs e esqueletos
- **Stub** (lado-cliente)
  - Transforma parâmetros (serializados) em formato independente de máquina e envia requisição pela rede através do ORB
- **ORB**: barramento comum
  - ORB do cliente passa 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

```
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;
    }
}
```

Classe base para todos os objetos remotos RMI-IIOP

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

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.

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

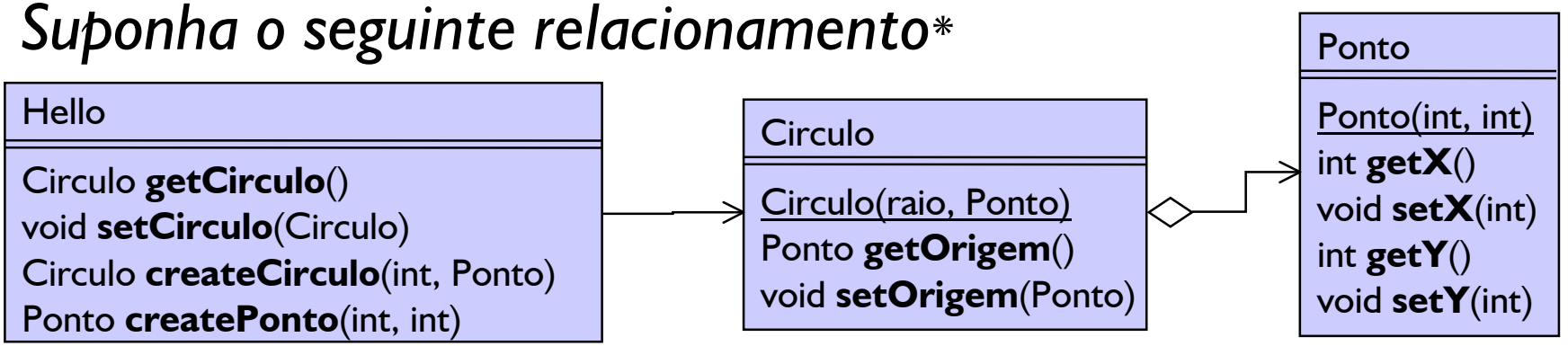
* É possível implementar a serialização de forma personalizada implementando métodos writeObject(), readObject(), writeReplace() e readResolve(). Veja especificação.

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();
```

Este método chama new Circulo(raio, Ponto)

  - *(b) Trocar o Circulo por outro*

```
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)*

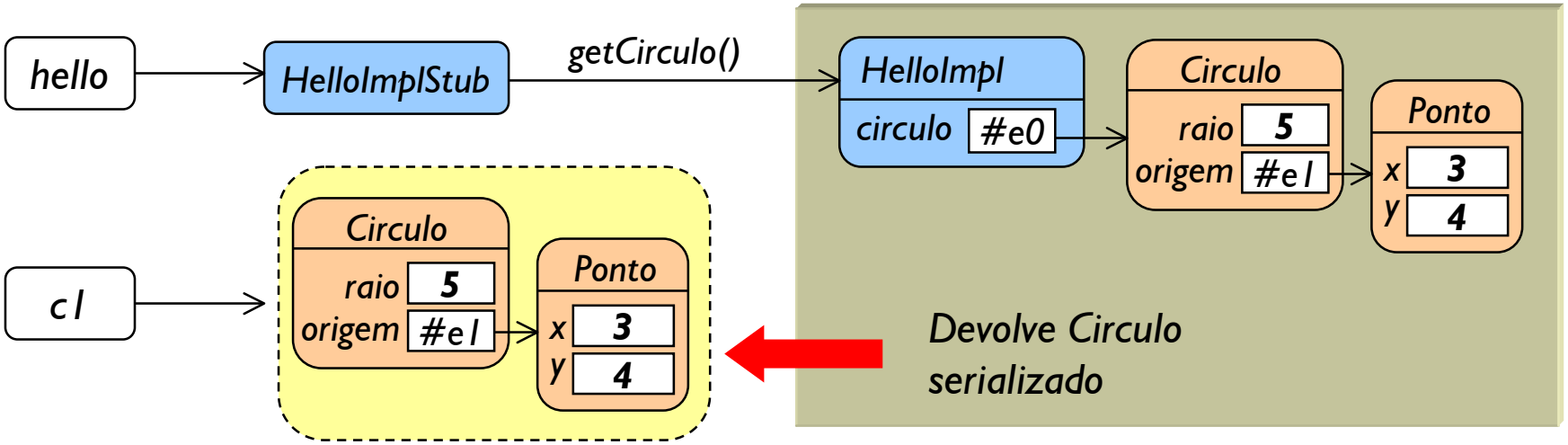
```
c2.getOrigem().setX(3000);
```

... e remotamente?

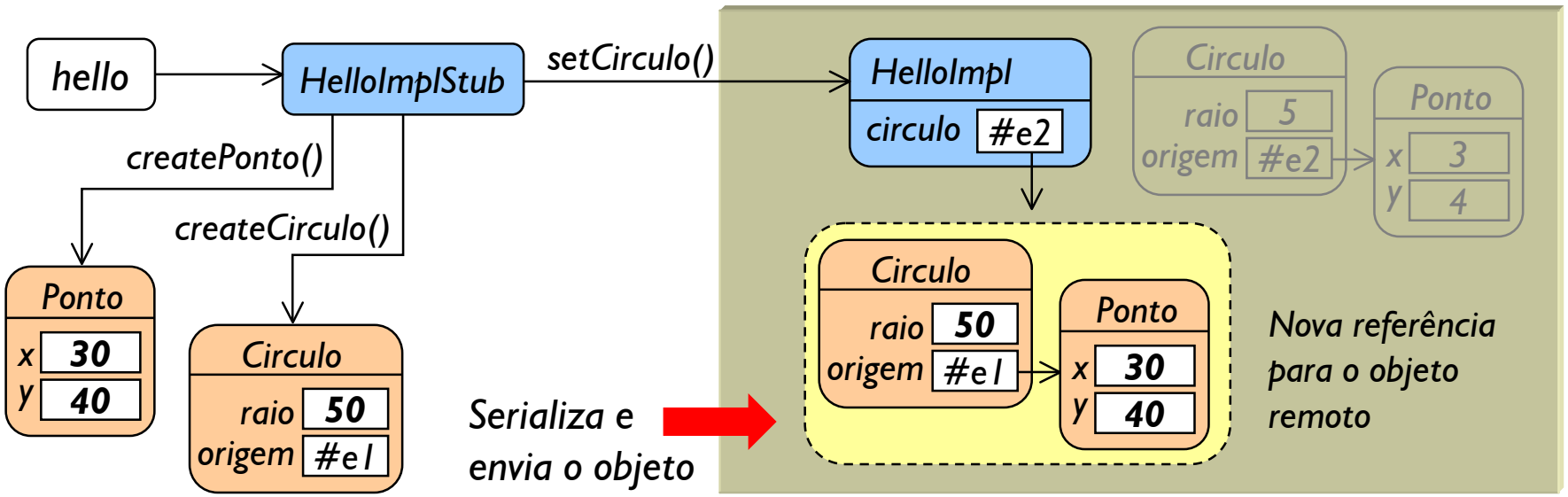
* *veja cap03/src*

Passagem de parâmetros em RMI

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

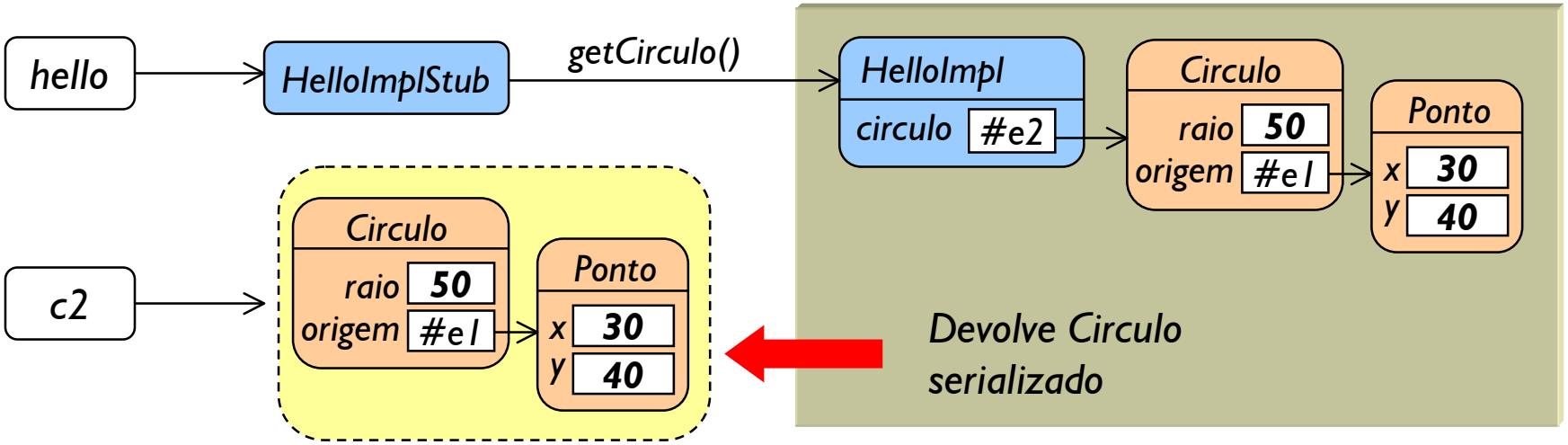


b `hello.setCirculo(hello.createCirculo(50, hello.createPonto(30, 40)));`



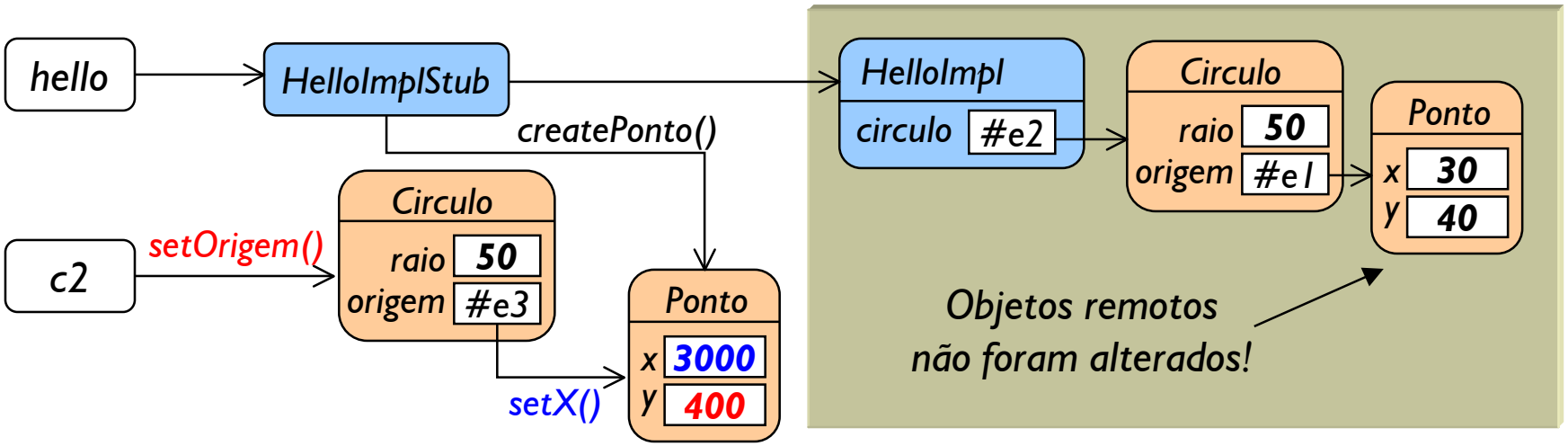
Conseqüências da passagem por valor

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!***

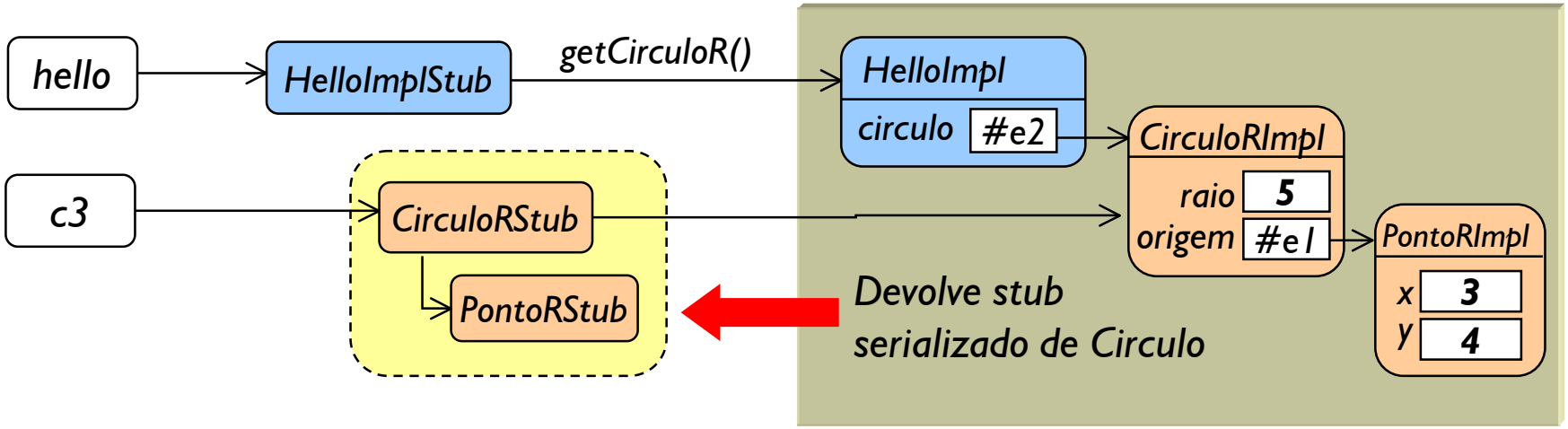


Passagem de parâmetros por referência

- 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)

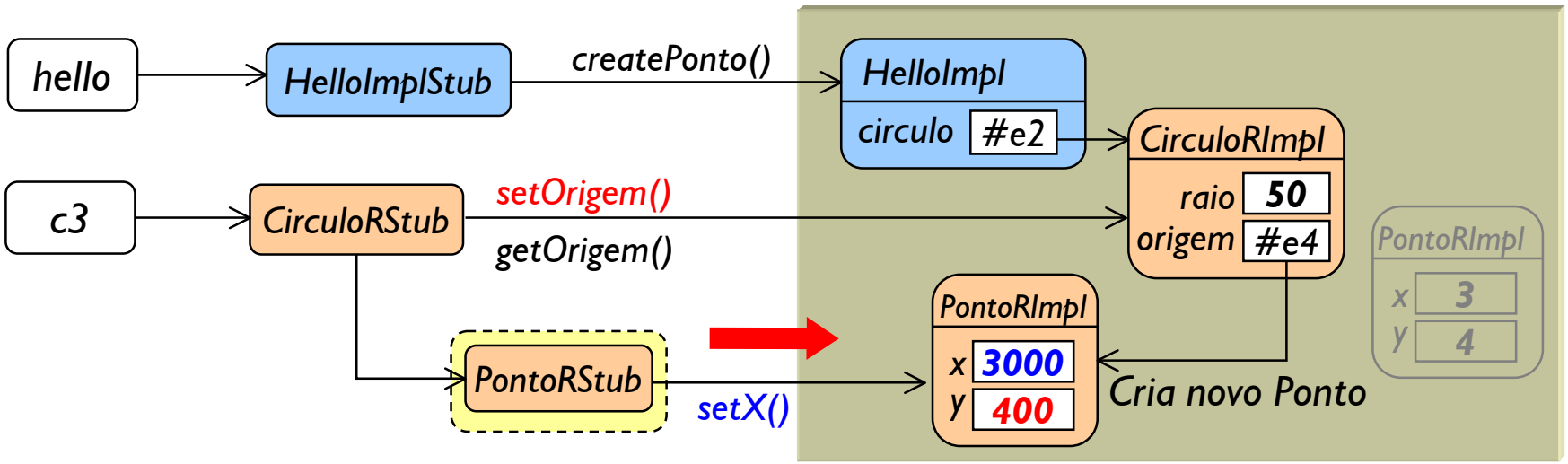
Passagem por referência

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*

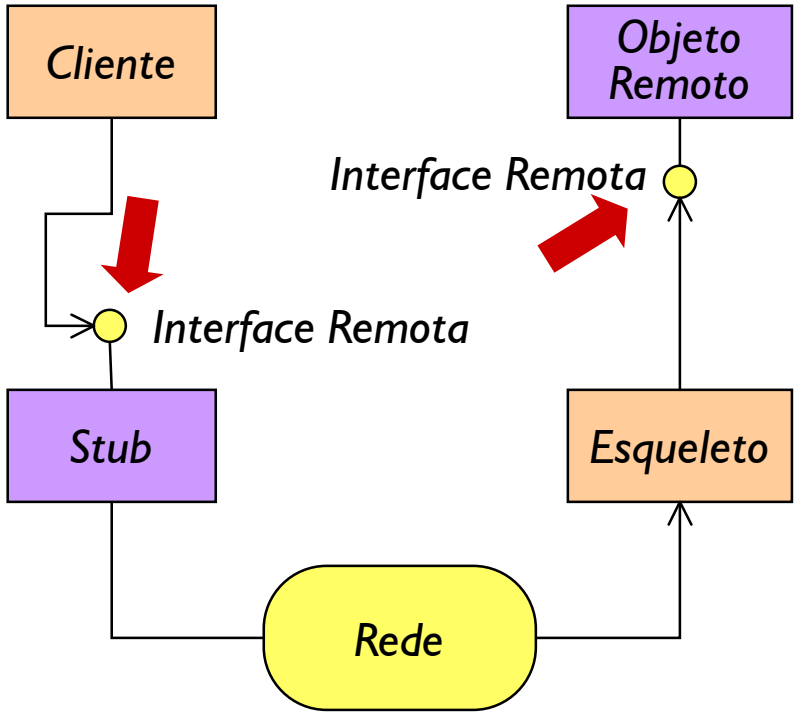
Veja demonstração: Rode, a partir do diretório cap04, **`ant build`**, *depois, em janelas separadas,* **`ant orbd`**, **`ant runrmiopserver`** e **`ant runrmiopclient`**

Enterprise JavaBeans e RMI/IIOP

- Enterprise JavaBeans são objetos distribuídos controlados por interceptadores que podem ser objetos remotos RMI sobre IIOP, logo
  - EJBs podem comportar-se como objetos remotos RMI
  - EJBs podem se comunicar com objetos CORBA
- Dados em EJBs são passados por interceptadores locais, remotos, ou copiados para objetos
- Aplicações que usam EJBs tipicamente lidam com os três tipos de passagem de parâmetro
  - 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)

Por que Enterprise JavaBeans? (I)

- Sistemas de objetos distribuídos têm em comum uma arquitetura que consiste de
  - O objeto remoto
  - Um stub que representa o objeto remoto no cliente
  - Um esqueleto, que representa o cliente no servidor
- O cliente conversa com o stub pensando tratar-se do próprio objeto



- O esqueleto conversa com o objeto remoto que pensa que é o cliente quem fala com ele
- O stub e o objeto remoto implementam a **mesma interface**!

Fonte da ilustração: [4] Ed Roman, Mastering EJB 2

Por que EJB? (2)

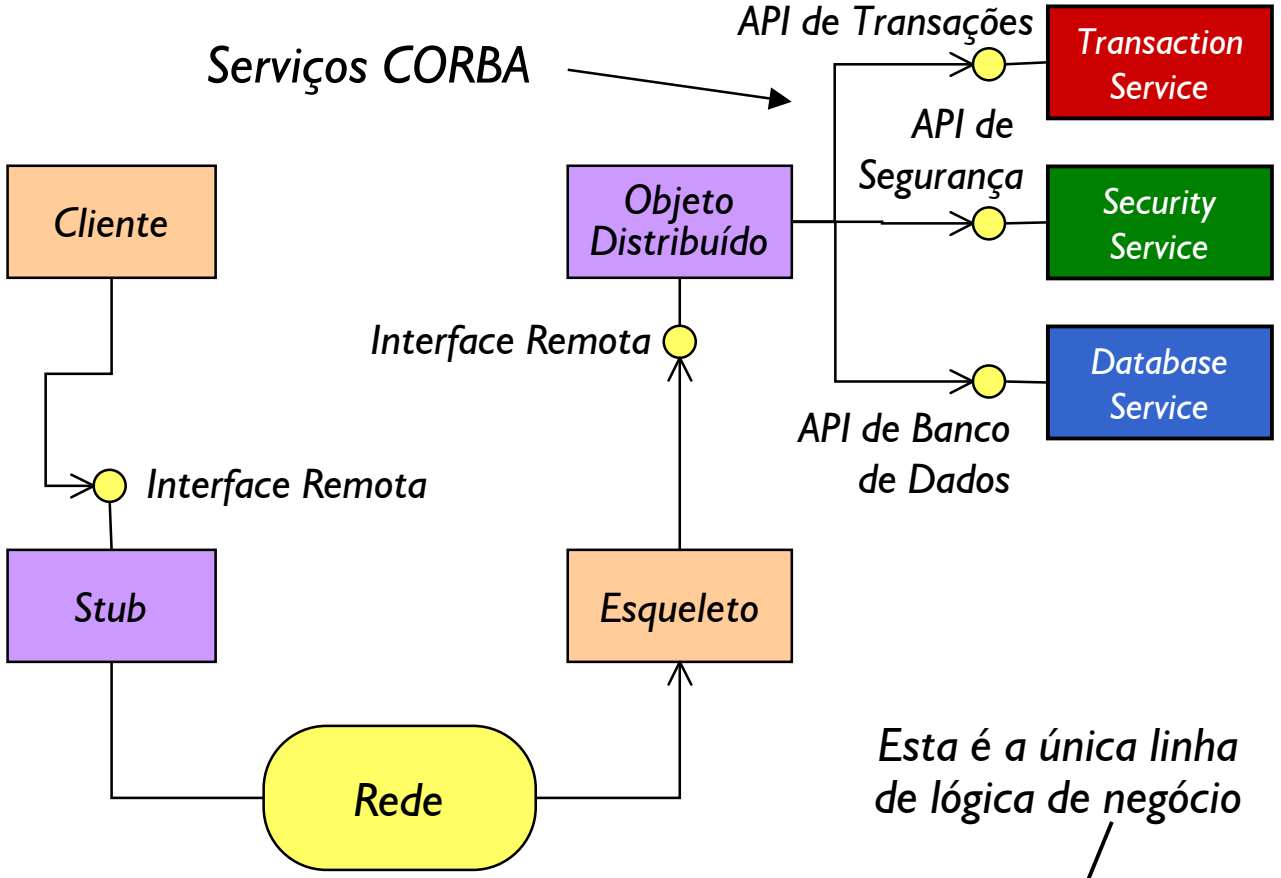
- Desenvolver um sistema distribuído não é simples.
  - Exige se concentrar em aspectos da aplicação que nada tem a ver com o problema de negócio a ser solucionado
  - Exige a observação de condições que geralmente são menos importantes ou dispensáveis em aplicações locais
- É preciso se preocupar com
  - A performance, que pode sofrer abalos em rede
  - Os custos de uma possível ampliação da capacidade
  - A segurança dos dados, controle de acesso, permissões
  - Transações: tarefas poderão falhar no meio do caminho
  - Integridade dos dados: clientes irão acessá-los simultaneamente
  - O que acontecerá se parte do sistema sair do ar?
- Mas esses são problemas de qualquer sistema distribuído
  - Será que precisamos implementar tudo isto?
  - Não! Use middleware!

Tipos de middleware

- Temos uma aplicação com vários objetos distribuídos e um servidor que oferece serviços de middleware. E agora? Como acessar os serviços?
- Serviços podem ser acessados através de uma API
  - Sua aplicação precisará escrever o código para usar a API, por exemplo, JDBC para controlar acesso a bancos de dados, ou JTA para controlar transações, ou APIs de segurança: middleware explícito
- Serviços podem ser configurados através de interfaces ou arquivos de configuração externos ao código
  - Sua aplicação não precisa conter código algum de acesso a recursos externos. Você declara os serviços que precisa externamente: middleware implícito

Middleware explícito

- *Tem que ser implementado em cada método de cada objeto que deseja utilizar o serviço*

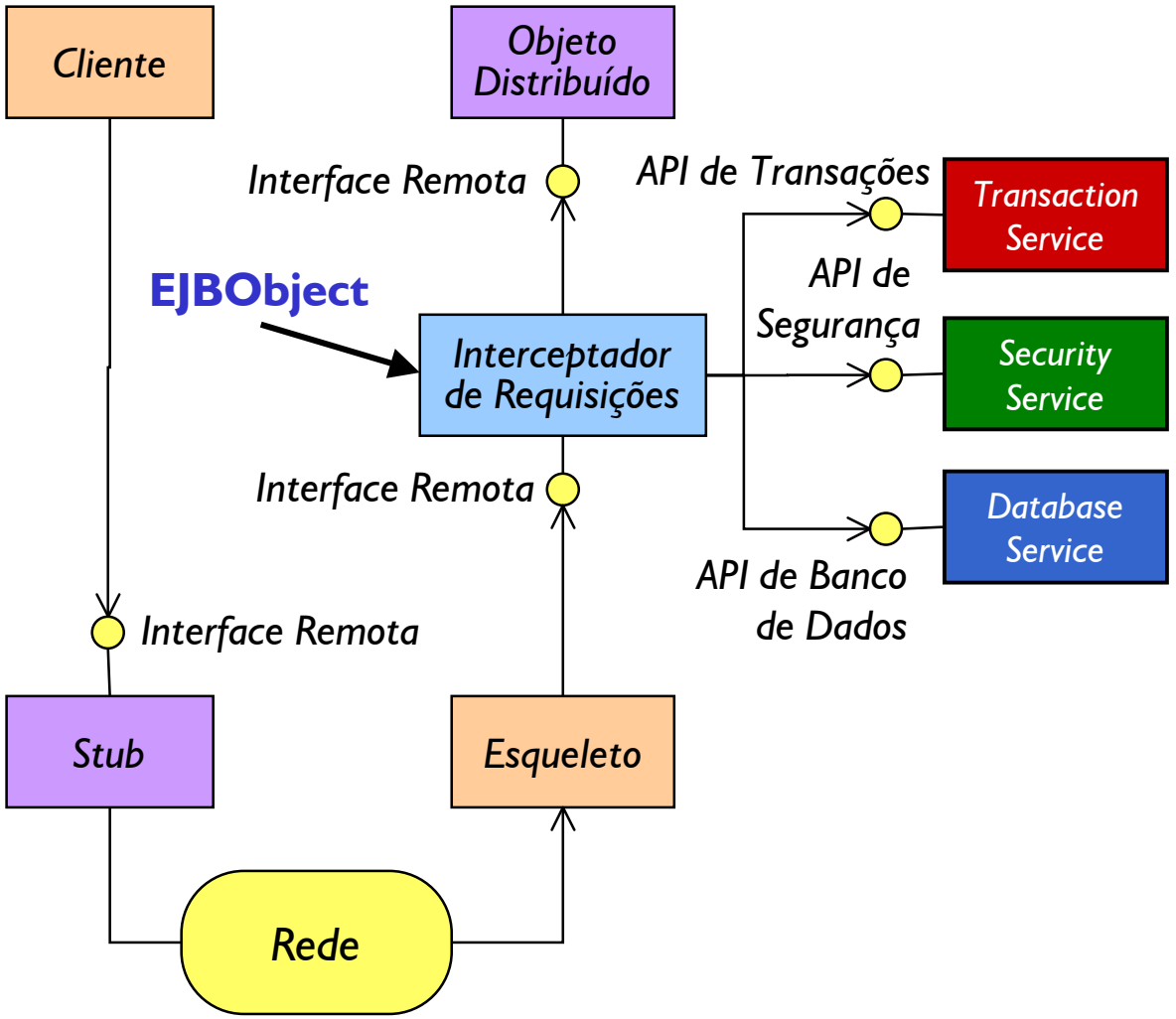



```
public void transferir(Conta um, Conta dois, double valor) {
   // chama API para verificar segurança da requisição
   // chama API para iniciar a transação
   // chama API para carregar dados do banco
   // Subtraia o valor do saldo da conta um e some na conta dois
   // chama API para guardar dados no banco
   // chama API para fechar a transação
}
```

Fonte da ilustração: [4]

Middleware implícito

- Escreva seu objeto para conter apenas lógica de negócio
- Declare os serviços de middleware que seu objeto precisa em um arquivo separado
- Rode uma ferramenta que pega o descritor e gera um objeto interceptador de requisições




```
public void transferir(Conta um, Conta dois, double valor) {
   // Subtraia o valor do saldo da conta um e some na conta dois
}
```

Fonte da ilustração: [4]

O que é um Enterprise JavaBean?

- Um Enteprise JavaBean (EJB) é um componente que pode ser implantado em um servidor de aplicações e utilizar seus serviços de forma declarativa ou não
- O servidor oferece o container que executa o bean
  - O container gera os interceptadores que isolam o bean dos seus clientes locais ou remotos
  - O deployer (instalador da aplicação) pode configurar os serviços que desejar e o código para chamá-los será gerado nos interceptadores durante a implantação
- Um enterprise bean é um objeto distribuído, mas não é um objeto remoto
  - Seu interceptador pode ser um. Se for, usa RMI-IIOP.

Componentes de um EJB

- Para que o container possa gerar o código necessário é preciso que o bean seja implantado corretamente. Um EJB deve estar empacotado em um JAR contendo
  - Uma classe que implementa o bean
  - Interface(s) do(s) interceptador(es) (Remote ou Local)
  - Interface(s) de fábrica: Home (Remote ou Local)
  - Deployment descriptor
- A estrutura é definida na especificação e formalizada em interfaces, esquemas XML e validadores
  - Antes de implantar uma aplicação, o servidor verifica se todos os itens estão de acordo com a especificação. Se um não estiver, a aplicação poderá não ser instalada

Classe do EJB

- *É o objeto distribuído. Contém a lógica de negócio*
  - *Pode também implementar rotinas de persistência*
- *É uma classe Java que implementa os mesmos métodos declarados nas interfaces do interceptador*
  - *Não é objeto remoto (não implementa java.rmi.Remote)*
- *Implementações diferem, dependendo do tipo*
  - *Session Beans - lógica relacionada a processos de negócio (computar preços, transferir fundos)*
  - *Entity Beans - lógica relacionada a dados (mudar o nome de um cliente, reduzir o saldo)*
  - *Message-driven Beans - lógica orientada a eventos (lógica assíncrona)*

Interfaces padrão do EJB

- Todo bean implementa a *javax.ejb.EnterpriseBean*

```
package javax.ejb;
public interface EnterpriseBean {
}
```

- Na verdade, todo bean implementa uma interface derivada de EnterpriseBean

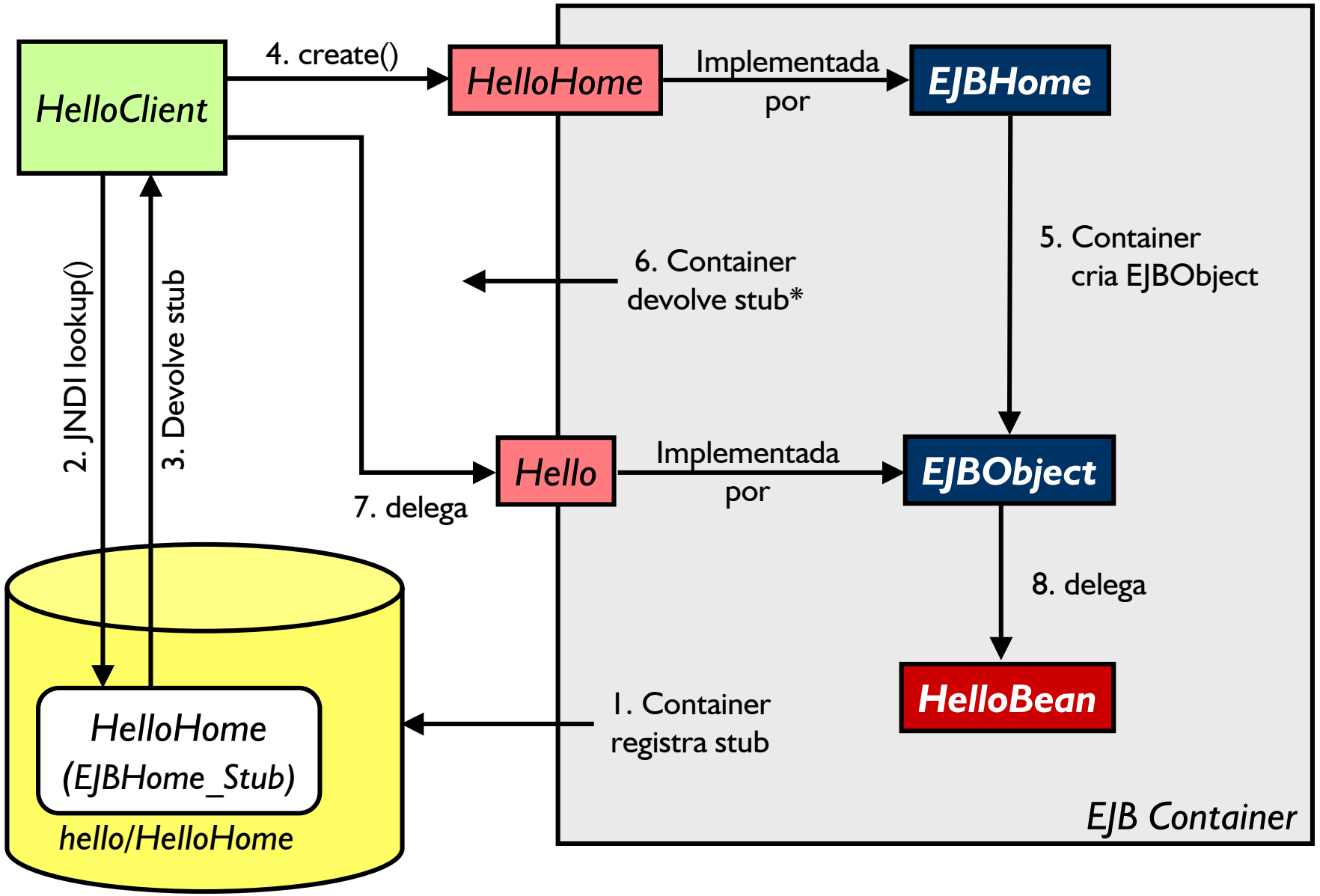
```
SessionBean extends EnterpriseBean
EntityBean extends EnterpriseBean
MessageDrivenBean extends EnterpriseBean
```

- Cada interface tem métodos próprios que precisam ser implementados em cada bean
  - Além deles, a especificação pode exigir outros

Interface do Componente

- *Component interface. Há dois tipos*
  - *Remote component interface*
  - *Local component interface*
- *Para criar uma interface de componente, é preciso estender interfaces EJBObject ou EJBLocalObject*
  - *O container criará automaticamente interceptadores locais ou remotos contendo todos os métodos declarados*
  - *Um bean só precisa de um tipo de interceptador (ou Local ou Remoto) mas pode ter ambos, se necessário*
- *Interceptador EJBObject é objeto remoto RMI-IIOP*
  - *É gerado pelo container e delega chamadas ao bean*
  - *Interface javax.ejb.EJBObject estende java.rmi.Remote*
  - *Define métodos de negócio remotos expostos pelo bean*

Arquitetura

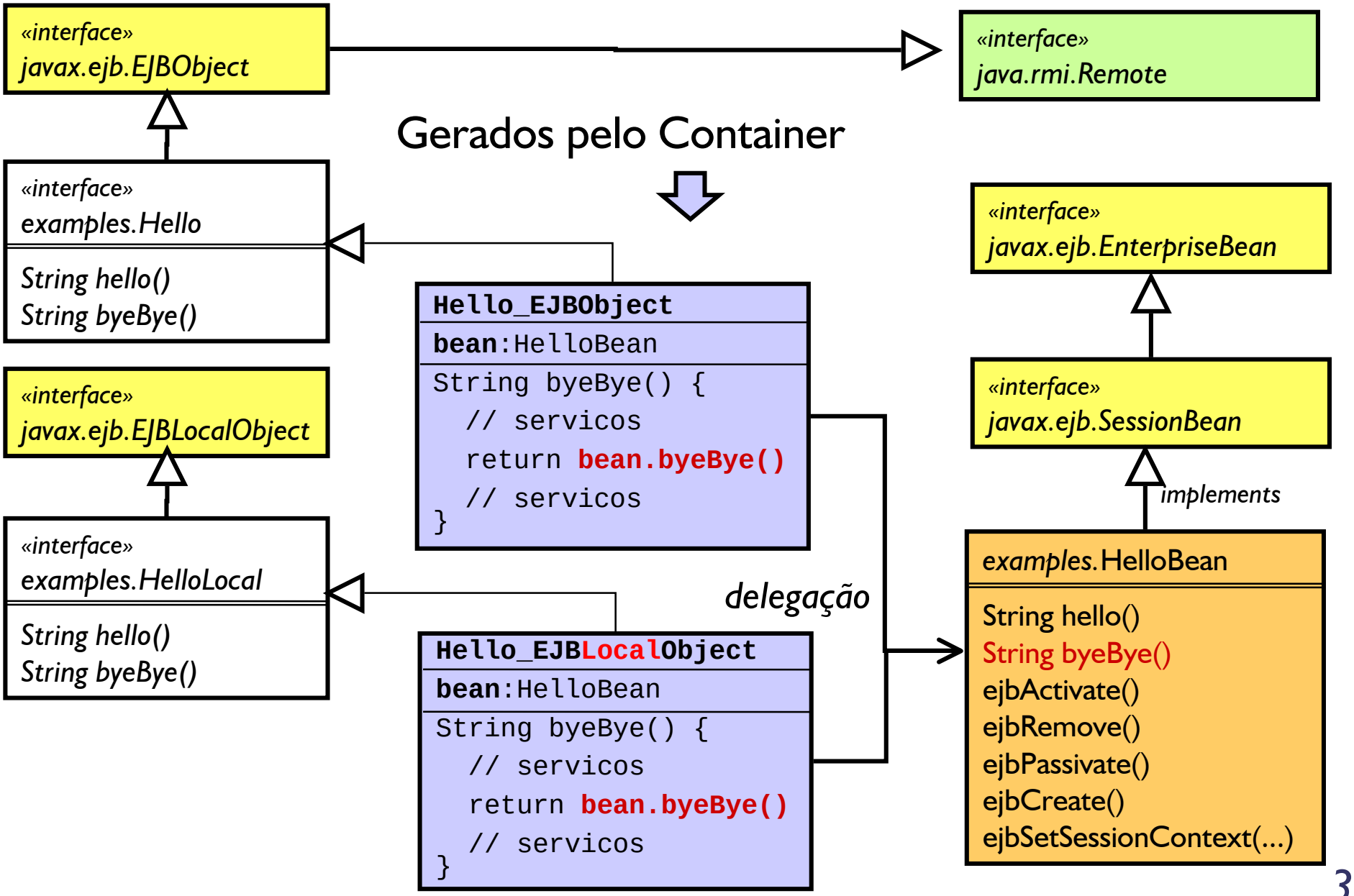


* *detalhes como comunicacão entre stubs e skeletons foram omitidos*

EJB Object

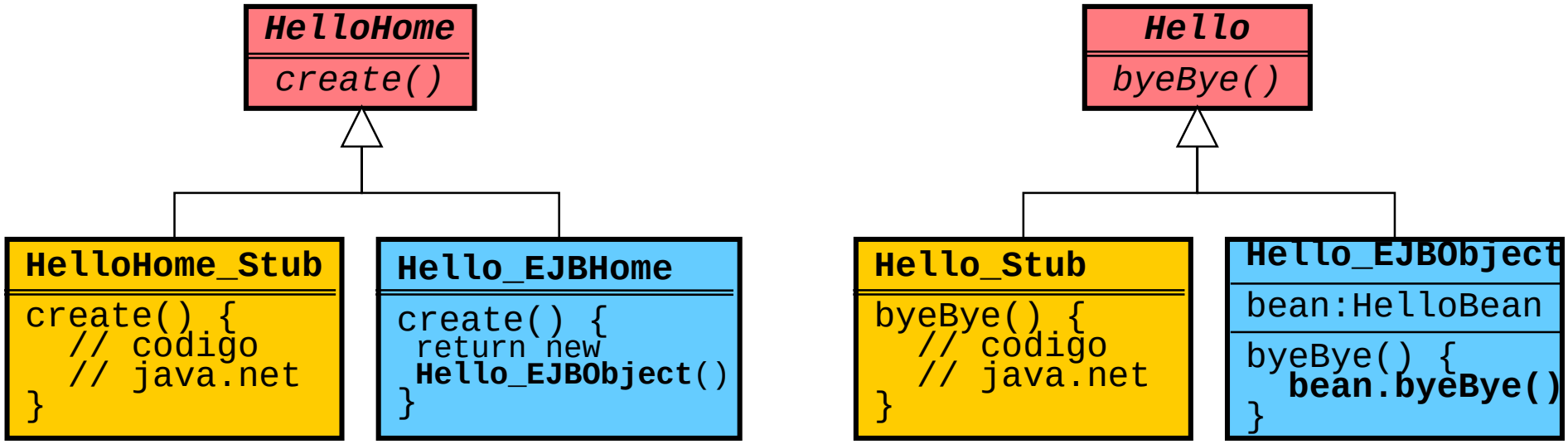
- Enterprise Beans podem ser acessados remotamente, mas não são objetos remotos
  - Um cliente (ou o skeleton) nunca chama o método diretamente em uma instância do bean: a chamada é interceptada pelo Container e delegada à instância
  - Ao interceptar requisições, o container pode automaticamente realizar middleware implicito
- O **EJBObject** é quem implementa a interface remota. Ele é o interceptador que chama o Bean
  - É objeto inteligente que sabe realizar a lógica intermediária que EJB container requer antes que uma chamada seja processada
  - Expõe cada método de negócio do bean e delega as requisições
- O container gera o EJBObject a partir da interface criada pelo programador para o componente

EJBObject, EJBLocalObject e Enterprise Bean



Objetos gerados pelo container

- *Durante o deployment*
  - *Interceptadores remotos EJBHome e EJBObject são gerados*
  - *Interceptadores locais EJBLocalHome e EJBLocalObject são gerados*
  - *Stubs são gerados para os interceptadores remotos. Stub da interface Home é mapeado a nome no servidor JNDI*
- *Hierarquias dos objetos gerados pelo container**



- *EJBObject delega requisições para o Bean*

** Apenas para efeito de ilustração. Implementação exata depende, na verdade, do fabricante*

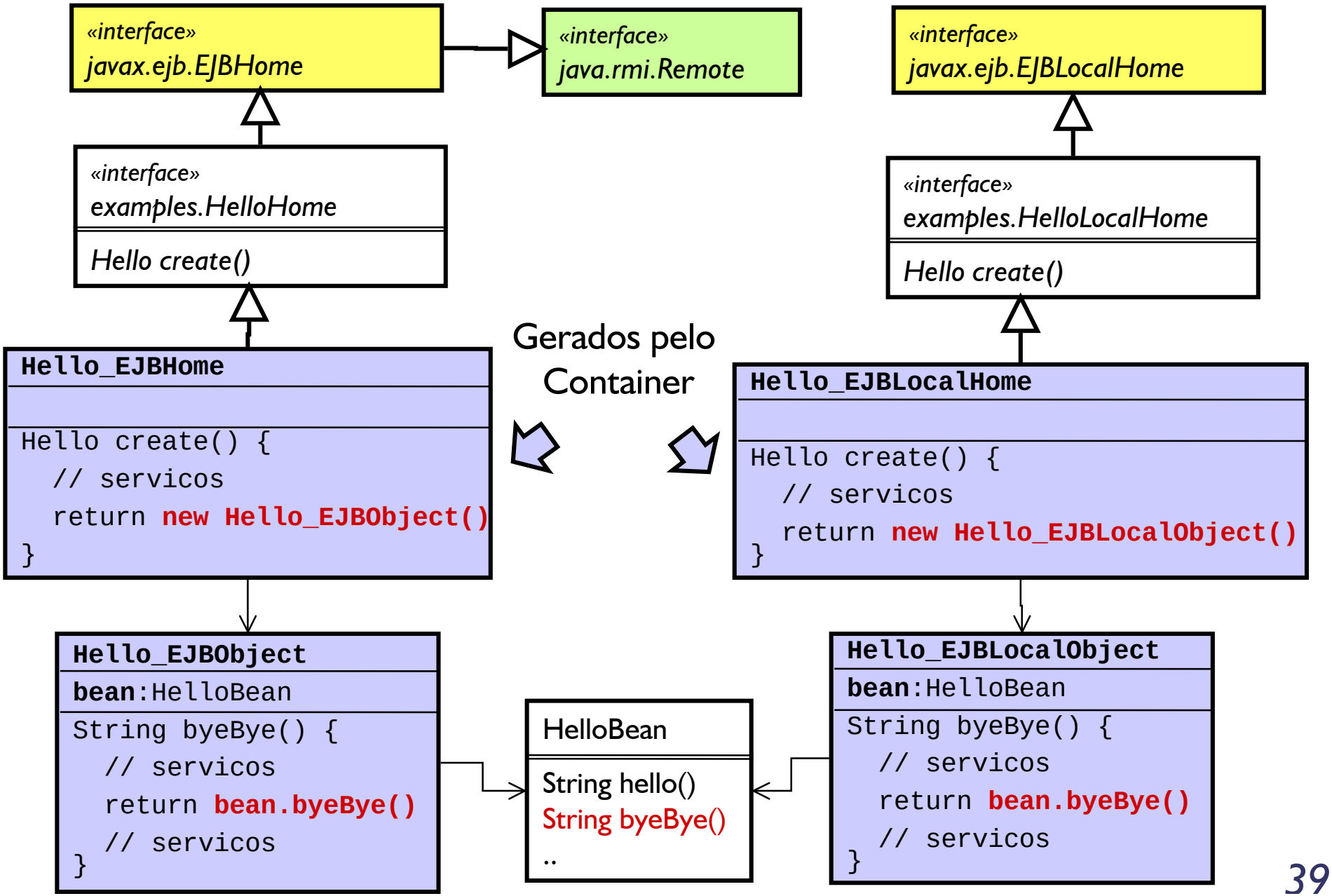
RMI-IIOP versus EJB Objects

- *Qualquer objeto que implementa javax.rmi.Remote é um objeto remoto e chamável de outra JVM*
  - *Todos os EJB Objects (gerados pelo Container) são objetos RMI-IIOP*
  - *Interfaces Remote EJB são interfaces Remote RMI-IIOP, só que precisam aderir à especificação EJB (têm métodos adicionais que o container implementa)*
- *Regras de java.rmi.Remote valem para EJBObjects*
  1. *Todos os métodos provocam RemoteException*
  2. *Interfaces Remote precisam suportar tipos de dados que podem ser passados via RMI-IIOP.*
     Tipos permitidos: primitivos, objetos serializáveis e objetos remotos RMI-IIOP

EJB Home

- *Fábrica usada para que clientes possam adquirir referências remotas a objetos EJB*
  - *Evita ter que registrar cada objeto: registra-se só a fábrica*
  - *Para obter uma referência a um objeto EJB, cliente chama métodos do EJBHome*
  - *O EJBHome é responsável por operações do ciclo de vida de um objeto EJB (criar, destruir, encontrar)*
- *Objetos **EJBHome** servem para*
  - *Criar objetos EJB*
  - *Encontrar objetos EJB existentes (em Entity Beans)*
  - *Remover objetos EJB*
- *São parte do container e gerados automaticamente*
  - *Implementam interface Home definida pelo programador*

EJB(Local)Home e EJB(Local)Objects



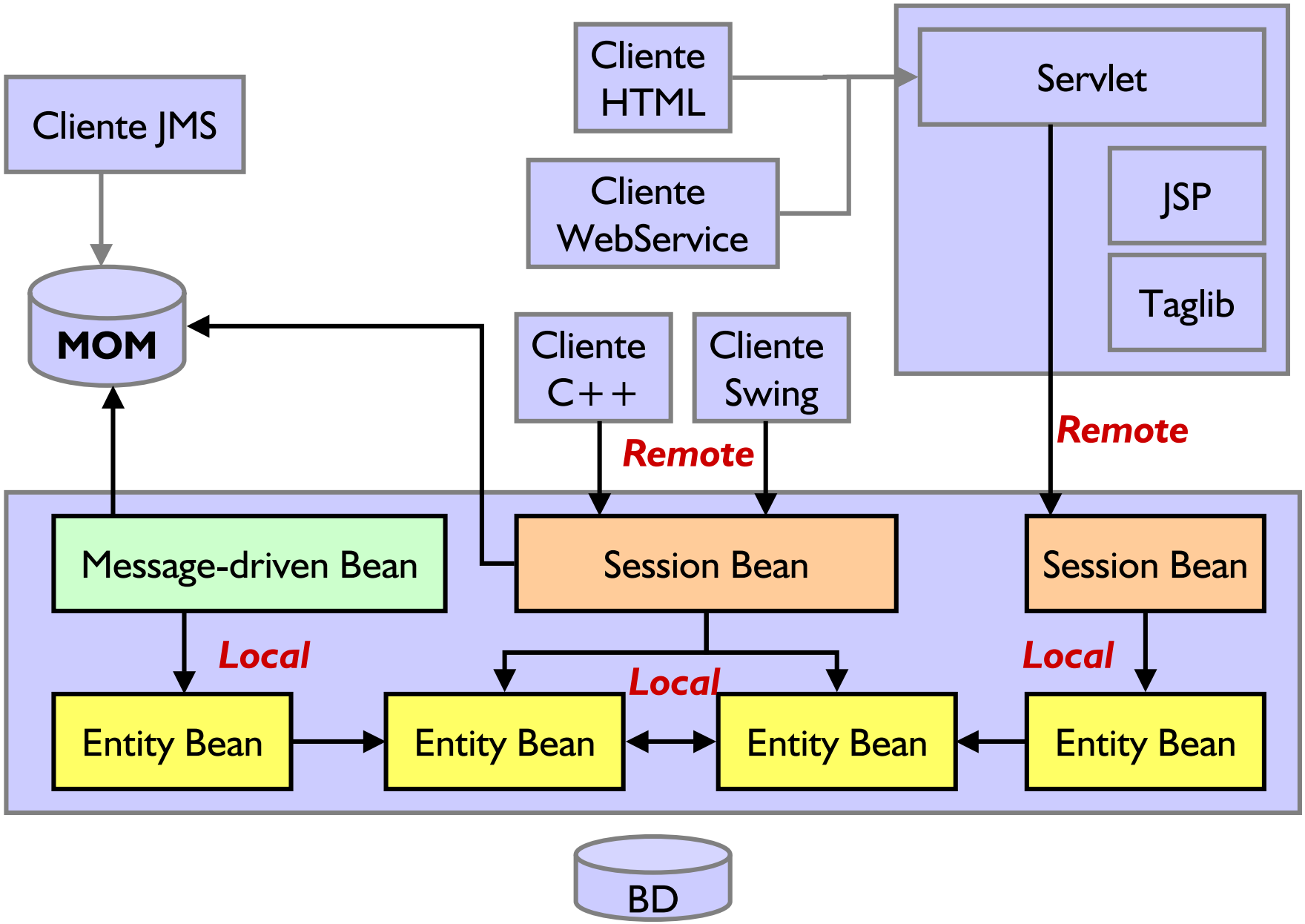
Por que usar interfaces locais

- Um problema das interfaces remotas é que criar objetos através dela é um processo lento
  1. Cliente chama um stub local
  2. Stub transforma os parâmetros em formato adequado à rede
  3. Stub passa os dados pela rede
  4. Esqueleto reconstrói os parâmetros
  5. Esqueleto chama o EJBObject
  6. EJBObject realiza operações de middleware como connection pooling, transações, segurança e serviços de ciclo de vida
  7. Depois que o EJBObject chama o Enterprise Bean, processo é repetido no sentido contrário
- Muito overhead! Em EJB 2.0 é possível chamar EJBs através de sua interface local
  1. Cliente chama objeto local
  2. Objeto EJB local realiza middleware
  3. Depois que o trabalho termina devolve o controle a quem chamou

Interfaces locais: conseqüências

- Interfaces locais são definidas para objetos EJBHome e para EJBObject (interfaces EJBLocal e EJBLocalHome)
- Benefícios
  - Pode-se escrever beans menores para realizar tarefas simples sem medo de problemas de performance
  - Uso típico: fachadas Session que acessam Entities que não são acessíveis remotamente
- São opcionais
  - Substituem ou complementam as interfaces remotas existentes
- Efeitos colaterais
  - Só funcionam ao chamar beans do mesmo processo (não dá para mudar o modelo de deployment sem alterar o código)
  - Parâmetros são passados por referência e não por valor: muda a semântica da aplicação!!!
- Uso típico: Entity Beans geralmente só têm interfaces locais

Uso de interfaces locais e remotas



javax.ejb.EJBContext

- Usada em todos os beans como forma de ter acesso ao contexto de tempo de execução do bean
  - Todo bean tem um método get/set<Tipo>Context, onde <Tipo> é Entity, Session ou MessageDriven
- Através desse contexto, o bean pode ter acesso...
  - Ao objeto que implementa sua interface Home
  - Ao usuário e perfil do usuário logado
  - Ao status de transações
- Sub-interfaces de EJBContext acessam
  - O interceptador que implementa sua interface do componente (SessionContext e EntityContext)
  - A chave primária do Entity Bean

Deployment descriptors

- Para informar ao container sobre suas necessidades de middleware, o bean provider deve declarar suas necessidades de middleware em um deployment descriptor (arquivo der configuração em XML)
  - Informações de gerência e ciclo de vida (quem é o Home, o Remote, se é Session, Entity ou MDB)
  - Requisitos de persistencia para Entity Beans
  - Configuração de transações
  - Segurança (quem pode fazer o que com que beans, que metodos pode usar, etc.)
  - ...
- Pode-se usar uma ferramenta (geralmente a ferramenta do próprio container) para gerar o arquivo

Deployment descriptor simples: exemplo

```
<!DOCTYPE ejb-jar PUBLIC
"-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
"http://java.sun.com/dtd/ejb-jar_2_0.dtd">

<ejb-jar>
 <enterprise-beans>
  <session>
   <ejb-name>Hello</ejb-name>
   <home>examples.HelloHome</home>
   <remote>examples.Hello</remote>
   <local-home>examples.HelloLocalHome</local-home>
   <local>examples.HelloLocal</local>
   <ejb-class>examples.HelloBean</ejb-class>
   <session-type>Stateless</session-type>
   <transaction-type>Container</transaction-type>
  </session>
 </enterprise-beans>
</ejb-jar>
```

ejb-jar.xml

Vendor-specific files

- Cada servidor tem recursos adicionais que podem ser definidos em arquivos específicos
  - Como configurar load-balancing
  - Clustering, pooling
  - Monitoramento
  - Mapeamento JNDI
- Geralmente são gerados por ferramentas no próprio servidor
- Podem também ser codificados à mão (jboss.xml)
- Devem ser incluidas no bean antes do deployment

Arquivos proprietários do fabricante

```
<jbosscmp-jdbc>
   <enterprise-beans>
      <entity>
         <ejb-name>ProdutoEJB</ejb-name>
         <table-name>produtos</table-name>
         <cmp-field>
            <field-name>name</field-name>
            <column-name>nome</column-name>
         </cmp-field>
         <cmp-field>
            <field-name>codigo</field-name>
            <column-name>id</column-name>
         </cmp-field>
         <cmp-field>
            <field-name>quantidade</field-name>
            <column-name>qte</column-name>
         </cmp-field>
                ...
      </entity>
   </enterprise-beans>
</jbosscmp-jdbc>
```

Mapeamentos para bancos de dados

jbosscmp-jdbc.xml

Mapeamentos para serviço de nomes

```
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>Hello</ejb-name>
      <jndi-name>hello/HelloHome</jndi-name>
    </session>
  </enterprise-beans>
</jboss>
```

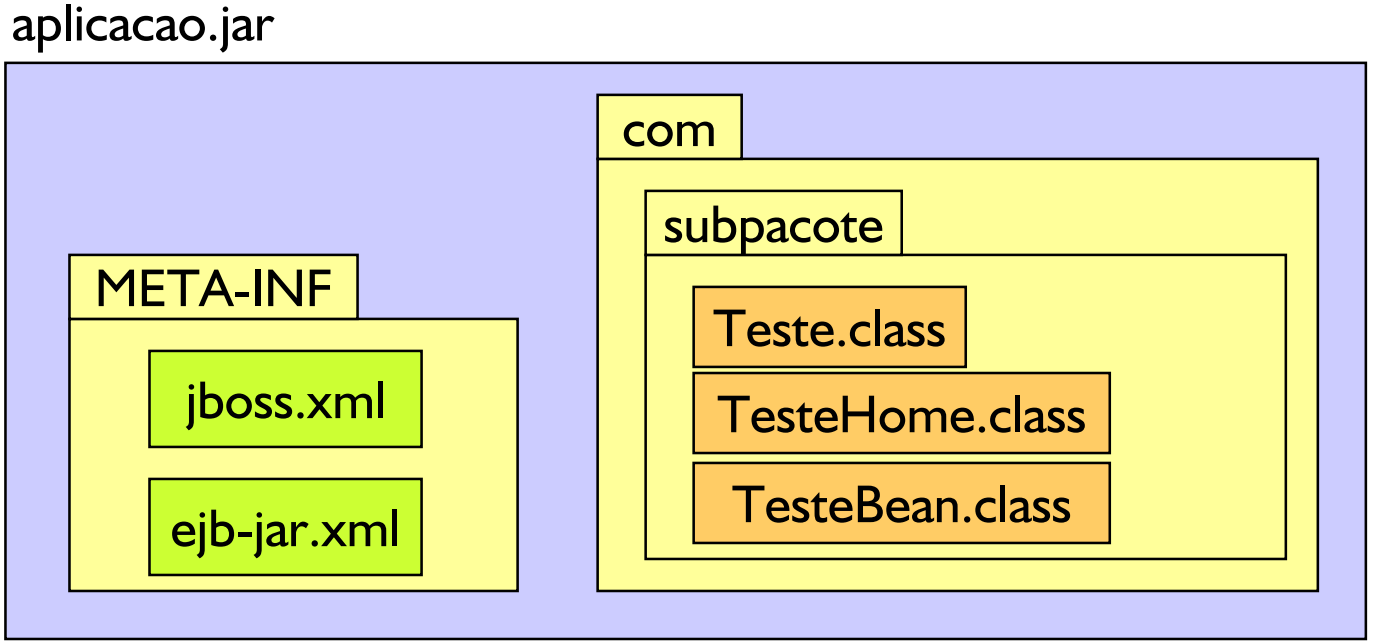
jboss.xml

EJB-JAR

- Arquivo JAR que contém tudo o que descrevemos
- Pode ser gerado de qualquer forma
  - ZIP, Ant, ferramentas dos containers
  - IDEs (NetBeans e deploytool)
- Uma vez feito o EJB-JAR, seu bean está pronto e é *unidade implantável* em application server
  - Ferramentas dos containers podem decomprimir, ler e extrair informações contidas no EJB-JAR
  - Depois, ferramentas realizam tarefas específicas do fabricante para gerar EJB Objects, EJB Home, importar seu bean no container, etc.
- Pode-se ter vários beans em um ejb-jar

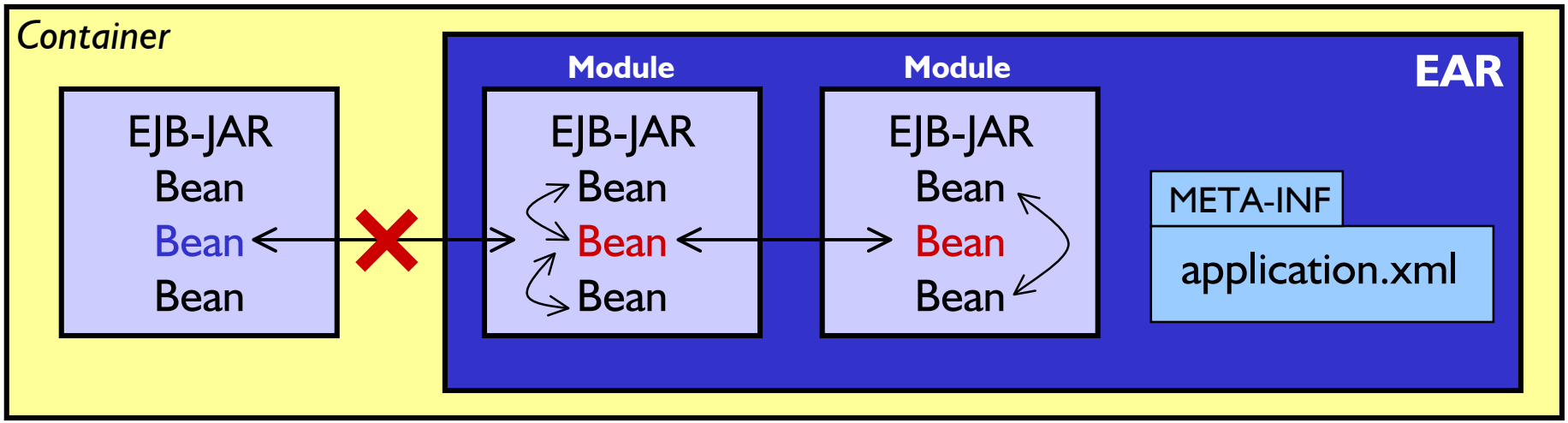
Estrutura de um EJB-JAR

- *O EJB-JAR é um JAR comum. Não possui uma estrutura especial exceto quanto à localização dos deployment descriptors (padrão e do fabricante).*
  - *Coloque as classes em seus pacotes a partir da raiz*
  - *Coloque os deployment descriptors no META-INF*



EARs

- *Classes em um componente JAR não são visíveis em outro componente JAR no container*
- *Para que beans em um JAR possam ter acesso a classes de um bean em outro componente, é preciso que esteja em um EAR*
  - *Há outras formas de resolver este problema, mas são todas não portáveis*



application.xml de um Enterprise ARchive

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC
'-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN'
'http://java.sun.com/dtd/application_1_3.dtd'>

<application>
  <display-name>ConverterEAR</display-name>
  <module>
    <ejb>Converter-ejb.jar</ejb>
  </module>
  <module>
    <java>utils.jar</java>
  </module>
  <module>
    <web>
      <web-uri>Converter-web.war</web-uri>
      <context-root>converter</context-root>
    </web>
  </module>
</application>
```

JARs conseguem ver classes uns dos outros

WAR vê classes dos JARs

JARs não vêem classes do WAR

Exercícios

- 1. Escreva um build.xml para montar a aplicação EJB fornecida. As classes estão prontas e compiladas. É preciso
  - a) Preencher o deployment descriptor informando quem é cada componente (use os comentários)
  - b) Montar o EJB-JAR
  - c) Teste a aplicação fazendo o deployment no JBoss e executando o target: `ant run.client`
- 2. Exercício extra: coloque o EJB-JAR e o WAR fornecido em um EAR
  - Faça deploy do EAR e acesse a aplicação via Web (o nome do contexto você escolhe)

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

helder@argonavis.com.br

argonavis.com.br

J500 - Aplicações Distribuídas com J2EE e JBoss
Revisão 1.5 (junho de 2003)

Java em Ambientes Distribuídos, 1998
Java RMI x CORBA x RMI-IIOP, 1999
jav500 - Introdução a Java 2 Enterprise Edition, Junho 2002