Usando Java™ em ambientes distribuídos

RMI

RMI-IIOP

Java IDL (CORBA)

Helder da Rocha

www.argonavis.com.br

Sumário

- *RMI/RPC e Java em ambientes distribuídos*
- *Java IDL (~25%)*
  - *Arquitetura CORBA e mapeamento Java-IDL*
  - *Construção de uma aplicação distribuída com Java IDL*
- *Java RMI (~25%)*
  - *Fundamentos (overview) de Java RMI*
  - *Construção de uma aplicação distribuída com Java RMI*
- *Java RMI sobre IIOP (~25%)*
  - *Diferenças entre Java RMI e RMI sobre IIOP*
  - *Construção de uma aplicação distribuída com RMI-IIOP*
  - *Passagem de parâmetros por valor e por referência*
- *Conclusão: Java RMI x CORBA x RMI-IIOP*
- *Demonstração, exemplos e exercícios (~25%)*

Tempo útil da apresentação: 6h (4 x 1,5h)

Objetivo deste minicurso

- *Oferecer uma introdução a objetos distribuídos em Java:* ***Java RMI***, ***Java IDL*** *e* ***RMI-IIOP***
- *Pré-requisitos*
  - *Experiência em Java e conhecimentos básicos de rede (java.net)*
- *Exploraremos* ***apenas*** *as tecnologias de objetos distribuídos. Não são abordadas*
  - *Sockets, JMS (modelo alternativo de comunicação), EJB (arquitetura de componentes baseada em RMI-IIOP), Servlets / JSP (tecnologias baseadas em HTTP)*
- *Também não são abordados (devido ao tempo)*
  - *RMI Activation e Socket factories*
- *Código exemplo:*
  - ***exemplos.zip*** *(requer Ant). Cria diretório exemplos. Configure arquivo build.properties e use o Ant para montar e rodar os exemplos.*
  - ***extras.zip*** *(usados em demonstrações).*

RMI e RPC

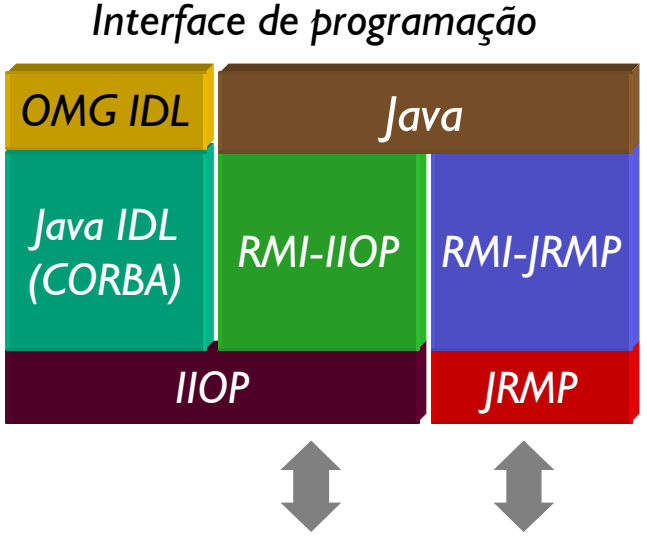
- *RMI e RPC são técnicas usadas para isolar, dos clientes e servidores, os detalhes da comunicação em rede*
  - *Utilizam protocolos padrão, stubs e interfaces*
  - *Lidam com diferentes representação de dados*
- *RPC: Remote Procedure Call*
  - *Chamada procedural de um processo em uma máquina para 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*



Java IDL

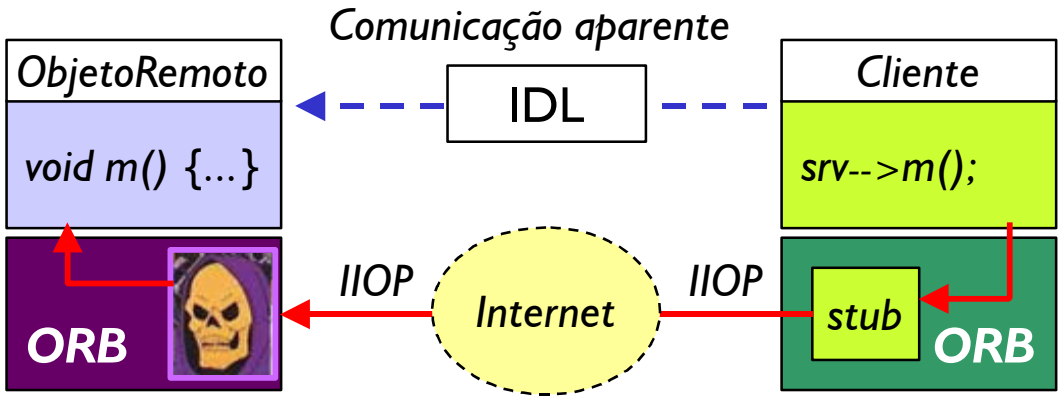
- *Tecnologia Java para objetos distribuídos baseada na arquitetura CORBA*
  - *Permite desenvolver aplicações ou componentes distribuídos em Java capazes de se comunicarem com aplicações ou objetos distribuídos escritos em outras linguagens*
- *Parte integrante do Java 2 SE SDK*
- *Oferece*
  - *Uma API: org.omg.CORBA.**
  - *Serviço de nomes: org.omg.CORBA.CosNaming*
  - *Object Request Broker (ORB) com servidor de nomes persistente (**orbd**) ou transiente (**tnameserv**)*
  - *Ferramentas para geração de código Java a partir de interfaces IDL: **idlj***

Arquitetura CORBA

- **C**ommon **O**bject **R**equest **B**roker **A**rchitecture
  - Especificação da OMG - Object Management Group
  - Padrão para interoperabilidade de objetos distribuídos
- Componentes
  - IIOP - Internet Inter ORB Protocol
  - ORB - Object Request Broker
  - OMG IDL - Interface Definition Language
  - COS - CORBA Object Services (opcionais)
- Principais características
  - Transparência de localidade: não precisa saber onde estão objetos
  - Independência de plataforma: objetos podem estar distribuídos em plataformas diferentes
  - Neutralidade de linguagem: objetos se comunicam mesmo escritos em linguagens diferentes

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*

OMG IDL

- *OMG Interface Definition Language*
  - *É a linguagem que CORBA usa para definir interfaces usadas por clientes para acessar objetos remotos*
  - *Qualquer implementação de objeto CORBA deve ter uma*
- *Exemplo de OMG IDL*

```
#include "orb.idl"
module hello {
    interface Ponto {   // Tipo de objeto remoto
        long getX();
        void setX(in long x);
    };
    interface Circulo {
        long getRaio(); // Assinatura de método remoto
        void setRaio(in long raio);
        ::hello::Ponto getOrigem();
        void setOrigem (in ::hello::Ponto origem);
    };
};
```

Mapeamento IDL

- *IDL serve apenas para descrever interfaces*
  - *Não serve para ser executada*
  - *Nenhuma aplicação precisa da IDL (não faz parte da implementação - é design puro)*
- *IDL pode ser mapeada a outras linguagens*
  - *Mapeamento: correspondência entre estruturas, palavras-chave, representação de tipos*
  - *Compiladores IDL traduzem IDL para outras linguagens, gerando partes essenciais para a comunicação (stubs para o cliente, esqueletos para o servidor)*
  - *Pode-se gerar a parte do cliente em uma linguagem e a parte do servidor em outra, viabilizando a comunicação entre aplicações escritas em linguagens diferentes*

Mapeamentos Java-IDL-Java

- *Os mapeamentos Java-IDL-Java são definidos na especificação CORBA e usados para pelas ferramentas do J2SDK:*
  - **`idlj`** *(compilador IDL): gera classes Java a partir de IDL*
  - **`rmic -iiop -idl`***: gera IDLs a partir de interfaces Java*

Relacionamento entre tipos

Java	IDL
boolean	boolean
byte	octet
short	short
char	wchar
int	long
long	long long
float	float
double	double
java.lang.String	wstring
java.math.BigDecimal	fixed

Equivalência entre algumas estruturas

Java	IDL
package	module
class, interface	interface
throws	raises
java.lang.Exception	exception
void	void
class	enum, union, struct
byte[ ], int[ ], ...	sequence <octet>
objeto serializado	sequence <long>
extends	:
.	: :

Fontes: IDL to Java Mapping (2.4) ptc/2000-11-03 e Java to IDL ptc/00-01-06.

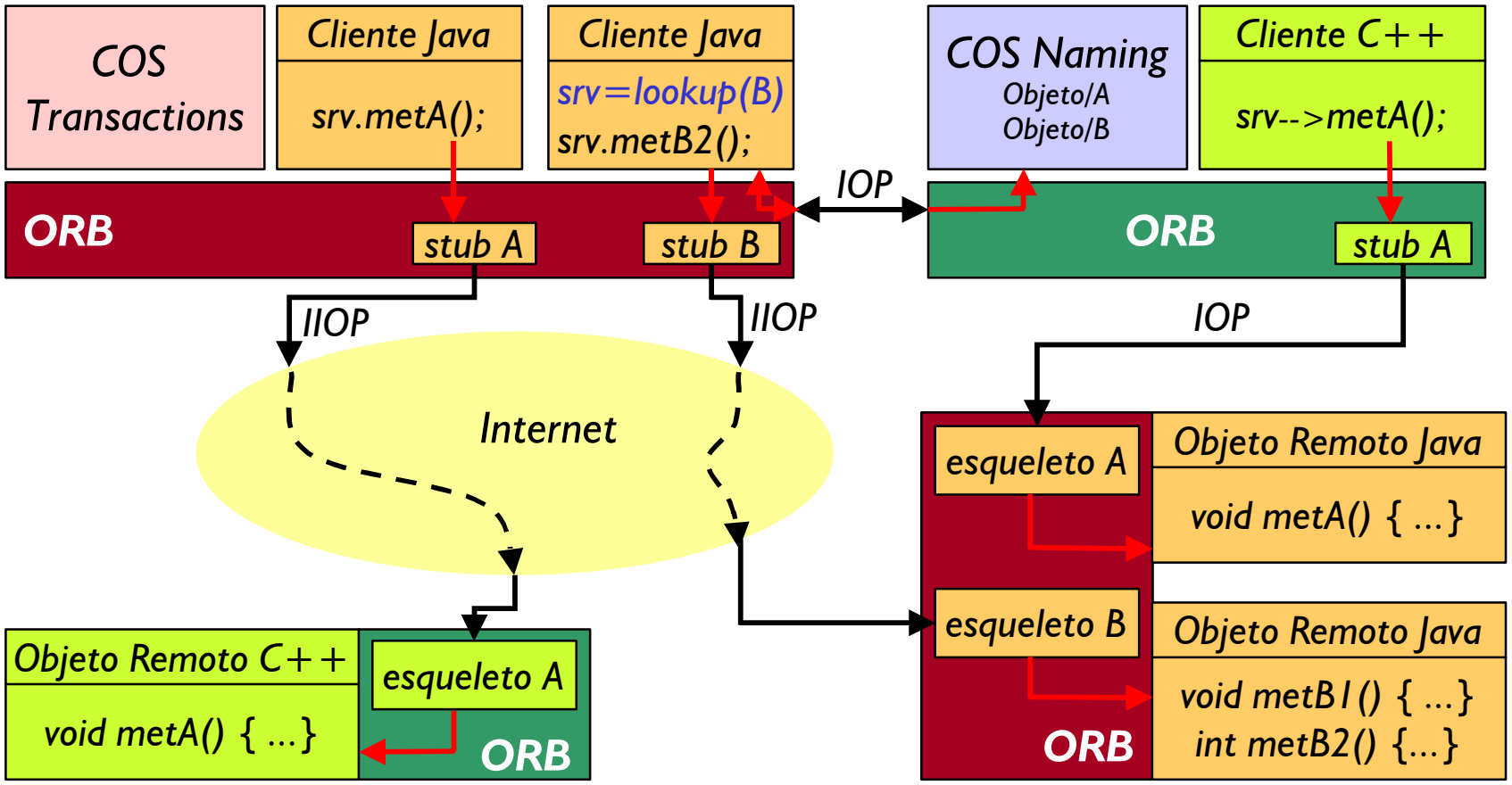
CORBA Object Services (COS)

Coleção de serviços genéricos de middleware

- COS Naming Service
  - Localiza objetos CORBA pelo nome
  - Acessível em Java via JNDI
  - É o único serviço COS implementado no J2SE SDK
- COS Object Transaction Service (OTS)
  - Permite o controle de transações em objetos CORBA
- COS Concurrency Control Service (CCS)
  - Permite que multiplos clientes acessem um recurso concorrentemente (sincronização)
- COS Security Service
  - Autenticação, autorização, controle de acesso
- ... vários outros:
  - Event Service, Life Cycle, Trader, Persistency, Time, Relationship,...

Comunicação CORBA: IIOP e serviços

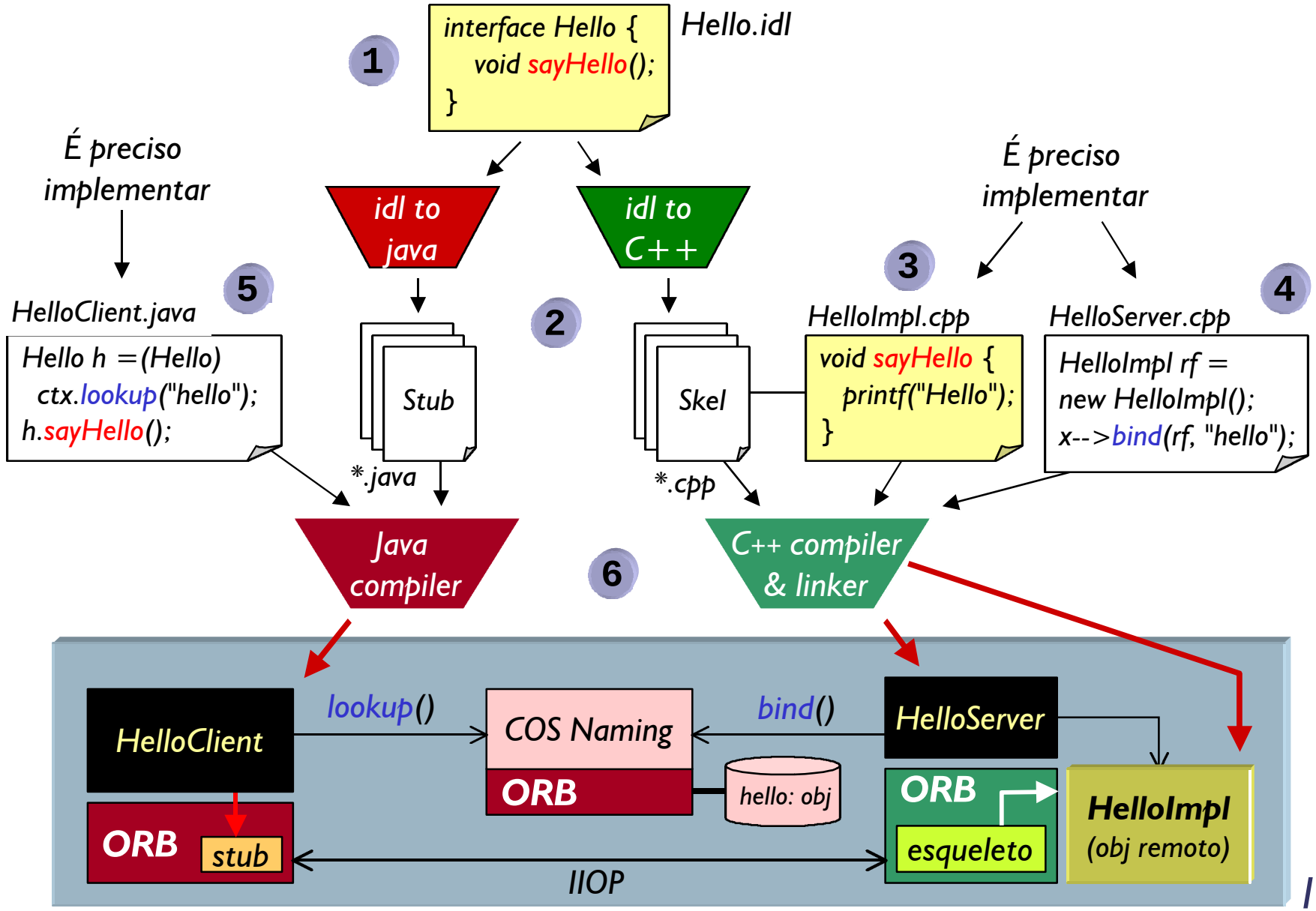
- *Com vários ORBs se comunicando via IIOP, clientes tem acesso a objetos remotos e serviços em qualquer lugar*
  - *ORB encarrega-se de achar o objeto (location transparency)*



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



Compilação do IDL

- *O IDL pode ser compilado para gerar todas as classes necessárias (cliente e servidor) ou apenas um dos lados*

```
> idlj -fclient hello.idl     (só classes essenciais para o cliente)
> idlj -fserver hello.idl     (só classes essenciais para o servidor)
> idlj -fall hello.idl        (todas as classes)
```

- *Pode-se escolher também o modelo de implementação*
  - *Herança é default; para delegação usa-se opções -fallTIE*
  - *Implementação tipo POA - Portable Object Adapter é default*

```
module hello {
  module corba {
    interface HelloRemote {
      string sayHello();
      void sayThis(in string msg);
    };
  };
};
```

hello.idl

- *Resultado*

```
hello/
  corba/
    _HelloRemoteStub.java
    HelloRemotePOA.java
    HelloRemoteHelper.java
    HelloRemoteHolder.java
    HelloRemote.java
    HelloRemoteOperations.java
```

Implementação do objeto remoto

- *O objeto remoto estende esqueleto HelloRemotePOA*
  - *Deve implementar métodos da interface HelloRemoteOperations (que é o equivalente Java da interface IDL)*

```
package hello.corba;

import org.omg.PortableServer.*;
import org.omg.CORBA.*;

public class HelloImpl extends HelloRemotePOA {

    private String nome = "Hello, World!";

    public String sayHello() {
        return nome;
    }
    public void sayThis(String toSay) {
        nome = toSay;
    }
}
```

Servidor: liga o objeto ao ORB

```
package hello.corba;
import org.omg.CORBA.*; import org.omg.CosNaming.*;
import org.omg.PortableServer.*;

public class HelloServer {
  public static void main (String[] args) {
    try {
      ORB orb = ORB.init(args, System.getProperties());
      POA rootPOA =
          POAHelper.narrow( orb.resolve_initial_references("RootPOA") );
      rootPOA.the_POAManager().activate();
      HelloImpl hello = new HelloImpl();
      hello.setORB(orb); hello.initPOA();
      org.omg.CORBA.Object ref = rootPOA.servant_to_reference(hello);
      HelloRemote helloRef = HelloRemoteHelper.narrow(ref);
      org.omg.CORBA.Object objRef =
          orb.resolve_initial_references("NameService");
      NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
      ncRef.rebind(ncRef.to_name( "hellocorba" ), helloRef);
      System.out.println("Remote object bound to 'hellocorba'.");
      orb.run();
    } catch (Exception e) {
      if (e instanceof RuntimeException) throw (RuntimeException)e;
      System.out.println("" + e);
}}}
```

Inicializa ORB e do POA Manager

Inicializa objeto

Transforma objeto em referência

Obtém referência do NameService

Liga objeto a nome no e roda ORB

Cliente

```
package hello.corba;

import org.omg.CORBA.*;
import org.omg.CosNaming.*;

public class HelloClient {
  public static void main (String[] args) {
    try {
      ORB orb = ORB.init(args, null);
      org.omg.CORBA.Object objRef =
          orb.resolve_initial_references("NameService");

      NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
      HelloRemote hello =
            HelloRemoteHelper.narrow(ncRef.resolve_str("hellocorba"));

      hello.sayHello();
      hello.sayThis("Goodbye!");
      hello.sayHello();

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

Obtém raiz do serviço de nomes

Procura nome no NameService e converte (narrow) referência obtida

Para executar

- *Inicie o ORB*

```
> orbd -ORBInitialPort 1900 -ORBInitialHost localhost &
```

- *Rode o servidor*

```
> java hello.corba.HelloServer -ORBInitialPort 1900
                               -ORBInitialHost localhost &
  Remote object bound to 'hellocorba'.
```

- *Rode o cliente*

```
> java hello.corba.HelloClient -ORBInitialPort 1900
                               -ORBInitialHost localhost &

  Hello, World!
  Goodbye!
```

- *Para rodar exemplo semelhante a este (CD)*
  - *No subdiretório* exemplos/, rode o ant para montar a aplicação:*

```
> ant buildcorba
```

  - *Depois, em janelas diferentes, inicie os servidores e cliente*

```
> ant orbd
> ant runcorbaserver
> ant runcorbaclient
```

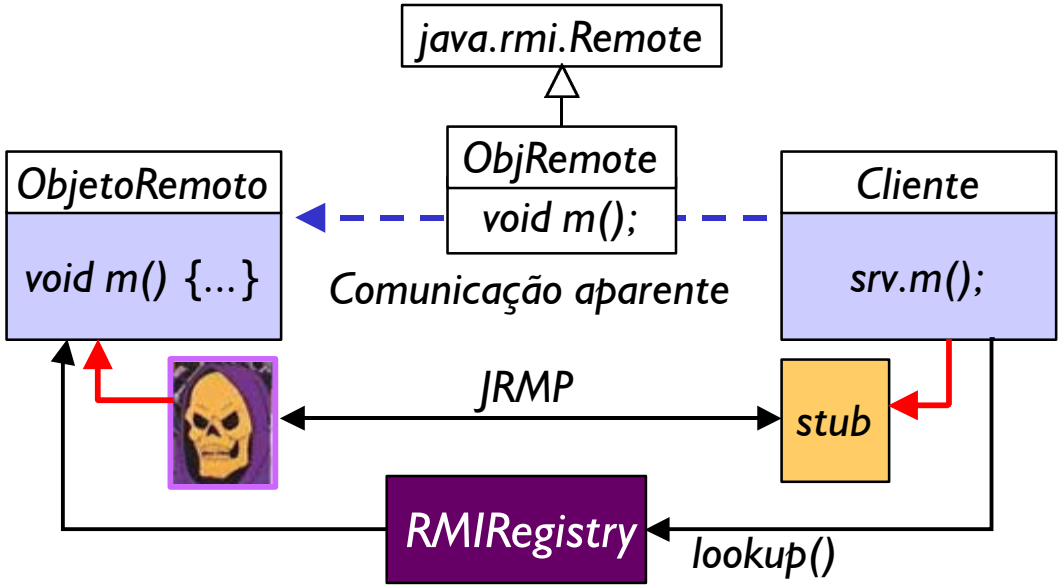
** informe antes no arquivo build.properties o caminho completo para este diretório*

Arquitetura Java RMI

- A arquitetura Java RMI pode ser comparada com a arquitetura CORBA
  - Usa um protocolo: JRMP - Java Remote Method Protocol
  - O RMI Registry funciona como um serviço de nomes centralizado e não hierárquico (todos os nomes estão no mesmo nível)
  - A interface java.rmi.Remote é base para sua interface "IDL"
- Principais características
  - Centralizada: cliente tem que saber em que servidor está o objeto remoto (não é transparente quanto à localização)
  - Independente de plataforma: por ser 100% Java (não suporta comunicação com objetos escritos em outras linguagens)
  - Suporta transferência de bytecode: objetos remotos podem ser transferidos fisicamente para o cliente (por ser 100% Java)
  - Suporta coleta de lixo distribuída
  - Suporta passagem de parâmetros por valor e por referência

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)

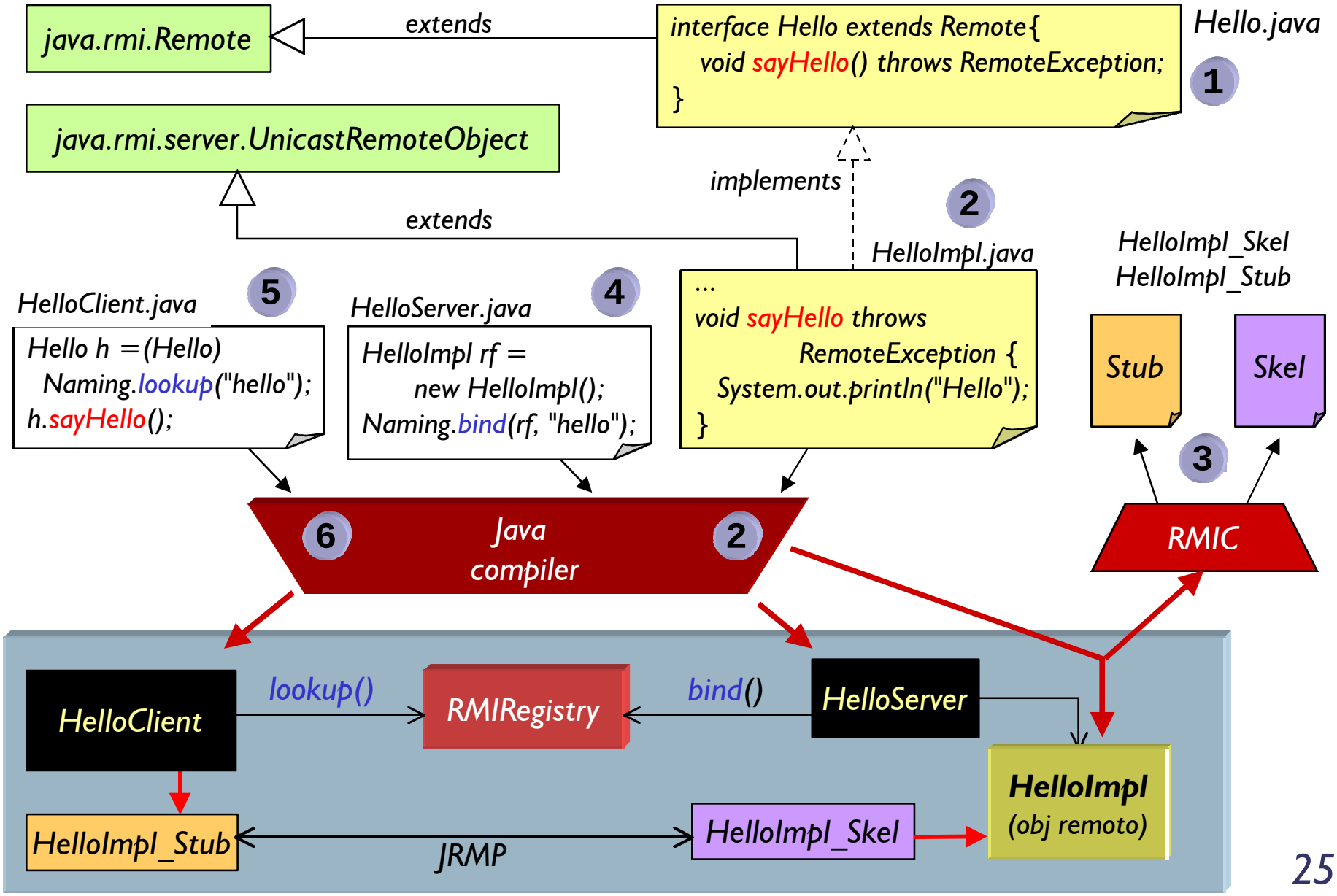


- **Esqueleto** *(lado-servidor)*
  - Recebe a requisição, desserializa os parâmetros e chama o método no objeto remoto. Depois serializa objetos retornados.
- **RMIRegistry**
  - Servidor registra o objeto usando *java.rmi.Naming.bind()*
  - Cliente recupera o objeto usando *java.rmi.Naming.lookup()*

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



Interface Remote

- *Interface de comunicação entre cliente e servidor*
  - *Stub e objeto remoto implementam esta interface de forma que cliente e esqueleto enxergam a mesma fachada*
- *Declara os métodos que serão visíveis remotamente*
  - *Todos devem declarar provocar RemoteException*

```
package hello.rmi;

import java.rmi.*;

public interface HelloRemote extends Remote {

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

}
```

Implementação do objeto remoto

```
package hello.rmi;

import java.rmi.*;

public class HelloImpl
                    extends java.rmi.server.UnicastRemoteObject
                    implements HelloRemote {

    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;
    }
}
```

Principal classe base para objetos remotos RMI (outra opção é Activatable)*

Construtor declara que pode provocar RemoteException!

* *javax.rmi.activatable.Activatable (modelo de implementação não abordado neste minicurso)*

Geração de Stubs e Skeletons

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

```
> rmic hello.rmi.HelloImpl
```

- *As classes são automaticamente compiladas (e as fontes são descartadas)*
- *Resultados*
  - *_HelloImpl_Stub.class*
  - *_HelloImpl_Skel.class*
- *Para preservar (e visualizar) o código-fonte gerado (.java) use a opção -keep:*

```
> rmic -keep hello.rmi.HelloImpl
```

Servidor e rmi.policy

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

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

SecurityManager viabiliza download de código

Associando o objeto com um nome no RmiRegistry

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.rmi;

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

public class HelloClient {
  public static void main (String[] args)  {
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
    }
    try {
      HelloRemote hello = (HelloRemote) Naming.lookup("hellormi");
      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"hellormi" no serviço de nomes

Execução

- *Inicie o RMIRegistry*

```
> rmiregistry &
```

- *Rode o servidor*

O diretório onde você instalou os exemplos

```
> java hello.rmi.HelloServer
  -Djava.rmi.server.codebase=file:///cursos/j433/exemplos/build/
  -Djava.security.policy=../lib/rmi.policy
Remote object bound to 'hellormi'.
```

- *Rode o cliente*

```
> java hello.rmiop.HelloClient
    -Djava.rmi.server.codebase=file:///aulaj2ee/cap03/build/
    -Djava.security.policy=../lib/rmi.policy
Hello, World!
Goodbye!
```

- *Você pode rodar esta aplicação no usando o Ant. Mude para o subdiretório* exemplos/ e monte uma aplicação semelhante a esta com*

```
> ant buildrmi
```

- *Depois, em janelas diferentes, inicie os servidores e cliente digitando*

```
> ant runrmiserver
> ant runrmiclient
```

** informe antes no arquivo build.properties o caminho completo para este diretório*

Necessidade de RMI-IIOP

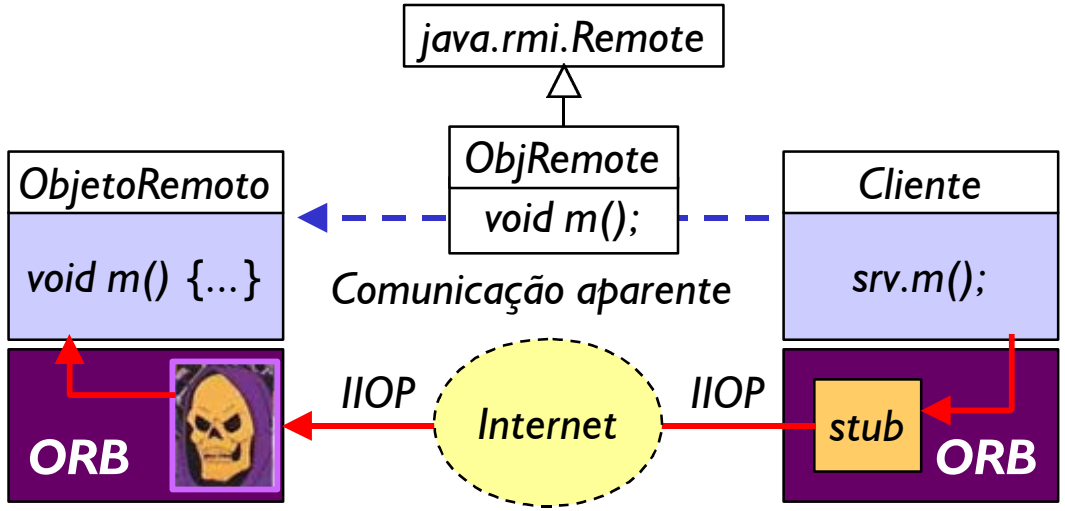
- *Queremos programar em Java*
  - *Não queremos aprender outra linguagem (IDL)*
  - *Não queremos escrever linhas e linhas de código para transformar, registrar e localizar objetos CORBA (queremos transparência)*
- *Queremos comunicação IIOP por causa de*
  - *integração com objetos em outras linguagens*
  - *vantagens do modelo ORB (transparência de localidade, escalabilidade, serviços, etc.)*
  - *comunicação com clientes escritos em outras linguagens*
- *Solução:* ***RMI sobre IIOP***
- *Principal benefício: comunicação com mundo CORBA*
- *Desvantagens: limitações de CORBA*
  - *Não faz download de bytecode*
  - *Não faz Distributed Garbage Collection - DGC*

Arquitetura Java RMI sobre IIOP

- Modelo de programação RMI com comunicação CORBA
  - Usa protocolo IIOP
  - Comunica-se através de ORB e IIOP
  - A interface java.rmi.Remote é base para sua interface remota
  - Gera IDL se necessário
- Principais características
  - Transparência de localidade: não precisa saber onde estão objetos
  - Independência de plataforma: objetos podem estar distribuídos em plataformas diferentes
  - Neutralidade de linguagem é possível: pode se comunicar com objetos escritos em linguagens diferentes (gera IDL)
  - Suporta passagem de parâmetros por valor e por referência
  - Sem suporte à transferência de bytecode
  - Sem suporte à coleta de lixo distribuída
- Melhor dos dois mundos para programadores Java!

Comunicação RMI-IIOP

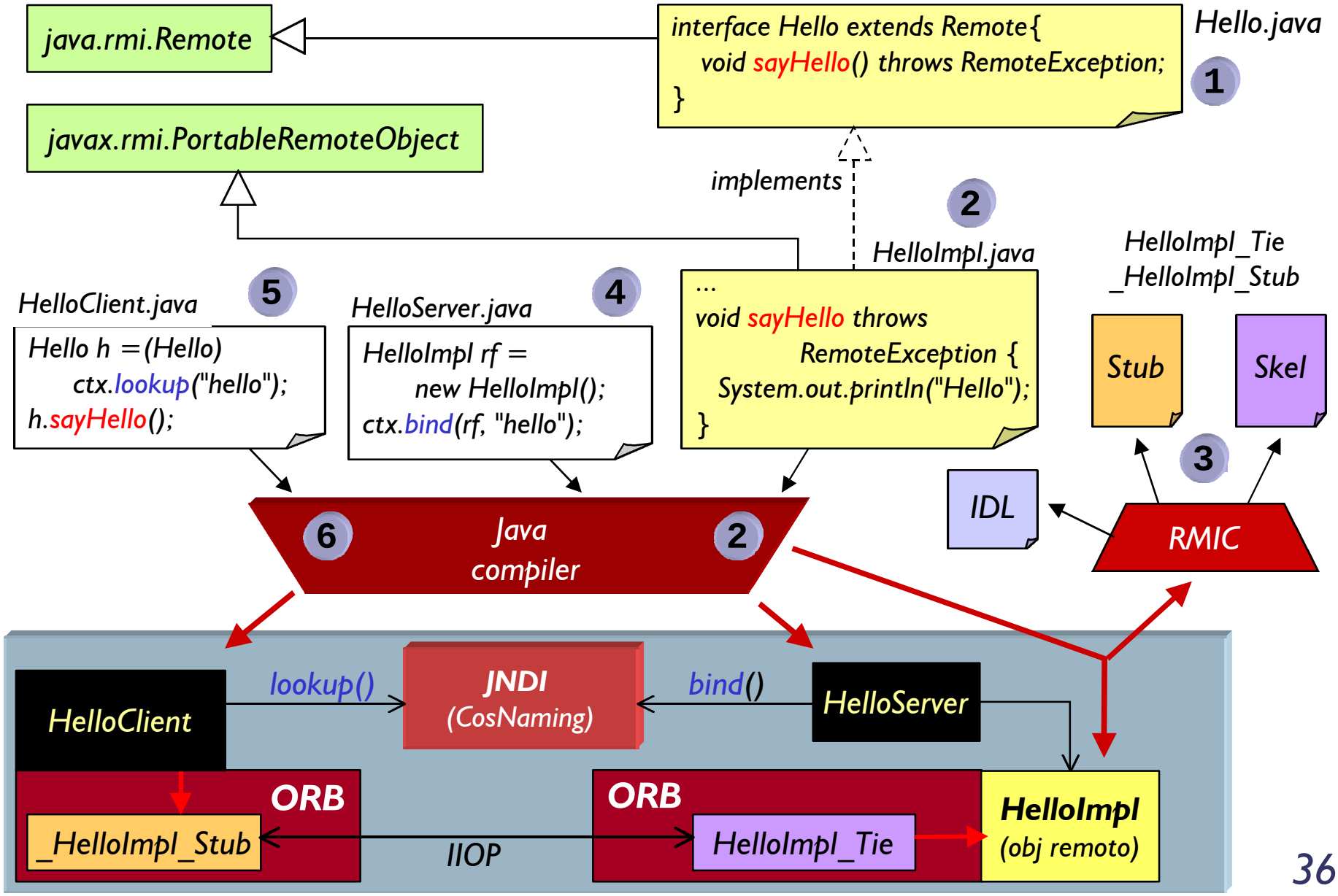
- **Interface Remote**
  - Abstração da interface do objeto remoto
  - Usada para gerar stubs e esqueletos
- **Stub** (lado-cliente)
  - Transforma parâmetros (serializados) em formato independente de máquina e envia requisição pela rede através do ORB
- **ORB**: barramento comum
  - ORB do cliente passa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

    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*
  - *_HelloRemote_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*

HelloRemote.idl

```
#include "orb.idl"
module hello {
  module rmiop {
    interface HelloRemote {
       ::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 {
         HelloRemote 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";
};
```

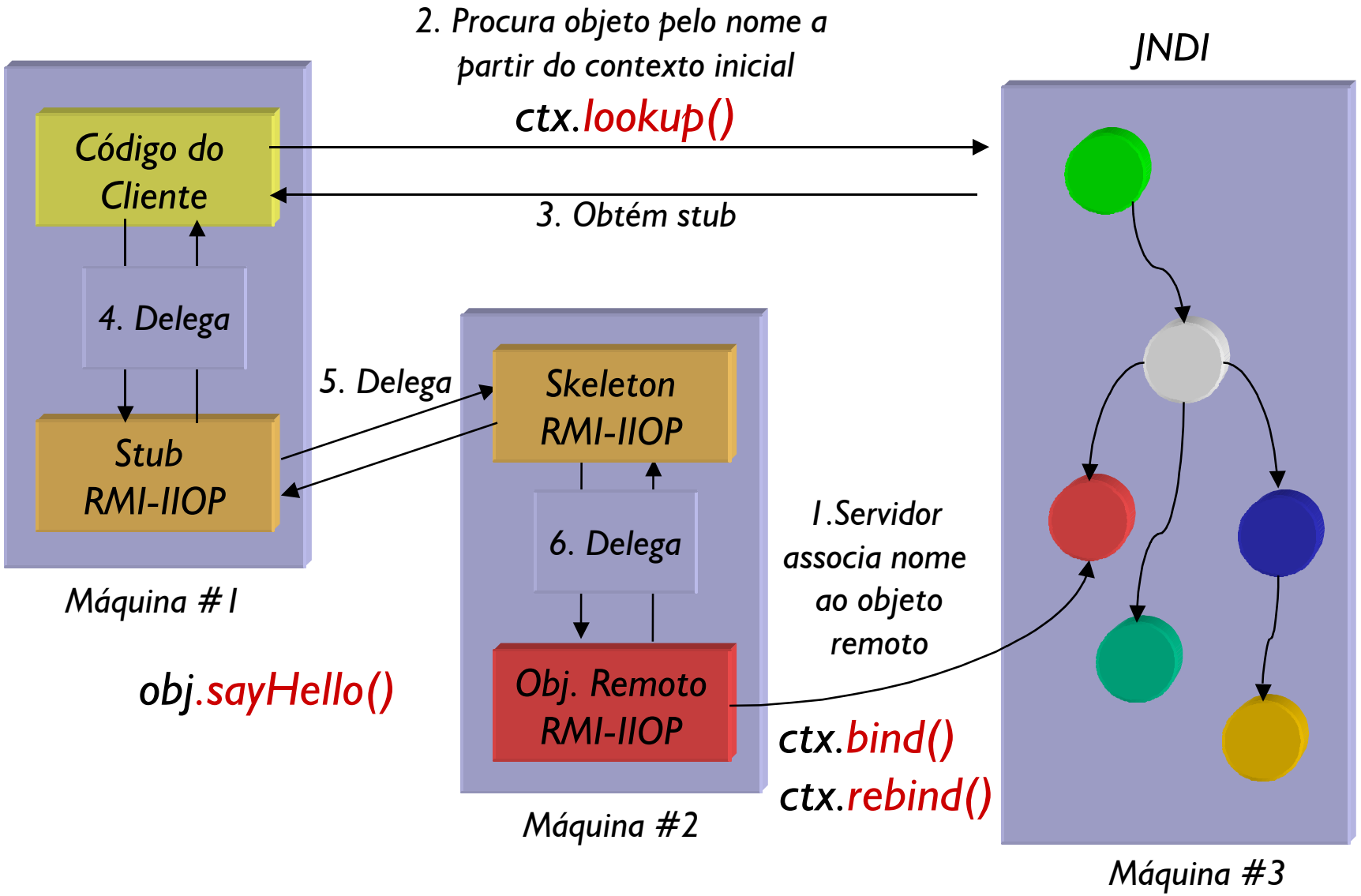
Cliente

```
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");
      HelloRemote hello = (HelloRemote)
         javax.rmi.PortableRemoteObject.narrow(obj,
                                    hello.HelloRemote.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

RMI-IIOP e JNDI



Fonte da ilustração: Mastering EJB2, Ed Roman et al.

Execução

- *Inicie o ORB*

```
> orbd -ORBInitialPort 1900 -ORBInitialHost localhost &
```

- *Rode o servidor*

O diretório onde você instalou os exemplos

```
> java hello.rmiop.HelloServer
 -Djava.rmi.server.codebase=file:///cursos/j433/exemplos/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'.
```

- *Rode o cliente*

```
> java hello.rmiop.HelloClient <mesmas propriedades -D do servidor>
Hello, World!
Goodbye!
```

- *Alternativas para reduzir a quantidade de parâmetros de execução*
  - *Defina as propridades no código ou crie um arquivo rmi.properties (melhor!)*
  - *Use um script shell, .bat ou o Ant*
- *Ant: no subdiretório* exemplos/, monte aplicação semelhante a esta com*

```
> ant buildrmiop
```

- *Depois, em janelas diferentes, inicie os servidores e cliente digitando* "**ant orbd**", "**ant runrmiopserver**" e "**ant runrmiopclient**"

* *informe antes no arquivo build.properties o caminho completo para este diretório*

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

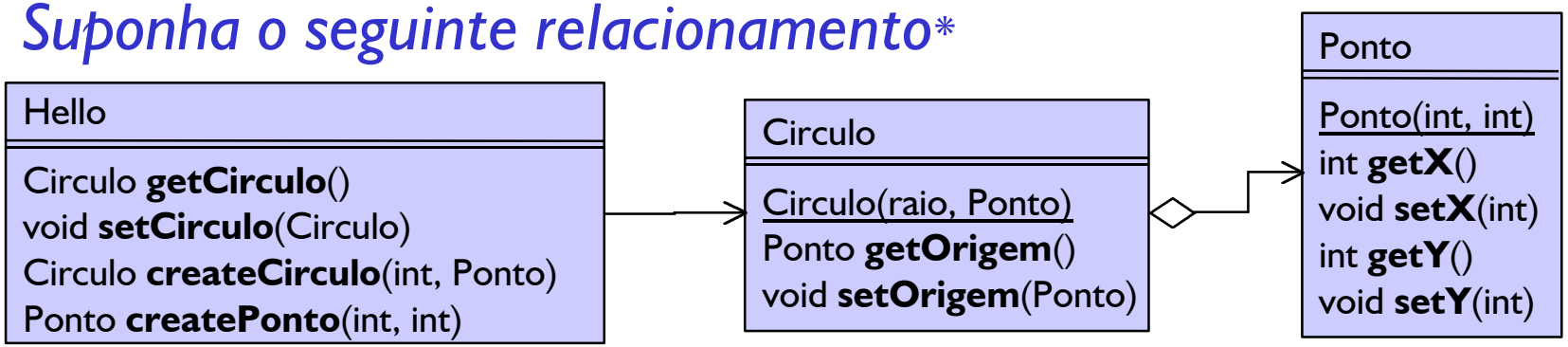
* É possível implementar a serialização de forma personalizada implementando métodos writeObject(), readObject(), writeReplace() e readResolve(). Veja especificação.

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)*

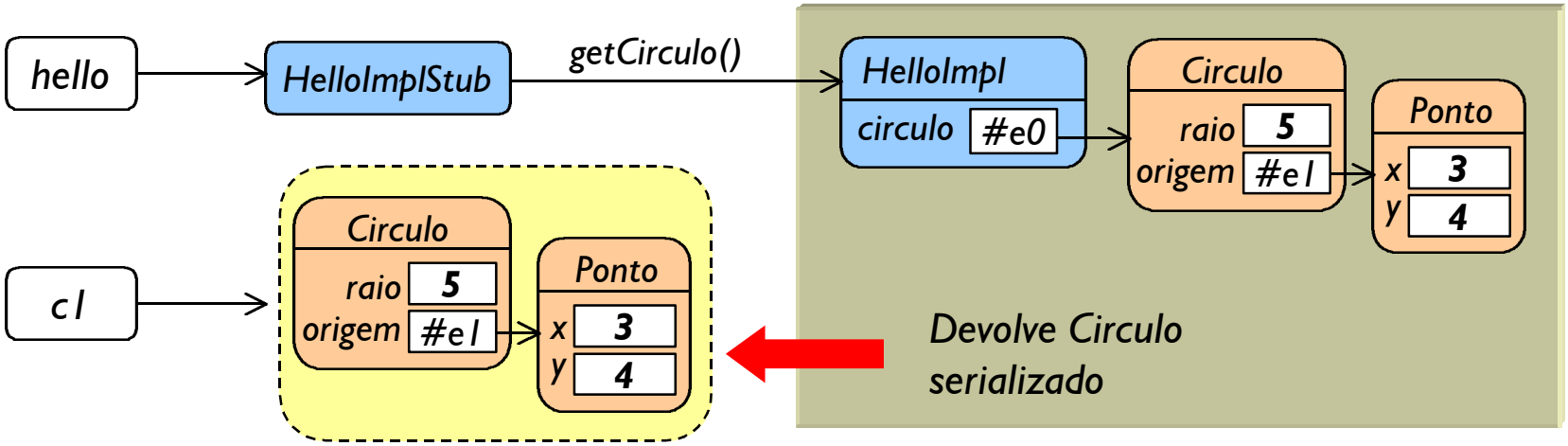
```
c2.getOrigem().setX(3000);
```

… e remotamente?

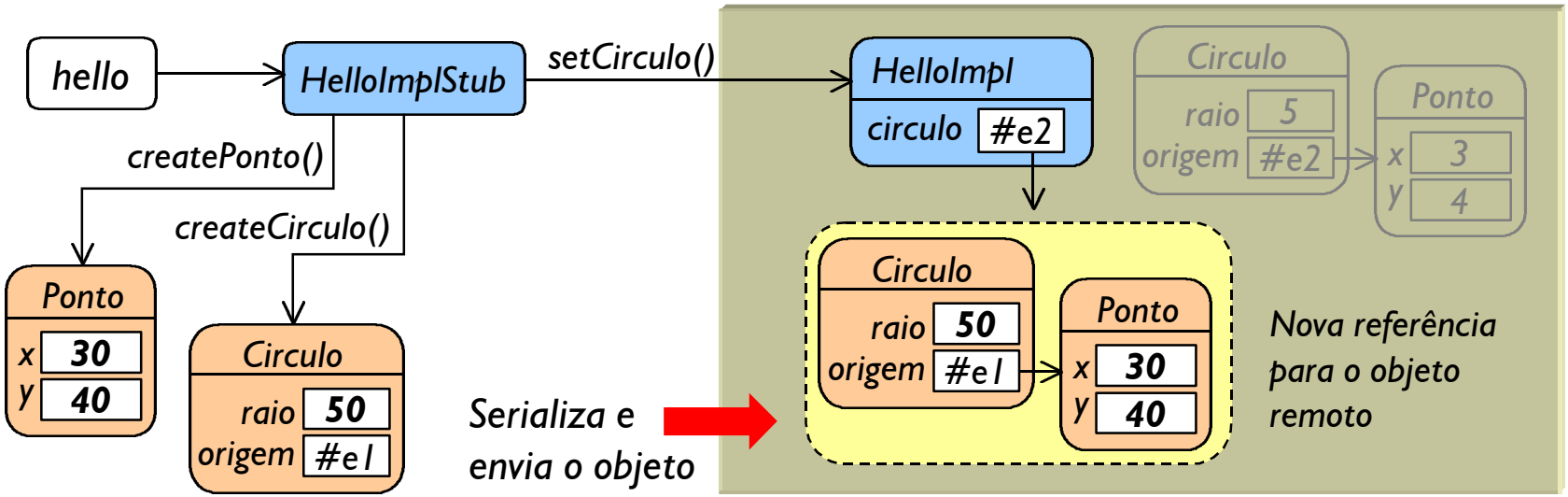
* *veja exemplos/src*

Passagem de parâmetros em RMI

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

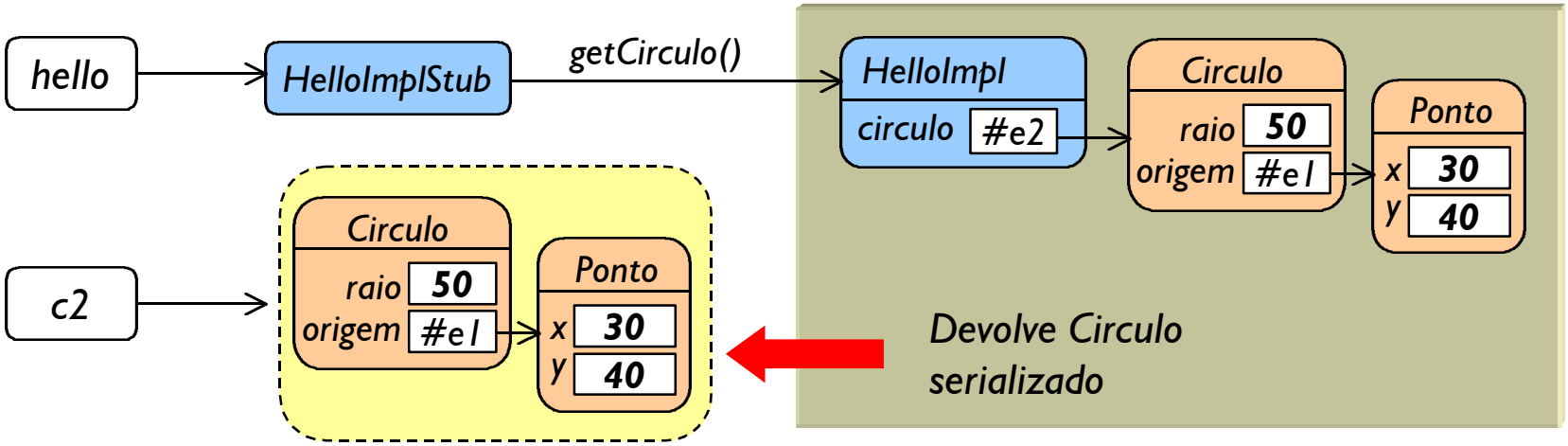


b `hello.setCirculo(hello.createCirculo(50, hello.createPonto(30, 40)));`



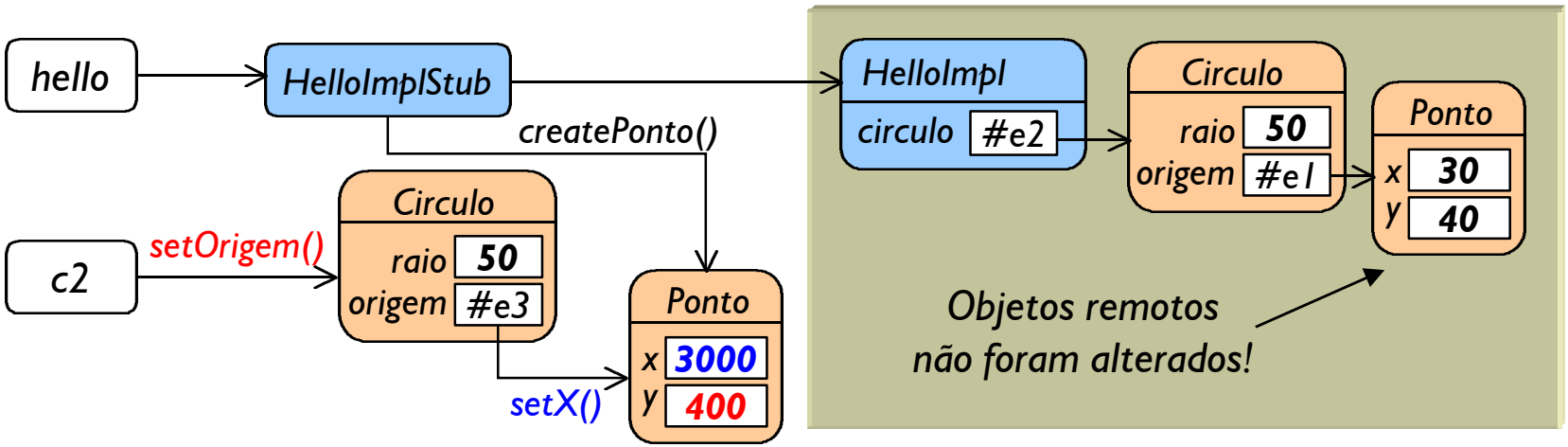
Conseqüências da passagem por valor

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!***

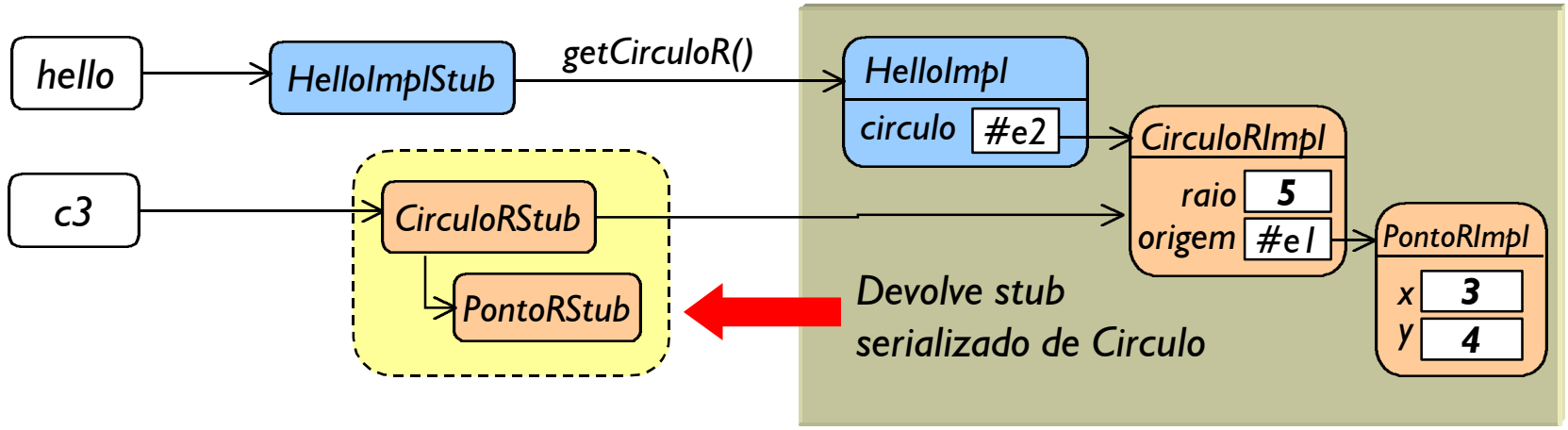


Passagem de parâmetros por referência

- 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)

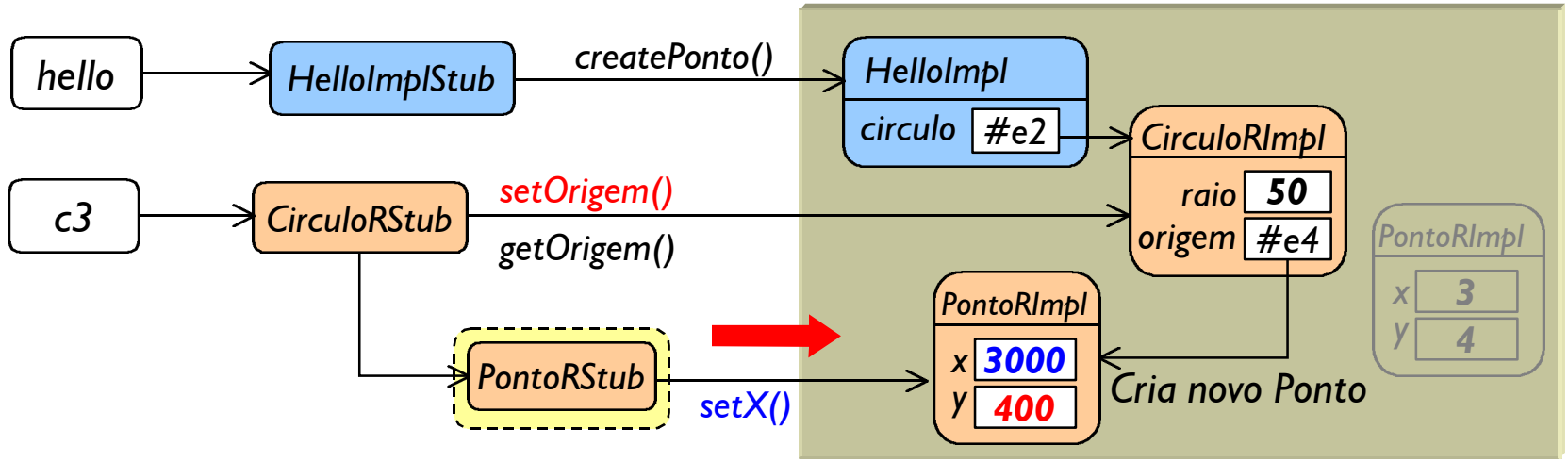
Passagem por referência

(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 exemplos, **`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*
- *Use RMI sobre JRMP se*
  - *Sua aplicação é 100% Java e sempre vai ser 100% Java*
  - *Você não precisa de um sistema de nomes distribuído e hierárquico*
  - *Os seus objetos nunca mudam de servidor*
  - *Quiser usar recursos exclusivos do RMI (activation, DGC, download)*
- *Use RMI sobre IIOP se você*
  - *Prefere o modelo de desenvolvimento Java RMI (deseja especificar suas interfaces em Java e não em IDL)*
  - *Quer usufruir da melhor escalabilidade do modelo CORBA*
  - *Precisa garantir interoperabilidade com sistemas CORBA*
- *Use Java IDL se você*
  - *Já tem um sistema especificado através de IDLs e precisa criar clientes ou servidores que se comuniquem via IIOP*

Exercício

- *Considere o seguinte arquivo IDL:*

```
module exercicio {
   interface Sorte {
      long numero();
   };
};
```

Número retorna um valor aleatório entre 1 e 10000:
`Math.ceil(Math.random()*10000)`

1. Implemente-o em CORBA usando Java IDL.

- *Compile o IDL para gerar stubs e skeletons*
- *Implemente o objeto remoto, servidor e cliente*
- *Inicie os ORBs e inicie a aplicação*

2. Implemente-o em (a) JAVA RMI e (b) RMI sobre IIOP.

- *Escreva uma interface Java (ou use a que foi gerada na fase 1)*
- *Implemente objeto remoto, cliente e servidor*
- *Inicie os ORBs (ou RMI daemons + Registry) e rode a aplicação*

3. Refaça o exercício 1 usando o IDL gerado no exercício 2b.

- *Implemente apenas o cliente (idlj -fclient)*
- *Inicie o cliente CORBA e o servidor RMI-IIOP*

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 (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 RMI-IIOP (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

[8] *David Flanagan et al. Java Enterprise in a Nutshell, 2nd. edition. Abril 2002 (O'Reilly).*
Oferece tutoriais sobre as principais tecnologias Java para ambientes distribuídos, incluindo RMI, RMI-IIOP e Java IDL

hlsr@uol.com.br

www.argonavis.com.br

Ministrado pela primeira vez
no Java Open Brasil 1998
em Brasília, DF

Minicurso Java em Ambientes Distribuídos, 1998
jav400 - Curso Programação distribuída em Java, 1999
jav433 - Minicurso Java RMI x CORBA x RMI-IIOP, 2000