

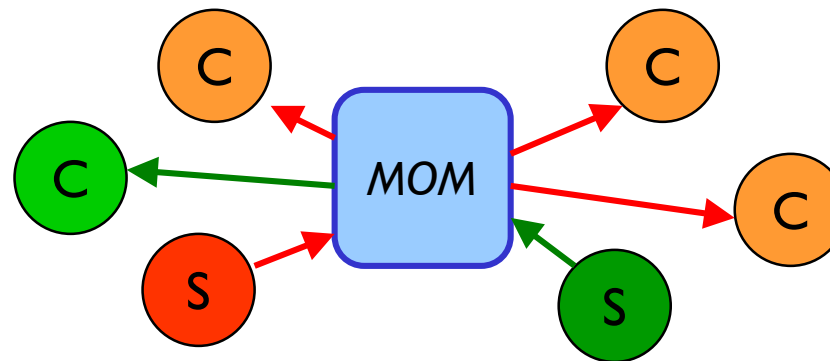
# Java 2 Enterprise Edition

# JMS e Message-driven Beans

*Helder da Rocha*  
[www.argonavis.com.br](http://www.argonavis.com.br)

# O que é Messaging

- **Método de comunicação** entre componentes ou aplicações
  - Arquitetura **peer-to-peer** com **serviço centralizado** para repasse de mensagens recebidas e enviadas
  - Clientes e servidores enviam e recebem mensagens para canais administrados por serviço central de mensagens (MOM)



- **Viabiliza comunicação distribuída com acoplamento fraco**
  - Interface genérica: MOM ignora conteúdo e repassa qualquer coisa. Formato do conteúdo deve ser conhecido pelas partes
  - Assíncrona: Comunicação pode ocorrer mesmo que o cliente e servidor não estejam disponíveis ao mesmo tempo

# Message-oriented Middleware (MOM)

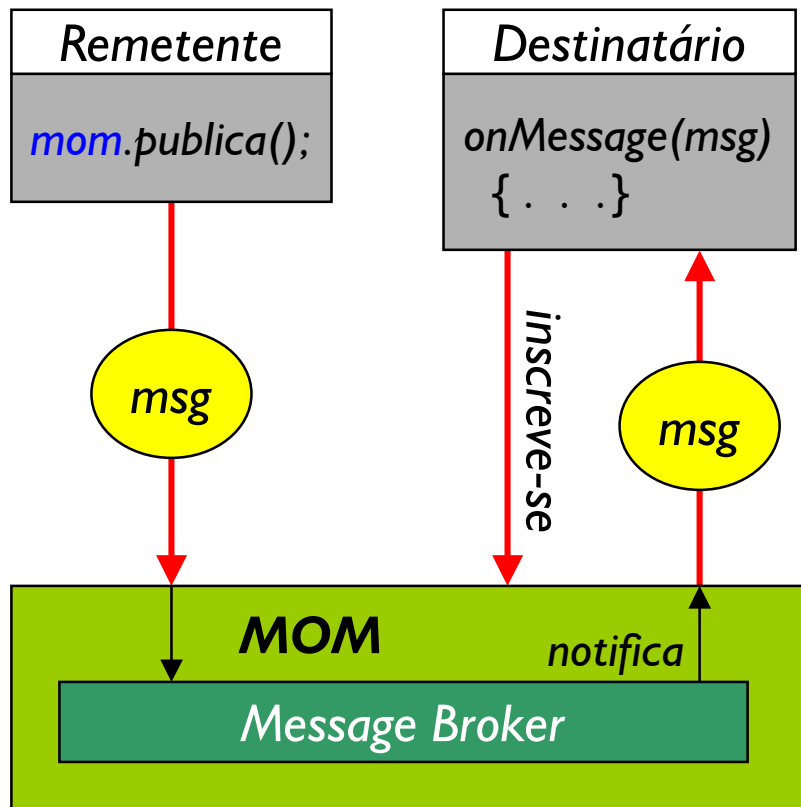
- *Sistemas de messaging são freqüentemente chamados de **Message-Oriented Middleware (MOM)***
  - *Conceito não é novo: já existiam há algum tempo em implementações proprietárias (e incompatíveis)*
  - *JMS API: solução independente de fabricante para acessar serviços MOM a partir de clientes Java*
- *Alguns produtos MOM compatíveis com JMS:*
  - ***Open source:** JBossMQ, OpenJMS, JORAM*
  - *IBM MQSeries, IPlanet MQ, Bea WebLogic, HP-MS, Progress SoniqMQ*
  - *Mais em: [java.sun.com/products/jms/](http://java.sun.com/products/jms/)*

# Messaging vs. RMI/RPC vs. E-mail

- **Messaging**
  - Mensagens são representadas como **eventos**
  - **Interface genérica** (reutilizada em aplicações diferentes)
  - Arquitetura **centralizada** (tudo passa pelo MOM)
  - Serviços de diretórios localizam **canais de comunicação** (destinos)
- **RMI/RPC (Corba, Java RMI, etc.)**
  - Mensagens são representadas como **chamadas** para métodos remotos
  - Cada aplicação se comunica através de uma **interface definida**
  - Pode ser **descentralizado** (rede de ORBs ligados por IIOP)
  - Serviços de diretórios localizam **objetos**

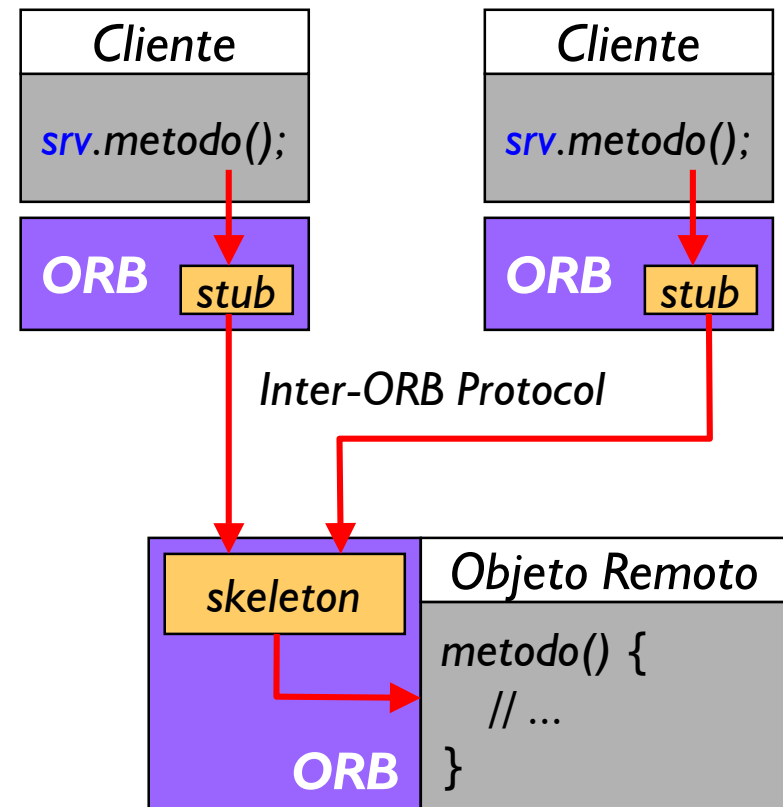
# Messaging vs. RMI/RPC

- Sistema de Messaging com paradigma pub/sub



Protocolo comum:  
**mensagem**

- Sistema RMI/RPC (Java RMI ou CORBA)



Protocolo comum:  
**interface dos objetos**

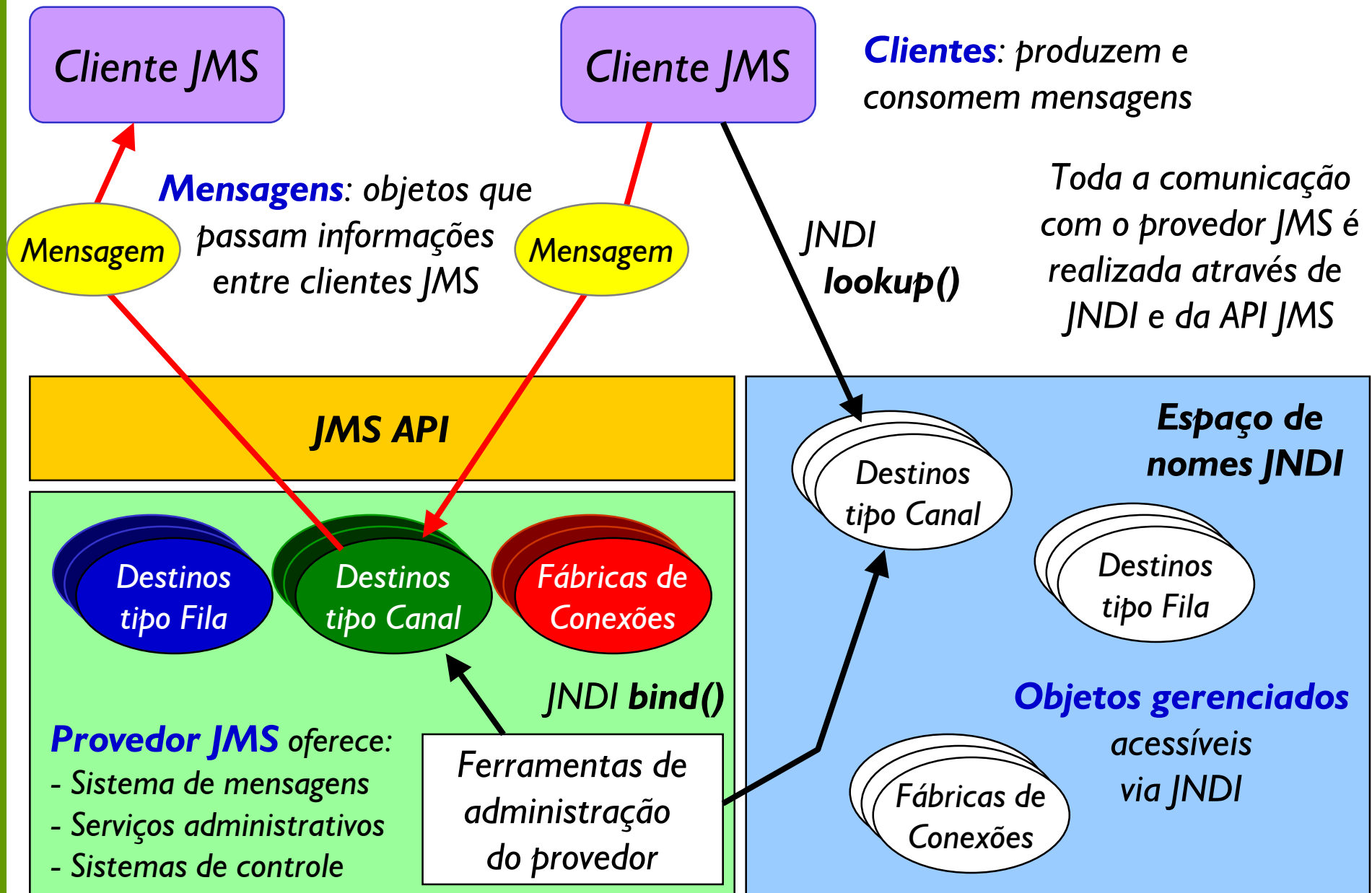
# Quando usar um MOM em vez de RMI/RPC?

- ... ou, quando decidir por acoplamento mais fraco?
    - Quando a comunicação se baseia mais no formato de mensagens que em interfaces rígidas (componentes **não dependem da interface** de outros componentes)
    - Quando a disponibilidade dos componentes é imprevisível, mas sua aplicação precisa rodar mesmo que componentes **não estejam todos acessíveis**
    - Quando for preciso suportar comunicação assíncrona: componente pode enviar informações para outro e **continuar a operar mesmo sem receber resposta imediata**
- ➡ Cenário comum em muitas aplicações B2B!**

# Java Message Service

- *Interface Java única para unir as MOMs incompatíveis*
- **API** que permite que aplicações criem, enviem, recebam e leiam mensagens através de um MOM
  - API consiste principalmente de **interfaces** (implementadas pelo fabricante do MOM)
  - Parte integral da **plataforma J2EE** (acrescenta possibilidade de comunicação assíncrona a EJBs)

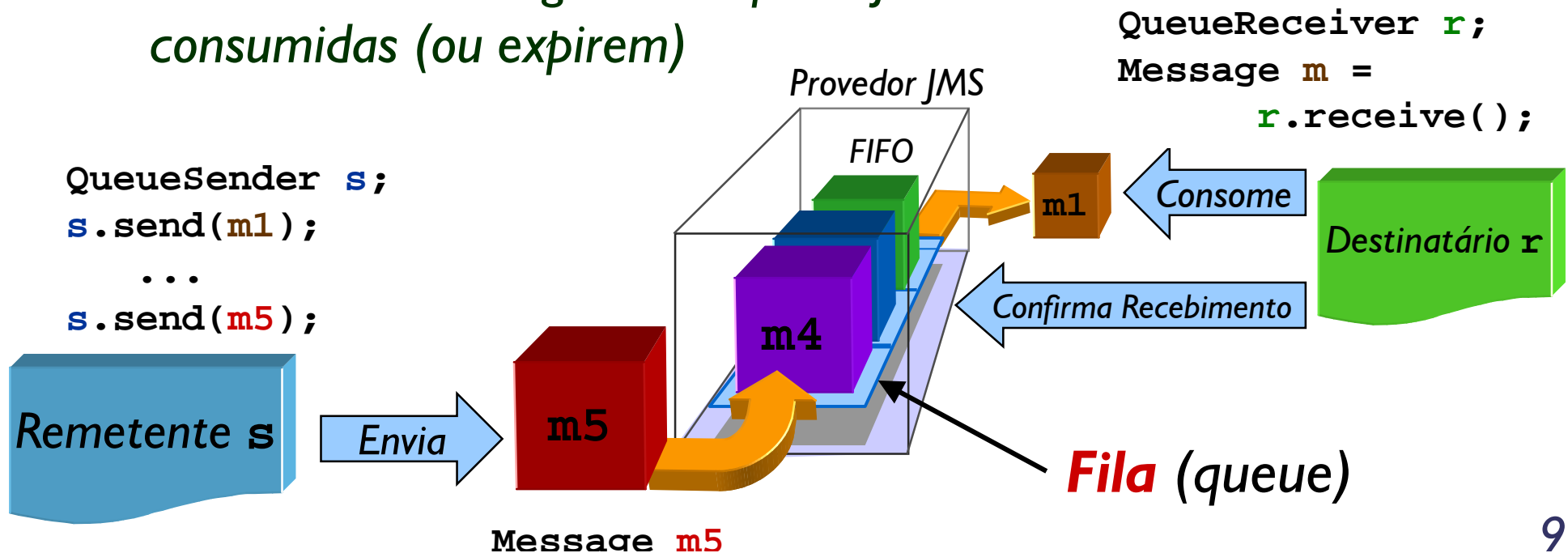
# Arquitetura JMS





# Domínio ptp (ponto-a-ponto)

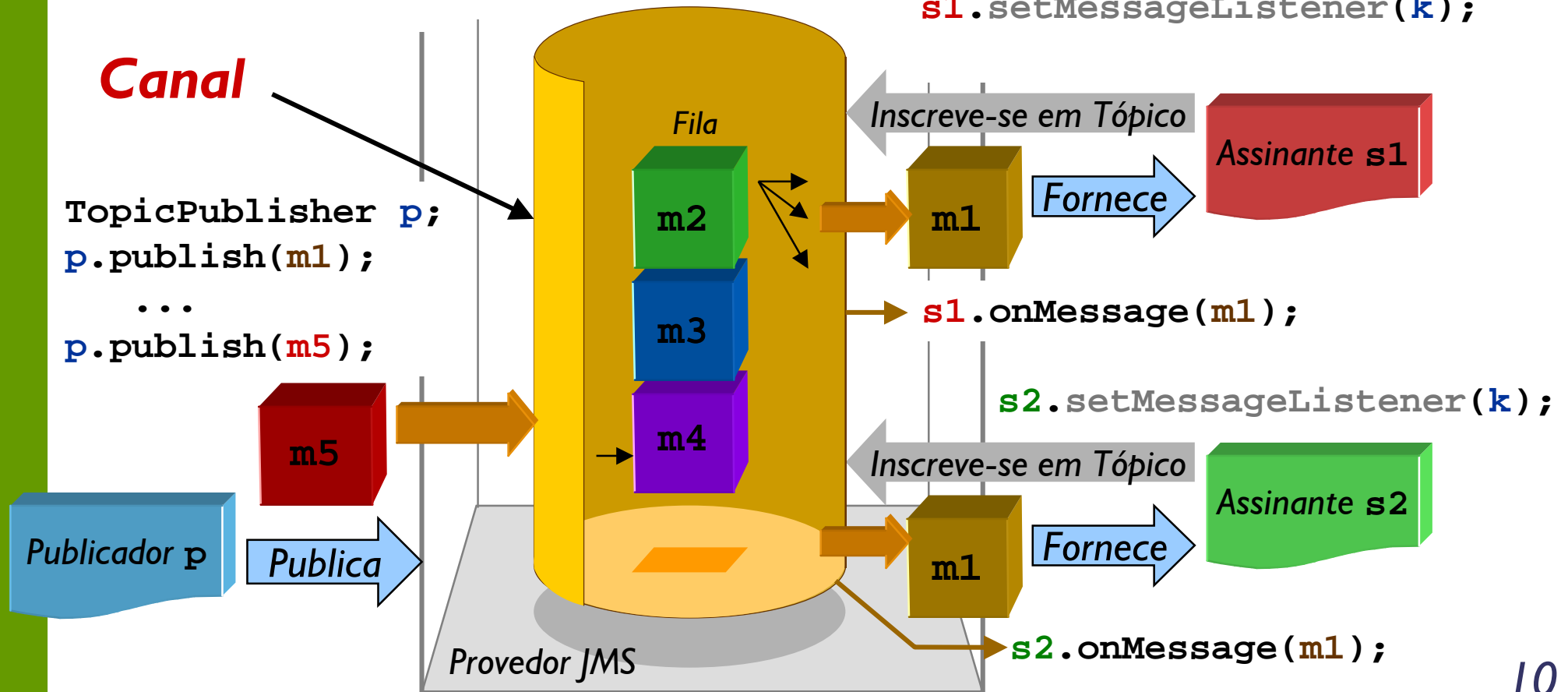
- Baseado no conceito de filas, remetentes e destinatários
  - Ou ou muitos para um: cada mensagem é enviada para uma fila específica e é **consumida** por um destinatário (que pode ou não estar disponível no momento)
  - Destinatário confirma que a mensagem foi recebida e processada corretamente (**acknowledgement**)
  - Filas retêm mensagens até que sejam consumidas (ou expirem)



# Domínio pub/sub (publica/inscreve)

- Baseado em canais (tópicos)
  - Muitos para muitos: mensagens são enviadas a um canal onde todos os assinantes do canal podem retirá-la

```
TopicSubscriber s1;  
MessageListener k =  
    new MessageListener() {  
        public void  
            onMessage(Message m){...}  
    };  
s1.setMessageListener(k);
```

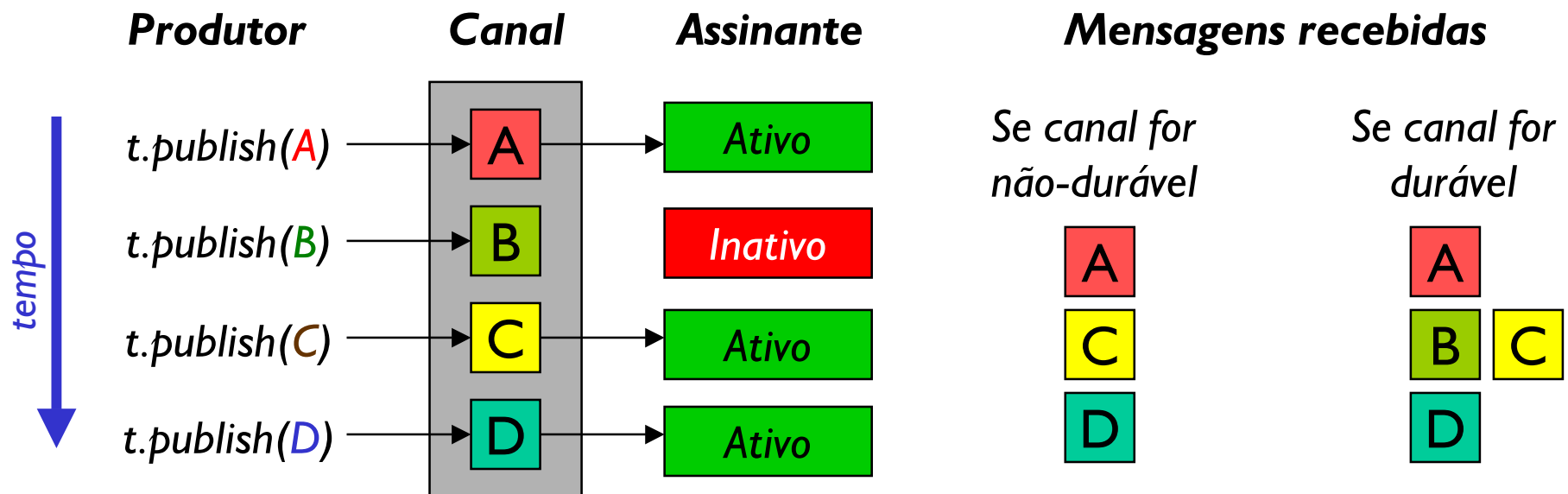


# Consumo síncrono e assíncrono

- Mensagens podem ser retiradas do destino através de uma chamada `receive()`
  - Esta é a forma **síncrona** de retirar mensagens
  - Cliente bloqueia enquanto mensagem não chega
- Mensagens também podem ser enviadas ao destinatário quando chegarem
  - Esta é a forma **assíncrona** de retirar mensagens
  - Para ser notificado, destinatário precisa ser implementado como um ouvinte e cadastrado para receber notificações
- Nos exemplos anteriores, o modelo P2P foi implementado com destinatário síncrono e o modelo `publish-subscribe` com destinatário assíncrono

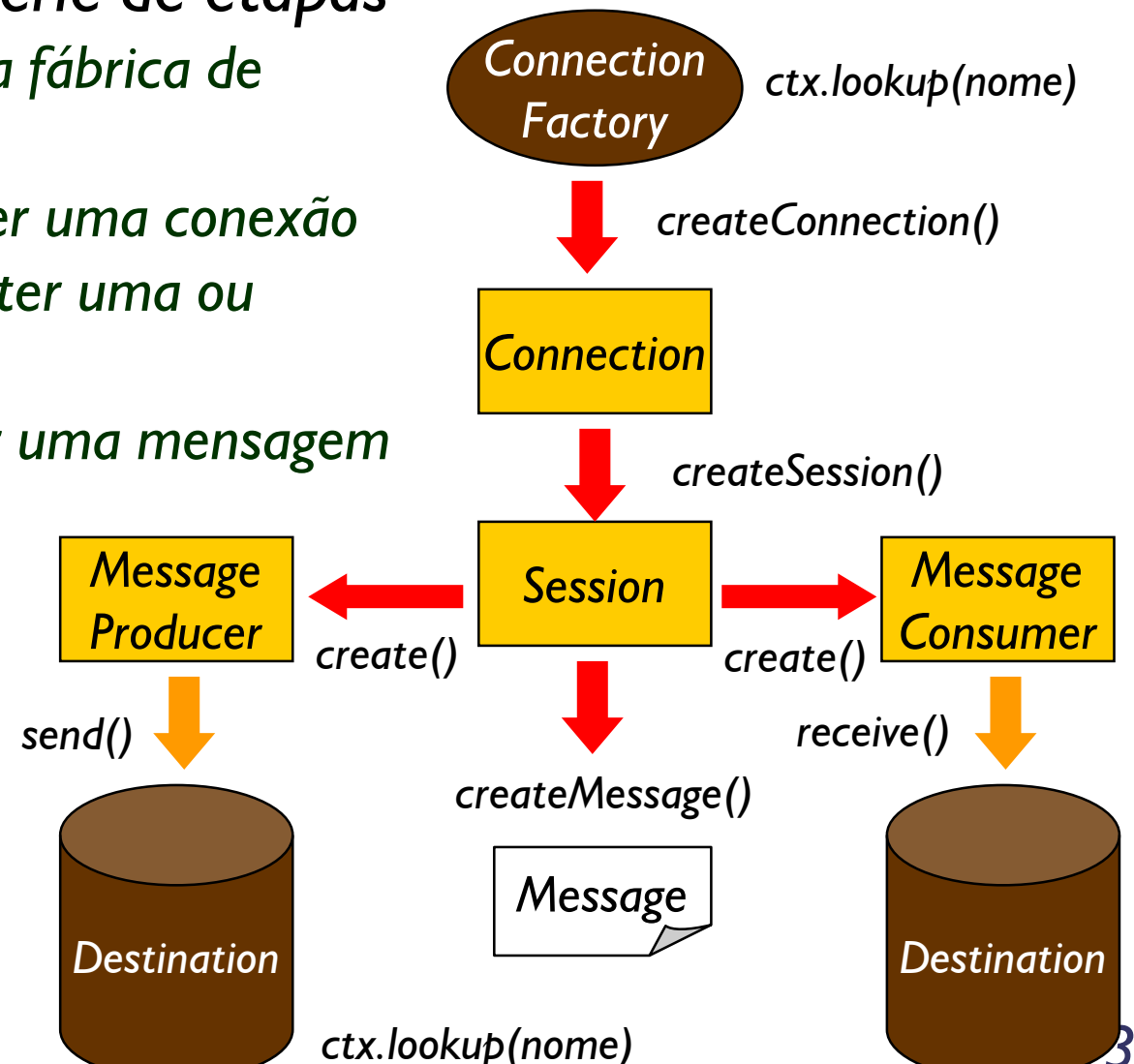
# Canais duráveis e não-duráveis

- Cada mensagem enviada a um canal pode ter múltiplos consumidores
  - Mensagem permanece disponível até que todos os assinantes a tenham retirado
  - Em canais **não-duráveis** assinante perde as mensagens enviadas nos seus períodos de inatividade



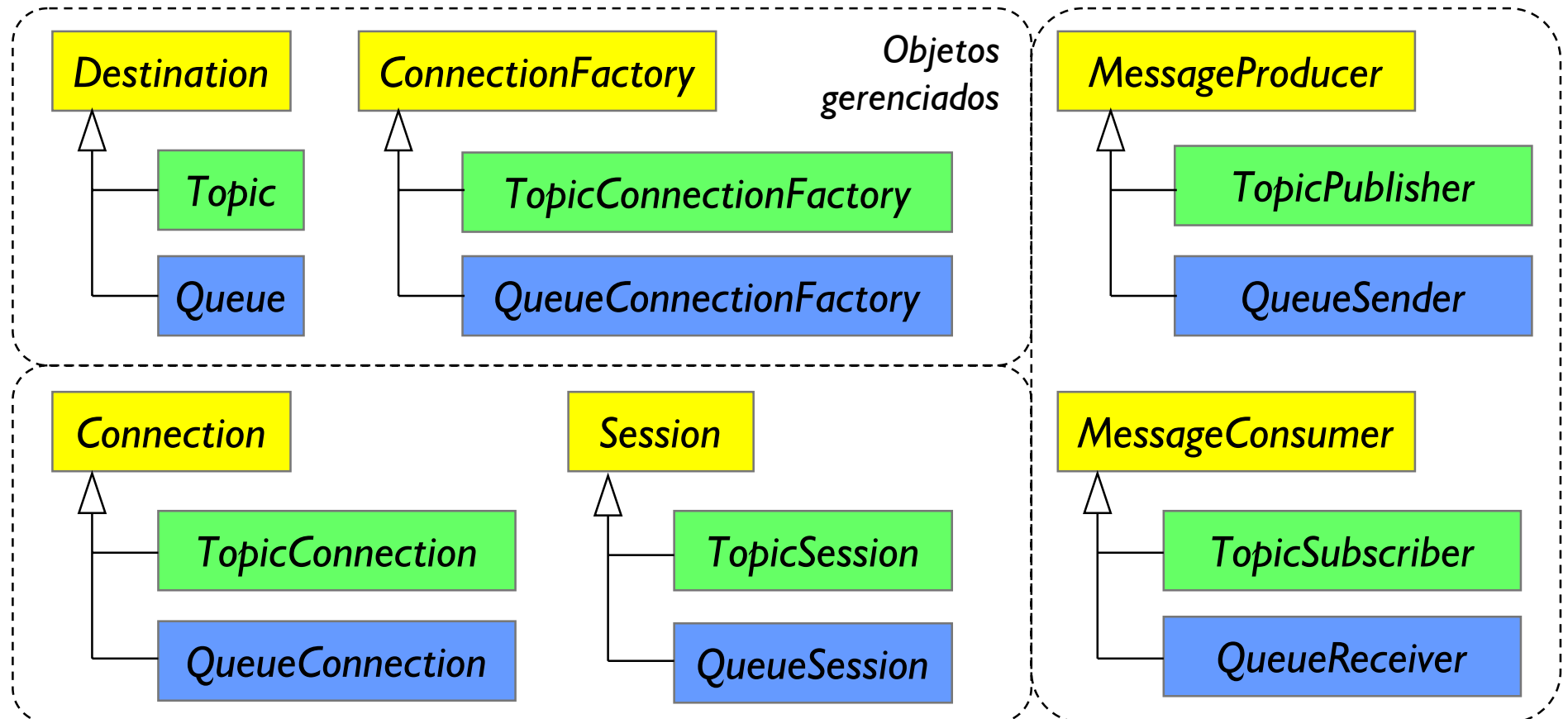
# Desenvolvimento de aplicações JMS

- Para escrever aplicações que enviam ou recebem mensagens é preciso realizar uma série de etapas
  - Obter um destino e uma fábrica de conexões via JNDI
  - Usar a fábrica para obter uma conexão
  - Usar a conexão para obter uma ou mais sessões
  - Usar a sessão para criar uma mensagem
  - Iniciar a sessão
- Depois, pode-se
  - Enviar mensagens
  - Receber mensagens
  - Cadastrar ouvintes para receber mensagens automaticamente



# Interfaces: dois domínios

- É preciso escolher um domínio
  - Para cada domínio há um destino, fábrica de conexões, conexão, sessão, produtor e consumidor
  - Interfaces são semelhantes



- Domínio pub/sub
- Domínio ptp

# Objetos gerenciados

- **Destinos** (filas e tópicos) e **fábricas de conexões** não podem ser criados através da API
  - São criados pelo provedor JMS e ligados a JNDI através de suas ferramentas administrativas. No J2EE-RI:

```
> j2eeadmin -addJmsDestination queue/MinhaFila queue
```

- Para usar um objeto gerenciado
  - O cliente precisa obtê-lo através de uma consulta (lookup) no serviço de nomes JNDI
  - Não há padronização do nome usado em JNDI (nomes default variam entre fabricantes)
- Exemplo (obtenção de um destino do tipo Topic):

```
String nomeJRI = "jms/Topic"; //Default J2EE RI  
String nomeJBoss = "topic/testTopic"; // Default JBossMQ
```

```
Context ctx = new InitialContext();  
Topic canal = (Topic) ctx.lookup(nomeJBoss);
```

# Há dois tipos de destinos JMS

## ■ **Filas** (Queue)

- *Retêm todas as mensagens que recebem até que sejam retiradas ou expirem*
- *Para cada mensagem enviada, apenas um cliente pode retirá-la*

```
Queue fila = (Queue) ctx.lookup("jms/Queue");
```

## ■ **Canais** (Topic)

- *Cada canal pode ter vários clientes assinantes*
- *Cada assinante recebe uma cópia das mensagens enviadas*
- *Para receber uma mensagem publicada em um canal, clientes precisam já ser assinantes dele antes do envio.*
- *Em canais "duráveis", assinantes não precisam estar ativos no momento do envio. Retornando, receberão mensagens não lidas.*

```
Topic canal = (Topic) ctx.lookup("jms/Topic");
```



# Fábricas de conexões

- *Antes que se possa*
  - **enviar** uma mensagem para uma fila,
  - **publicar** uma mensagem em um canal,
  - **consumir** uma mensagem de uma fila ou
  - **fazer uma assinatura** de um canal
- é preciso obter uma **conexão** ao provedor JMS
- Isto é feito através de uma **fábrica de conexões**. Há duas:
  - **TopicConnectionFactory** - para conexões no domínio Topic
  - **QueueConnectionFactory** - para conexões no domínio Queue
- É preciso conhecer o **nome JNDI**

```
String nomeJRI = "TopicConnectionFactory"; //default J2EE-RI
String nomeJBoss = "ConnectionFactory"; // default JBossMQ
```

```
Context ctx = new InitialContext();
TopicConnectionFactory factory =
    (TopicConnectionFactory) ctx.lookup(nomeJBoss);
```

Precisa ser definido (geralmente arquivo `jni.properties` no CPATH)

- *Encapsulam uma conexão virtual com o provedor JMS*
  - *Suportam múltiplas sessões (threads)*
- *Uma vez obtida uma fábrica de conexões, pode-se obter uma conexão*

```
QueueConnection queueCon =  
    queueConnectionFactory.createQueueConnection();
```

```
TopicConnection topicCon =  
    topicConnectionFactory.createTopicConnection();
```

- *Quando a conexão terminar, é preciso fechá-la*
  - *O fechamento fecha todas as seções, produtores e consumidores associados à conexão*

```
queueCon.close();  
topicCon.close();
```

- *Contexto onde se produz e se consome mensagens*
  - *Criam produtores, consumidores e mensagens*
  - *Processam a execução de ouvintes*
  - *Single-threaded*
- *Podem ser configuradas para definir*
  - *forma de acknowledgement*
  - *uso ou não de transações (tratar uma série de envios/recebimentos como unidade atômica e controlá-la via commit e rollback)*
- *Exemplos*

```
TopicSession topicSession =  
    topicCon.createTopicSession(false,  
        Session.AUTO_ACKNOWLEDGE);  
QueueSession queueSession =  
    queueCon.createQueueSession(true, 0);
```

Sem transações

Confirmação automática após recebimento correto

Sessão controlada por transações

Tratamento de confirmações não especificado

# Produtores de mensagens

- Objeto criado pela sessão e usado para enviar mensagens para um destino

- *QueueSender*: domínio ponto-a-ponto
- *TopicPublisher*: domínio pub/sub

```
QueueSender sender =  
    queueSession.createSender( fila );  
TopicPublisher publisher =  
    topicSession.createPublisher( canal );
```

- Uma vez criado o produtor, ele pode ser usado para enviar mensagens

- *send()* no domínio ponto-a-ponto
- *publish()* no domínio pub/sub

```
sender.send( message );  
publisher.publish( message );
```

➔ Durante o envio cliente pode definir **qualidade do serviço** como modo (persistente ou não), prioridade (sobrepõe comportamento FIFO) e tempo de vida (TTL)

# Consumidores de mensagens

- Objeto criado pela sessão e usado para receber mensagens
  - *QueueReceiver* e *QueueBrowser*: domínio ponto-a-ponto
  - *TopicSubscriber*: domínio pub/sub

```
QueueReceiver receiver =  
    queueSession.createReceiver(fila);  
TopicSubscriber subscriber =  
    topicSession.createSubscriber(canal);
```

- É preciso iniciar a conexão antes de começar a consumir:  
`topicCon.start();`
- Depois, pode consumir mensagens de forma *síncrona* (método é o mesmo para domínios PTP e pub/sub)  

```
Message queueMsg = receiver.receive();  
Message topicMsg = subscriber.receive(1000);
```
- Para consumir mensagens de forma *assíncrona* é preciso criar um *MessageListener*

# MessageListener

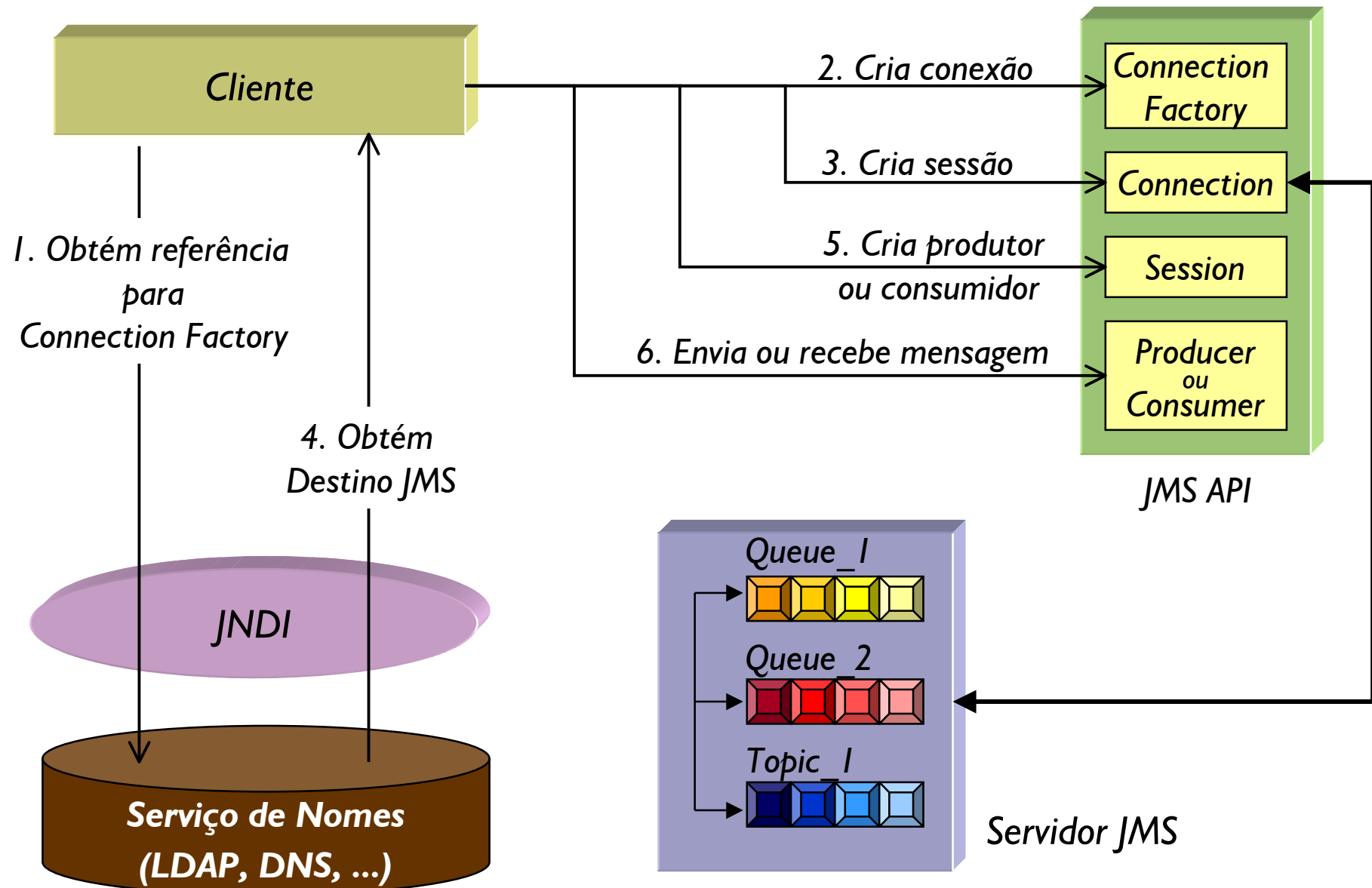
- *Event handler que detecta o recebimento de mensagens*
  - *Para usar, implemente **MessageListener** e seu método `onMessage()`:*

```
public class MyListener implements MessageListener {
    public void onMessage(Message msg) {
        TextMessage txtMsg = (TextMessage) msg;
        System.out.println( "Mensagem recebida: " +
            txtMsg.getText() )
    }
}
```

- *Método **onMessage()** não deve deixar escapar exceções*
  - *Código acima deve estar em um bloco try-catch*
- *Para que objeto seja notificado, é preciso registrá-lo em um `QueueReceiver` ou `TopicSubscriber`*

```
subscriber.setMessageListener( new MyListener() );
topicCon.start(); // iniciar a conexão
```

# Envio e recebimento de mensagens



- Mensagens são compostas de três partes
  - **Propriedades** (opcionais): pares nome/valor (nomes e valores arbitrários definidos pela aplicação); contém tipos primitivos Java (int, boolean, etc.) ou String.
  - **Cabeçalhos**: propriedades com nomes e tipos de valores padrão definidos na especificação JMS
  - **Corpo**: conteúdo que não pode ser representado através de propriedades
- Os **tipos** de mensagem correspondem a **formatos** de dados armazenados no corpo de mensagem
  - Texto, objetos serializados, bytes, valores primitivos, etc.
- Mensagens são criadas a partir de uma Session



# Cabeçalhos e propriedades

- Cabeçalhos: conjunto de propriedades (chave: valor) **definidas na especificação JMS** e usadas pelo sistema para identificar e rotear mensagens
  - Chaves começam com "JMS". Ex: JMSMessageID, JMSTDestination, JMSExpiration, JMSPriority, JMSType
  - A maioria são criadas automaticamente durante a chamada do método de envio (dependem dos parâmetros do método)
- Propriedades **definidas pelo programador**
  - Pode guardar informações de conteúdo textual em mensagens simples sem corpo (onde a mensagem consiste das propriedades)
  - Usadas para qualificar a mensagem e permitir sua filtragem
  - Úteis para preservar a compatibilidade com outros sistemas de messaging

```
message.setStringProperty("Formato", "Imagem JPEG");
```

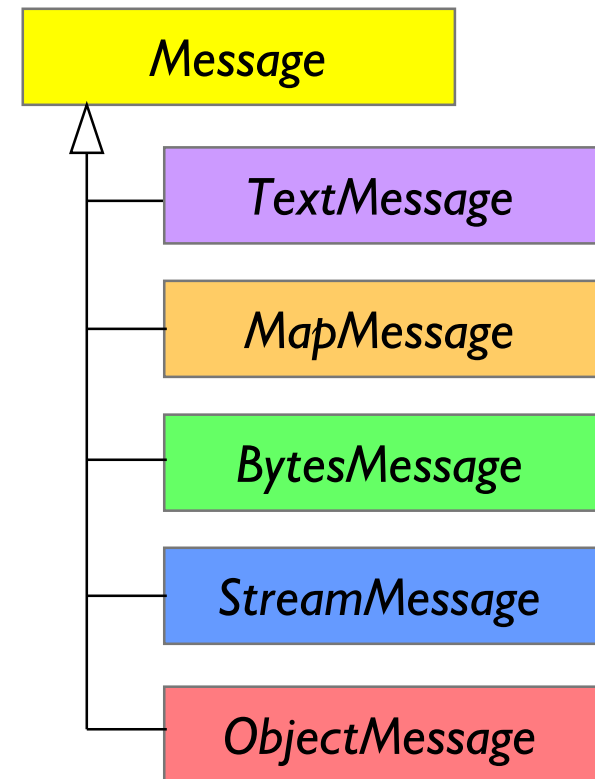
# Filtros (seletores) de mensagens

- *Consumidor pode especificar quais as mensagens que lhe interessam através de expressões SQL*
  - *Expressão deve retornar valor booleano*
  - *Consiste de String passado como argumento de métodos `createReceiver()` e `createSubscriber()`*
  - *Expressão é tipicamente usada para comparar valores de propriedades de mensagens*
  - *Consumidor só consome as mensagens em que propriedades, aplicadas à expressão resultem verdadeiro*
- *Exemplos*

```
String seletor = "Formato LIKE '%Imagem%' AND " +  
                "JMSEExpiration > 0 AND " +  
                "Valor IS NOT NULL";  
topicSession.createSubscriber (canal, seletor)
```

# Seis tipos de mensagens

- **Message**
  - Mensagem genérica sem corpo (contendo apenas cabeçalho e possíveis propriedades)
- **TextMessage**
  - Objeto do tipo *String* (ex: conteúdo XML)
- **MapMessage**
  - Conjunto de pares nome/valor onde nomes são *Strings* e valores são tipos primitivos
- **BytesMessage**
  - Stream de bytes não interpretados
- **StreamMessage**
  - Seqüência de tipos primitivos Java
- **ObjectMessage**
  - Objeto Java serializado



# Criação de mensagens

- Para cada tipo de mensagem, Session fornece método create()
  - createMessage(), createTextMessage(), createBytesMessage(), createObjectMessage(), createMapMessage(), createStreamMessage()

```
TextMessage message =  
    queueSession.createTextMessage();  
message.setText(msg_text); // msg_text é String  
sender.send(message);
```

- Após receber uma mensagem, via receive() ou onMessage(), é preciso fazer o cast para ter acesso aos métodos específicos de cada tipo de mensagem

```
Message m = receiver.receive();  
if (m instanceof TextMessage) {  
    TextMessage message = (TextMessage) m;  
    System.out.println("Recebido: " + message.getText());  
}
```

- *A maior parte das operações JMS potencialmente causam **javax.jms.JMSException***
- *Há várias sub-classes de JMSException para tratar situações específicas*
  - *Veja documentação da API ou JMS Tutorial (3.7) para mais detalhes*

# Exemplos de aplicações

- Os exemplos são do capítulo 4 do JMS Tutorial (Sun)
  - As versões em `cap07/exemplos/` foram alteradas
  - Todas estão nos pacotes `jmstut.topic` ou `jmstut.queue`
- Há alvos prontos no Ant. Rode-os em duas ou três janelas e mude a ordem (rode o produtor antes e depois inverta)
- Aplicação ponto-a-ponto (rode em janelas diferentes)
  - > `ant runqueue sender`
  - > `ant runqueue receiver`
- Aplicação pub/sub
  - > `ant runtopic publisher`
  - > `ant runtopic subscriber`

# O que é um Message-Driven Bean (MDB)?

- Consumidor assíncrono
- Bean ativado por mensagens (eventos)
- MDBs **consomem** mensagens enviadas a filas e canais JMS
  - Não é possível enviar uma mensagem diretamente a um message-driven bean (envia-se mensagens a um canal que o bean escuta)
  - Acoplamento entre clientes e MDB resume-se à conhecimento de canal comum de comunicações
- Um message-driven bean é um EJB mas
  - **Não possui interfaces** Home ou Remote
  - Possui apenas **um método** que recebe qualquer tipo de mensagem
  - **Não devolve tipos ou exceções ao cliente**
  - Não participa de contexto transacional iniciado no cliente
  - Não possui estado (é **stateless**)
  - Pode ser ouvinte de uma **fila** ou assinante durável ou não-durável de um **canal** (Topic) JMS

# Desenvolvendo MDB

- *Message-driven beans implementam duas interfaces*

```
public interface javax.jms.MessageListener
```

```
public interface javax.ejb.MessageDrivenBean
```

```
extends javax.ejb.EnterpriseBean
```

- *Métodos de MessageDrivenBean e da especificação*

- **ejbRemove()**: método da interface. Chamado pelo container quando o message-driven bean está sendo removido do pool

- **ejbCreate()**: método definido na especificação (não está na interface). É chamado quando um novo bean é criado pelo container

- **setMessageDrivenContext(MessageDrivenContext ctx)**: método da interface. Chamado antes do `ejbCreate()` quando o bean está sendo adicionado ao pool

- *Método de MessageListener*

- **onMessage(Message m)**: chamado cada vez que uma mensagem é enviada para o canal do bean (se o bean estiver ativado).



# Exemplo: implementação de um MDB

```
import javax.ejb.*;
import javax.jms.*;
```

```
public class LogBean implements MessageDrivenBean, MessageListener {
```

```
    protected MessageDrivenContext ctx;
```

```
    public void setMessageDrivenContext(MessageDrivenContext ctx) {
        this.ctx = ctx;
    }
```

```
    public void ejbCreate() {}
```

```
    public void ejbRemove() {}
```

Métodos da interface  
MessageDrivenBean ou do  
contrato com a especificação

```
    public void onMessage(Message msg) {
        TextMessage tm = (TextMessage) msg;
        try {
            String text = tm.getText();
            System.err.println("Received new message : " + text);
        } catch (JMSEException e) {
            throw new EJBException();
        }
    }
}
```

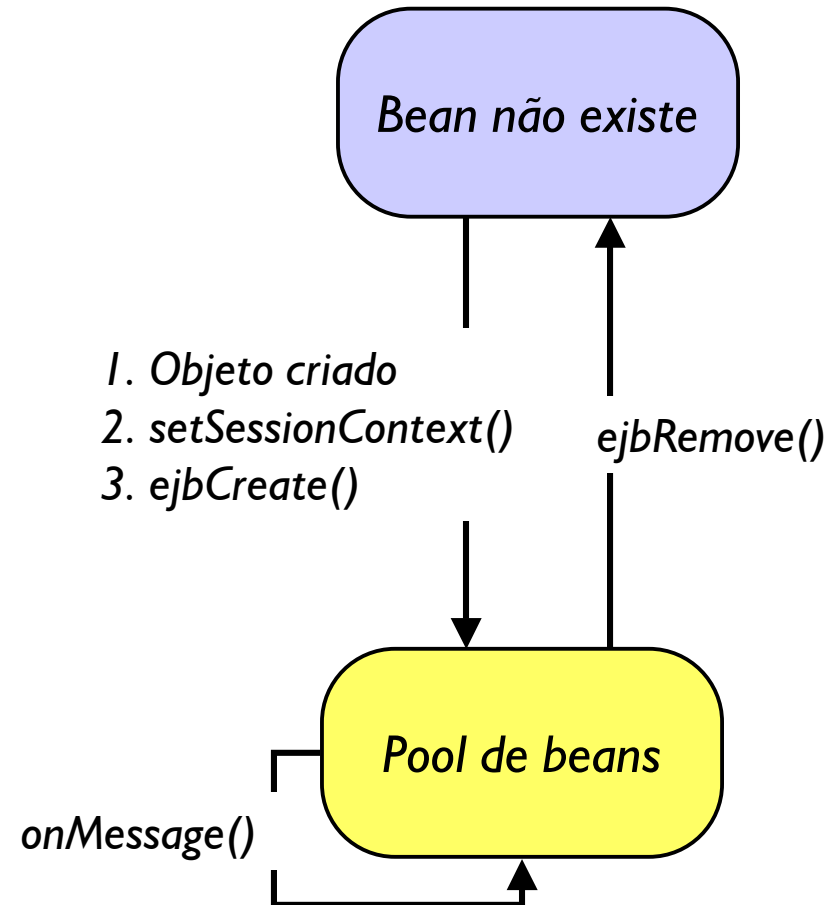
Método da interface  
MessageListener

} RuntimeException para container capturar!  
Só faça isto se houver chance de corrigir erro: falta  
de acknowledgement pode causar loop infinito!

Este bean simplesmente  
imprime o texto contido na  
mensagem recebida

- *Ciclo de vida muito simples*
  - *Apenas dois estados*

*Apenas o container chama os métodos de um Message-driven Bean*



# Deployment Descriptor

```
<ejb-jar>
  <enterprise-beans>
    <message-driven>

      <ejb-name>LogEJB</ejb-name>
      <ejb-class>examples.LogBean</ejb-class>
      <transaction-type>Container</transaction-type>

      <message-driven-destination>
        <destination-type>javax.jms.Topic</destination-type>
        <subscription-durability>NonDurable</subscription-durability>
      </message-driven-destination>

      <resource-ref>
        <res-ref-name>jms/TopicFactory</res-ref-name>
        <res-type>javax.jms.TopicConnectionFactory</res-type>
        <res-auth>Container</res-auth>
      </resource-ref>

    </message-driven>
  </enterprise-beans>
</ejb-jar>
```

*Canal (apenas a classe e tipo)*

*Usado apenas em Topics*

*Fábrica de filas e canais*

# Outros sub-tags de <message-driven>

- <message-selector>

- Usado para incluir um filtro SQL para cabeçalhos de mensagens. Apenas mensagens que combinarem com o filtro serão recebidas pelo bean. Exemplo:

```
<message-selector><![CDATA[  
    Formato LIKE '%Imagem%'  
    AND JMSEExpiration > 0  
    AND Valor IS NOT NULL  
]]></message-selector>
```

- <acknowledge-mode>

- Permite definir o modo de acknowledgment. Pode ser *Auto-acknowledge* ou *Dups-ok-acknowledge*. O último permite o recebimento de mensagens duplicadas
- *Client-acknowledge* não é suportado

# Vendor-specific: JBoss

- Os nomes JNDI dos recursos que serão acessados globalmente são definidos na configuração do fabricante.
- No JBoss:

```
<jboss>
  <enterprise-beans>
    <message-driven>
      <ejb-name>LogEJB</ejb-name>
      <destination-jndi-name>
        topic/testTopic
      </destination-jndi-name>
    </message-driven>
  </enterprise-beans>

  <resource-managers>
    <resource-manager>
      <res-name>jms/logtopic</res-name>
      <res-jndi-name>topic/testTopic</res-jndi-name>
    </resource-manager>
  </resource-managers>
</jboss>
```

Nome JNDI global do destino compartilhado pelo bean e pelos clientes

Referência do ENC `java:comp/env`

Mapeamento do nome do ENC para nome JNDI global

# Cliente é um cliente JMS comum

- Pode ser outro bean, cliente Web ou standalone

```
public class Client {  
  
    public static void main(String[] args) throws Exception {  
  
        Context ctx = new InitialContext(System.getProperties());  
        TopicConnectionFactory factory =  
            (TopicConnectionFactory) ctx.lookup("ConnectionFactory");  
        TopicConnection connection =  
            factory.createTopicConnection();  
        TopicSession session =  
            connection.createTopicSession(  
                false, Session.AUTO_ACKNOWLEDGE);  
  
        Topic topic = (Topic) ctx.lookup("topic/testTopic");  
        TopicPublisher publisher = session.createPublisher(topic);  
        TextMessage msg = session.createTextMessage();  
        msg.setText("This is a test message.");  
        publisher.publish(msg);  
  
    }  
}
```

Cliente externo (fora do ENC) usa nomes JNDI globais

# Execução dos exemplos

- *Diretório cap7/exemplos/mejb2/*
- *Para montar e instalar a aplicação, use*
  - > **ant jboss.deploy**
- *Uma vez instalada, rode o cliente:*
  - > **ant run.jboss.client**
- *A execução não deve causar nenhuma reação no cliente, já que foi assíncrona "one-way".*
- *No servidor, aparece a mensagem enviada:*  

```
[STDERR] Received new message : This is a test message.
```
- *Veja também o exemplo do J2EE Tutorial (usando Queue) no subdiretório cap7/exemplos/sun2/*

- *1. Utilizando os exemplos como base, escreva um produtor JMS que envie mensagens para uma fila. As mensagens tem o seguinte formato*
  - Código: 100* (cabeçalho: cod. produto)
  - Quantidade: 5* (cabeçalho: quantidade)
  
  - Descrição do produto* (corpo da mensagem texto)
- *Use a fila do JBoss **queue/A***
- *Varie quantidades entre 1 e 500*
- *2. Escreva um consumidor JMS para ler e consumir as mensagens enviadas sincronamente*
  - *Filtre as mensagens com menos de 10 produtos*



- 3. *Implemente um bean que receba as mensagens enviadas pelo cliente que você desenvolveu nos exercícios 1 e 2 (transforme o consumidor de mensagens em um bean)*
  - *O nome JNDI da fila é queue/A*
  - *O bean deve filtrar as mensagens e receber apenas aquelas cujo cabeçalho **Quantidade** contiver mais que 10 itens e menos que 100*

- [1] Kim Haase. *The JMS Tutorial*. Sun Microsystems, 2002  
<http://java.sun.com/products/jms/tutorial/> *Este capítulo é baseado no JMS Tutorial.*
- [2] Sun Microsystems. *The Java Message Service Specification*  
<http://www.javasoft.com/products/jms/docs.html>
- [3] Peter Antman. *JBoss and JMS*. JBoss User's Manual, Chapter 8.  
<http://www.jboss.org/online-manual/HTML/ch08.html> *Contém breve tutorial sobre JMS*
- [4] Todd SundSted. *Messaging helps move Java into the Enterprise*. JavaWorld, Jan 1999.  
<http://www.javaworld.com/jw-01-1999/jw-01-jms.html>  
*Introdução aos conceitos fundamentais de sistemas de mensagens.*
- [5] Todd SundSted. *Messaging makes its move, Parts I & II*. JavaWorld, Fev e Mar 1999.  
[http://www.javaworld.com/javaworld/jw-02-1999/jw-02-howto\\_p.html](http://www.javaworld.com/javaworld/jw-02-1999/jw-02-howto_p.html)  
*Série de dois artigos mostrando como usar JMS.*
- [6] Gordon Van Huizen. *JMS: An infrastructure for XML-based B2B communication*  
<http://www.javaworld.com/javaworld/jw-02-2000/jw-02-jmsxml.html> JavaWorld, 2000.  
*Introdução a JMS motivada pelo seu potencial para soluções B2B com XML*
- [7] Michael Schoffner. *Write your own MOM*. JavaWorld, 1998.  
<http://www.javaworld.com/javaworld/jw-05-1998/jw-05-step.html>
- [8] Ed Roman et al. *Mastering EJB 2, Chapter 8: Message Drive Beans*  
<http://www.theserverside.com/books/masteringEJB/index.jsp>  
*Contém um breve e objetivo tutorial sobre JMS no início do capítulo*

*helder@argonavis.com.br*

***argonavis.com.br***

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

*Minicurso de JMS, 2002*