

# ACADEMIA

Accelerating the world's research.

# Infraestrutura para sistemas Java EE

Helder da Rocha

## Related papers

[Download a PDF Pack](#) of the best related papers ↗



[SISTO ENAM EIN TREINAMENTOS T R EIN AM EN TO TR S](#)

Geyson Breno

[Engenharia Informática e de Computadores](#)

João Leitão

[Desenvolvimento-distribuído](#)

Ivoney Ferreira



# Infraestrutura de sistemas Java EE

Java Message Service  
Enterprise JavaBeans

Web Services com JAX-WS e JAX-RS  
Clientes Ajax com JavaScript, JQuery, HTML e CSS

# Sobre este tutorial

- Este material foi originalmente escrito por Helder da Rocha em agosto de 2013 (para um treinamento contratado sob demanda), e poderá ser usado de acordo com os termos da licença **Creative Commons BY-SA (Attribution-ShareAlike)** descrita em  
<http://creativecommons.org/licenses/by-sa/3.0/br/legalcode>
- Parte do código usado em exemplos e exercícios pode ser baixado dos seguintes repositórios
  - <https://github.com/helderdarocha/cursojaxrs>
  - <https://github.com/helderdarocha/cursojaxws>
  - <https://github.com/helderdarocha/ExercicioMinicursoJMS>
  - <https://github.com/helderdarocha/JavaXMLExamples>
- Para treinamento presencial, particular ou online entre em contato:
  - Helder da Rocha ([www.agonavis.com.br](http://www.agonavis.com.br))  
[helder.darocha@gmail.com](mailto:helder.darocha@gmail.com)

WebServices em Java EE 7

(31-08-2013)

Java Message Service 2.0

(31-08-2013)

Integração de sistemas com Java EE

(05-12-2013)

Aplicações Web dinâmicas

(18-03-2014)



# Conteúdo resumido

- 1. Enterprise JavaBeans e Java Message Service:**  
Arquiteturas de sistemas distribuídos,  
Serviços síncronos e assíncronos  
Java RMI e Java Messaging; EJB e MDB
- 2. Tecnologias para representação de dados:**  
**XML**, XML Schema, XPath, **JSON**, DOM, SAX, JAXP e **JAXB**
- 3. SOAP Web Services:**  
Aplicações em Java usando **JAX-WS** e Java EE 7
- 4. REST Web Services:**  
Aplicações em Java usando **JAX-RS** e Java EE 7
- 5. Clientes** Ajax para serviços Java  
Infraestrutura: HTML5, CSS e JavaScript  
Clientes dinâmicos para Web Services usando Ajax e JQuery



# Conteúdo detalhado (1)

## (1) Enterprise JavaBeans e Java Message Service

1. Métodos de comunicação síncrono e assíncrono
  - Arquiteturas
  - Tecnologias
2. Session Beans com clientes locais e remotos
3. Messaging e Java Message Service (JMS)
  - Configuração de ambiente
  - JMS API: conexões, sessões, produtores, consumidores, mensagens
  - Criação de produtores e consumidores JMS
4. Message Driven Beans
5. JMS 2.0 (Java EE 7)



# Conteúdo detalhado

## (2) Representação de dados: XML e JSON

1. Formatos de dados usados em Web Services
  - XML
  - JSON
2. XML
  - Tecnologias XML: DTD, XML Schema, XPath
  - APIs de manipulação do XML - JAXP: DOM, SAX, StAX e TrAX
  - XML Java Binding: **JAXB**
3. JSON
  - JSON
  - JSON Java APIs
  - JSON Java Binding



# Conteúdo detalhado

## (3) SOAP Web Services

1. Introdução a SOAP Web Services
  - Fundamentos da tecnologia
  - Protocolos e mensagens: SOAP, WSDL
2. Como criar um serviço SOAP com EJB e JAX-WS
  - Como componente Web
  - Como session bean
3. Como criar um cliente SOAP usando JAX-WS
  - Tipos de clientes
  - Cliente estático
  - Clientes dinâmicos
4. Tópicos avançados
  - Handlers e MessageContext
  - WS-Addressing
  - WS-Security
  - SOAP sobre JMS



# Conteúdo detalhado

## (4) REST Web Services

1. Introdução a REST
  - Arquitetura REST
  - Fundamentos de WebServices usando REST
2. Como criar um serviço RESTful usando JAX-RS
  - MediaTypes, resources, parâmetros
  - Anotações @Path, @PathParam, etc.
  - Metadados, ResponseBuilder e UriBuilders
  - Estratégias de segurança e contextos
3. Como criar um cliente RESTful
  - Usando comunicação HTTP simples
  - Usando APIs RESTEasy e Jersey
  - Clientes RESTful em aplicativos Web e mobile
  - WADL e JAX-RS 2.0



# Conteúdo detalhado

## (5) Clientes Web

1. Introdução a interfaces Web estáticas
  - **HTML e CSS** essencial
  - Formulários do HTML
  - Document Object Model
  - **JavaScript** essencial
2. Interfaces Web dinâmicas
  - Manipulação da árvore DOM com JavaScript
  - Alteração de classes CSS
  - Recursos essenciais do HTML5 e CSS3
  - Ajax e comunicação com Web Services
3. **JQuery** essencial
  - Seletores e acesso a dados em HTML, XML e JSON
  - Ajax usando JQuery
  - Formulários dinâmicos
  - Boas práticas e técnicas essenciais para aplicações assíncronas



# Agenda de treinamento (40h)

	Dia 1	Dia 2	Dia 3	Dia 4	Dia 5 (opcional)
8h	<b>1</b> EJB JMS	<b>2</b> XML JSON	<b>3</b> SOAP Web Services JAX-WS	<b>4</b> REST Web Services JAX-RS	<b>5</b> JavaScript JQuery HTML/CSS

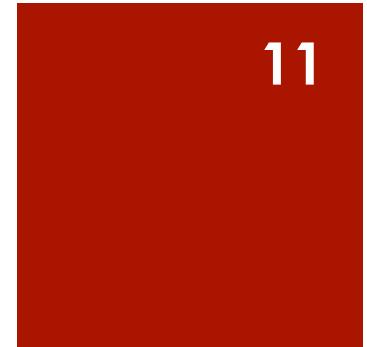


# Estrutura das aulas

- Apresentação e discussão
- Demonstrações e exemplos
- Exercícios práticos em sala
- Referências adicionais
  - Especificações
  - Tutoriais
  - Artigos
  - Documentação de produtos
  - Fontes para pesquisa, discussão e aprofundamento
  - Ferramentas



# Instrutor



- Helder da Rocha (Argo Navis)
  - Mestre em informática (UFCG, 1999)
  - Autor de cursos e apostilas sobre Java, Java EE, Arquitetura da JVM, Design Patterns, Ant & Maven, HTML5 e CSS, SVG, JavaScript, Perl, iOS 3, 4 e 5, Objective-C. Autor de livros de HTML, CSS e JavaScript (1996, IBPI Press)
  - Autor de tutoriais do site [argonavis.com.br](http://argonavis.com.br)
  - Também ator, músico, artista visual e paleoartista
  - [www.argonavis.com.br](http://www.argonavis.com.br) (apenas TI)
  - [www.helderdarocha.com.br](http://www.helderdarocha.com.br) (tudo menos TI)
  - [helder@argonavis.com.br](mailto:helder@argonavis.com.br)



1

# Infraestrutura de sistemas Java EE

## EJB e JMS

Arquitetura de sistemas distribuídos, serviços síncronos e assíncronos, Java RMI e Java Messaging, EJB e MDB



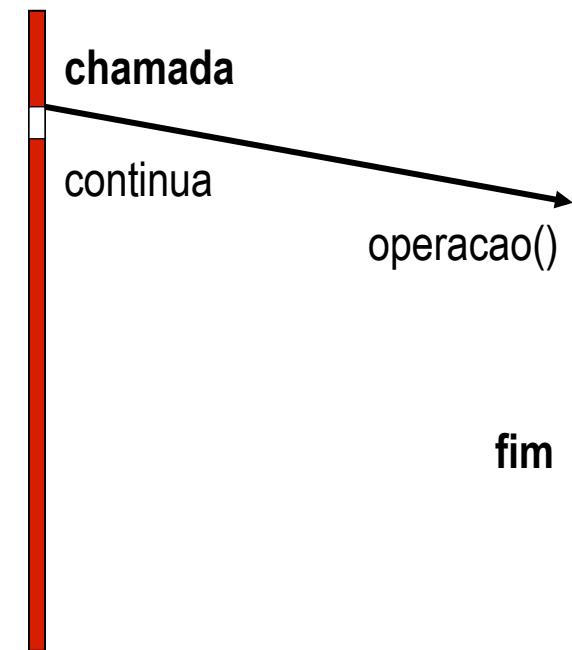
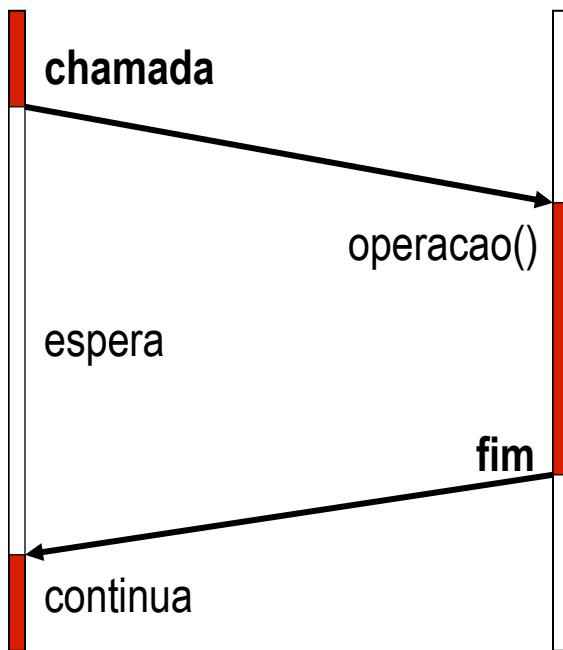
# Conteúdo

1. Métodos de comunicação síncrono e assíncrono
  - Arquiteturas
  - Tecnologias
2. Session Beans com clientes locais e remotos
3. Messaging e Java Message Service (JMS)
  - Configuração de ambiente
  - JMS API: conexões, sessões, produtores, consumidores, mensagens
  - Criação de produtores e consumidores JMS
4. Message Driven Beans
5. JMS 2.0 (Java EE 7)



# Métodos de comunicação

- **Síncrono:** cliente chama o serviço e **espera** pelo fim da execução da operação
  - RPC
- **Assíncrono:** cliente chama o serviço e **não espera** a operação terminar
  - Messaging



# Comunicação entre objetos

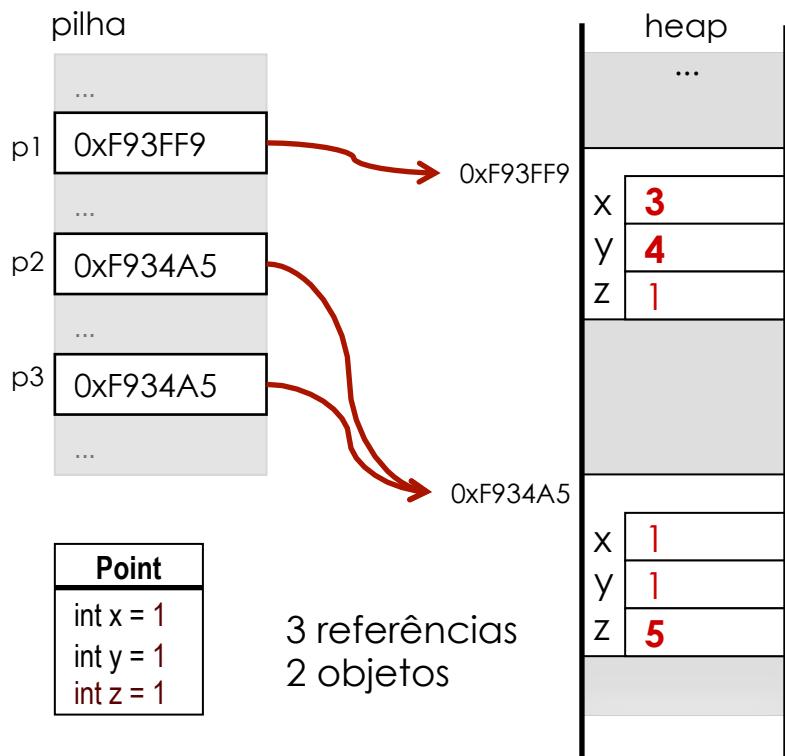
- Local (mesma JVM)
  - **Síncrono**, passagem do objeto por **referência** (comportamento default para objetos)
  - **Síncrono**, passagem do objeto por **valor** (cópia do estado de objetos via `clone()` ou serialização)
  - **Assíncrono** (eventos, listeners, observers)
- Remota (JVMs diferentes)
  - **Síncrono**, via proxy (passagem do objeto por **referência**) – tecnologias RMI/JRMP, RMI/IOP, SOAP
  - **Síncrono**, via objeto serializado (passagem do objeto por **valor**) – tecnologia Java (`java.io.Serialization`), XML (JAXB, Web Objects), JavaScript (JSON)
  - **Assíncrono**, via mediador (JMS, SOAP Messaging)



# Acesso local síncrono

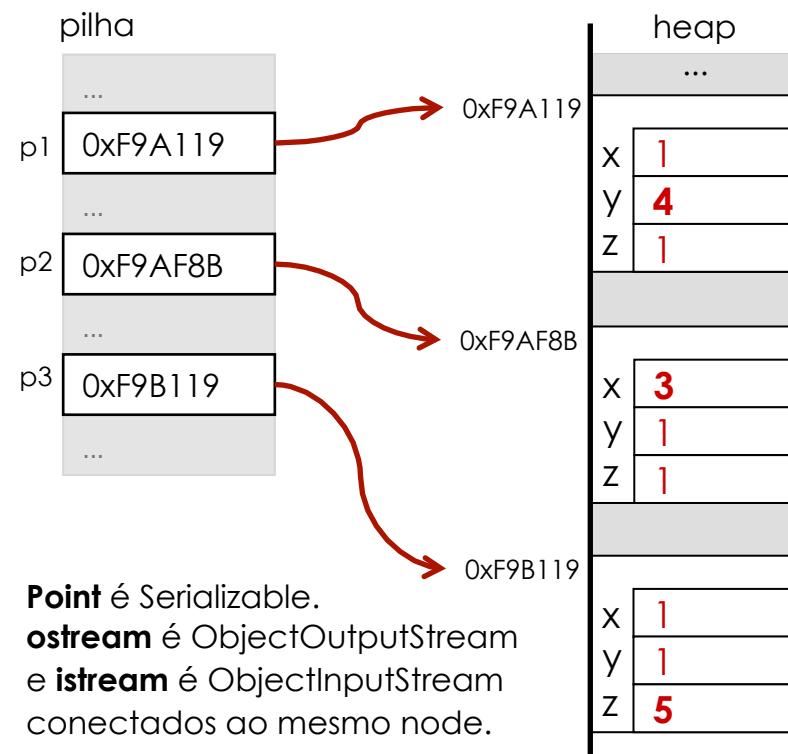
- Referência de memória

```
Point p1, p2, p3;
p1 = new Point();
p2 = new Point();
p3 = p2;
p2.x = 3; p1.y = 4; p3.z = 5;
```



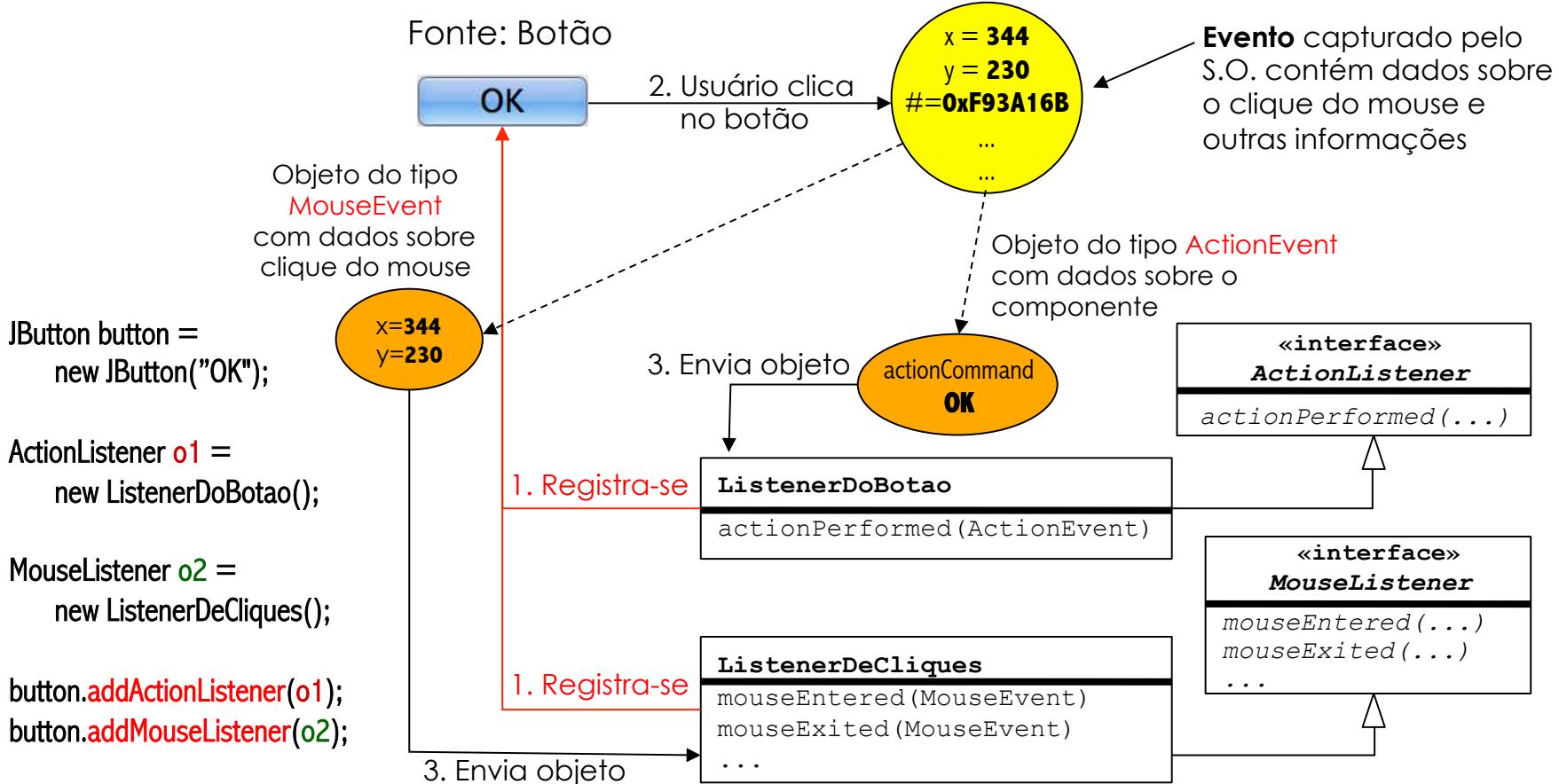
- Value objects

```
Point c1, c2, c3;
c1 = new Point();
c2 = c1.clone();
ostream.writeObject(c2);
c3 = (Point)istream.readObject();
p2.x = 3; p1.y = 4; p3.z = 5;
```



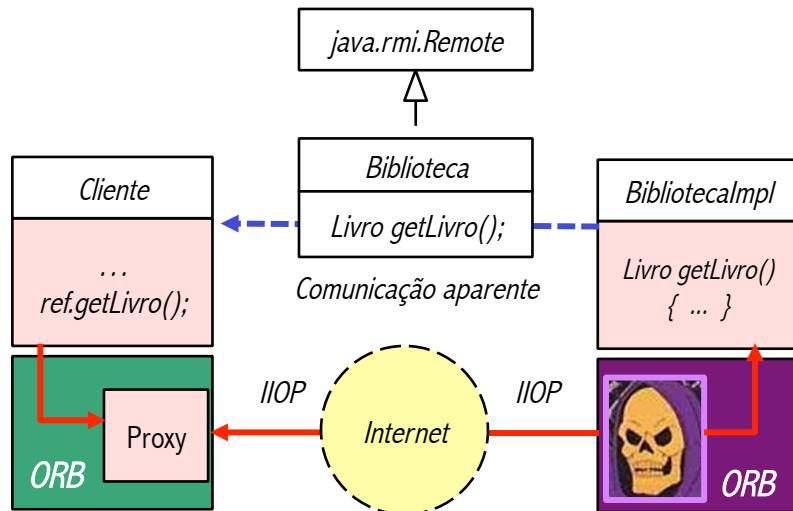
# Acesso local assíncrono

- **Notificações** usando referências locais
- Padrão **Observer**
- Exemplo: listeners de eventos do sistema operacional

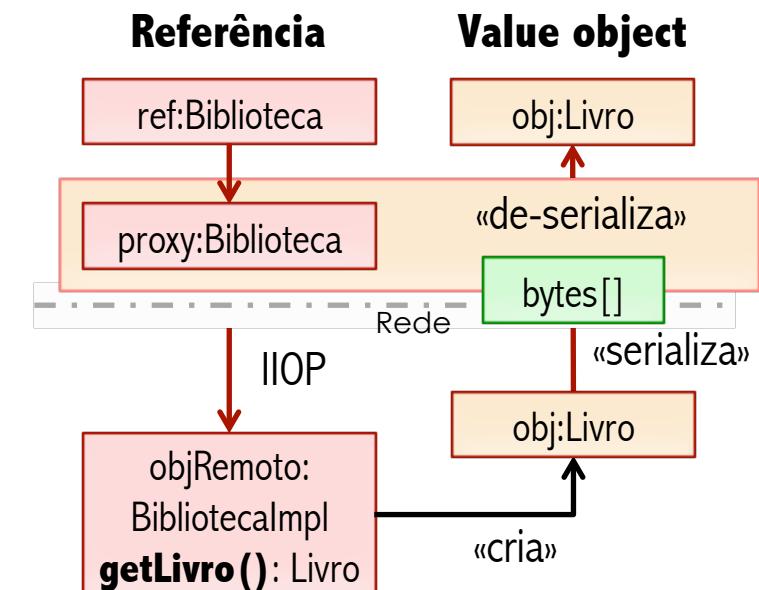


# Acesso remoto síncrono (RMI/IIOP)

- Por referência: **proxy RMI/IIOP** (`java.rmi.Remote`)
  - Geralmente encapsulam comportamento
  - Acessados via referência de rede (proxy, stub)
- Por valor: **value objects** (não implementam `Remote`)
  - Geralmente objetos que representam estado
  - Podem ser transferidos por valor se `Serializable`



```
Biblioteca ref = (Biblioteca) jndi.lookup("proxy");
Livro obj = ref.getLivro(); // chamada de método remoto
```



# Web Services: arquiteturas

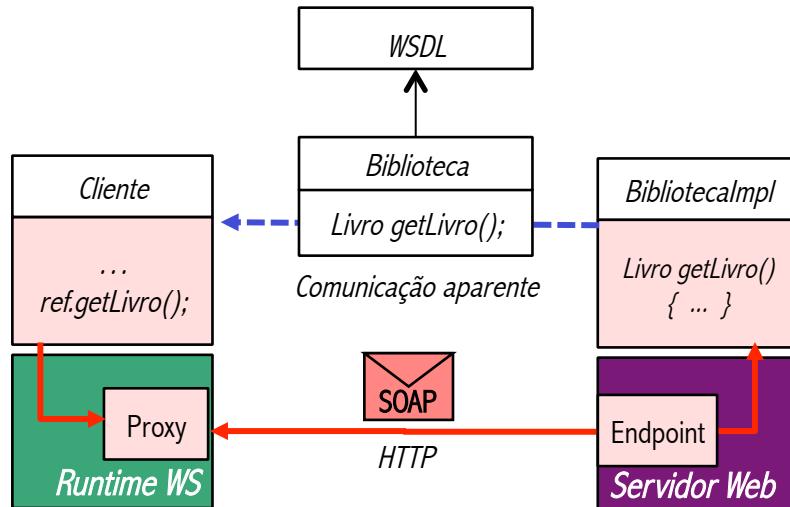
- Web Services\*: ambiente de computação distribuída que usa o protocolo HTTP como camada de transporte e texto (XML, JSON) como formato de dados
  - Solução de integração independente de linguagem (cliente e servidor podem ser implementados em linguagens diferentes)
- Duas arquiteturas: SOAP e REST
- **SOAP Web Services**
  - Solução de objetos distribuídos (RMI, similar a CORBA, DCOM, DCE) que usa XML (SOAP) como protocolo de comunicação e como IDL (WSDL) para descrição de interface de serviços
  - Também usada para messaging assíncrono
  - Geralmente implementada sobre HTTP
- **RESTful Web Services**
  - Arquitetura de Web Services que aproveita a infraestrutura da Web como protocolo HTTP, tipos MIME e URIs (sem acrescentar o overhead do SOAP)

\* Em 2013 (definição de 'web services' tem mudado desde que foi criado)

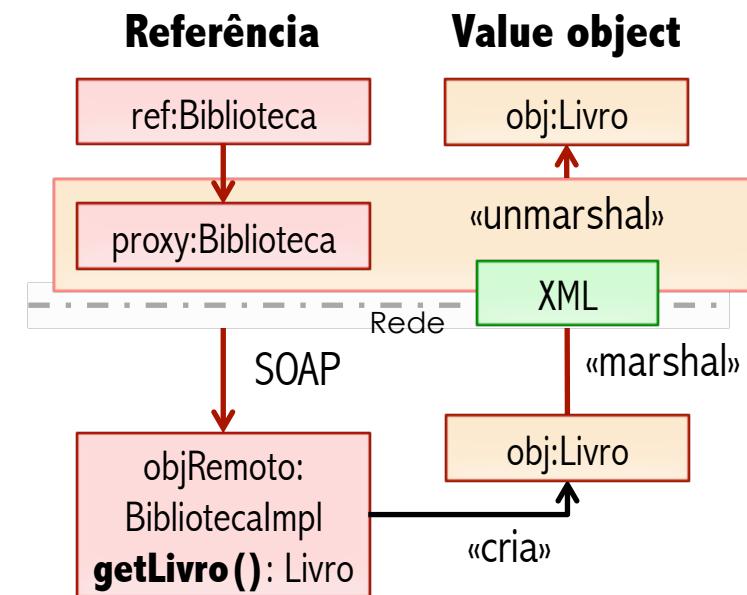


# Acesso remoto síncrono (SOAP)

- Por referência: **proxy JAX-WS** (Web Service SOAP)
  - Gerados a partir de WSDL
  - Runtime JAX-WS sincroniza mensagens SOAP de requisição e resposta
- Por valor: **value objects** (XML mapeado a objetos)
  - Declarados em WSDL usando XML Schema
  - Serializados (marshaled) com JAXB e anexos na msg SOAP



Biblioteca **ref** = (Biblioteca) stub.getProxy(wsdl);  
 Livro obj = **ref**.getLivro(); // chamada de método remoto

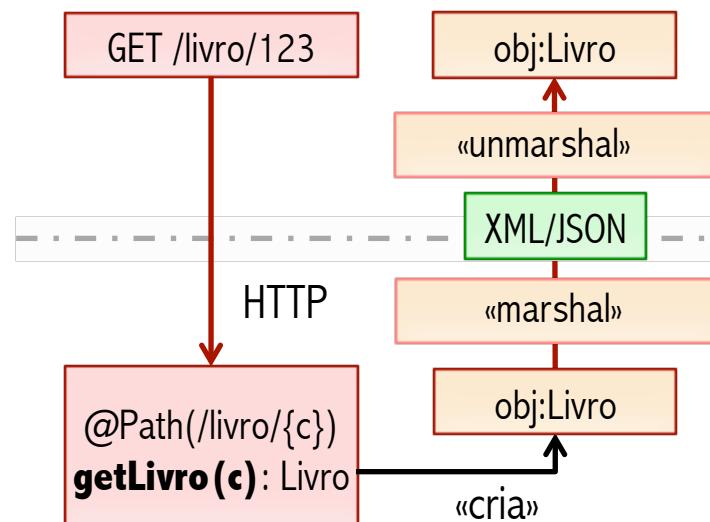
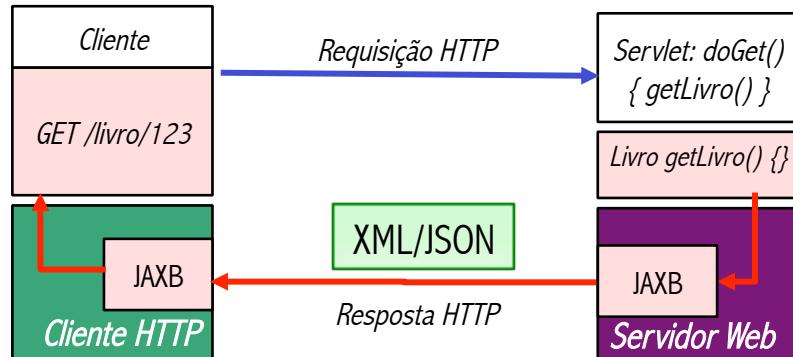


JAXB



# Acesso remoto síncrono (REST)

- Interface remota é formada por combinação **método HTTP + URI**
  - Método Java remoto mapeado à URI e método HTTP responde a requisição
  - Cliente Java local precisa construir requisição HTTP e depois decodificar dados da resposta HTTP (ex: com JAXB)
- Pode haver **value objects** (XML ou JSON mapeado a objetos)
  - Serializados (marshaled) com JAXB em formato JSON ou XML e anexados em requisições e respostas



```

HttpURLConnection con = new URL("http://host/livro/123").openConnection();
BufferedReader r = new BufferedReader(new InputStreamReader(con.getInputStream()));
String json = rd.readLine(); // objeto recebido como JSON string precisa ser decodificado
  
```



# Session Beans

- Objeto gerenciado (Enterprise JavaBean)
  - Ciclo de vida gerenciado pelo container (com callbacks configuráveis via anotações)
  - Permite configuração declarativa de serviços (transações, segurança, etc.)
- Pode oferecer serviço síncrono local ou remoto
- Três tipos
  - **Stateless** – propósito geral, sem garantir estado entre chamadas
  - **Stateful** – preserva estado entre chamadas de mesmo cliente
  - **Singleton** – compartilha estado entre clientes, suporta inicialização prévia **@Startup** e cadeias de beans dependentes **@DependsOn**
- Pode ser configurado via anotações e/ou ejb-jar.xml



# Stateless local bean

- Mais simples

- Anotação **@Stateless** e vários comportamentos default (transações, ciclo de vida, etc)
- Se não houver declaração **@Local** ou **@Remote**, todos os métodos automaticamente são incluídos(no-interface)
- Se houver interface **@Local**, apenas métodos declarados na interface serão gerenciados

```
@Stateless
```

```
public class CalculatorBean implements Calculator {  
    public float add (int a, int b) {  
        return a + b;  
    }  
    public float subtract (int a, int b) {  
        return a - b;  
    }  
}
```

```
@Local
```

```
public interface Calculator {  
    public float add (int a, int b);  
    public float subtract (int a, int b);  
}
```



# Exemplo de cliente SB local

- Outros objetos dentro do servidor de aplicações podem referir-se ao serviço do EJB através de JNDI ou injeção de recursos/dependências
  - Nome da classe do bean (tipo) é usado em JNDI e em DI
  - Chamada de métodos é igual ao uso normal de Java standalone
  - Objetos retornados são referências, sejam ou não gerenciados

```
@ManagedBean
public class CalculatorWebBean {
    @EJB
    private Calculator calc;
    private float p1, p2, result;

    public String calculate() {
        result = calc.add(p1, p2);
        return null;
    }
    ...
}
```

Inicialização usando JNDI em construtor  
(em vez de usar de @EJB)

```
...
public CalculatorWebBean () {
    Context jndi =
        new InitialContext();
    calc = (Calculator)
        jndi.lookup("Calculator");
}
...
```



# Stateless remote (IIOP) bean

- Permite acesso remoto (clientes de outras JVMs)
- Exporta OMG IDL (independente de linguagem)
- Como criar
  - Fornecer uma interface anotada com **@Remote** para o session bean

```
@Stateless
```

```
public class CalculatorBean implements Calculator {  
    public float add (int a, int b) {  
        return a + b;  
    }  
    public float subtract (int a, int b) {  
        return a - b;  
    }  
}
```

```
@Remote
```

```
public interface Calculator {  
    public float add (int a, int b);  
    public float subtract (int a, int b);  
}
```



# Cientes remotos (RMI/IOP)

- Precisam obter autorização de acesso e baixar proxy de rede (stub) para realizar a comunicação
  - **Cliente Java standalone** pode obter via **JNDI** (pode ser preciso configurar autorização de acesso externo para o serviço EJB)
  - **Cliente executando em container Java EE** pode obter stub através de injeção de dependências **@Inject** ou resource **@EJB**
  - **Clientes IOP** podem usar o **IDL** do serviço e ferramentas para gerar componentes em outras linguagens (mas isto, hoje, é 100x mais fácil com Web Services)
- Uma vez obtido o **proxy**, métodos remotos podem ser chamados normalmente como se fossem locais
  - Valores retornados podem ser **value objects** (serializados) ou **proxies** para objetos remotos: há diferença se comparado ao acesso local

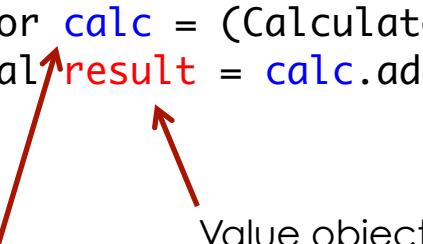


# Exemplo de cliente SB remoto

- Se não estiver em container, precisa ter acesso a remoto a componente cliente JNDI
  - Requer configuração (veja configuração JNDI para clientes JMS)
  - Objetos retornados em métodos são value objects (serializados) a menos que sejam também proxies para objetos remotos (SB)

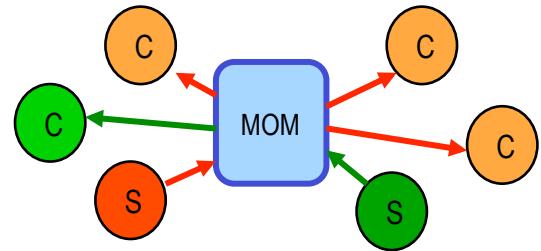
```
public class CalculatorClient {  
  
    public static void main(String[] args) throws Exception {  
        Context jndi = new InitialContext(); // conf. via jndi.properties  
        String nome = "java:/global/mathbeans/Calculator";  
        Calculator calc = (Calculator) jndi.lookup(nome);  
        BigDecimal result = calc.add(3, 4);  
    }  
}
```

Proxy (referência)      Value object



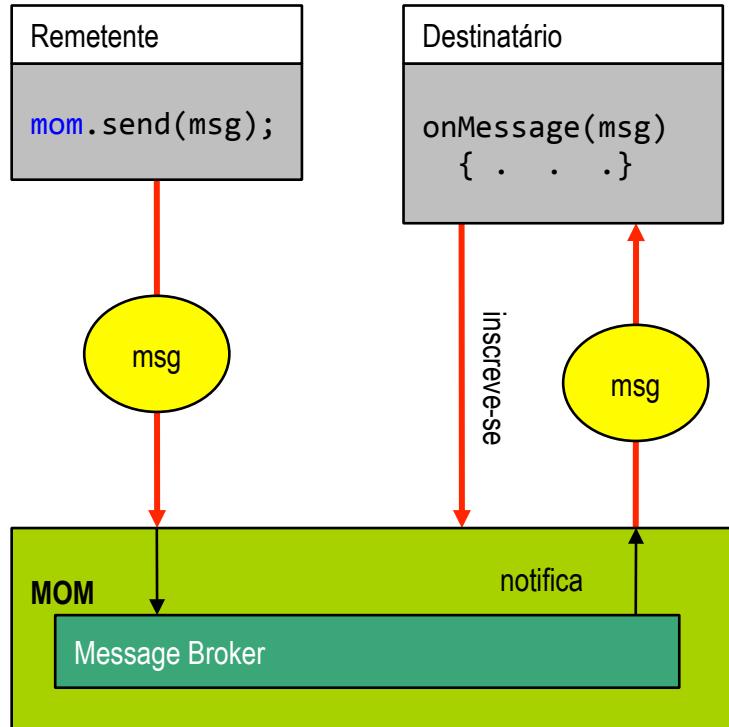
# Messaging e JMS

- Messaging
  - Método de comunicação assíncrono
  - Permite comunicação com baixo acoplamento
- Provedor de serviços de messaging (Message Oriented Middleware – MOM)
  - Aplicação que oferece o serviço de enfileiramento de mensagens
  - Disponibiliza infraestrutura de mediação de mensagens, fábrica para criação de conexões, filas e tópicos para troca de mensagens
  - Oferece serviços adicionais como transações, segurança, QoS, etc.
- Java Message Service
  - **API Java** que consiste de uma coleção de interfaces uniformes para serviços de messaging
  - Permite escrever aplicações que irão rodar em diferentes provedores

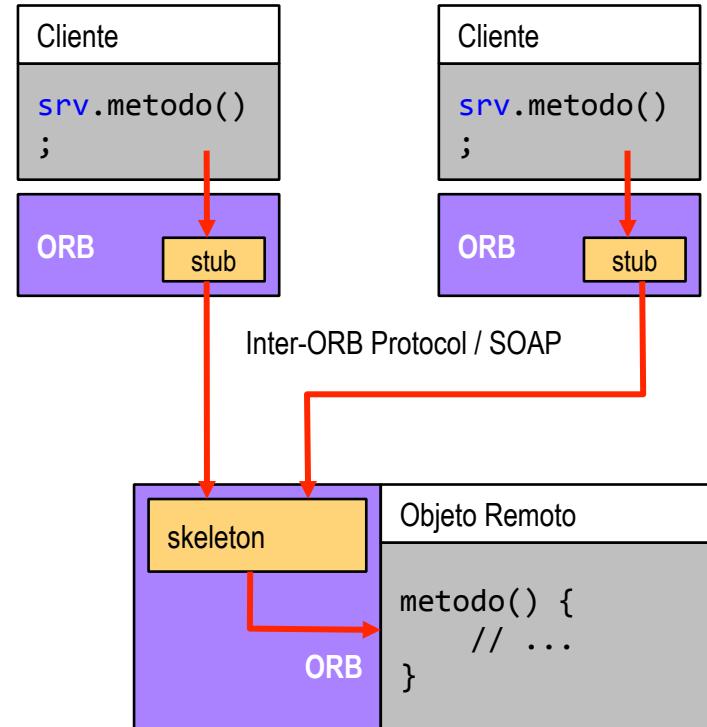


# Messaging vs. RMI/RPC

- Sistema de Messaging



- Sistema RMI/RPC  
(Java RMI, CORBA, WS)



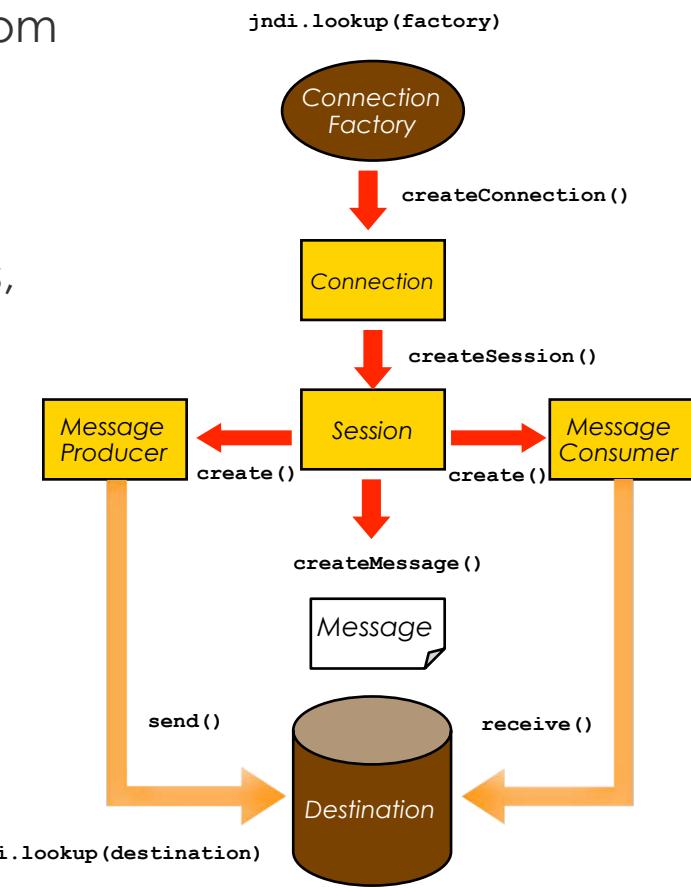
- Protocolo comum:  
mensagem

- Protocolo comum:  
interface dos objetos



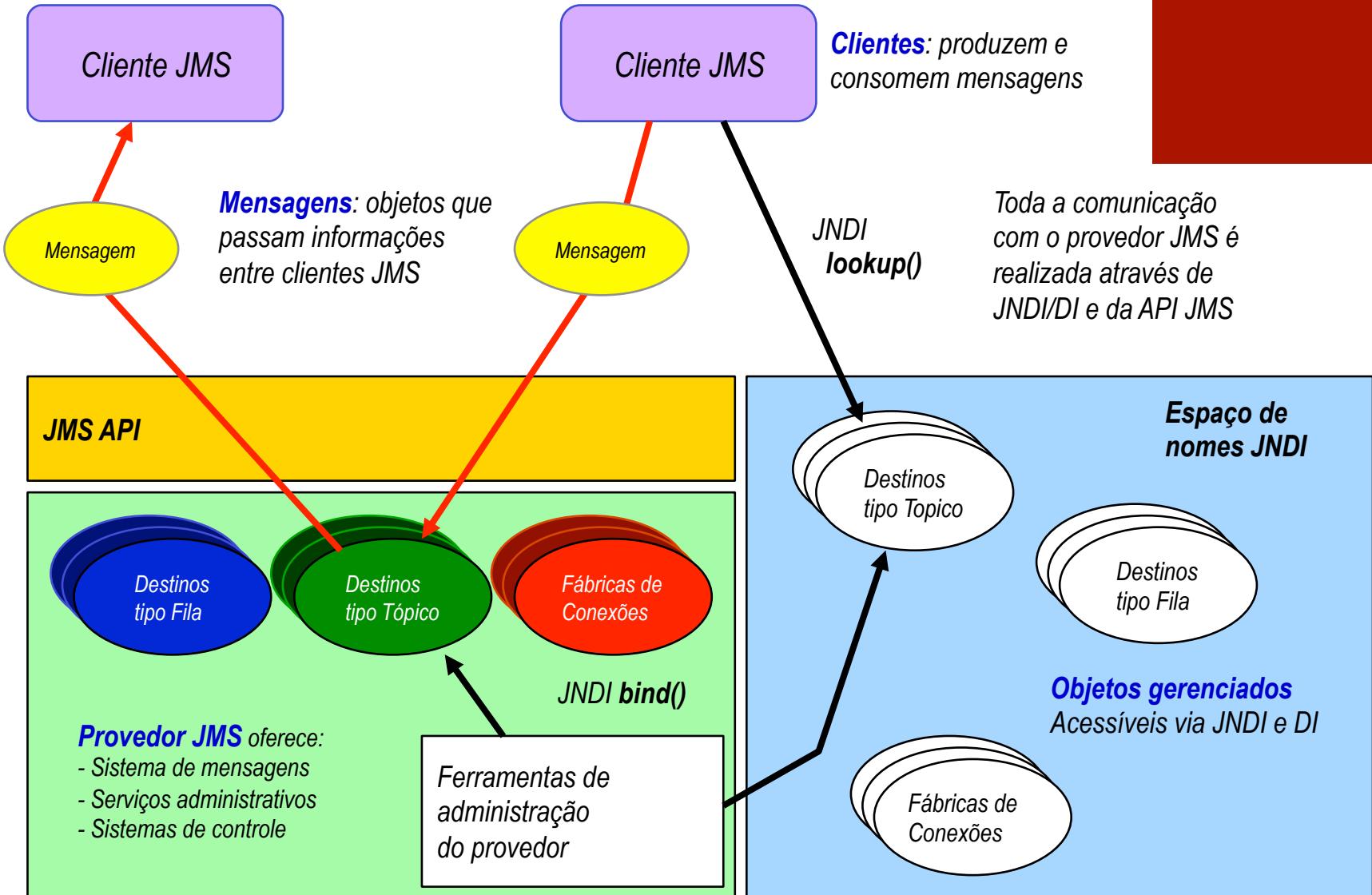
# Java Message Service 1.1

- Serviço JMS age como um **mediador** entre clientes produtores e consumidores
  - Ponto de comunicação (**destino**): **fila** ou **tópico**
  - **Produtores** e **consumidores** comunicam-se com um destino comum através de **mensagens**
  - Permite **recebimento síncrono** (blocking method) ou **assíncrono** (listener de eventos)
  - Permite transferir dados, texto, tipos primitivos, objetos serializados, XML, JSON, etc.
- JMS API
  - **Objetos gerenciados** (obtidos via injeção de recursos ou JNDI): destinos e conexões
  - **Abstract Factory**: interfaces para conexões (multithreaded), sessões (um thread), produtores, consumidores e mensagens



# Arquitetura JMS

31



# Domínios de messaging

- Provedores de messaging distinguem dois estilos, suportados por JMS
  - Ponto a ponto (**PTP**)
  - Publish/subscribe (**pub/sub**)
- Desde JMS 1.1 pode-se usar o mesmo código para criar aplicações que usam qualquer um dos estilos
- Características de PTP
  - Cada mensagem tem apenas **um consumidor**
  - O consumidor pode consumir a mensagem mesmo que não esteja rodando quando ela foi enviada pelo produtor
- Características de pub/sub
  - Consumidores têm **assinaturas** para um tópico
  - Cada mensagem é **enviada para todos os assinantes**
  - Clientes podem consumir mensagens depois que a assinatura foi iniciada, e ele precisa estar **ativo** quando a mensagem for iniciada para recebê-la (a menos que ele tenha uma assinatura **durável**)

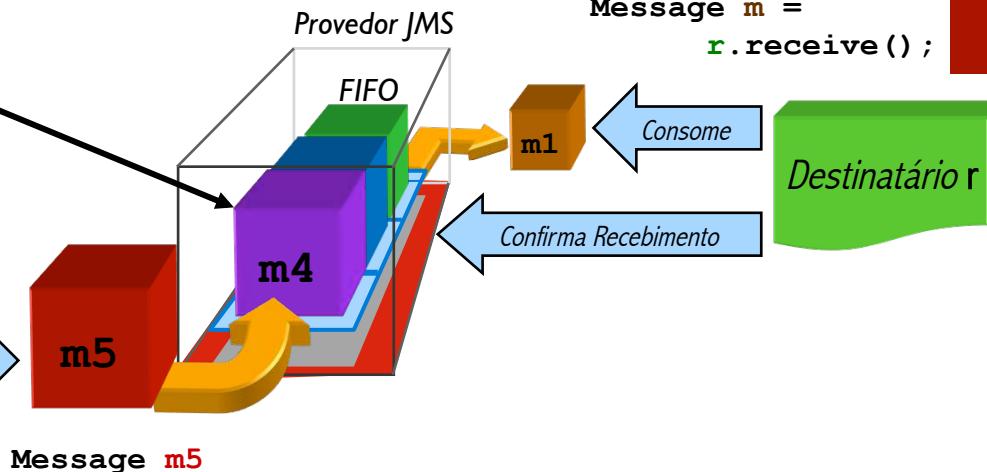


# Arquitetura JMS: domínios

- Estilo **ponto-a-ponto**: filas, remetentes, destinatários

## Fila (queue)

```
MessageProducer s;
s.send(m1);
...
s.send(m5);
```



```
MessageConsumer r;
Message m =
r.receive();
```

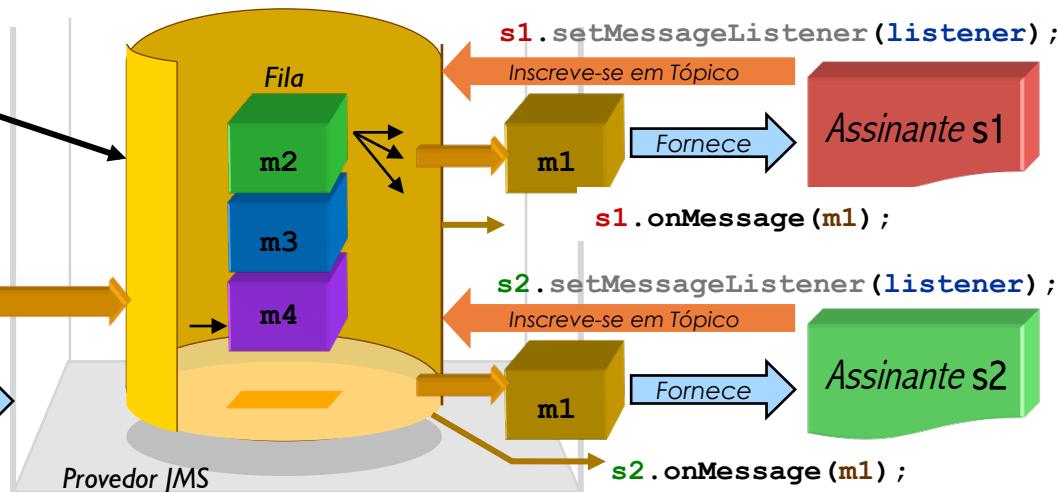
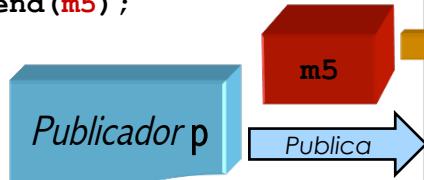
Consumo síncrono através de blocking method **receive()**



Estratégias de **consumo** síncrono ou assíncrono podem ser usados nos dois estilos

## Tópico (topic)

```
MessageProducer p;
p.send(m1);
...
p.send(m5);
```



```
s1.setMessageListener(listener);
```

Inscreve-se em Tópico

```
m1 Fornece
```

s1.onMessage(m1);

```
s2.setMessageListener(listener);
```

Inscreve-se em Tópico

```
m1 Fornece
```

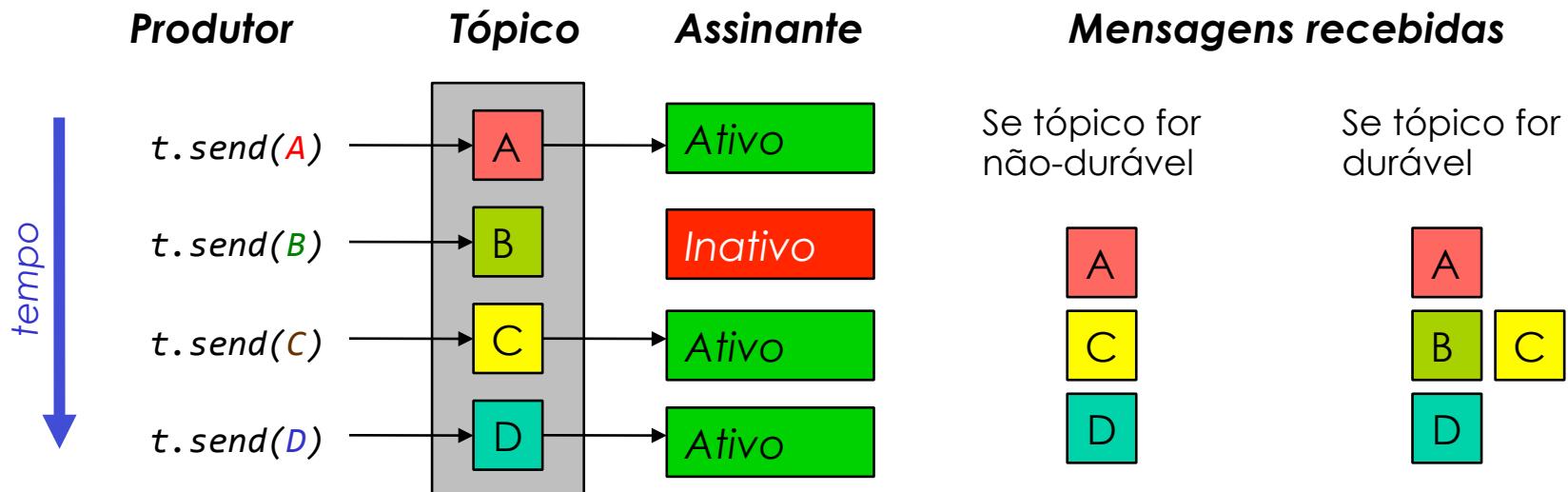
s2.onMessage(m1);

Consumo assíncrono via implementação de **MessageListener** e event handler method **onMessage()**



# Tópicos duráveis e não-duráveis

- Cada mensagem enviada a um tópico pode ter múltiplos consumidores
  - **Depois que assinatura é iniciada**, consumidor pode receber mensagens
  - Em tópicos **duráveis** mensagem permanece disponível até que todos os assinantes a tenham retirado
  - Em tópicos **não-duráveis** assinante perde as mensagens enviadas nos seus períodos de inatividade



# Configuração

- Clientes JMS **internos** a um servidor Java EE têm várias alternativas para obter acesso a fábricas de conexões e destinos
  - Via lookup **JNDI**
  - Via injeção de recursos **@Resource**
  - Via injeção de dependências (CDI) **@Inject**
- Para clientes JMS **externos**, servidores de aplicação podem importar diversas restrições (autenticação, autorização, nomes JNDI distintos, etc.) que precisam ser configuradas
  - Configurar o acesso JMS/EJB remoto no servidor de aplicação; cada servidor requer JARs e propriedades JNDI próprias
  - Para aplicações JMS que não necessitam dos recursos de um servidor de aplicações, pode-se usar servidores JMS standalone como o **ActiveMQ** ou o **HornetQ**



# Configuração para clientes

- Aplicações-cliente **externas** precisam **configurar cliente JNDI** para ter acesso a recursos gerenciados (conexões, destinos)
  - Incluir classes do cliente JNDI no classpath da aplicação
  - Configurar **InitialContext** do JNDI com **propriedades**
  - Os detalhes da configuração dependem do servidor usado
- Exemplo de configuração do contexto JNDI via **jndi.properties**
  - Conteúdo de arquivo **jndi.properties** (acessível pelo classpath do cliente)  
`java.naming.factory.initial=nome-da-classe  
java.naming.provider.url=url  
java.naming.outra.propriedade.requerida.pelo.fabricante=valor`
  - No código Java  
`javax.naming.Context ctx = new InitialContext();`
- Exemplo de configuração do contexto JNDI via objeto **java.util.Properties**

```
Properties props = new Properties();  
props.setProperty(Context.INITIAL_CONTEXT_FACTORY,"nome-da-classe");  
props.setProperty(Context.PROVIDER_URL,"url");  
props.setProperty("outra.propriedade.requerida.pelo.fabricante","valor");  
javax.naming.Context ctx = new InitialContext(props);
```



# Servidores JMS standalone

- **ActiveMQ** 5.x (veja detalhes em <http://activemq.apache.org>)
  - JARs a incluir no classpath (além da API JMS)
    - **activemq-5.x.jar, spring-1.x.jar**
  - Propriedades JNDI:
    - java.naming.factory.initial (Context.INITIAL\_CONTEXT\_FACTORY):  
**org.apache.activemq.jndi.ActiveMQInitialContextFactory**
    - java.naming.provider.url (Context.PROVIDER\_URL) :  
**vm://localhost** ou **tcp://host:port**
- **HornetQ** (veja detalhes em <http://hornetq.sourceforge.net/docs>)
  - JARs para clientes JMS standalone usando JNDI
    - **hornetq-core-client.jar, hornetq-jms-client.jar, jnp-client.jar**
  - Fábricas, queues e topics configurados em **hornetq-jms.xml**
  - Não é necessário configurar JNDI (pode ser feito em **hornetq-beans.xml**). Há uma API cliente para instanciar fábricas e filas

```
ConnectionFactory cf = HornetQJMSClient.createConnectionFactory(...);  
Queue orderQueue = HornetQJMSClient.createQueue("fila");
```



# Configuração JBoss AS 7.1 (HornetQ)

38

- Dependências para cliente externo standalone
  - Inclua no classpath o arquivo **jboss-client.jar** (localizado em **\$JBOSS\_HOME/bin/client** (não use em projetos Maven - leia README.txt))
- Configuração do JNDI via **java.util.Properties**

Properties **props** = new Properties();

**props**.put(Context.INITIAL\_CONTEXT\_FACTORY, org.jboss.naming.remote.client.InitialContextFactory");

**props**.put(Context.PROVIDER\_URL,"remote://localhost:4447");

**props**.put(Context.SECURITY\_PRINCIPAL, "usuario"); // cadastre com add-user.sh em ApplicationRealm

**props**.put(Context.SECURITY\_CREDENTIALS, "senha");

**props**.put("jboss.naming.client.ejb.context", true);

Context ctx = new InitialContext(**props**);

Para usar JMS rode JBoss usando  
bin/standalone.sh –c standalone-full.xml

- Ou arquivos **jndi.properties** e **jboss-ejb-client.properties** no classpath

**jndi.properties**

Context ctx = new InitialContext();

java.naming.factory.initial=org.jboss.naming.remote.client.InitialContextFactory  
java.naming.factory.url.pkgs=org.jboss.ejb.client.naming  
java.naming.provider.url=remote://localhost:4447  
java.naming.security.principal=usuario  
java.naming.security.credentials=senha

**jboss-ejb-client.properties**

remote.connections=default

remote.connection.default.host=localhost

remote.connection.default.port = 4447

remote.connectionprovider.create.options.org.xnio.Options.SSL\_ENABLED=false

remote.connection.default.connect.options.org.xnio.Options.SASL\_POLICY\_NOANONYMOUS=false

Isto é um resumo! Configuração  
muda de versão para versão  
Consulte a documentação!



# Configuração Glassfish 3 & 4

39

- Dependências para cliente externo standalone
  - Inclua no classpath o arquivo **gf-client.jar** (localizado em **\$GLASSFISH\_HOME/glassfish/lib**)
- Configuração JNDI para clientes externos standalone no mesmo host
  - Não é necessário definir nenhuma propriedade. Initialize com  
**Context jndi = new InitialContext();**
- Clientes em outro host/porta: propriedades do ORB
  - Use ferramenta para cliente remoto ou inclua **todos** os JARs do MQ e JNDI remoto
  - Defina as propriedades via **jndi.properties** ou via **System.setProperty(prop, valor)**
    - org.omg.CORBA.ORBIInitialHost=localhost (ou outro endereço)
    - org.omg.CORBA.ORBIInitialPort=3700 (ou outra)
- Para clientes externos que possuem JNDI local (ex: Tomcat), adicione no **jndi.properties** (ou via **java.util.Properties**)
  - java.naming.factory.initial=com.sun.enterprise.naming.SerialInitContextFactory
  - java.naming.factory.url.pkgs=com.sun.enterprise.naming
  - java.naming.factory.state=com.sun.corba.ee.impl.presentation.rmi.JNDIStateFactoryImpl

Isto é um resumo! Configuração  
muda de versão para versão  
Consulte a documentação!



# Fábrica de conexões (via JNDI)

- Objeto gerenciado pelo container
  - Clientes internos podem usar injeção de recursos (@Resource) ou dependências (@Inject)

```
ConnectionFactory factory = (ConnectionFactory)jndi.lookup("nome-jndi")  
@Resource(lookup="nome-jndi")  
ConnectionFactory factory;  
@Inject ConnectionFactory factory;
```
  - Nome do recurso não segue padrão: depende do servidor usado e pode ter restrições de acesso para clientes externos
- Exemplos (**nomes default** – usuário pode configurar outros)
  - GlassFish 3 & 4 (crie via ferramenta asadmin)
    - “**jms/ConnectionFactory**”
  - JBoss 7.1 (configurável via XML, ferramentas ou interface Web)
    - “**java:jboss/exported/jms/RemoteConnectionFactory**” (externo)
    - “**java:/JmsXA**” (interno)
  - ActiveMQ 5 (configurável via jndi.properties, interface Web, XML)
    - “**ConnectionFactory**”



# Destinos (via JNDI)

## ■ Filas (Queue)

- Retêm mensagens até que sejam retiradas da fila ou expirem
- Apenas um cliente pode retirar cada mensagem enviada

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

OU

```
@Resource(lookup="jms/nome-jndi") Queue fila;
```

## ■ Tópicos (Topic)

- Cada tópico pode ter vários clientes assinantes, cada qual recebe uma cópia das mensagens enviadas
- Clientes precisam já ser assinantes antes do envio.
- Em tópicos "duráveis", assinantes não precisam estar ativos no momento do envio: mensagens são preservadas

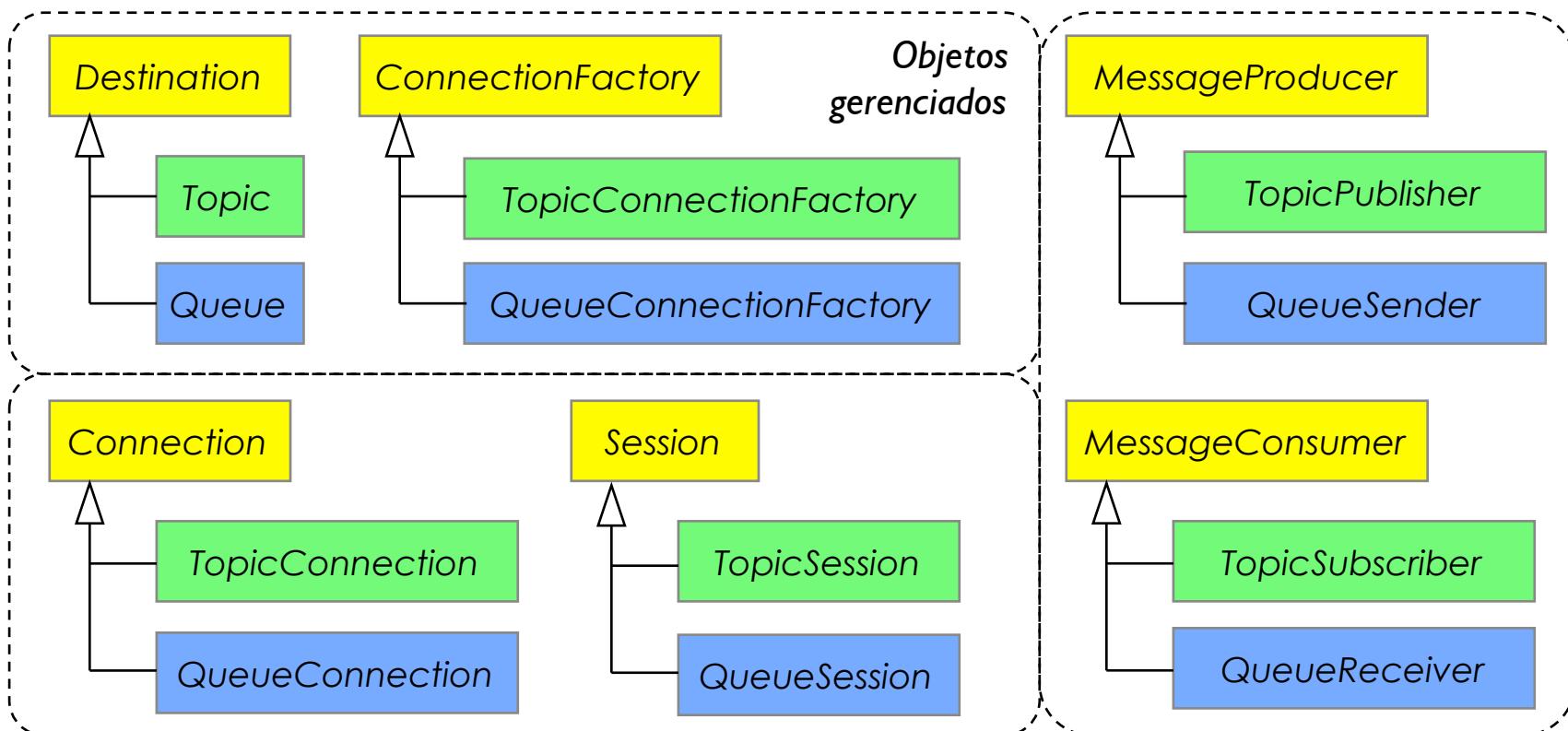
```
Topic topico = (Topic) ctx.lookup("jms/nome-jndi");
```

OU

```
@Resource(lookup="jms/nome-jndi") Topic topico;
```



- Cada domínio tem uma Abstract Factory
  - Desde **JMS 1.1** recomenda-se utilizar as classes genéricas (e não as implementações de cada domínio) para construir aplicações JMS (ex: use MessageProducer e não TopicPublisher)



- Domínio pub/sub
- Domínio ptpt



# Conexões e sessões

- Conexões suportam múltiplas sessões paralelas

```
Connection con = factory.createConnection();
```

- Métodos **start()** e **stop()** iniciam e interrompem (temporariamente) o envio de mensagens; **close()** encerra a conexão.

```
con.start();
```

- Para clientes que exigem autenticação

```
Connection con = factory.createConnection(userid, senha);
```

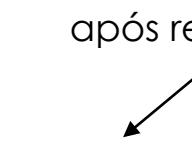
- Cada sessão é um thread, obtida de uma conexão

```
Session session =  
    con.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

↓

Sem transações

Confirmação automática  
após recebimento correto



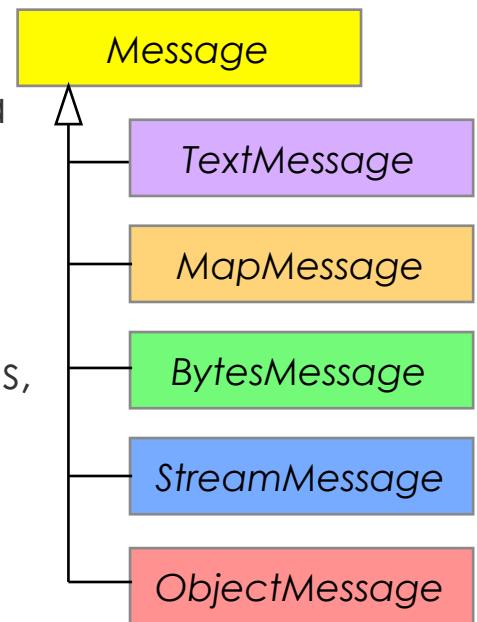
- Modos de Session

- AUTO\_ACKNOWLEDGE
- CLIENT\_ACKNOWLEDGE
- DUPS\_OK\_ACKNOWLEDGE



# Mensagens

- Têm duas partes
  - **Cabeçalho** (contém cabeçalhos JMS e possíveis propriedades definidas pelo usuário)
  - **Corpo** (opcional)
- Seis tipos
  - **Message**: contém apenas cabeçalho; suficiente para enviar propriedades (strings, números, etc.)
  - **TextMessage**: pode conter corpo de texto (ex: XML)
  - **MapMessage**: contém Map como corpo
  - **BytesMessage**: corpo de stream de bytes (ex: imagens, dados comprimidos)
  - **StreamMessage**: corpo de tipos primitivos (interface DataInput/DataOutput)
  - **ObjectMessage**: corpo contém objeto serializado



# Criação de mensagens

- Session possui métodos para criar cada tipo de mensagem

```
TextMessage tm = session.createTextMessage();
```

```
Message m = session.createMessage();
```

- Message possui métodos para guardar e recuperar tipos primitivos e strings como propriedades no cabeçalho

```
m.setStringProperty("Codigo", "123");
```

```
m.setIntProperty("Quantidade", 900);
```

```
String codigo = m.getStringProperty("Codigo");
```

- Cada subclasse de message possui métodos próprios para anexar e recuperar dados anexados no corpo

```
tm.setText("<resultado><codigo>200</codigo></resultado>");
```

```
String anexo = tm.getText();
```



# Cabeçalho e seletores

- Propriedades no cabeçalho da mensagem que iniciam em “JMS” são geradas pelo sistema
  - Ex: `JMSMessageID`, `JMSDestination`, `JMSExpiration`, `JMSPriority`
- Propriedades definidas pelo programador têm métodos get/set declarados na classe `Message`

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

- **Seletores booleanos:** durante o consumo de mensagens, propriedades podem ser usadas em filtro de seleção
  - Seletor é **string** com expressão **SQL** “where”:

```
String seletor = "Formato LIKE '%Imagen%' AND " +
    "JMSExpiration > 10 AND " +
    "Size IS NOT NULL";
```

- Usado durante a criação de um consumidor

```
session.createConsumer (topico, seletor);
```



# Produtores

## ■ MessageProducer

- Criado através de uma sessão
- Passa a referência para um Destination (Queue ou Topic)

```
MessageProducer producer =  
    session.createProducer(fila);
```

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

```
producer.send( message );
```
- Métodos `send(msg, modo, prioridade, ttl)` aceita outros parâmetros onde define qualidade do serviço
  - **modo**: `DeliveryMode.NON_PERSISTENT` (default) ou `PERSISTENT`
  - **prioridade**: 0 – 9 (default é 4 = `Message.DEFAULT_PRIORITY`)
  - **tempo de vida** (TTL) em ms – default é zero
- QoS também pode ser configurado via métodos
  - `get/setDeliveryMode()`, `get/setPriority()`, `get/setTimeToLive()`



# Consumidores síncronos

- **MessageConsumer**

```
MessageConsumer consumer =  
    session.createConsumer(fila);
```

- É preciso iniciar a conexão antes de começar a consumir:  
`con.start();`
- O consumo de forma síncrona é realizado através de `receive()`, `receive(timeout)` ou `receiveNoWait()`

```
Message message = consumer.receive();      // blocking  
consumer.receive(20000); // 20 segundos  
consumer.receiveNoWait(); // no blocking
```

- Se um seletor for definido, apenas as mensagens que passarem pelo seletor serão consumidas

```
session.createConsumer (fila, "Codigo-produto = 'L940');
```



# Consumidores assíncronos

- Consumo assíncrono requer a criação de um **event handler**
  - Implementação da interface **MessageListener** que possui método **onMessage(javax.jms.Message)**
- Código acima deve estar em um bloco try-catch
- Para que objeto seja **notificado**, é preciso registrá-lo em um MessageConsumer

```
consumer.setMessageListener( new MyListener() );
con.start(); // iniciar a conexão
```



# Exercício: JMS

1. Configure o seu servidor de aplicações para JMS
  - Configure o acesso JNDI para as aplicações cliente
  - Descubra como obter uma fábrica de conexões e uma fila (**queue**)
2. Escreva um cliente JMS para produzir mensagens contendo a data e hora (guarde o valor em uma propriedade Long)
  - Escreva uma aplicação standalone, com main()
  - Se possível use conexões e filas já existentes no servidor
3. Escreva dois clientes JMS para consumir mensagens da fila
  - Um que use recebimento síncrono
  - Outro que use recebimento assíncrono
  - Execute um dos consumidores primeiro, em seguida execute um produtor, depois inverta a ordem; rode o produtor várias vezes
  - Escreva um seletor para consumir apenas mensagens produzidas há mais de 1 minuto e aplique-o a um dos consumidores
4. Mude o exercício acima para usar um tópico (topic)



# Message Driven Beans

- Objeto gerenciado (Enterprise JavaBean)
  - Anotado com **@MessageDriven**
  - Pode injetar **@Resource MessageDrivenContext**
  - Permite configuração declarativa de serviços Java EE
- Implementa consumidor JMS assíncrono (listener)
  - Implementa interface **MessageListener** (método **onMessage**)
  - Permite a configuração declarativa de seletores JMS, filas, tópicos, etc.

```
@MessageDriven(activationConfig={
    @ActivationConfigProperty(propertyName="destinationType",
        propertyValue="javax.jms.Queue"),
    @ActivationConfigProperty(propertyName="destination",
        propertyValue="queue/ProdutoQueue")
}
public class ProdutoMDB implements MessageListener {
    @Resource
    private MessageDrivenContext mdc;
    public void onMessage(Message message) { ... }
}
```



# Produtores JMS para MDB

- MDB é apenas um listener para um destino
  - É um **consumidor assíncrono** de mensagens
- Produtores podem ser quaisquer clientes que enviam mensagens para o mesmo destino
  - Produtores externos e **clientes standalone** (como os mostrados anteriormente)
  - Outros **EJBs** e componentes dentro do servidor de aplicações
- Produtores dentro do mesmo container podem injetar filas e conexões

**@Stateless**

```
public class ClienteEJB {  
    public void metodo() throws Exception {  
  
        @Resource(mappedName="java:/JmsXA") ConnectionFactory connectionFactory;  
  
        Connection con = factory.createConnection();  
        Session session = con.createSession(false, Session.AUTO_ACKNOWLEDGE);  
  
        @Resource(mappedName="topic/testTopic") Topic topic;  
  
        Producer publisher = session.createProducer(topic);  
        TextMessage msg = session.createTextMessage();  
        msg.setText("Teste");  
        publisher.send(msg);  
    }  
}
```



# JMS 2.0 (Java EE 7)

- JMS 2.0 simplificou bastante a interface JMS combinando a conexão e sessão JMS em um único objeto: **JMSPContext**
  - Com um JMSPContext é possível criar produtores, consumidores, mensagens, destinos temporários e queue browsers
  - Pode ser criado a partir de um ConnectionFactory ou injetado

```
JMSPContext ctx = connectionFactory.createContext();  
@Inject @JMSPConnectionFactory("jms/MyConnectionFactory")  
private JMSPContext ctx2;
```

- Ideal é criar dentro de um bloco try-with-resources, que fecha o contexto automaticamente ao final:

```
try(JMSPContext ctx = connectionFactory.createContext()) { ... }
```
- Novos produtores e consumidores são JMSProducer e JMSPConsumer

```
JMSProducer producer = jmsCtx.createProducer();
```
- Outras novidades do JMS 2.0
  - Assinaturas compartilhadas: permite criar uma assinatura cujas mensagens serão distribuídas a mais de um consumidor
  - Criação de destinos temporários (que duram o tempo da conexão)
  - Envio assíncrono de mensagens



# Envio de mensagens JMS 2.0

## ■ Antes (JMS 1.1)

```
public void sendJMS11(ConnectionFactory conFactory, Queue queue, String text) {  
    try {  
        Connection con = conFactory.createConnection();  
        try {  
            Session session = con.createSession(false,Session.AUTO_ACKNOWLEDGE);  
            MessageProducer messageProducer = session.createProducer(queue);  
            TextMessage textMessage = session.createTextMessage(text);  
            messageProducer.send(textMessage);  
        } finally {  
            connection.close();  
        }  
    } catch (JMSException ex) { // handle exception } }
```

## ■ JMS 2.0

```
public void sendJMS2(ConnectionFactory conFactory, Queue queue, String text) {  
    try (JMSContext context = conFactory.createContext();) {  
        context.createProducer().send(queue, text);  
    } catch (JMSRuntimeException ex) { // handle exception }  
}
```



# Recebimento síncrono JMS 2.0

## ■ Antes (JMS 1.1)

```
public void sendJMS11(ConnectionFactory conFactory, Queue queue, String text) {  
    try {  
        Connection con = conFactory.createConnection();  
        try {  
            Session session = con.createSession(false,Session.AUTO_ACKNOWLEDGE);  
            MessageConsumer messageConsumer = session.createConsumer(queue);  
            con.start();  
            TextMessage textMessage = (TextMessage)messageConsumer.receive();  
            this.messageContents = textMessage.getText();  
        } finally {  
            connection.close();  
        }  
    } catch (JMSException ex) { // handle exception } }
```

## ■ JMS 2.0

```
public void sendJMS2(ConnectionFactory conFactory, Queue queue, String text) {  
    try (JMSContext context = conFactory.createContext()){  
        JMSConsumer consumer = context.createConsumer(queue);  
        this.messageContents = consumer.receiveBody(String.class);  
    } catch (JMSRuntimeException ex) { // handle exception }  
}
```



# Outras simplificações

- **Message.getBody( tipo.class)** em vez de **getText()**, **getBytes()**, etc. de cada tipo de mensagem

```
void onMessage(Message message) { // BytesMessage JMS 1.1
    int len = ((BytesMessage)message).getBodyLength();
    byte[] bytes = new byte[len];
    int bytes = ((BytesMessage)message).readBytes(bytes);
```

...

```
void onMessage(Message message){ // BytesMessage JMS 2.0
    byte[] bytes = message.getBody(byte[].class);
```

...

- **JMSConsumer.receiveBody( tipo.class)**

- Usado em recebimento síncrono (não é preciso obter a mensagem  
– pode-se retirar o payload diretamente)



# Preparação do ambiente

Infraestrutura para exercícios: leia o README.txt

1. Implante a aplicação-exemplo no servidor de aplicações para que as tabelas sejam geradas corretamente
  - a. Rode o teste **TabelasTest**
  - b. Acesse a aplicação Web e use algumas de suas funções
2. Configure as seguintes filas
  - a. jms/biblioteca/solicitacoes
  - b. jms/biblioteca/autorizacoes
3. Verifique na interface do seu servidor de aplicações se as filas foram criadas. Anote
  - a. Nome JNDI da fábrica de conexões
  - b. Nomes JNDI das filas criadas
4. Se tudo estiver OK, rode o teste **FilasTest**
5. Analise o código das classes fornecidas (completas e incompletas)



# Exercícios

Use os comentários incluídos no código como roteiro

58

1. Transforme as classes **AutorizacaoEmprestimo**, **SolicitacaoEmprestimo** e **ListarLivros** em **SSLBs**
  - a. AutorizacaoEmprestimo e SolicitacaoEmprestimos devem ter interface **Local**
  - b. ListarLivros deve ter interface **Remote**
  - c. Teste a aplicação (rode o **LivrosClient** remota)
2. Crie um **MDB** SolicitarEmprestimo que receba uma propriedade String “Código-do-livro” que envie uma nova mensagem para o **MDB** AutorizarEmprestimo (que está pronto) contendo **duas propriedades**
  - a. **Código-do-livro** – copiada do valor recebido
  - b. **Tempo-autorizado** – valor inteiro
3. Teste a aplicação através de sua interface Web
  - a. Solicite o empréstimo de um livro
  - b. Observe a lista de empréstimos autorizados e veja se a autorização aparece e quantos dias foram autorizados



# Referências

- Especificações Java EE
  - EJB e MDB <http://jcp.org/aboutJava/communityprocess/final/jsr345/>
  - JMS <https://java.net/projects/jms-spec/pages/Home>
  - Java EE <https://java.net/projects/javaee-spec/pages/Home>
- Tutorial Java EE
  - Java EE 6 <http://docs.oracle.com/javaee/6/tutorial/doc/>
  - Java EE 7 <http://docs.oracle.com/javaee/7/tutorial/doc/>
- JMS 2.0 (Java EE 7)
  - Nigel Deakin. "What's New in JMS 2.0". Oracle. 2013.
    - Part I <http://www.oracle.com/technetwork/articles/java/jms20-1947669.html>
    - Part II <http://www.oracle.com/technetwork/articles/java/jms2messaging-1954190.html>
- Configuração de JNDI externo
  - Glassfish [https://glassfish.java.net/javaee5/ejb/EJB\\_FAQ.html#StandaloneRemoteEJB](https://glassfish.java.net/javaee5/ejb/EJB_FAQ.html#StandaloneRemoteEJB)
  - JBoss <http://stackoverflow.com/questions/14336478/jboss-7-jndi-lookup>
  - ActiveMQ <http://activemq.apache.org/jndi-support.html>



2

Infraestrutura de sistemas  
Java EE

Tecnologias para  
representação de dados

XML, XML Schema, XPath, JSON, DOM, SAX, JAXP e JAXB



# Conteúdo

## 1. Formatos de dados usados em Web Services

- XML
- JSON

## 2. XML

- Tecnologias XML: DTD, XML Schema, XPath
- APIs de manipulação do XML - JAXP: DOM, SAX, StAX e TrAX
- XML Java Binding: JAXB

## 3. JSON

- JSON
- JSON Java APIs
- JSON Java Binding



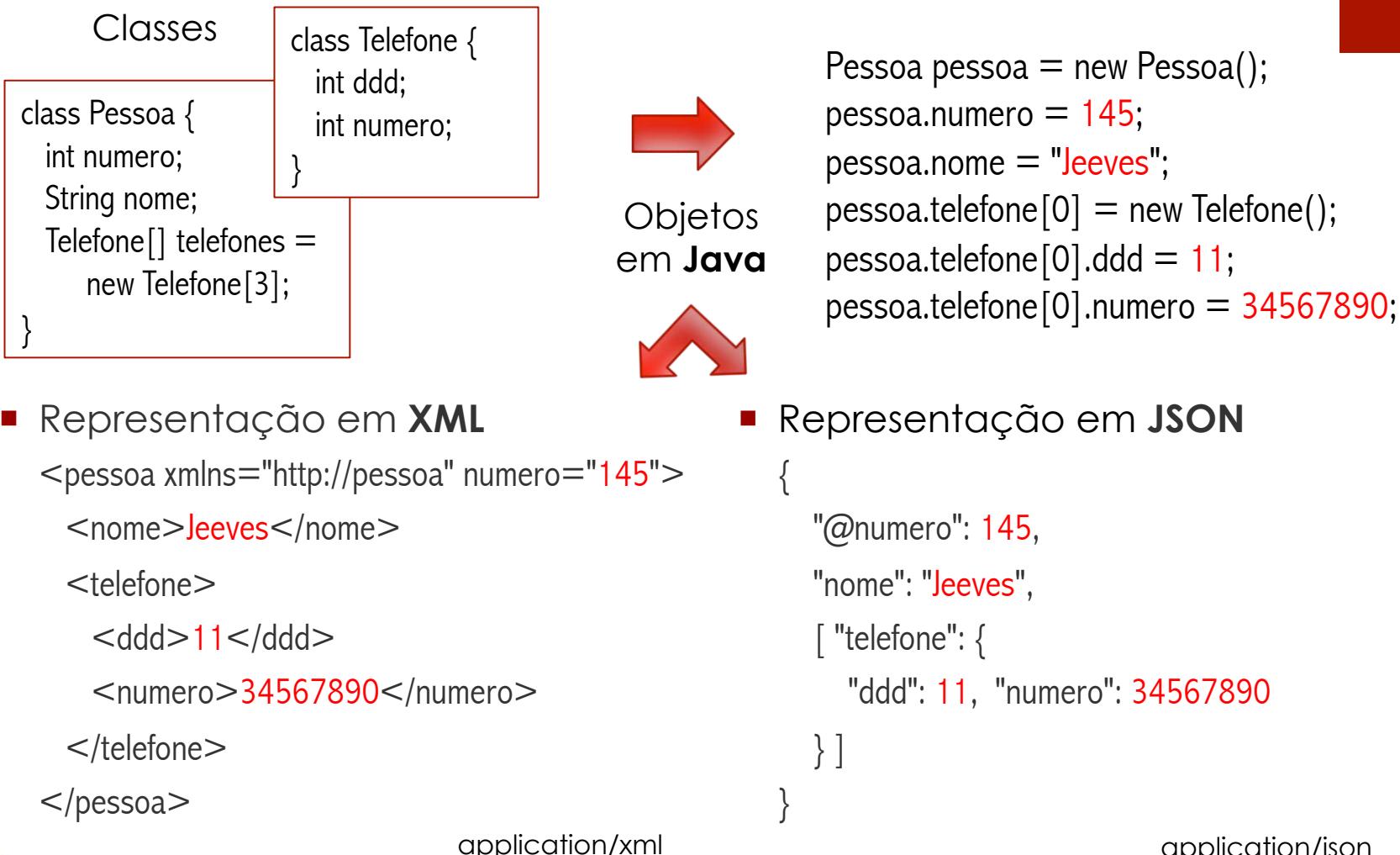
# Formatos de dados

- Os dois formatos de dados (texto)mais usados em Web Services são XML e JSON
  - Ambos são independentes de linguagem e plataforma
  - Ambos representam dados como uma **árvore** e têm APIs para streaming, modelo de objetos e binding de objetos
- XML `<dados><x>123</x><y>456</y></dados>`
  - Especificação W3C
  - Media type: **application/xml**
  - APIs SAX (streaming), DOM (objetos), JAXB (Java binding)
- JSON `dados: {x :123, y : 456}`
  - Estruturas baseadas em JavaScript
  - Media type: **application/json**
  - APIs de streaming e objetos nativas Java EE 7, binding via JAXB



# Representação de objetos

- É possível usar XML e JSON para obter diferentes **representações** dos mesmos objetos Java



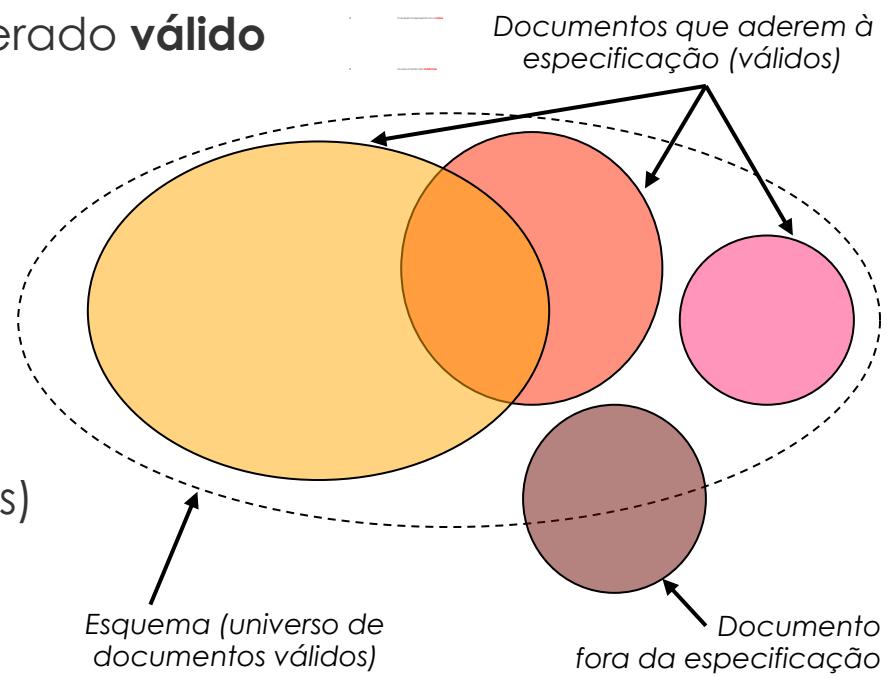
# XML 1.0 (W3C, 1996)

- XML é um padrão W3C que define sintaxe para documentos estruturados com dados marcados por **tags** e **atributos**
  - Não define vocabulário, apenas regras mínimas de formação – facilita a construção de parsers
- Documentos XML devem ser **bem formados** para que informações sejam processadas
  - Características de documentos XML bem formados:
    - Têm um, e apenas um, elemento raiz (é uma árvore)
    - Atributos não se repetem no mesmo elemento, Valores entre aspas ou apóstrofes
    - Todos os elementos têm etiqueta de fechamento e corretamente aninhados
  - **Fragments XML** podem não ser bem formados, mas ainda podem ser usados
    - Podem ser lidos/gravados mas só podem ser **processados** depois de inseridos no contexto de documentos bem-formados
- Tecnologias relacionadas a XML (relevantes neste curso)
  - Esquemas: **DTD** e **XML Schema** (especificação e validação de XML)
  - **XML Namespaces**
  - **XPath** (linguagem para localizar elementos em árvore XML)
  - APIs de programação em Java: **JAXP**, **DOM**, **SAX**, **StAX**, **JAXB**



# XML válido e esquemas

- A **validade** de um XML é um conceito relativo
  - Um XML bem formado pode ser válido em relação a determinada aplicação e não ser válido em outra
- Validade pode ser formalizado através de um **esquema**
  - Um esquema **especifica** um vocabulário de elementos e atributos, regras de formação, etc.
  - Documento XML pode ser considerado **válido em relação a um esquema**
- Um esquema é essencial para usar XML em **comunicação**
- Linguagens usadas para especificar esquemas em XML
  - **DTD** (limitado, porém muito simples)
  - **XML Schema**



# DTD (Document Type Definition)

- Linguagem de esquema que declara todos os elementos e atributos de um documento XML
  - Define quais elementos e atributos são válidos e em que contexto
- Exemplo: DTD para um documento simples

```

<!ELEMENT pessoa (nome, web?, telefone+)>
<!ELEMENT nome (prenome, inicial*, sobrenome)>
<!ELEMENT prenome (#PCDATA)>
<!ELEMENT inicial (#PCDATA)>
<!ELEMENT sobrenome (#PCDATA)>
<!ELEMENT web (email|website)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT website (#PCDATA)>
<!ELEMENT telefone (#PCDATA)>

```

**pessoa** tem **nome**, seguido de zero ou um **web** e um ou mais **telefone**

**nome** tem um **prenome**, seguido de zero ou mais **inicial** e um **sobrenome**

**web** pode conter ou um **email** ou um **website**

Elementos que só podem conter texto



# XML Namespaces

- Estabelecem um **contexto** para elementos e atributos
  - Formalmente declarados através de um identificador (string, geralmente uma URI) em atributo reservado do XML: **xmlns**
- A declaração pode associar o namespace a um **prefixo** para **qualificar** elementos e atributos
  - Quando o prefixo não é usado, estabelece-se um **namespace default** no contexto formado pelo elemento onde é declarado e seus elementos-filho

```
<simulacao>
  <tempo unidade="segundos">130</tempo>
  <clima xmlns="uri://app-clima">
    <tempo>chuvisco</tempo>
  </clima>
</simulacao>
```

Escopo do namespace vale para elemento **<clima>** e herdado por todos os seus descendentes

```
<simulacao xmlns:w="uri://app-clima">
  <tempo unidade="segundos">130</tempo>
  <w:clima>
    <w:tempo>chuvisco</w:tempo>
    <tempo unidade="horas">2.5</tempo>
  </w:clima>
</simulacao>
```

Escopo do namespace vale para descendentes de **<simulacao>** qualificados com o prefixo 'w'

Nos dois casos, elementos **<tempo>** significam coisas diferentes, mas não há conflito porque pertencem a namespaces diferentes (um deles não tem namespace declarado)



- Linguagem usada para especificar uma **classe** de documentos XML
  - Mesma finalidade que DTD, com mais recursos e sintaxe XML
  - Define coleção de **tipos simples** (números, strings)
  - Permite derivar novos tipos simples por **restrição** (ISBNs, CNPJs, limites de valores, etc.)
  - Permite declarar **tipos complexos** (elementos com atributos e elementos) e derivar novos tipos complexos por **extensão**

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" ... >
  <xs:element name="oferta">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="codigo"/>
        <xs:element ref="preco"/>
        <xs:element ref="itens"/>
      </xs:sequence>
      <xs:attribute name="validade" use="required" type="xs:date"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="preco">
    <xs:simpleType>
      <xs:restriction base="xs:decimal">
        <xs:fractionDigits value="2" />
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
  <xs:element name="codigo"> ... </xs:element>
  <xs:element name="itens"> ... </xs:element>
</xs:schema>
```



# Anatomia de XML Schema

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="astro" type="astroType" />
  <xs:element name="imagem" type="imagemType"/>

  <xs:attribute name="href" type="xs:anyURI"/>
  <xs:attribute name="id" type="xs:ID"/>
  <xs:attribute name="nome" type="xs:string"/>
  <xs:attribute name="diametrokm" type="xs:decimal"/>

  <xs:complexType name="imagemType">
    <xs:attribute ref="href" use="required"/>
  </xs:complexType>

  <xs:complexType name="astroType">
    <xs:sequence>
      <xs:element ref="imagem" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute ref="id" use="required"/>
    <xs:attribute ref="nome" use="required"/>
    <xs:attribute ref="diametrokm"/>
  </xs:complexType>
</xs:schema>
```

← Elementos

← Atributos

Tipos simples

Definição de tipos complexos



# Compare com DTD

Exemplo de documento válido  
em relação a este DTD ou  
XML Schema mostrado

Modelo de conteúdo  
(tipo de dados complexo)

```
<astro id="p5" nome="Jupiter">
  <imagem href="jup31.jpg" />
  <imagem href="jup32.jpg" />
</astro>
```

```
<!ELEMENT astro (imagem*)>
<!ELEMENT imagem EMPTY>
```

← Elementos

Atributos

<!ATTLIST imagem href	CDATA	#REQUIRED	>
<!ATTLIST astro id	ID	#REQUIRED	>
<!ATTLIST astro nome	CDATA	#REQUIRED	>
<!ATTLIST astro diametrokm	NMTOKEN	#IMPLIED	>

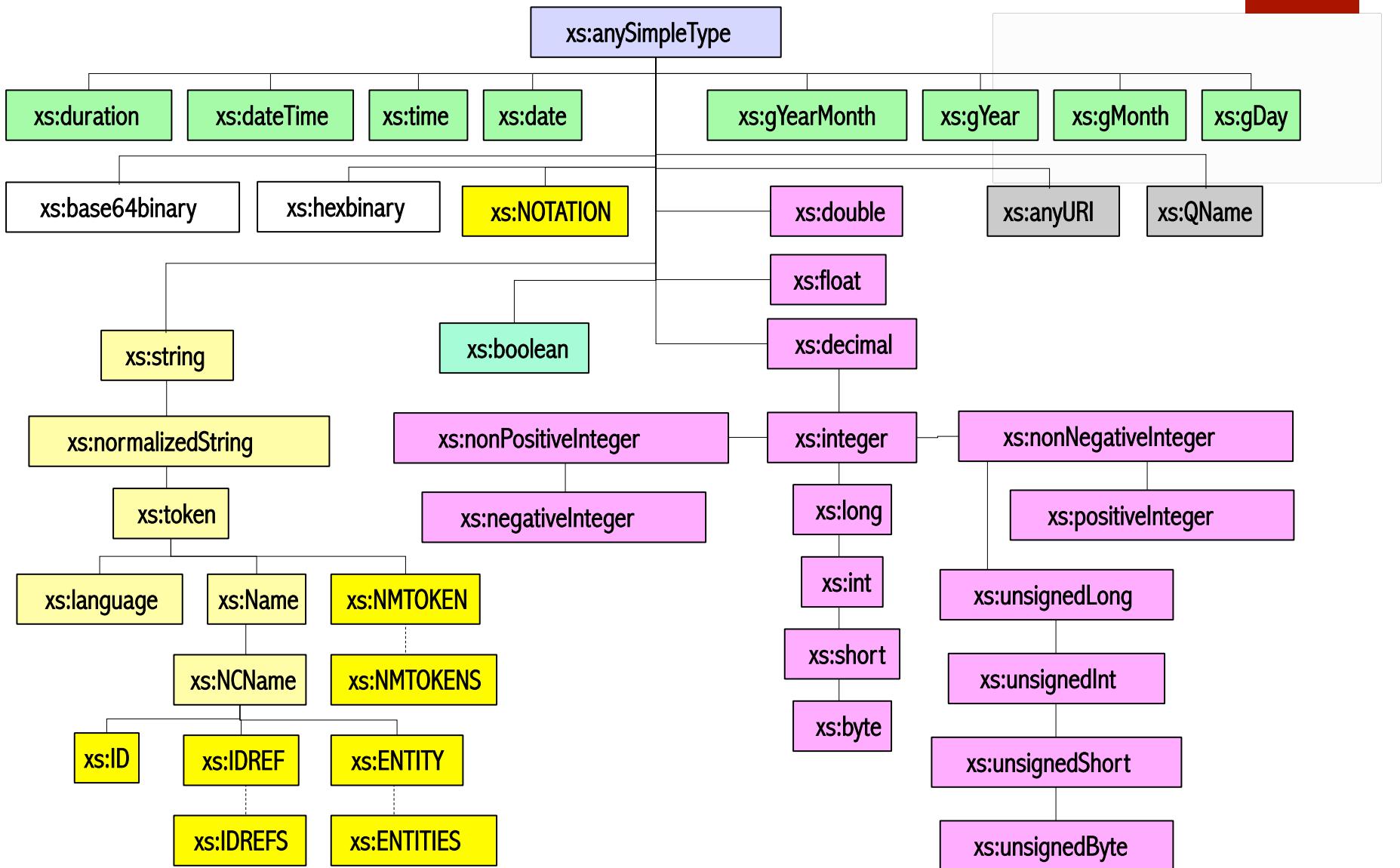
Atributo sempre  
associado a elemento

Tipos de dados simples  
(somente para atributos)



# Tipos simples nativos do XML Schema

71



# Derivação de um tipo simples

- Tipo **base**: xs:NMTOKEN
- Tipo **derivado**: isbn
  - Faceta de comprimento <xs:length>:  
exatamente 10 caracteres
  - Faceta de formato <xs:pattern>:  
descrito por expressão regular

```
<xs:simpleType name="isbn">
  <xs:restriction base="xs:NMTOKEN">
    <xs:length value="10"/>
    <xs:pattern value="[0-9]{9}[0-9X]"/>
  </xs:restriction>
</xs:simpleType>
```



# Declaração de tipo complexo

- <xs:complexType>
  - Tipo que pode conter outros elementos ou atributos
  - Pode ser usado para descrever a estrutura de um objeto

```
<xs:complexType name="imagemType">
    <xs:attribute name="href" type="xs:anyURI"/>
</xs:complexType>

<xs:complexType name="astroType">
    <xs:sequence>
        <xs:element name="imagem" type="imagemType" minOccurs="0"/>
        <xs:element name="satelite" type="sateliteType"
                    minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="id" type="astroID" use="required"/>
    <xs:attribute name="nome" type="xs:token" />
    <xs:attribute name="diametrokm" type="xs:decimal"/>
</xs:complexType>
```



# Validação de instâncias XML

- Uma instância pode ser formalmente vinculada a um XML Schema através de atributos globais declarados em um namespace padrão
  - Namespace XML Schema Instance:  
**<http://www.w3.org/2001/XMLSchema-instance>**
- Este namespace deve ser atribuído a um prefixo no documento XML a ser vinculado, para que se possa usar um ou mais dos seus 4 atributos:
  - **schemaLocation** – para vincular um ou mais XML Schemas à instância
  - **noNamespaceSchemaLocation** – para vincular documento a um XML Schema que não usa namespace
  - **nil** e **type** – usados em elementos e atributos



# Exemplos com namespace

```
<oferta xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.empresa.com/pontodevenda
                            ofertaElementsAllImport.xsd"
        xmlns="http://www.empresa.com/pontodevenda"
        validade="2010-07-22">

    <codigo numero="4599" tipo="T"/>
    <preco>15.2</preco>
    <itens>14</itens>
</oferta>
```

```
<pdv:oferta xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.empresa.com/pontodevenda
                            ofertaElementsAllImport.xsd"
        xmlns:pdv="http://www.empresa.com/pontodevenda"
        validade="2010-07-22">

    <pdv:codigo numero="4599" tipo="T"/>
    <pdv:preco>15.2</pdv:preco>
    <pdv:itens>14</pdv:itens>
</pdv:oferta>
```



# Múltiplos namespaces

- Schemas compostos requerem a declaração de múltiplos namespaces na instância
  - Esquema principal

```
<xs:schema  
    targetNamespace="http://cosmos.org.br"  
    xmlns:cm="http://cosmos.org.br/com"  
    xmlns:st="http://cosmos.org.br/sat"  
    xmlns="http://cosmos.org.br"  
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

Target  
namespace

- Instância

```
<se:sistemaEstelar xmlns:se="http://cosmos.org.br"  
    xmlns:sat="http://cosmos.org.br/sat"  
    xmlns:cmt="http://cosmos.org.br/com"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xsi:schemaLocation="http://cosmos.org.br sistema.xsd  
                      http://cosmos.org.br/sat satelites.xsd  
                      http://cosmos.org.br/com cometas.xsd">
```



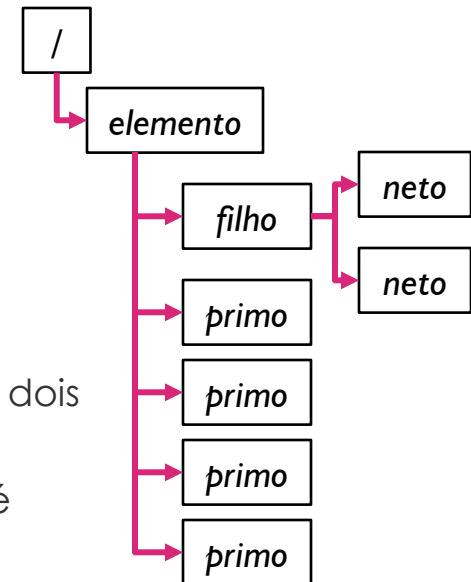
# Exemplo com múltiplos namespaces

```
<html xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.w3.org/1999/xhtml  xhtml.xsd
                           http://www.empresacom/pontodevenda  oferta.xsd"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:pdv="http://www.empresacom/pontodevenda">
  <head>...</head>
  <body><div class="oferta">
    <pdv:oferta validade="2010-07-22">
      <pdv:codigo numero="4599" tipo="T"/>
      <pdv:preco>15.2</pdv:preco>
      <pdv:itens>14</pdv:itens>
    </pdv:oferta>
  </div></body>
</html>
```

```
<html xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.w3.org/1999/xhtml  xhtml.xsd
                           http://www.empresacom/pontodevenda  oferta.xsd"
      xmlns="http://www.w3.org/1999/xhtml" >
  <head>...</head><body><div class="oferta">
    <oferta validade="2010-07-22" xmlns="http://www.empresacom/pontodevenda">
      <codigo numero="4599" tipo="T"/>
      <preco>15.2</preco>
      <itens>14</itens>
    </oferta>
  </div></body>
</html>
```



- Linguagem usada para **localizar informações** em XML
  - É usada para configurar mapeamento Java-XML em bindings (JAXB, REST, SOAP Web Services)
  - Expressão XPath é um **caminho: passos** de navegação na árvore XML separados por “/”; cada passo tem um **eixo, teste e predicado**
    - **eixo::teste[predicado]**
    - Sintaxe permite variações e atalhos. Ex: `produto[@nome='livro']` é equivalente a `child::produto[attribute::nome='livro']`
  - Resultados da expressão podem ser: **texto, número, booleano, elemento, atributo, nó de texto ou node-set** (sub-árvore XML)
- XPath opera sobre o **XML processado**
  - O arquivo-fonte usado pelo XPath não contém entidades
    - Todas as entidades e seções CDATA são convertidas em XML e texto antes do processamento XPath (usa ‘>’ e não ‘&gt;’)
  - Exemplos (navegando na árvore ao lado)
    - **/elemento/filho/neto** : resultado é node-set 'neto', que contém dois elementos
    - **.../.../primo[3]**: expressão relativa (a partir de 'neto'); resultado é elemento <primo> na posição 3 (do node-set de tamanho 4)



# Caminhos XPath equivalentes

(1)

- `parent::* / following-sibling::paragrafo`
- `../following-sibling::paragrafo`

(2)

- `descendant-or-self::capitulo[position()=1] / child::secao[position()=4] / child::paragrafo[position()=1]`
- `//capitulo[1]/secao[4]/paragrafo[1]`

(3)

- `self::elemento | descendant::elemento`
- `//elemento`
- `descendant-or-self::elemento`

Veja referências no final desta apresentação para links com maiores informações sobre XPath

(4)

- `/child::cursos / child::curso / child::topicos / child::item[position()=5]`
- `/cursos / curso / topicos / item[5]`



# JAXP

## ■ Java API for XML Processing

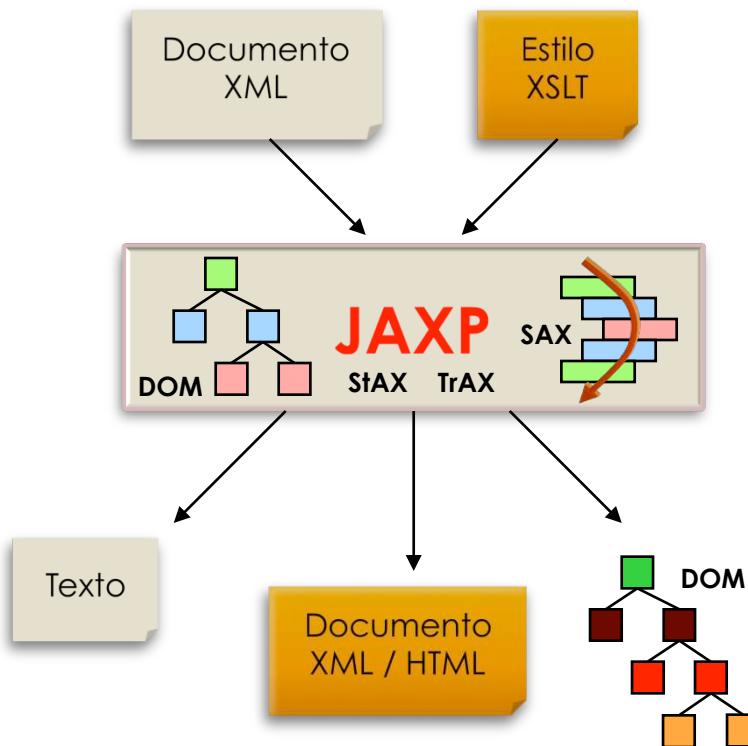
- Para leitura, criação, manipulação, transformação de XML
- Parte integrante do JAVA SE

## ■ Pacotes

- javax.xml.parsers
- javax.xml.transform (TrAX)
- javax.xml.stream (StAX)
- org.w3c.dom (DOM)
- org.xml.sax (SAX)

## ■ Componentes

- Parsers estilo pull e push
- APIs para streaming XML
- API W3C DOM
- Transformação XSLT



# Manipulação do XML

81

- **SAX** (Simple API for XML) – **org.xml.sax**
  - API baseada em **eventos** que lê o documento sequencialmente
  - Eficiente para pesquisar documentos longos e quando não é necessário montar toda a árvore
  - Requer a criação de listeners que irão capturar eventos do SAX (localização de documento, elementos, texto)
- **DOM** (Document Object Model) – **org.w3c.dom**
  - API baseada em **modelo de objetos**
  - Realiza a montagem de toda a **árvore na memória**
  - Necessário para usar tecnologias como XPath, Xquery e XSLT
  - Interface Java SE é padrão (W3C) – há alternativas: JDOM, DOM4J
- **StAX** – Streaming API - **javax.xml.streaming**
  - API de Java I/O streaming (mais fácil de usar que SAX)
- **TrAX** – Transformador XSLT – **javax.xml.transform**
  - Permite transformar árvores DOM em outros default (texto, XML formatado)
  - Aceita documentos XSLT para customizar transformação



# Processamento SAX

- Se um processador SAX receber o documento ...

```
<carta>
  <mensagem id="1">Bom dia!</mensagem>
</carta>
```

- ... ele irá disparar os seguintes eventos:

- **startDocument()**
- **startElement("carta", [])**
- **startElement("mensagem", [Attribute("id","1")])**
- **characters("Bom dia!")**
- **endElement("mensagem")**
- **endElement("carta")**
- **endDocument()**

- Programador deve implementar um listener para capturar os eventos e extrair as informações desejadas



# Como usar SAX

- Crie listener estendendo **org.w3c.sax.DefaultHandler**

```
public class MySaxHandler extends DefaultHandler {...}
```
- Implemente os métodos de evento desejados nessa classe
  - characters()
  - startElement()
  - endElement(), etc.
- Crie outra classe para inicializar o parser e processar o documento XML
  - Importe as classes **SAXParserFactory**, **SAXParser** e **XMLReader** de **org.w3c.sax**

```
SAXParserFactory spf = SAXParserFactory.newInstance();
SAXParser sp = spf.newSAXParser();
XMLReader reader = sp.getXMLReader();
reader.setContentHandler(new MySaxHandler());
reader.parse("documento.xml");
```

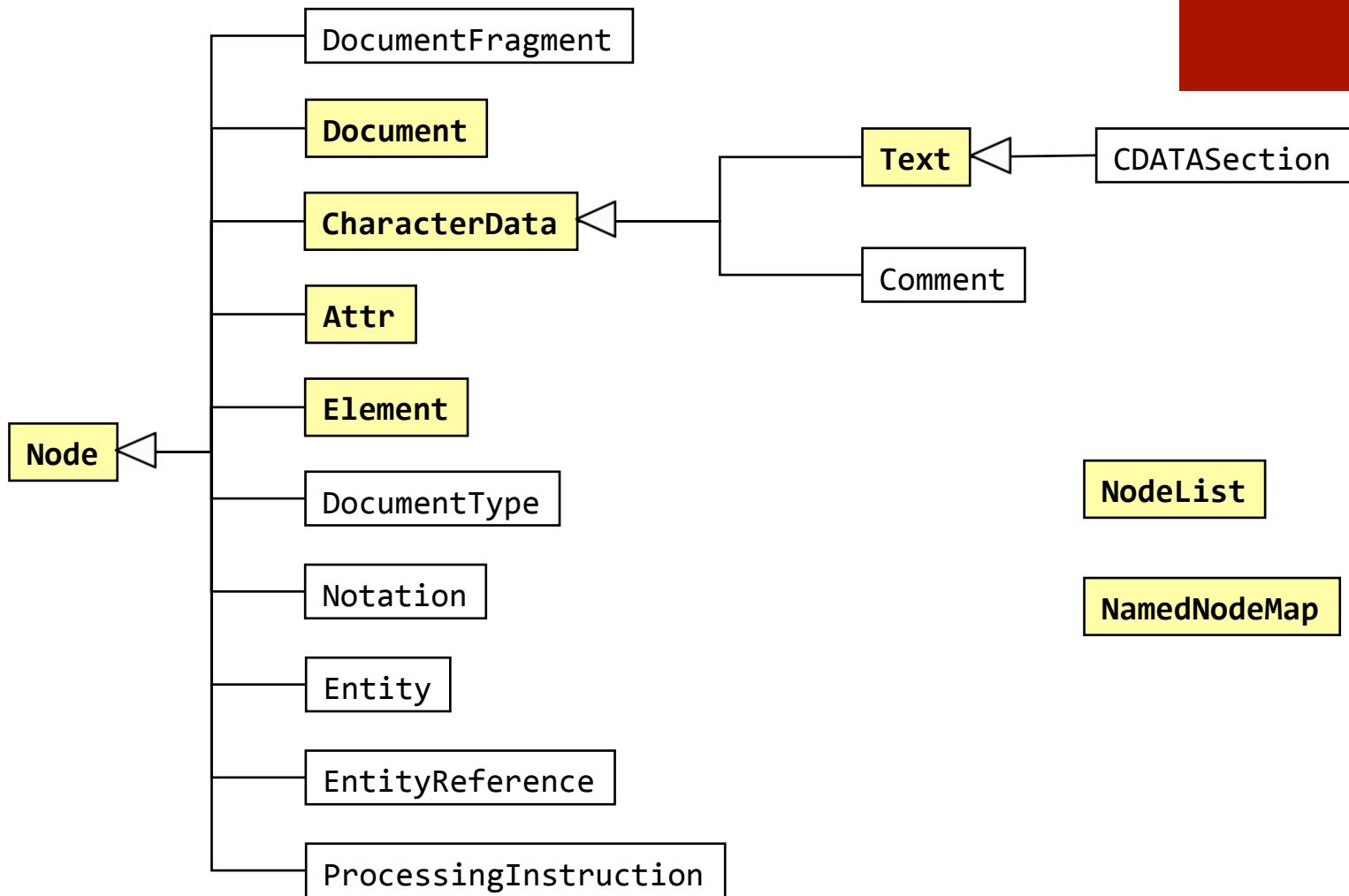


# Implementação de SAX Handler

```
public class MySaxHandler extends DefaultHandler {  
  
    public void characters(char[] ch, int start, int length) {  
        for (int i = start; i < length; i++) {  
            System.out.println(ch[i]);  
        }  
    }  
  
    public void startElement(String uri, String localName,  
                            String qName, Attributes att) {  
        System.out.print("<" + qName);  
        for (int i = 0; i < att.getLength(); i++) {  
            System.out.print(" " + att.getQName(i) + "="'  
                            + att.getValue(i) + "'");  
        }  
        System.out.println(">");  
    }  
  
    public void endElement(String uri, String localName,  
                          String qName) {  
        System.out.println("</> " + qName + ">");  
    }  
}
```



# W3C DOM: Hierarquia



# Document Object Model API

86

## Node

- `Node appendChild(Node)`
- `Node cloneNode(boolean)`
- `NamedNodeMap getAttributes()`
- `NodeList getChildNodes()`
- `boolean hasAttributes()`
- `boolean hasChildNodes()`
- `Node insertBefore(Node, Node)`
- `Node removeChild(Node)`
- `Node replaceChild(Node, Node)`
- `Node getFirstChild()`
- `Node getLastChild()`
- `Node getNextSibling()`
- `Node getPreviousSibling()`
- `String getNodeName()`
- `short getNodeType()`
- `String getNodeValue()`
- `Document getOwnerDocument()`
- `Node getParentNode()`

## Text e CharacterData

- `void appendData(String)`
- `String getData()`
- `int getLength()`
- `void insertData(int, String)`
- `void setData(String)`
- `void replaceData(int, int, String)`

## Document

- `Attr createAttribute(String)`
- `Attr createAttributeNS(String, String)`
- `Element createElement(String)`
- `Element createElementNS(String, String)`
- `Text createTextNode(String)`
- `DocumentType getDocType()`
- `Element getDocumentElement()`
- `Element getDocumentById(String)`
- `NodeList getElementsByTagName(String)`
- `NodeList getElementsByTagNameNS(String, String)`

## NamedNodeMap

- `Node item(int)`
- `Node getNamedItem(String)`
- `Node nextNode()`
- `void reset()`
- `int getLength()`

## NodeList

- `Node item(int)`
- `Node nextNode()`
- `void reset()`
- `int getLength()`

## Attr

- `String getName()`
- `Element getOwnerElement()`
- `String getValue()`
- `void setValue(String)`

## Element

- `String getAttribute(String)`
- `String getAttributeNS(String, String)`
- `Attr getAttributeNode(String)`
- `Attr getAttributeNodeNS(String, String)`
- `String getTagName()`
- `boolean hasAttribute(String)`
- `boolean hasAttributeNS(String, String)`
- `void removeAttribute(String)`
- `void removeAttributeNS(String, String)`
- `void setAttribute(String, String)`
- `void setAttributeNS(String, String, String)`
- `NodeList getElementsByTagName(String)`
- `NodeList getElementsByTagNameNS(String, String)`



# Criação de árvore DOM

## ■ Criação de elementos e nó de texto

/

Document

Objeto **document** é obtido através da  
inicialização do parser

<carta>

Element

`carta := document.createElement("carta")`

<mensagem>

Element

`mens := document.createElement("mensagem")`

Bom dia!

String

`texto := document.createTextNode("Bom dia!")`

## ■ Criação de atributos

<carta id="1">

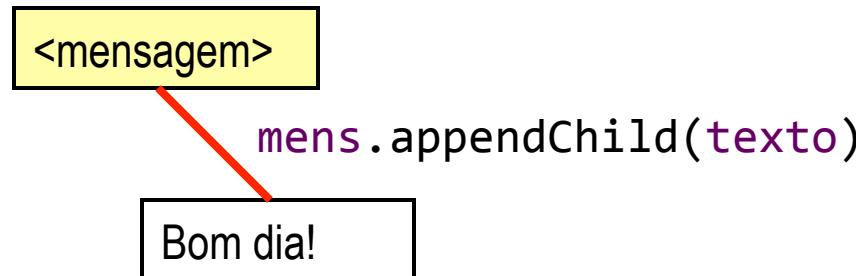
`carta.setAttribute("id", "1")`



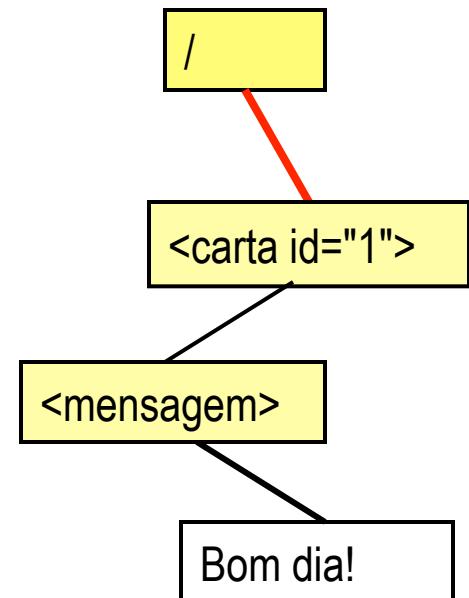
# Criação de árvore DOM (2)

- Interligando elementos para construir a árvore

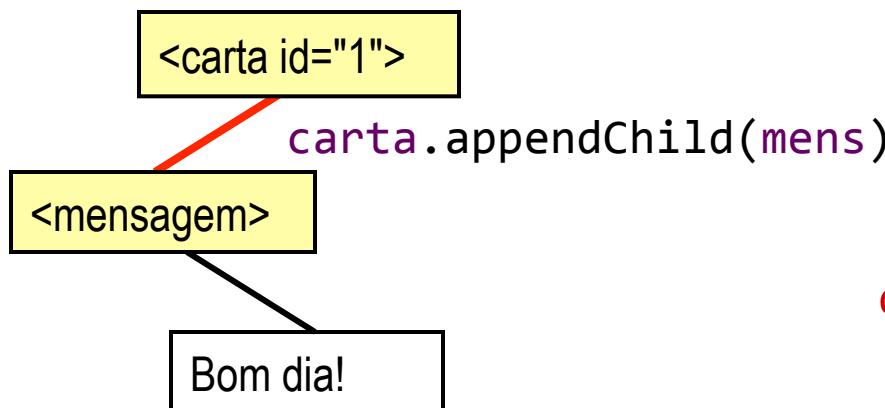
- 1. Sub-árvore <mensagem>



- 3. Árvore completa



- 2. Sub-árvore <carta>



`document.appendChild(carta)`



# Obtenção do Document

- É preciso obter o **Document** para trabalhar com DOM
- Use os pacotes javax.xml.parsers.\* e org.w3c.dom.\*

- Crie um **javax.xml.parsers.DocumentBuilder**

```
DocumentBuilderFactory factory =
    DocumentBuilderFactory.newInstance();
```

```
DocumentBuilder builder = factory.newDocumentBuilder();
```

- Chame **builder.newDocument()** para obter o elemento **raiz** de um documento vazio (org.w3c.dom.Document)

```
Document document = builder.newDocument();
```

- Ou chame **builder.parse("documento.xml")** para obter o elemento raiz de um documento XML existente

```
Document document = builder.parse("documento.xml");
```

- Exemplo de uso de DOM com Java

```
Element elemento = document.getElementById("secao");
elemento.appendChild(document.createElement("p"));
```



# Serialização → XML

- Uma vez criada a árvore DOM, ela pode ser serializada para XML (arquivo de texto)
- Solução padrão é usar XSLT (**javax.transform**)
  - javax.xml.transform.\*
  - javax.xml.transform.dom.DOMSource;
  - javax.xml.transform.stream.StreamResult;
- O trecho abaixo imprime o documento XML contido em document na saída padrão (System.out)

```
TransformerFactory tFactory = TransformerFactory.newInstance();
Transformer transformer = tFactory.newTransformer();
DOMSource source = new DOMSource(document);
StreamResult result = new StreamResult(System.out);
transformer.transform(source, result);
```



# Transformação XSLT (TrAX)

- Inicialize o ambiente (DocumentBuilder, pacotes, etc.)
- Carregue o arquivo-fonte em uma árvore DOM

```
Document document = builder.parse("fonte.xml");
```

- Inicialize a árvore DOM do arquivo resultado

```
Document resDocument = builder.newDocument();
```

- Crie os objetos (e crie um InputStream com a folha XSLT)

```
Source xmlSource = new DOMSource(document);
Result result = new DOMResult(resDocument);
Source xslStyle = new StreamSource(estilo);
```

- Inicialize o transformador XSL

```
TransformerFactory tf = TransformerFactory.newInstance();
Transformer t = tf.newTransformer(xslStyle);
```

- Faça a transformação

```
t.transform(xmlSource, result);
```

- A árvore DOM resultante está em resDocument



# StAX streaming API

- StAX possui duas APIs: **Cursor API** e **Iterator API**
- Cursor API – navega por um componente XML de cada vez
  - Interface **XMLStreamReader** (similar a SQL ResultSet)
    - Métodos de iterator next()/hasNext()
    - Métodos de acesso ao XML getText(), getLocalName(), etc.
  - Interface **XMLStreamWriter** (similar a usar SAX para gravação)
    - writeStartElement(), writeEndElement(), writeCharacters(), ...
- Iterator API – XML como um stream de eventos
  - **XMLEvent** – encapsula os eventos SAX - subtipos
    - StartDocument, StartElement, Characters, etc.
  - **XMLEventReader** extends Iterator
    - XMLEvent nextEvent(), boolean hasNext(), XMLEvent peek()
  - **XMLEventWriter**
    - Add(XMLEvent), add(Attribute), flush(), close()
- **XMLInputFactory**, **XMLOutputFactory**, **XMLEventFactory**
  - Configuram e inicializam o parser para as duas APIs



# Como usar StAX cursor API

- Leitura

```
XMLInputFactory if = XMLInputFactory.newInstance();
XMLStreamReader sr =
    if.createXMLStreamReader(path, new FileInputStream(path));
while(sr.hasNext()) {
    sr.next();
    ...
    System.out.println(sr.getText());
}
```

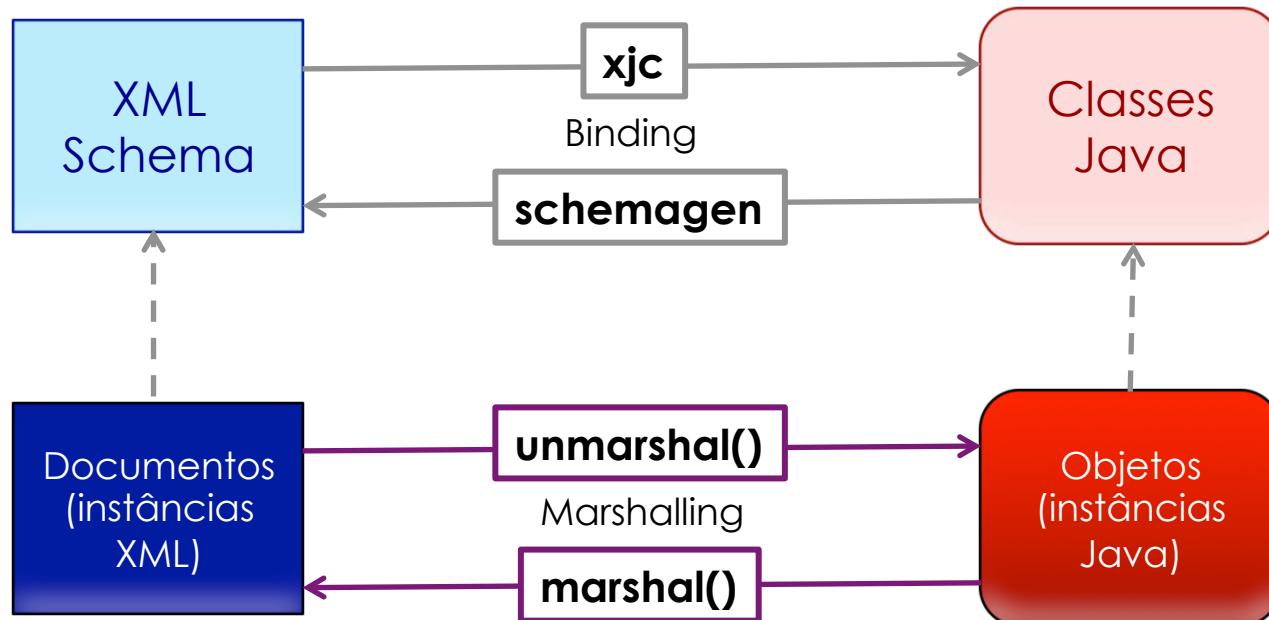
- Gravação

```
XMLOutputFactory of = XMLOutputFactory.newInstance();
XMLStreamWriter w =
    of.createXMLStreamWriter(new FileWriter(path));
w.writeStartDocument("utf-8","1.0");
w.writeStartElement("http://namespace", "mensagem");
w.writeCharacters("Hello!");
w.writeEndElement();
w.writeEndDocument();
w.close();
```



# JAXB

- Java API for XML Binding
- Mapeia classes Java a XML Schema
  - Classes mapeadas a XML Schema (ferramentas **xjc** e **schemagen**)
  - Objetos mapeados a documentos XML (através da API **javax.xml.bind** e operações de serialização em XML (**marshalling** e **unmarshalling**)



# Geração de classes

- Classes geradas a partir de um XML Schema são criadas em um pacote, que inclui
  - Uma **classe Java** derivada de um elemento XML
  - Uma **classe ObjectFactory** usada para produzir instâncias
- Comportamento default XML Schema → Java
  - Tipos **complexos** geram classes
  - Tipos **simples** geram elementos ou atributos (configurável) mapeados a tipos primitivos ou nativos do Java
  - Casos em que a informação é insuficiente para inferir o tipo geram **JAXBElement** (que possui métodos para obter nome e valor do objeto)
  - Pode ser configurado através de anotações no XML Schema **<xs:annotation><xs:appinfo>** ou XML de configuração **<bindings>** lido pelo **xjc**
- Comportamento default para Java → XML Schema mapeia tipos de forma diferente
  - Pode ser configurado através de anotações



# Mapeamento XML → Java

- **xs:string** → java.lang.String
- **xs:integer** → java.math.BigInteger
- **xs:decimal** → java.math.BigDecimal
- **xs:int** → int, **xs:long** → long, **xs:double** → double, etc.
- **xs:unsignedInt** → long
- **xs:unsignedShort** → int
- **xs:unsignedByte** → short
- **xs:QName**, **xsd:NOTATION** → javax.xml.namespace.Qname
- **xs:base64Binary**, **xs:hexBinary** → byte[]
- **xs:time**, **xs:date**, **xs:gYear**, **xs:gMonth**, **xs:dateTime**, **xs:gDate**, **xs:gMonthYear** → javax.xml.datatype.XMLGregorianCalendar
- **xs:anySimpleType** → java.lang.String
- **xs:duration** → javax.xml.datatype.Duration



# Mapeamento Java → XML: tipos

- `java.lang.Object` → **xs:anyType**
- `java.lang.String`, `java.util.UUID`, `java.net.URI` → **xs:string**
- `java.math.BigInteger` → **xs:integer**
- `java.math.BigDecimal` → **xs:decimal**
- `int` → **xs:int**, `long` → **xs:long**, `float` → **xs:float**, `double` → **xs:double**, etc.
- `java.awt.Image`, `javax.activation.DataHandler`, `javax.xml.transform.Source` →  
**xs:base64Binary**
- `java.util.Calendar`, `java.util.Date` → **xs:dateTime**
- `javax.xml.datatype.Duration` → **xs:duration**
- `javax.xml.datatype.XMLGregorianCalendar` → **xs:anySimpleType**
- `javax.xml.namespace.QName` → **xs:QName**



# Ferramentas

- **xjc**: gera classes Java a partir de XML Schema
  - Sintaxe: **xjc opcoes <xml-schema ...> [-b <config.xjb> ...]**
  - Exemplos:  
`xjc -d generated -p com.acme.xml.gen Satelites.xsd`  
`xjc -d . -p com.argonavis.filmes FilmeFacade.xsd`  
`xjc schema1.xsd schema2.xsd -b bindings1.xjb -b bindings2.xjb`
  - Exemplos e detalhes de customização via **<xs:appinfo>** ou **arquivo de bindings**
    - <http://docs.oracle.com/javase/tutorial/jaxb/intro/custom.html>
- **schemagen**: gera XML Schema a partir de código Java
  - Sintaxe: **schemagen -cp classpath -d destino Arquivos.java ...**
  - Exemplo:  
`schemagen -cp classes -d generated src/pac/A.java src/B.java`
  - Por default, cada pacote é um namespace e XML Schema
  - Maior parte da configuração do XML Schema gerado pode ser feito usando anotações nas classes
  - XML Schema é gerado a partir de arquivos Java no JAX-RS (RESTful WS)



# Algumas anotações JAXB

- Usadas para configurar geração de XML Schema
- Anotações de pacote e classe
  - **@XmlSchema**: configura mapeamento de pacote-namespace e se elementos e atributos serão qualificados por default
  - **@XmlAttributeType**: configura mapeamento de propriedades (get/set) e atributos
- Anotações de classe/enum
  - **@RootElement**: associa elemento global ao tipo XML Schema mapeado à classe
- Anotações de propriedades (derivados de get/set) e atributos
  - **@XmlElement**: mapeia propriedade ou atributo Java a elemento XML
  - **@XmlAttribute**: mapeia propriedade ou atributo Java a atributo XML
  - **@XmlElementWrapper**: gera um elemento de agrupamento (tipicamente usado em coleções). Ex: `<objetos><objeto/><objeto/></objetos>`
  - **@Transient**: impede o mapeamento (tipicamente usado em atributos para evitar o duplo mapeamento causado ao mapear propriedades)
- Referência oficial (JAXB Tutorial) com exemplos de uso
  - <http://docs.oracle.com/javase/tutorial/jaxb/intro/customize.html>
  - <http://docs.oracle.com/javase/tutorial/jaxb/intro/examples.html>



# Exemplo de classe anotada

```

@XmlSchema(namespace = "http://filmes.agonavis.com/")
package com.agonavis.filmes.generated; ← Namespace do Schema  

                                         mapeado a pacote

@XmlElement
@XmlAccessorType(XmlAccessType.PUBLIC_MEMBER) ← PUBLIC_MEMBER é default. Mapeia propriedades e  

@XmlType(name = "sala", propOrder = {  

    "nome",  

    "lotacao",  

    "assentos",  

})  

public class Sala implements Serializable {  

    @XmlTransient ← Sem estas anotações default seria  

    private List<Assento> assentos;  

    private String nome;  

    private int lotacao;  

    @XmlElementWrapper ← Com anotações o resultado será  

    @XmlElement(name="assento") ←  

    public List<Assento> getAssentos() {  

        return assentos;  

    }  

...
}

```

Não irá persistir o atributo <assentos> (não precisa, pois já está mapeando <assentos> a getAssentos())

<sala>  
 <assentos>...</assentos>  
 <assentos>...</assentos>  
</sala>

<sala>  
 <assentos>  
 <assento>...</assento>  
 <assento>...</assento>  
 </assentos>  
</sala>



# JAXBContext

- Antes de iniciar a serialização XML (marshalling e unmarshaling) é necessário configurar o **JAXBContext**
- A forma mais simples é passando o(s) **pacote(s)** das classes geradas como argumento ou uma lista de classes

```
JAXBContext jc1 = JAXBContext.newInstance("com.acme.gen");  
JAXBContext jc2 =  
    JAXBContext.newInstance("pacote.um.gen:pacote.dois.gen:outro.pacote.gen");  
JAXBContext jc3 =  
    JAXBContext.newInstance(new Class[]{Filme.class, Evento.class});
```

- É possível passar propriedades de configuração e ter acesso a recursos proprietários do provedor JAXB usado
  - Exemplo: propriedade **media-type** do **EclipseLink Metro** configura o JAXB a produzir e consumir JSON em vez de XML

```
Map properties = new HashMap();  
properties.put("eclipselink.media-type", "application/json");  
JAXBContext jc3 =  
    JAXBContext.newInstance(new Class[]{Filme.class}, properties);
```



# JAXB Marshal / Unmarshal

- Unmarshalling converte XML em uma árvore de objetos Java

```
JAXBContext jc =
    JAXBContext.newInstance( "com.agonavis.filmes.gen" );
Unmarshaller u = jc.createUnmarshaller();
Filme filme = (Filme)u.unmarshal( new File( "tt1937390.xml" ) );
```

- Pode-se também criar novas instâncias usando os métodos do **ObjectFactory** gerado

```
Filme filme = ObjectFactory.createFilme();
```

- Marshalling converte uma árvore de objetos Java em XML

```
JAXBContext jc =
    JAXBContext.newInstance( "com.agonavis.filmes.gen" );
Filme filme = ObjectFactory.createFilme();
filme.setAno(2014); // alterando objeto
Marshaller m = jc.createMarshaller();
m.marshal( filme, System.out );
```



# JSON (JavaScript Object Notation)

- Formato de dados baseado em JavaScript
  - RFC 4627 <http://tools.ietf.org/search/rfc4627>
  - Tipo MIME: **application/json**
- Possui sete tipos de **valores**:
  - Duas **estruturas**: objeto e array
  - Dois tipos **escalares**: string (entre aspas) e número
  - Dois literais **booleanos**: os valores **true** e **false**
  - O valor **null**
- Estruturas
  - **Objeto**: coleção de pares nome:valor representados entre {} e separados por vírgula. Exemplo:

```
{id:123, cidade:"Paris", voos:["M344","J919"], pos:{S:9.75, W:40.33}}
```
  - **Array**: lista ordenada de valores representados entre [] e separados por vírgula. Exemplo:

```
[1, "texto", {x:233, y:991}, [0,0,1]]
```



# APIs JSON em Java

- JSON pode ser codificado diretamente em String (se for muito simples) e processado com métodos de String como split(), substring(), indexOf() e outros

```
String json = "{\"+ nome1+\":\"+valor1+\",\"+nome2+\":\"+valor2+\"}";  
String[] objetos = json.split("\\\\,\"");  
String nome = objetos[0].split("\\\\:")[0];  
String valor = objetos[0].split("\\\\:")[1];
```

- Java EE 7 disponibiliza uma API para construir objetos JSON e para converter strings JSON em mapas
  - É um par de APIs de baixo nível (não é mapeamento objeto-JSON)
- Existem várias implementações que fazem mapeamento (binding) objeto-JSON automático (não são parte do Java EE)
  - **MOXy, Jettison, Jersey, Jackson**, etc.



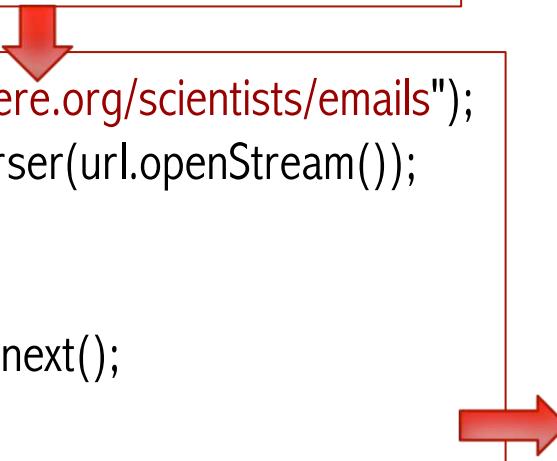
- JSON Streaming API (**javax.json.stream**)
  - Análogo a SAX: leitura sequencial (baixo nível)
  - **JsonParser** – permite ler um stream JSON e capturar eventos (ex: Event.KEY\_NAME, Event.START\_OBJECT, Event.START\_ARRAY, etc.)
  - **JsonGenerator** – métodos para criar uma estrutura JSON
- JSON Object Model API (**javax.json**)
  - Análogo a DOM: estrutura em árvore; I/O streaming via decorators
  - **JsonObject** – representa um objeto JSON – contém um Map; criado com JsonObjectBuilder
  - **JsonArray** – representa um array JSON – contém um List; criado com JsonArrayBuilder
  - Leitura e gravação usando **JsonReader** (um java.io.Reader) e **JsonWriter** (um java.io.Writer)
  - Tipos **JsonValue**: JsonObject, JsonArray, JsonString, JsonNumber



# JSON Streaming

```
[ {"nome" : "James Clerk Maxwell",  
  "email" : "maxwell@somewhere.org"},  
 {"nome" : "Nicola Tesla",  
  "email" : "tesla@somewhere.org"} ]
```

```
URL url = new URL("http://somewhere.org/scientists/emails");  
JsonParser parser = Json.createParser(url.openStream());  
  
while (parser.hasNext()) {  
    JsonParser.Event event = parser.next();  
    if (event == Event.KEY_NAME)  
        System.out.print(parser.getString() + "\n  ");  
    if (event == Event.VALUE_STRING)  
        System.out.println(parser.getString());  
}
```



```
nome  
James Clerk Maxwell  
email  
maxwell@somewhere.org  
nome  
Nicola Tesla  
email  
tesla@somewhere.org
```



# JSON Object Model

## ■ Lendo uma estrutura JSON

```
URL url = new URL("http://somewhere.org/scientists/emails");
JsonReader reader = Json.createReader(new InputStreamReader(url.openStream()));
JsonArray emails = reader.readArray();
for(int i = 0; i < emails.length(); i++) {
    System.out.println("nome: " + object.getJSONObject(i).getString("nome"));
    System.out.println("email: " + object.getJSONObject(i).getString("email"));
}
```

## ■ Criando uma estrutura JSON e enviando para a console

```
JsonArray emails = Json.createArrayBuilder()
    .add(Json.createObjectBuilder()
        .add("nome", "James Clerk Maxwell").add("email", "maxwell@somewhere.org"))
    .add(Json.createObjectBuilder()
        .add("nome", "Nicola Tesla").add("email", "tesla@somewhere.org"))
    .build();
JsonWriter out = Json.createWriter(System.out);
out.writeArray(emails);
```



# JSON-Java binding

- Binding possibilita a realização de transformação objeto-JSON-objeto de forma transparente (como acontece em JAXB ou JPA)
- Ainda não há uma solução padrão, mas maior parte das implementações aproveitam a API do JAXB
  - Usam `jaxbMarshaller.marshal()` para gerar JSON, e `jaxbUnmarshaller.unmarshal()` para ler JSON
  - Configuração varia entre implementações
- Implementações que suportam JSON binding
  - EcliseLink **MOXy** (implementação JAXB nativa no Glassfish 4)
  - **Jersey** (implementação de REST, JAXB e JSON) <http://jersey.java.net>
  - **Jettison** (implementação JAXB) <http://jettison.codehaus.org/>
  - **Jackson** (biblioteca JSON) <http://jackson.codehaus.org> - não usa JAXB no processo



- Para substituir o provedor JAXB default por MOXy:

- Inclua JARs do MOXy como dependências do projeto
  - Inclua arquivo **jaxb.properties** no **classpath**, contendo

```
javax.xml.bind.context.factory=org.eclipse.persistence.jaxb.JAXBContextFactory
```

- Configuração de JAXBContext

```
Map properties = new HashMap();
props.put("eclipselink.media-type", "application/json");
JAXBContext ctx = JAXBContext.newInstance(new Class[] { Produto.class }, props);
```

- JSON → Java

```
Unmarshaller u = ctx.createUnmarshaller();
Produto produto= u.unmarshal(new StreamSource("produto123.json"));
```

- Java → JSON

```
Marshaller m = ctx.createMarshaller();
m.marshal(produto, System.out);
```



# Exercícios: XML e JAXP

1. Use JAXP (DOM, SAX, StAX – qual o melhor neste caso?) para ler os documentos XML da pasta **filmes**, extrair seus **títulos** e **códigos do IMDB**
2. Crie um documento XML **novo** contendo todos os títulos dos filmes e código IMDB lidos no exercício anterior da forma

```
<filmes>
    <titulo imdb="yyy1">xxx1</titulo>
    <titulo imdb="yyy2">xxx2</titulo>
    ...
</filmes>
```

3. Qual a expressão XPath para localizar no XML resultante
  - O título do terceiro filme?
  - Todos os códigos IMDB



# Exercícios: JAXB e JSON

4. A pasta **/filmes** contém 10 arquivos XML que foram produzidos por outra aplicação. Eles são especificados pelo esquema **filmes.xsd**.
- Crie uma aplicação (standalone ou Web) que permita a conversão desses arquivos em objetos Java
  - Liste os objetos disponíveis e seus atributos
  - Permita a criação de novos objetos. Eles devem ser gravados como XML e aparecerem na lista de filmes disponíveis
  - Crie um método que retorne a lista de filmes como um XML único
5. Uma aplicação JavaScript precisa receber dados dos objetos Filme fornecidos por um método de um Managed Bean. Os objetos são retornados como uma List<Filme>.
- Crie um método que devolva essa lista como um string JSON (adapte o método criado no exercício anterior)



- Especificações
  - XML: <http://www.w3.org/TR/REC-xml/>
  - XML Namespaces <http://www.w3.org/TR/xml-names/>
  - XML Schema <http://www.w3.org/XML/Schema>
  - XPath <http://www.w3.org/TR/xpath/>
  - DOM <http://www.w3.org/TR/REC-DOM-Level-1/>
  - SAX <http://www.saxproject.org/>
  - OASIS WSS [https://www.oasis-open.org/committees/tc\\_home.php?wg\\_abbrev=wss](https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss)
  - JSON: descrição resumida: <http://json.org/>
  - JSON: especificação RFC 4627 <http://tools.ietf.org/search/rfc4627>
  - JSON Java API: <http://jcp.org/en/jsr/detail?id=353>
- Artigos, documentação e tutoriais
  - Helder da Rocha. Tutorial XML <http://argonavis.com.br/cursos/xml/x100/>
  - Helder da Rocha. Tutorial XPath [http://argonavis.com.br/cursos/xml/x100/x100\\_8\\_XPath.pdf](http://argonavis.com.br/cursos/xml/x100/x100_8_XPath.pdf)
  - Helder da Rocha. Tutorial XML Schema <http://argonavis.com.br/cursos/xml/x170/>
  - Oracle. Java Tutorial. JAXP (SAX, DOM, TrAX) <http://docs.oracle.com/javase/tutorial/jaxp>
  - Oracle. Java Tutorial. StAX <http://docs.oracle.com/javase/tutorial/jaxp/stax>
  - Oracle. Java Tutorial. JAXB <http://docs.oracle.com/javase/tutorial/jaxb>
  - Kotamraju, J. Java API for JSON Processing. Oracle, 2013. <http://www.oracle.com/technetwork/articles/java/json-1973242.html>
  - JSON & XML Java Binding (EclipseLink MOXY): <http://www.eclipse.org/eclipselink/moxy.php>
  - EclipseLink MOXY. “10 Using JSON Documents”, in “Developing JAXB Applications Using EclipseLink MOXY”, Release 2.4. 2013. <http://www.eclipse.org/eclipselink/documentation/2.4/moxy/json.htm>
  - Jersey. “Chapter 5. JSON Support”. <https://jersey.java.net/documentation/1.17/json.html>



3

Infraestrutura de sistemas  
Java EE

# SOAP Web Services

Aplicações Java usando JAX-WS e Java EE 7



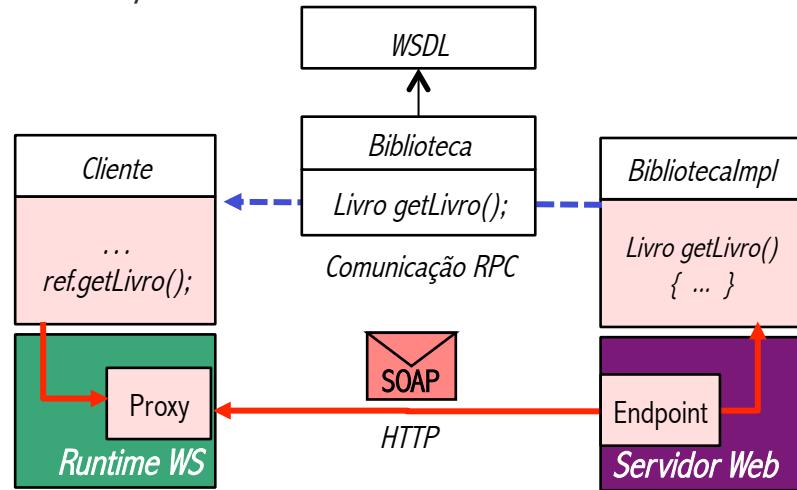
# Conteúdo

1. Introdução a SOAP Web Services
  - Fundamentos da tecnologia
  - Protocolos e mensagens: SOAP, WSDL
2. Como criar um serviço SOAP com EJB e JAX-WS
  - Como componente Web
  - Como session bean
3. Como criar um cliente SOAP usando JAX-WS
  - Tipos de clientes
  - Cliente estático
  - Clientes dinâmicos
4. Tópicos avançados
  - Handlers e MessageContext
  - WS-Addressing
  - WS-Security
  - SOAP sobre JMS



# SOAP Web Services

- Arquitetura de objetos distribuídos (método RPC – Remote Procedure Call ou Messaging/Notificação)
  - Objeto remoto implementa interface comum via proxy
  - Interface comum é expressa em uma IDL: **WSDL**
  - Cliente acessa objeto remoto transparentemente através do **proxy** em container (**runtime**) que se comunica com URI do serviço em um servidor Web (**endpoint**), usando um protocolo em XML (**SOAP**)
  - Dados e objetos são serializados (marshaled) em formato XML na transferência
  - Baseado em padrões: especificações **OASIS**, **W3C**, **WS-I**
- **SOAP** – Simple Object Access Protocol
  - Linguagem XML para requisições e respostas (**protocolo**)
- **WSDL** – Web Service Description Language
  - Linguagem XML usada para **descrever** interfaces (métodos remotos)
- **XML Schema**
  - Linguagem XML usado para **representar tipos de dados** usados na comunicação
  - XML Schema é usado dentro da WSDL



# Origens de SOAP WS

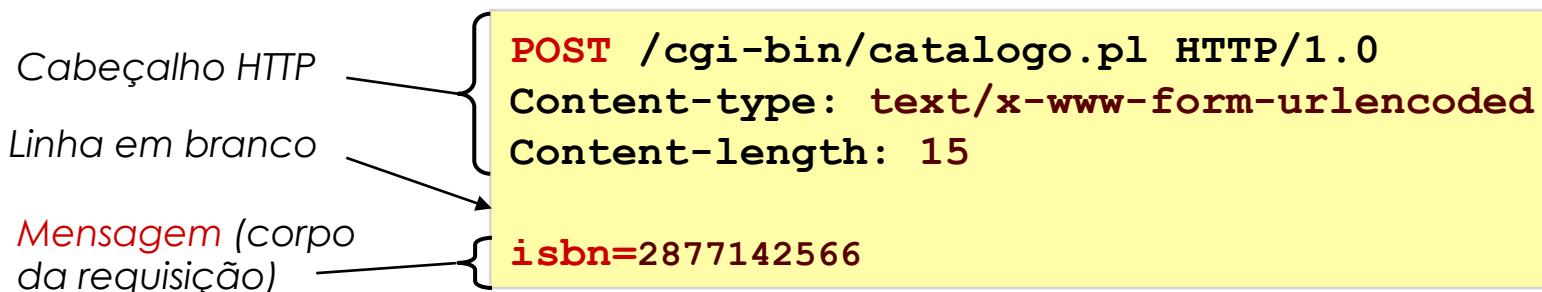
- Aproveitar requisição HTTP **POST**\*
  - Geralmente as portas do HTTP estão abertas nos Firewalls
- Clientes HTTP usam método POST para enviar dados
  - Tipicamente usado por browsers para enviar dados de formulários HTML e fazer upload de arquivos
  - Exemplo: formulário HTML



```

<FORM ACTION="/cgi-bin/catalogo.pl"
      METHOD="POST">
  <H3>Consulta preço de livro</H3>
  <P>ISBN: <INPUT TYPE="text" NAME="isbn">
  <INPUT TYPE="Submit" VALUE="Enviar">
</FORM>
  
```

- Requisição POST gerada pelo browser com dados do formulário



\* Web Services SOAP suportam outros protocolos de transporte mas HTTP tem maior interoperabilidade

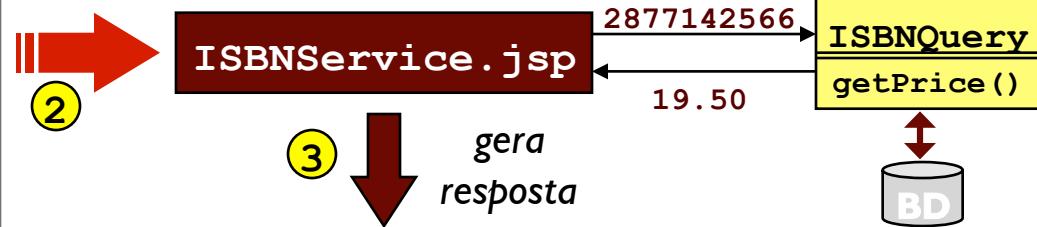


# Como criar um Web Service?

- Exemplo de um **serviço RPC simples** que troca mensagens XML via HTTP POST
  - Clientes e servidores compartilham esquema (DTD) simples que descreve as mensagens de chamada e resposta
  - Cliente envia requisições POST para servidor Web
  - Aplicação Web recebe requisições (JSF, JSP, ASP, PHP, servlet)

```
POST /ISBNService.jsp HTTP/1.0
Content-type: text/xml
Content-length: 90

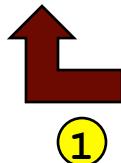
<chamada>
  <funcao>
    <nome>getPrice</nome>
    <param>2877142566</param>
  </funcao>
</chamada>
```



```
HTTP/1.1 200 OK
Content-type: text/xml
Content-length: 77

<resposta>
  <funcao>
    <param>19.50</param>
  </funcao>
</resposta>
```

gera  
requisição



ISBNClient



# XML-RPC

- Primeira especificação (1998) para RPC em XML via HTTP POST (criado no grupo de discussões xml-dev)
  - Projetada para ser a solução mais simples possível
  - Implementações em várias linguagens
- Exemplo anterior com XML-RPC (cabeçalhos HTTP omitidos)

```
<methodCall>
  <methodName>getPrice</methodName>
  <params>
    <param>
      <value><string>2877142566</string></value>
    </param>
  </param>
</methodCall>
```

Requisição

```
<methodResponse>
  <params>
    <param>
      <value><double>19.5</double></value>
    </param>
  </param>
</methodResponse>
```

Resposta



# Simples requisição SOAP-RPC

- XML-RPC evoluiu para SOAP
  - Requisição HTTP POST com serviço RPC em SOAP:

```
POST /xmlrpc-bookstore/bookpoint/BookstoreIF HTTP/1.0
Content-Type: text/xml; charset="utf-8"
Content-Length: 585
SOAPAction: ""

<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope
    xmlns:env="http://www.w3.org/2001/12/soap-envelope"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:enc="http://www.w3.org/2001/12/soap-encoding/"
    env:encodingStyle="http://www.w3.org/2001/12/soap-encoding/">
    <env:Body>
        <ans1:getPrice xmlns:ans1="http://mybooks.org/wsdl">
            <String_1 xsi:type="xsd:string">2877142566</String_1>
        </ans1:getPrice>
    </env:Body>
</env:Envelope>
```

Mensagem (envelope) SOAP

Parâmetro (ISBN)

Payload



# Resposta SOAP-RPC

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
SOAPAction: ""
Date: Thu, 08 Aug 2002 01:48:22 GMT
Server: Apache Coyote HTTP/1.1 Connector [1.0]
Connection: close

<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope
    xmlns:env="http://www.w3.org/2001/12/soap-envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:enc="http://www.w3.org/2001/12/soap-encoding/"
    xmlns:ns0="http://mybooks.org/types"
    env:encodingStyle="http://www.w3.org/2001/12/soap-encoding/">
    <env:Body>
        <ans1:getPriceResponse xmlns:ans1="http://mybooks.org/wsdl">
            <result xsi:type="xsd:decimal">19.50</result>
        </ans1:getPriceResponse>
    </env:Body>
</env:Envelope>
```

Mensagem  
(envelope)  
SOAP

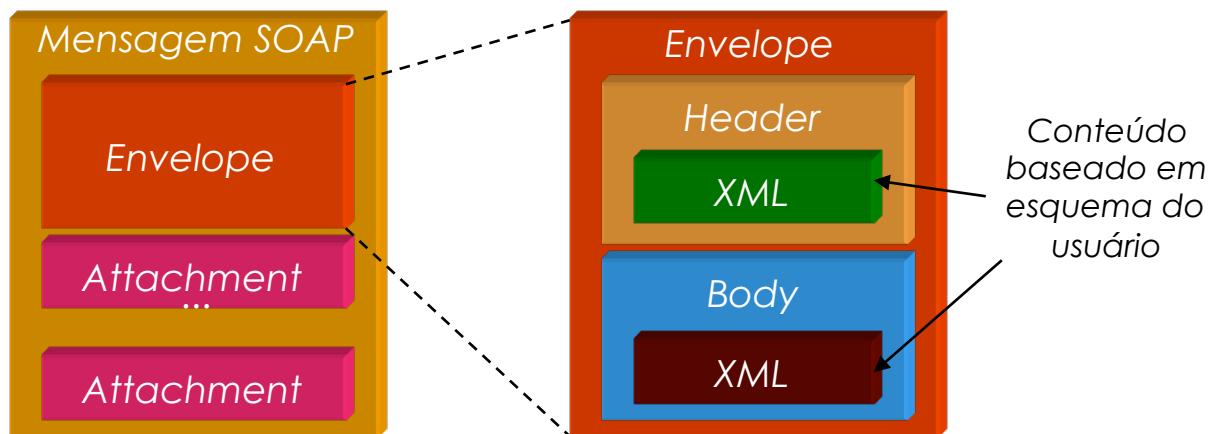
Resposta (Preço)

Payload



# SOAP

- Contém todas as informações necessárias à comunicação síncrona ou assíncrona
- SOAP não é um protocolo RPC
  - Um **par** de mensagens SOAP **pode** ser usado para RPC: operação tipo **request-response**; pode também ser **one-way** (há dois outros tipos)
  - Transporte pode ser HTTP, SMTP, ou outro
  - Mensagens podem conter qualquer coisa (texto, bytes): payload
  - É extensível
- Pode ser manipulado via APIs em Java: JAXP, SAAJ



# Estrutura do SOAP

- HTTP Content-type
  - **application/soap+xml**
- Namespace do Envelope:
  - "<http://www.w3.org/2001/12/soap-envelope>"
- **<Envelope>**
  - Contém **<Header>** (opcional, metadados) e **<Body>** (obrigatório, dados)
- Componentes de Body
  - Dados da mensagem: **payload** (declarados com namespace próprio)
  - **<Fault>** – mensagem de erro; contém **<faultcode>**, **<faultstring>**, **<faultactor>**, **<detail>**; faultcode “Client” indica erro no cliente, “Server” erro no servidor
- Formas de incluir anexos (**SOAP Attachments**) para dados binários
  - Incluídos **dentro** da mensagem SOAP como dados serializados usando **XOP** (XML-binary Optimized Packaging – formato base64): ineficiente, overhead > 30%
  - **MTOM** (Message Transmission Optimization Mechanism): anexos incluídos **fora** da mensagem SOAP(anexo no protocolo de transporte em container MIME) usando um placeholder XOP como anexo SOAP



# WSDL

- Usado para descrever (e localizar) um Web Service SOAP

- **<types>**

- Inclui XML Schema com definição de tipos usados no serviço

- **<message>**

- Descrevem as mensagens que são usadas na comunicação

- **<portType>**

- Descreve as operações suportadas por cada endpoint (4 tipos)

- **<operation>** descreve as mensagens que compõem uma operação

- **<binding>**

- Declara formato de dados e protocolo de transporte

- **<service>**

- Informa a localização do serviço

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions name="BookstoreService"
    targetNamespace="http://mybooks.org/wsdl"
    xmlns:tns="http://mybooks.org/wsdl"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <types>...</types>
    <message name="BookstoreIF_getPrice">
        <part name="String_1" type="xsd:string"/>
    </message>
    <message name="BookstoreIF_getPriceResponse">
        <part name="result" type="xsd:decimal"/>
    </message>
    <portType name="BookstoreIF">
        <operation name="getPrice" parameterOrder="String_1">
            <input message="tns:BookstoreIF_getPrice"/>
            <output message="tns:BookstoreIF_getPriceResponse"/>
        </operation>
    </portType>
    <binding ... > ...</binding>
    <service ... > ... </service> ← Informa onde está o serviço (endpoint)
</definitions>

```

Compare com a mensagem SOAP mostrada anteriormente



# Suporte Java

- APIs padrão Java
  - **JAX-WS** – API de **alto nível** para aplicações RPC com WSDL usando Web Services SOAP (é transparente e esconde todos os detalhes do XML)
  - **JAXP** – Java API for XML Processing – API para manipulação de XML em **baixo nível** (DOM, SAX, XML Schema, XSLT, XPath, etc.)
  - **SAAJ** – SOAP with Attachments for Java – API para construir e ler mensagens SOAP (usado em aplicações assíncronas)
  - **JAXB** – Java API for XML Binding – mapeamento Objeto-XML
- Implementações populares (aderem à especificação padrão **WS-I Basic Profile** e oferecem recursos adicionais)
  - Projeto **Metro** (Kenai): implementação de referência JAX-WS do Glassfish
  - Apache **CXF** (principal implementação usada em **JBossWS**)
- APIs e implementações antigas
  - JAX-RPC – API anterior a JAX-WS (parte do WS Dev Kit)
  - Apache Axis1/Axis2: implementações antigas (suportam JAX-RPC)



# Algumas ferramentas úteis

125

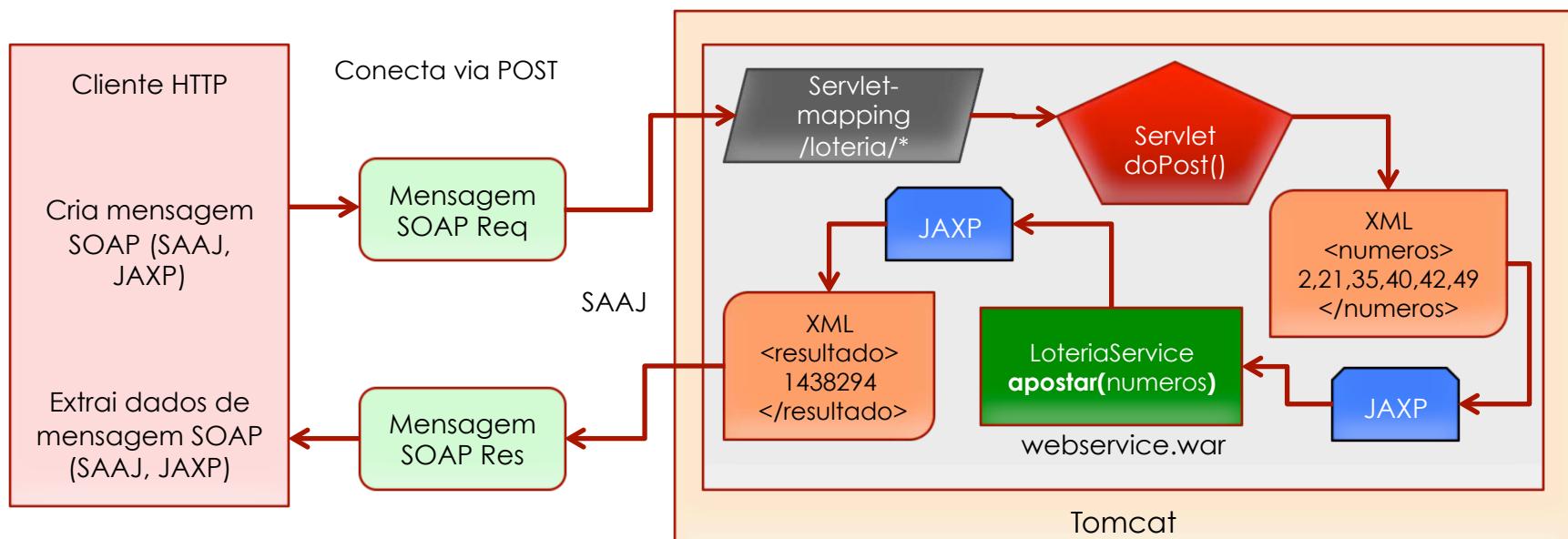
- Para **testar e depurar** Web Services SOAP sobre HTTP e REST
  - Browsers
    - Firebug <https://getfirebug.com>
    - Firefox RESTClient <https://addons.mozilla.org/en-US/firefox/addon/restclient>
    - Chrome Dev Tools <https://developers.google.com>
  - Aplicações gráficas
    - **Java**: REST Client <https://code.google.com/p/rest-client/>
    - **Windows**: Fiddler <http://fiddler2.com/>
    - **Linux** (e OS X): MITMProxy <http://mitmproxy.org/>
    - **OS X**: Cocoa REST Client <https://code.google.com/p/cocoa-rest-client/>
  - Para enviar requisições por linha de comando
    - cURL <http://curl.haxx.se/>
  - **Web proxies**
    - Wireshark, Squid, JMeter, Charles Proxy, WebScarab, ...
- Para **programar** clientes HTTP em Java
  - Apache HttpComponents (HttpClient) <http://hc.apache.org/>



# Como criar um serviço SOAP?

126

- Sem usar nenhuma API de Web Services, pode-se criar o serviço com um **servlet** mapeado a uma URL (**endpoint**)
  - Servlet implementa **doPost()** e extrai corpo da mensagem (XML) usando ferramentas XML como DOM ou SAX (JAXP)
  - A resposta requer a devolução de um documento XML SOAP, que pode ser gerado com JAXP ou simplesmente incluindo o string na **response** se for simples



- Criar um serviço SOAP desta forma serve como **aplicação didática**
  - Seria a melhor solução em 1998, mas hoje temos várias APIs para SOAP que facilitam o desenvolvimento e escondem totalmente o XML



# Como criar um serviço SOAP?

- Em Java, usando **JAX-WS** há duas alternativas
  - (1) Através da criação de um **componente Web**
  - (2) Através de um **EJB**
- (1) Componente Web
  - Criar **serviço** (SEI) e **interface** (opcional) configurado com **anotações**
  - Rodar **ferramentas** para gerar código
  - Empacotar classes compiladas em um **WAR** e fazer deploy em **servidor Web**
- (2) EJB
  - Criar **Stateless Session Bean** (SEI) configurado com **anotações**
  - Empacotar e fazer deploy do **EJB-JAR** ou **EAR** no **servidor de aplicações**
- Ferramentas de linha de comando disponíveis no **Java SE**
  - **wsimport**: recebe um **WSDL** e **gera código Java** usado para criar serviços e clientes
    - No JBossWS use **wsconsume** (em JBOSS\_HOME/bin – também tarefa Ant e Maven)
  - **wsgen**: recebe POJOs Java e gera artefatos JAX-WS, mapeamentos (JAXB) e WSDL
    - No JBossWS use **wsprovide** (em JBOSS\_HOME/bin – também tarefa Ant e Maven)



# Componente Web

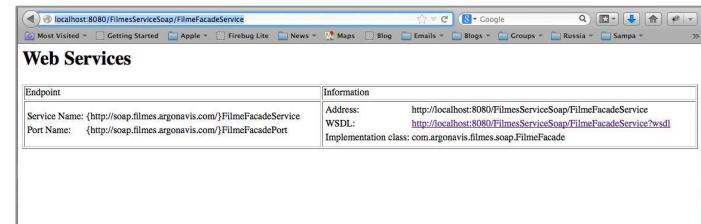
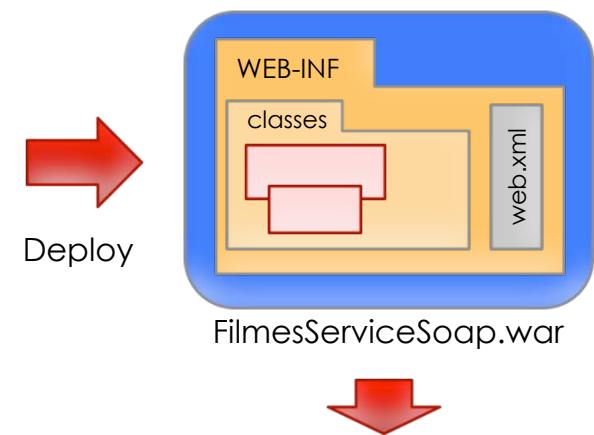
128

- O serviço é implementado em uma classe Java
  - Use anotação **@WebService** (javax.jws) para classe
    - Classe representa um **Service Endpoint Interface (SEI)**
    - Todos os métodos públicos de um SEI são automaticamente incluídos na interface do serviço
  - Empacote em um WAR e faça deploy

```
@WebService
public class FilmeFacade {
    @PersistenceContext(unitName = "FilmesServiceSoap")
    EntityManager em;

    public List<Filme> getFilmes() {
        String jpql = "select filme from Filme filme";
        Query query = em.createQuery(jpql);
        return (List<Filme>)query.getResultList();
    }

    @Entity
    public class Filme implements Serializable {
        @Id private Long id;
        private String titulo;
        private String diretor;
        private Long ano;
        private long duracao;
        private String imdb;
    }
}
```



Endpoint:  
<http://servidor/FilmesServiceSoap/FilmeFacadeService>



# Geração de código

- O servidor gera o código que implementa o Web Service na implantação do serviço, se ele tiver suporte nativo a JAX-WS
- O serviço também pode ser implantado em qualquer servidor com suporte a servlets. Nesse caso é preciso
  - Usar **ferramentas** que irão interpretar as anotações para gerar código Java e WSDL
  - Configurar no **web.xml** o servlet que irá receber as requisições SOAP (depende da implementação WS usada)
  - **Empacotar** todas as classes no WAR
- Exemplo
  - `wsgen -cp classpath com.argonavis.filmes.soap.FilmeFacade -wsdl -d genbin`
  - Para a implementação JBossWS (Apache CXF) use  
`wsprovide -c classpath --wsdl com.argonavis.filmes.soap.FilmeFacade -o genbin`

Classes geradas



```
pacote/jaxws/GetFilmes.class
pacote/jaxws/GetFilmesResponse.class
FilmeFacadeService.wsdl
FilmeFacadeService_schema1.xsd
```



# WSDL gerado

- Veja em <http://servidor/nome-do-war/NomeDaClasseService?wsdl>

```
<definitions targetNamespace="http://soap.filmes.agonavis.com/" name="FilmeFacadeService">
  <types> ... </types>
  <message name="getFilmes">
    <part name="parameters" element="tns:getFilmes"/>
  </message>
  <message name="getFilmesResponse">
    <part name="parameters" element="tns:getFilmesResponse"/>
  </message>
  <portType name="FilmeFacade">
    <operation name="getFilmes">...</operation>
  </portType>
  <binding name="FilmeFacadePortBinding" type="tns:FilmeFacade">... </binding>
  <service name="FilmeFacadeService">
    <port name="FilmeFacadePort" binding="tns:FilmeFacadePortBinding">
      <soap:address location="http://localhost:8080/FilmesServiceSoap/FilmeFacadeService">
    </port>
  </service>
</definitions>
```



# Tipos de dados <types>

131

- Inclui um **XML Schema** declarando tipos de dados
  - Operações (parâmetros na requisição, tipo de retorno na resposta)
  - Descrição de tipos enviados como parâmetros ou retorno

```
<xs:schema version="1.0" targetNamespace="http://soap.filmes.agonavis.com/">

    <xs:element name="getFilmes"          type="tns:getFilmes"/>
    <xs:element name="getFilmesResponse" type="tns:getFilmesResponse"/>

    <xs:complexType name="filme">
        <xs:sequence>
            <xs:element name="ano"      type="xs:long"   minOccurs="0"/>
            <xs:element name="diretor" type="xs:string" minOccurs="0"/>
            <xs:element name="duracao" type="xs:long"/>
            <xs:element name="id"      type="xs:long"   minOccurs="0"/>
            <xs:element name="imdb"    type="xs:string" minOccurs="0"/>
            <xs:element name="titulo"  type="xs:string" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="getFilmes"> <xs:sequence/> </xs:complexType>

    <xs:complexType name="getFilmesResponse">
        <xs:sequence>
            <xs:element name="return" type="tns:filme" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```



# Anotações do JAX-WS

- Além de **@WebService** (única anotação obrigatória, várias outras anotações (em javax.jws.\* e javax.jws.soap.\*) podem ser usadas para configurar o serviço)
- Na classe
  - **@SOAPBinding** – especifica mapeamento WS↔SOAP
  - **@BindingType** – especifica tipo de mapeamento
  - **@HandlerChain** – associa o Web Service a uma cadeia de handlers
- No método
  - **@WebMethod** – configura métodos da interface
  - **@OneWay** – declara método uma operação sem retorno (só mensagem de ida)
- Nos parâmetros de um método
  - **@WebParam** – configura nomes dos parâmetros
- Nos valores de retorno de um método
  - **@WebResult** – configura nome e comportamento



# @OneWay

- Métodos anotados com **@OneWay** têm apenas mensagem de requisição
  - Pode ser usada em métodos que retornam void
- Exemplos

```
@WebMethod  
@OneWay  
public void enviarAvisoDesligamento() {
```

```
    ...  
}
```

```
@WebMethod  
@OneWay  
public void ping() {  
    ...  
}
```



# @WebParam e @WebResult

- Essas anotações permitem configurar o WSDL que será gerado e o mapeamento entre o SEI e o SOAP
- **@WebResult** serve para configurar o elemento XML de retorno
  - No exemplo abaixo, a resposta estará dentro de um elemento `<filme>`; o default é `<return>`.

```
@WebMethod
```

```
@WebResult(name="filme")
```

```
public Filme getFilme(String imdbCode) {  
    return getFilmeObject(imdbCode);  
}
```

- **@WebParam** permite configurar nomes dos parâmetros
  - No exemplo abaixo, o parâmetro da operação getFilme no SOAP e WSDL é `imdb`. Seria `imdbCode` se o @WebParam não estivesse presente

```
@WebMethod
```

```
public Filme getFilme(@WebParam(name="imdb") String imdbCode) {  
    return getFilmeObject(imdbCode);  
}
```



- Métodos de WebServiceContext
  - MessageContext **getMessageContext()**: retorna objeto MessageContext que permite acesso a metadados da mensagem (cabeçalhos, porta, serviço, info do servlet, etc.)
  - Principal **getUserPrincipal()**: permite acesso ao javax.security.Principal do usuário autenticado
  - boolean **isUserInRole(String role)**: retorna true se usuário autenticado faz parte de um grupo de autorizações (role)
- Exemplo

```
@Resource  
private WebServiceContext ctx;  
  
@WebMethod()  
public String metodoSeguro(String msg) {  
    String userid = ctx.getUserPrincipal().getName();  
    if (userid.equals("czar")) {  
        ...  
    } else if (ctx.isUserInRole("admin")) {  
        ...  
    }  
}
```



# @SOAPBinding

- Permite configurar o estilo do mapeamento entre a mensagem SOAP e o Web Service. É opcional.
  - Afeta o formato do WSDL gerado e mensagens SOAP

- Atributos

- **style**

- SOAPBinding.Style.RPC ou
    - SOAPBinding.Style.DOCUMENT (default)

“RPC” aqui é um **estilo do protocolo** SOAP (não tem nada a ver com o modelo de programação RPC: tanto DOCUMENT como RPC são usados em comunicação request-response)

- **use**

- SOAPBinding.Use.ENCODED ou
    - SOAPBinding.Use.LITERAL (default)

O artigo “Which style of WSDL should I use?” descreve como esses atributos alteram o formato do SOAP e WSDL:  
[www.ibm.com/developerworks/webservices/library/ws-whichwsdl/](http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/)

- **parameterStyle**

- SOAPBinding.ParameterStyle.BARE
    - SOAPBinding.ParameterStyle.WRAPPED (default)

- Exemplo: use para anotar o SEI (todos os atributos são opcionais)

```
@SOAPBinding(style=SOAPBinding.Style.DOCUMENT,
              use=SOAPBinding.Use.LITERAL,
              parameterStyle=SOAPBinding.ParameterStyle.WRAPPED)
```



# APIs de serviços no JAX-WS

- Java Service Endpoint Interface - **SEI** (default)
  - Alto nível: usa WSDL para esconder detalhes da comunicação (a programação do cliente e servidor podem ignorar o XML)
- Endpoint **Provider** Interface
  - Baixo nível: trabalha diretamente com as mensagens XML
  - Serviço pode trabalhar apenas com o payload das mensagens (dados contidos no envelope SOAP) em vez de usar a mensagem inteira
    - Modos de serviço @ServiceMode: MESSAGE e PAYLOAD (default)
- Exemplo

```
@WebServiceProvider
@ServiceMode(value=Service.Mode.PAYLOAD)
public class MyService implements Provider<Source> {
    public Source invoke(Source request) {
        Source requestPayload = request.getPayload();
        String response = "<filmes><filme><titulo>...</filme></filmes>";
        StreamSource responsePayload = new StreamSource(new StringReader(response));
        return responsePayload;
    }
}
```



# Stateless remote (SOAP) bean

- Forma mais simples de criar e implantar um serviço
  - SEI é implementado por um stateless session bean
- Resultado idêntico ao Web Service via componente Web
  - Permite acesso remoto via porta HTTP
  - Exporta WSDL
- Para criar
  - Declare uma classe com `@Stateless`
  - Adicione a anotação `@WebService`

Declaração explícita de um **Service Endpoint Interface** que declara os métodos que serão expostos é **opcional**

```
@Stateless @WebService(endpointInterface="LoteriaWeb")
public class LoteriaWebBean implements LoteriaWeb {
    @Override
    public int[] numerosDaSorte() {
        int[] numeros = new int[6];
        for(int i = 0; i < numeros.length; i++) {
            int numero = (int)Math.ceil(Math.random() * 60);
            numeros[i] = numero;
        }
        return numeros;
    }
}
```

```
@WebService
interface LoteriaWeb {
    int[] numerosDaSorte();
}
```



# Cientes SOAP

139

- Podem ser criados de várias formas, em Java ou em outras linguagens
  - Formas mais simples consiste em gerar código **estaticamente** compilando o WSDL ou usar um container do fabricante
  - Outras estratégias permitem gerar stubs, proxies e classes **dinamicamente**, ou ainda usar reflection para chamar a interface dinamicamente
- Exemplo de cliente (estático) Java

```
public class WSClient {  
  
    public static void main(String[] args) throws Exception {  
  
        Classes  
        geradas  
  
        LoteriaWebService  
        = stub  
  
        LoteriaWeb  
        = proxy  
  
        LoteriaWebService service = new LoteriaWebService();  
        LoteriaWeb port = service.getLoteriaWebPort();  
        List<String> numeros = port.numerosDaSorte();  
  
        System.out.println("Numeros a jogar na SENA:");  
        for(String numero : numeros) {  
            System.out.println(numero);  
        }  
    }  
}
```



# Compilação do WSDL

- A ferramenta **wsimport** (Java) ou **wsconsume** (CXF) gera artefatos Java necessários aos clientes compilando o WSDL
  - Classes (DTO) representando tipos usados (parâmetros e valores de retorno)
  - Implementação do stub, representando o serviço
  - Implementação do port, representando o endpoint
- Geração de código com **wsimport**
  - `wsimport -keep -s gensrc -d genbin -p com.argonavis.filmes.client.soap.generated http://localhost:8080/FilmesServiceSoap/FilmeFacadeService?wsdl`
- Geração de código com **wsconsume**
  - `wsconsume.sh -k -s gensrc -o genbin -p com.argonavis.filmes.client.soap.generated http://localhost:8080/FilmesServiceSoap/FilmeFacadeService?wsdl`
- Classes geradas (inclua no classpath do cliente)

**Filme.class**  
    **FilmeFacade.class**  
**FilmeFacadeService.class**

Estas classes são as que o cliente precisará usar para utilizar o serviço remoto

**GetFilmes.class**  
    **GetFilmesResponse.class**  
    **ObjectFactory.class**  
**package-info.class**



# Exemplo de cliente SOAP

```
public class FilmeClient {  
    public static void main(String[] args) {  
        FilmeFacadeService service = new FilmeFacadeService();  
        FilmeFacade proxy = service.getFilmeFacadePort();  
        listarFilmes(proxy.getFilmes());  
    }  
  
    public static void listarFilmes(List<Filme> filmes) {  
        for(Filme f : filmes) {  
            System.out.println(f.getImdb()+" : " + f.getTitulo() + "(" + f.getAno() + ")");  
            System.out.println("           " + f.getDiretor());  
            System.out.println("           " + f.getDuracao() + " minutos\n");  
        }  
    }  
}
```

- Classes em destaque são classes geradas pela ferramenta wsimport (ou wsconsume)



```
$ java -jar FilmeClient.jar  
tt0081505: The Shining(1980)  
          Stanley Kubrick  
          144 minutos  
  
tt1937390: Nymphomaniac(2013)  
          Lars von Trier  
          330 minutos  
  
tt0069293: Solyaris(1972)  
          Andrei Tarkovsky  
          167 minutos  
  
tt1445520: Hearat Shulayim(2011)  
          Joseph Cedar
```



# Tipos de clientes

142

- Clientes podem ser mais dinâmicos. Duas estratégias
  - **Proxy dinâmico**: tem cópia local da interface do serviço, mas gera código em tempo de execução através do WSDL remoto
  - **Cliente totalmente dinâmico**: não depende de interface, WSDL ou quaisquer artefatos gerados para enviar requisições e obter resposta, mas é necessário trabalhar **no nível das mensagens XML**
- Cliente com proxy dinâmico

```
URL wsdl = new URL("http://servidor/app/AppInterfaceService?wsdl");
QName nomeServ = new QName("http://app.ns/", "AppInterfaceService");
Service service = Service.create(wsdl, nomeServ);
AppInterface proxy = service.getPort(AppInterface.class);
```

- Cliente 100% dinâmico (**Dispatch client** – trecho)

```
Dispatch<Source> dispatch = service.createDispatch(portName, Source.class, Service.Mode.PAYLOAD);
String reqPayload =
    "<ans1:getFilmes xmlns:ans1=\"http://soap.filmes.agonavis.com/\"></ans1:getFilmes>";
Source resPayload = dispatch.invoke(new StreamSource(new StringReader(reqPayload)));
DOMResult domTree = new DOMResult();
TransformerFactory.newInstance().newTransformer().transform(resPayload, domTree);
Document document = (Document)domTree.getNode();
Element root = document.getDocumentElement();
Element filmeElement = (Element)root.getElementsByTagName("return").item(0); // <return>...</return>
String tituloDoFilme = filmeElement.getElementsByTagName("titulo").item(0)
    .getFirstChild().getTextContent(); // <titolo> CONTEUDO </titolo>..."
```



# Exemplo de cliente dinâmico

- Proxy dinâmico
  - Compare com o cliente estático mostrado anteriormente

```
public class FilmeDynamicClient {

    public static void main(String[] args) throws MalformedURLException {
        URL wsdlLocation =
            new URL("http://localhost:8080/FilmesServiceSoap/FilmeFacadeService?wsdl");
        QName serviceName =
            new QName("http://soap.filmes.argonavis.com/", "FilmeFacadeService");
        Service service = Service.create(wsdlLocation, serviceName);

        FilmeFacade proxy = service.getPort(FilmeFacade.class);
        listarFilmes(proxy.getFilmes());
    }

    public static void listarFilmes(List<Filme> filmes) {
        for(Filme f : filmes) {
            System.out.println(f.getImdb() + ": " + f.getTitulo() + "(" + f.getAno() + ")");
            System.out.println("                " + f.getDiretor());
            System.out.println("                " + f.getDuracao() + " minutos\n");
        }
    }
}
```

Veja outro tipo de cliente dinâmico em  
FilmeDispatchClient.java



# Cliente em container

144

- Cientes localizados em container Java EE (ex: servlet ou managed bean) podem injetar o serviço através da anotação **@WebServiceRef**

```
@ManagedBean(name = "filmesBean")
public class FilmesManagedBean {
    @WebServiceRef(wsdlLocation=
        "http://localhost:8080/FilmesServiceSoap/FilmeFacadeService?wsdl")
    private FilmeFacadeService service;
    private List<Filme> filmes;

    @PostConstruct
    public void init() {
        FilmeFacade proxy =
            service.getFilmeFacadePort();
        this.filmes = proxy.getFilmes();
    }
    ...
}
```

IMDB	Título	Diretor	Ano	Duração
tt0081505	The Shining	Stanley Kubrick	1980	144 min
tt1527186	Melancolia	Lars von Trier	2011	136 min
tt0075686	Annie Hall	Woody Allen	1977	93 min
tt0060390	Fahrenheit 451	François Truffaut	1966	112 min
tt0013442	Nosferatu, eine Symphonie des Grauens	F. W. Murnau	1922	81 min
tt0071129	Amarcord	Frederico Fellini	1973	123 min
tt0069221	A Clockwork Orange	Stanley Kubrick	1972	136 min
tt0101765	La double vie de Véronique	Krzysztof Kieslowski	1991	98 min
tt0069293	Solyaris	Andrei Tarkovsky	1972	167 min
tt1832382	Jodaeyle Nader az Simin	Asghar Farhadi	2011	123 min
tt1445520	Hearat Shulayim	Joseph Cedar	2011	103 min

## JSF Managed Bean

```
<h1>Lista de Filmes</h1>
<h: dataTable value="#{filmesBean.filmes}" var="filme">
    <h: column>
        <f: facet name="header">IMDB</f: facet>
        <a href="http://www.imdb.com/title/# {filme.imdb} ">
            # {filme.imdb}</a>
    </h: column>
    <h: column>
        <f: facet name="header">Título</f: facet>
        # {filme.titulo}
    </h: column>
    ...
</h: dataTable>
```

## Facelets



# MessageContext

- Objeto que permite acesso a metadados de uma mensagem
  - Cabeçalhos SOAP, se mensagem é inbound ou outbound, dados do WSDL, servidor, etc.
- Pode ser obtida do WebServiceContext

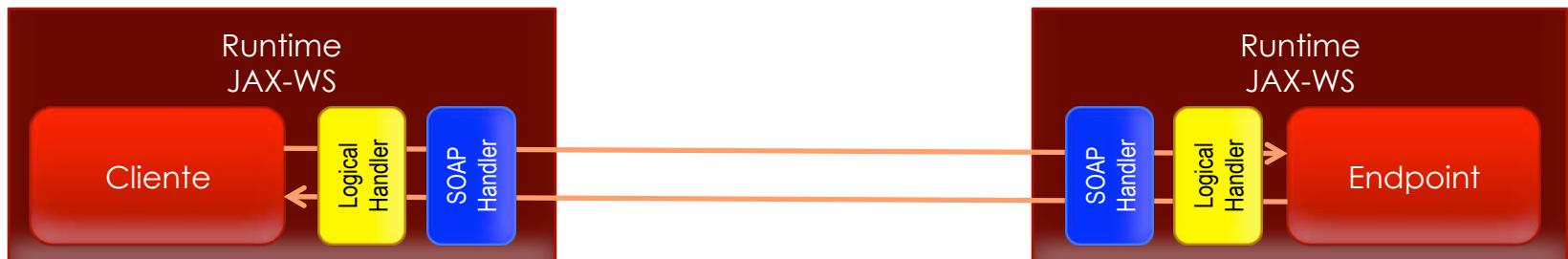
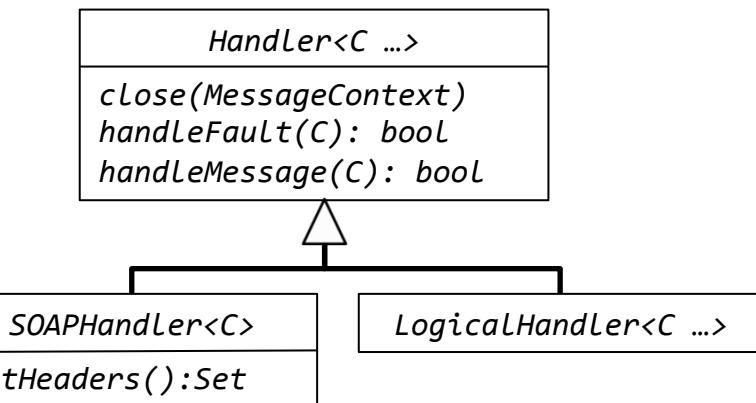
```
@Resource  
WebServiceContext wsctx;  
  
@WebMethod  
public void metodo() {  
    MessageContext ctx = wsContext.getMessageContext();  
    Map headers = (Map)ctx.get(MessageContext.HTTP_REQUEST_HEADERS);  
    ...  
}
```

- Algumas propriedades
  - MESSAGE\_OUTBOUND\_PROPERTY (boolean)
  - INBOUND\_MESSAGE\_ATTACHMENTS (Map)
  - HTTP\_REQUEST\_METHOD (String)
  - WSDL\_OPERATION (Qname)



# JAX-WS Handlers

- Interceptadores (filtros) de mensagens
  - Dois tipos: **protocol** (MESSAGE) e **logical** (PAYLOAD)
  - Podem ser usados para fazer alterações na mensagem antes de ser processada no servidor ou cliente
  - Geralmente são configurados em **corrente**
- Para criar
  - Implementar **SOAPHandler** ou **LogicalHandler**
  - Um XML para configurar a corrente
- Para usar
  - Anotar com **@HandlerChain(file="handlers.xml")** o SEI (no servidor) ou stub do serviço (código gerado no cliente)



# SOAP Handler

- Implementa javax.xml.ws.handler.soap.SOAPHandler
  - Interceptador para a mensagem inteira (permite acesso a cabeçalhos da mensagem)

```
public class MyProtocolHandler implements SOAPHandler {  
  
    public Set getHeaders() { return null; }  
  
    public boolean handleMessage(SOAPMessageContext ctx) {  
        SOAPMessage message = ctx.getMessage();  
        // fazer alguma coisa com a mensagem  
        return true;  
    }  
  
    public boolean handleFault(SOAPMessageContext ctx) {  
        String operacao = (String)  
            ctx.get(MessageContext.WSDL_OPERATION);  
        // logar nome da operacao que causou erro  
        return true;  
    }  
  
    public void close(MessageContext messageContext) {}  
}
```



# Logical Handler

- Implementa javax.xml.ws.handler.LogicalHandler
  - Acesso ao payload da mensagem

```
public class MyLogicalHandler implements LogicalHandler {  
  
    public boolean handleMessage(LogicalMessageContext ctx) {  
        LogicalMessage msg = context.getMessage();  
        Source payload = msg.getPayload(); // XML Source  
        Transformer transformer =  
            TransformerFactory.newInstance().newTransformer();  
        Result result = new StreamResult(System.out);  
        transformer.transform(payload, result); // Imprime XML na saida  
    }  
  
    public boolean handleFault(LogicalMessageContext ctx) {  
        String operacao = (String)  
            ctx.get(MessageContext.WSDL_OPERATION);  
        // logar nome da operacao que causou erro  
        return true;  
    }  
  
    public void close(MessageContext context) {}  
}
```



# Usando handlers

- Arquivo de configuração (handlers-chains.xml)

```
<javaee:handler-chains xmlns:javaee="http://java.sun.com/xml/ns/javaee"  
                         xmlns:xsd="http://www.w3.org/2001/XMLSchema">  
    <javaee:handler-chain>  
        <javaee:handler>  
            <javaee:handler-class>com.acme.MyProtocolHandler</javaee:handler-class>  
        </javaee:handler>  
        <javaee:handler>  
            <javaee:handler-class>com.acme.MyLogicalHandler</javaee:handler-class>  
        </javaee:handler>  
    </javaee:handler-chain>  
</javaee:handler-chains>
```

- Uso no servidor (no SEI)

```
@WebService  
@HandlerChain(file="handler-chains.xml")  
public class FilmesFacade { ... }
```

- Uso no cliente (edite classe gerada que implementa o Service)

```
@HandlerChain(file="handler-chains.xml")  
public class FilmesFacadeService extends Service { ... }
```



# Outros padrões

- Alguns padrões relacionados SOAP Web Services
  - **WS-I Basic Profile** – conjunto de especificações que promovem a interoperabilidade
  - Especificações de messaging: **WS-Notification**, **WS-Addressing**, **WS-ReliableMessaging**, **WS-ReliableMessagingPolicy**
  - Especificações de metadados: **WS-Policy**, **WS-Transfer**, **WS-MetadataExchange**, **UDDI**
  - Especificações de segurança: **WS-Security**, **WS-SecureConversation**, **WS-Trust**, **WS-SecurityPolicy**
  - Especificações de transações: **WS-Coordination**, **WS-AtomicTransaction**
- Vários especificações estão implementadas nos provedores de Web Services existentes no Glassfish e JBoss
  - JAX-WS abrange apenas às especificações relacionadas comunicação SOAP



# WS-Addressing

- Padronização de propriedades usadas no roteamento de mensagens SOAP
  - Propriedades: <To>, <From>, <ReplyTo>, <FaultTo>, <MessageID>, <Action>, <RelatesTo>
- Exemplo de uso em envelope SOAP

```
<soap:Envelope...>
  <soap:Header>
    <wsa:To> http://host/WidgetService </wsa:To>

    <wsa:ReplyTo>
      <wsa:Address>
        http://schemas.xmlsoap.org/ws/2003/03/addressing/role/anonymous
      </wsa:Address>
    </wsa:ReplyTo>

    <wsa:FaultTo>
      <wsa:Address>
        http://client/myReceiver
      </wsa:Address>
    </wsa:FaultTo>
    ...
  </soap:Header>
  <soap:Body> ... </soap:Body>
</soap:Envelope>
```

Fonte: <http://www.ibm.com/developerworks/webservices/library/ws-address/index.html>



# WS-Security

- Segurança na camada de mensagens
  - Garante integridade e confidencialidade
  - Permite criptografia de partes de mensagem SOAP
- Ainda não há uma implementação Java padrão. Alternativas:
  - **Projeto Metro** (Kenai) do **Glassfish**: metro.java.net disponibiliza uma implementação WSS em **xwss.java.net**
  - **Fundação Apache** tem uma implementação WSS4J em **ws.apache.org/wss4j**
- Exemplo de mensagem SOAP com WSS

```
<wsse:UsernameToken wsu:id="uuid_faf0159a-6b13-4139-a6da-cb7b4100c10c">
  <wsse:Username>Alice</wsse:Username>
  <wsse:Password Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordDigest">
    6S3P2EWNP3IQf+9VC3emNoT57oQ=</wsse:Password>
  <wsse:Nonce EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#Base64Binary">
    YF6j8V/CAqi+1nRsGLRbuZhi</wsse:Nonce>
  <wsu:Created>2008-04-28T10:02:11Z</wsu:Created>
</wsse:UsernameToken>
```

Fonte: [https://blogs.oracle.com/ashutosh/entry/hash\\_password\\_support\\_and\\_token](https://blogs.oracle.com/ashutosh/entry/hash_password_support_and_token)



# SOAP-JMS

- Enviar SOAP em mensagens JMS poderia resultar em **maior escalabilidade e confiabilidade** que sobre HTTP
  - Ainda não é padrão Java EE (mas há especificação no W3C)
  - Provedores de serviços JAX-WS (Apache CXF e Oracle WebLogic) já implementam o serviço (de forma proprietária) em seus servidores
- Desvantagens:
  - **Perde-se interoperabilidade** – JMS não é um padrão de protocolo de transporte como HTTP – é uma API **Java** – SOAP/JMS é uma solução Java
  - Complexidade maior na implementação de serviços requisição-resposta já que o mecanismo JMS é assíncrono – melhor para serviços **@OneWay**
- Soluções
  - Apache CXF (mais fácil de usar e menos preso à implementação)
    - <http://cxf.apache.org/docs/jms-transport.html>
    - <http://cxf.apache.org/docs/soap-over-jms-10-support.html>
  - Oracle WebLogic
    - <http://www.oracle.com/technetwork/articles/murphy-soa-jms-092653.html>



# Exercícios

1. Escreva um SOAP Web Service para acesso a uma lista de Produtos. Use qualquer modelo (Provider ou SEI, WAR ou EJB). Crie as seguintes operações

- listarProdutos
- detalharProduto
- criarProduto

Produto
<b>id:</b> long
<b>descricao:</b> string
<b>preco:</b> double

2. Escreva um cliente Web que utilize o Web Service criado no exercício anterior
3. Escolha um Web Service simples abaixo, use o WSDL para gerar código e escreva um cliente para testá-lo (retornam apenas o Payload XML)
- <http://ws.cdyne.com/ip2geo/ip2geo.asmx>
  - <http://wsf.cdyne.com/WeatherWS/Weather.asmx>
  - <http://www.currencyserver.de/webservice/currencyserverwebservice.asmx>



# Referências

155

- Especificações
  - WSDL <http://www.w3.org/TR/wsdl>
  - SOAP <http://www.w3.org/TR/soap/>
  - MTOM <http://www.w3.org/TR/soap12-mtom/>
  - XOP <http://www.w3.org/TR/xop10/>
  - WS-Addressing <http://www.w3.org/TR/ws-addr-core/>
  - WS-Security  
<http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf>
  - WS-I Basic Profile <http://ws-i.org/Profiles/BasicProfile-1.2-2010-11-09.html>
  - JAX-WS <https://jax-ws.java.net/>
  - SOAP sobre JMS <http://www.w3.org/TR/soapjms/>
- Artigos e tutoriais
  - Russell Butek. "Which style of WSDL should I use?" IBM Developerworks, 2005.  
<http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/>
  - Jaromir Hamala. "SOAP over JMS between WebLogic 12 and Apache CXF" C2B2, 2013.  
<http://blog.c2b2.co.uk/2013/09/soap-over-jms-between-weblogic-12-and.html>
  - Como implementar a interface Provider (Apache CXF documentation).  
<http://cxf.apache.org/docs/provider-services.html>
  - Rama Pulavarthi. "Introduction to handlers in JAX-WS". Java.net articles.  
[https://jax-ws.java.net/articles/handlers\\_introduction.html](https://jax-ws.java.net/articles/handlers_introduction.html)
  - JAX-WS WS-Addressing <https://jax-ws.java.net/nonav/jax-ws-21-ea2/docs/wsaddressing.html>



4

Infraestrutura de sistemas  
Java EE

# RESTful Web Services

Usando JAX-RS e Java EE 7



# Conteúdo

1. Introdução a REST
  - Arquitetura REST
  - Fundamentos de WebServices usando REST
2. Como criar um serviço RESTful usando JAX-RS
  - MediaTypes, resources, parâmetros
  - Anotações @Path, @PathParam, etc.
  - Metadados, ResponseBuilder e UriBuilders
  - Estratégias de segurança e contextos
3. Como criar um cliente RESTful
  - Usando comunicação HTTP simples
  - Usando APIs RESTEasy e Jersey
  - Clientes RESTful em aplicativos Web e mobile
  - WADL e JAX-RS 2.0



# REST (Fielding, W3C TAG, 2000)

## ■ Representational State Transfer

- **Estilo arquitetônico** baseado na World Wide Web
- Recursos (páginas) em arquitetura que facilita a transferência de estado no cliente (navegação via links, hierarquias, métodos HTTP) usando representação uniforme (URI, tipos MIME, representações)
- A World Wide Web é uma implementação de arquitetura REST
- As especificações HTTP 1.1 e URI (Uniform Resource Identifier) foram escritas em acordo com a arquitetura REST

## ■ Web Services baseados em REST (RESTful WS)

- Adotam a arquitetura REST, representação de URLs, tipos MIME, HTTP e restrições (estado, cache, etc) para fornecer infraestrutura que permite um ambiente de computação distribuída eficiente, simples e com overhead mínimo



# HTTP (RFC 2616)

- Protocolo de requisição-resposta sem estado
- Mensagem de requisição consiste de
  - linha inicial composta de método, URI e versão
  - linhas de cabeçalho propriedade:valor + linha em branco
  - corpo opcional (depende do método)
  - **Métodos:** GET, HEAD, POST, PUT, DELETE, TRACE, OPTIONS, CONNECT
- **Mensagem de resposta** consiste de
  - linha inicial composta de versão, código de status e mensagem
  - linhas de cabeçalho propriedade:valor + linha em branco
  - corpo opcional (depende do método)
  - **Códigos:** 1xx (informação), 2xx (sucesso), 3xx (redirecionamento), 4xx (erro no cliente) e 5xx (erro no servidor)



# RESTful Web Services

- Aproveitam a infraestrutura de um website
  - recursos (páginas), árvore de navegação (links, hierarquia)
  - representações de dados e tipos (URLs, MIME)
  - vocabulário de operações do protocolo HTTP (GET, POST, ...)
- URLs representam objetos (com estado e relacionamentos aninhados)
  - Um objeto contendo relacionamentos  
**pais.estado.cidade**  
de uma **aplicação** pode ser representado em REST pela URI  
**http://servidor/aplicacao/pais/estado/cidade/**
- Dados anexados a respostas podem ter diversas representações e tipos
  - XML, JSON, CSV, etc.
- Métodos HTTP permitem operações CRUD
  - Create:      **POST /pais/estado**      (contendo **<estado>SP</estado>**, no corpo)
  - Retrieve:     **GET /pais/estado/SP**    (Retrieve All com **GET /pais/estado**)
  - Update:       **PUT /pais/estado/PB**
  - Delete:       **DELETE /pais/estado/PB**



- **JAX-RS** é uma API Java para facilitar o desenvolvimento de aplicações que utilizam a arquitetura REST
- Componentes de uma aplicação JAX-WS
  - Subclasse de **Application**: usado para configurar a aplicação, mapear o nome do contexto e listar outras classes que fazem parte da aplicação (opcional)
  - **Root resource class**: classes que definem a raiz de um recurso mapeado a um caminho de URI
  - **Resource method**: os métodos de um Root resource class, que são associados a **designadores de métodos** HTTP (@GET, @POST, etc.)
  - **Providers**: operações que produzem ou consomem representações de entidades em outros formatos (ex: XML, JSON)
- JAX-RS usa anotações para configurar os componentes
  - **@ApplicationPath** na subclasse de Application
  - **@Path** nos Root resource classes
  - **@GET, @POST**, etc e **@Path** nos Resource methods
  - **@Produces, @Consumes** nos Providers
  - **@Context** para injetar diversos tipos de contexto disponíveis no ambiente Web



# HelloWorld com JAX-RS

- Para construir um serviço RESTful com JAX-RS

1. Criar uma subclasse de **Application** vazia e anote com **@ApplicationPath** informando o nome do contexto

```
@ApplicationPath("webapi")
```

```
public class HelloApplication extends Application { }
```

2. Criar um root resource anotado com **@Path**

```
@Path("sayhello")
```

```
public class HelloResource { ... }
```

3. Criar método no resource anotado com designador de método HTTP

```
@GET
```

```
public String getResposta() { return "Hello, world!"; }
```

4. Empacotar WAR (ex: **helloapp.war**) + deploy (ex: **localhost:8080**)

5. Envie GET para a URL abaixo (pode ser via browser) e veja a resposta

**http://localhost:8080/helloapp/webapi/sayhello**



# Como testar RESTful Web Services

- Pela interface do browser é possível testar apenas método **GET** e envio de **text/plain**
  - Testar outros métodos e representações é difícil
- Use um **cliente HTTP**
  - Linha de comando: **cURL**
  - Mozilla **Firebug**, Firefox RESTClient, Chrome Dev Tools, etc.
  - **Java REST Client** <https://code.google.com/p/rest-client/>
  - **Fiddler** (no Windows) <http://fiddler2.com/>
  - **MitmProxy** (no Linux, OS X) <http://mitmproxy.org/>
- Escreva programas simples usando uma API HTTP
  - Apache HTTP Components – **HTTP Client**  
<http://hc.apache.org/>



# Anotação @Path

- Mapeia um **caminho** a resource raiz ou método
  - Pode ser especificado antes de **classe** ou **método**
  - Se método não define @Path próprio, ele responde ao @Path declarado na classe
  - Se método define um @Path, ele é **subcontexto** do @Path raiz
    - Não deve haver ambiguidade nos mapeamentos resultantes

```
@Path("sala")
public class SalaResource {
    @POST
    public void create(int id, String nome, boolean ocupada) {...}

    @PUT
    public void ocuparProximaSala() {...} ← Responde a PUT /ctx/app/sala/

    @GET
    public int[] getSalas() {...} ← Responde a GET /ctx/app/sala/

    @GET @Path("livre")
    public id getProximaSalaLivre() {...} ← Responde a GET /ctx/app/sala/livre/
}
```

The code snippet illustrates the use of the `@Path` annotation at both the class and method levels. The class `SalaResource` is annotated with `@Path("sala")`. The first method, `create`, is annotated with `@POST`. The second method, `ocuparProximaSala`, is annotated with `@PUT`. The third method, `getSalas`, is annotated with `@GET`. The fourth method, `getProximaSalaLivre`, is annotated with `@GET @Path("livre")`. Annotations are highlighted in red and blue, while their descriptions are in black.



# Path templates e @PathParam

- **@Path** pode receber parâmetros (Path templates)
  - `@Path("/filmes/{imdb}")`
  - Aceita por exemplo: `http://abc.com/war/app/filmes/tt0066921`
- Parâmetro pode ser lido em método usando **@PathParam**

```
@Path("/filmes/{imdb}")
public class FilmesIMDBResource {

    @GET    @Produces("text/xml")
    public Filme getFilme(@PathParam("imdb") String codigoIMDB) {
        return entity.getFilmeByIMDBCode(codigoIMDB);
    }
}
```



# Path template e restrições

- Se URL não combinar com a resolução do path template, o servidor produzirá um **erro 404**
- Variável deve combinar com a expressão regular "[^/]+?"
- Pode-se **substituir** a expressão regular default por outra  
`@Path("filme/{imdb: tt[0-9]{4,7}}")`
- Template pode ter mais de uma variável

```
@Path("{cdd1:[0-9]}/{cdd2:[0-9]}")
public class AssuntoResource {
    @GET @Path("{cdd3:[0-9]}")
    public void getAssunto(@PathParam("cdd1") int d1,
                          @PathParam("cdd2") int d2,
                          @PathParam("cdd3") int d3) { ... }
    ...
}
```

Combina com  
GET /ctx/app/**5/1/0**

- Espaços e caracteres especiais devem ser substituídos por URL encodings
  - `/cidade/São Paulo` → `/cidade/S%E3o%20Paulo`



# Designadores de métodos HTTP

- Mapeia métodos HTTP a métodos do resource
  - @GET
  - @POST
  - @PUT
  - @DELETE
  - @HEAD
  - Podem ser criados outros designadores
- Métodos mapeados podem retornar
  - **void**: não requer configuração adicional
  - **Tipos Java** (objeto ou primitivo): requer conversão do tipo por um entity **provider** (MessageBodyReader ou MessageBodyWriter) que pode ser configurado usando @Produces ou @Consumes
  - Um objeto javax.ws.rs.core.**Response**: permite configurar metadados da resposta (ex: cabeçalhos)



# Idempotência em REST

- No contexto de uma aplicação REST, um método é **idempotente** se, quando chamado múltiplas vezes produz os mesmos resultados
- Os métodos idempotentes do HTTP são
  - GET, HEAD, PUT e DELETE
  - POST não é idempotente (chamadas sucessivas do mesmo método poderão produzir resultados diferentes)
- Ao mapear métodos HTTP a métodos do resource, deve-se observar se o método do resource também é idempotente, para manter a consistência de comportamento
  - Usar **POST** para criar novos dados (C)
  - Usar **GET** apenas para retornar dados (R)
  - Usar **PUT** apenas para alterar dados (U)
  - Usar **DELETE** apenas para remover dados (D)



# @GET e @DELETE

- **@GET** pode ser usado para retornar representações do resource
  - Representação **default** retornada é **text/plain**
  - Outras representações MIME podem ser configuradas usando um entity provider através da anotação **@Produces**
    - **@GET**

```
@Produces("text/html")
public String getHtml() {
    return "<html><body><h1>Hello!!</h1></body></html>";
}
```
- **@DELETE** deve mapeado a métodos responsáveis pela remoção de instâncias do resource
  - **@DELETE**

```
@Path("{id}")
public void deleteAssento(@PathParam("id") int id) {
    facade.removeAssento(id);
}
```



- POST e PUT podem ser usados para create e update
  - Decisão é questão de estilo e discussões acadêmicas
  - Como PUT é idempotente, é geralmente usado para fazer atualizações do resource (mas updates usando PUT **não podem ser parciais**: é preciso enviar o objeto inteiro)
- Nos exemplos deste curso, usaremos **@POST** para criar resources

```
@POST @Consumes({"application/xml", "application/json"})
public void create(Sala entity) {
    facade.create(entity);
}
```

- E **@PUT** para realizar updates

```
@PUT @Path("{id}")
@Consumes({"application/xml", "application/json"})
public void edit(@PathParam("id") Long id, Sala entity) {
    facade.update(entity);
}
```



# Provedores de entidades

171

- Os provedores de entidades fornecem **mapeamento** entre diferentes **representações** e tipos Java
  - JAX-RS disponibiliza provedores para diversos tipos Java e várias representações populares (texto, XML, JSON, etc.)
  - Provedores são **selecionados** nos métodos com anotações **@Produces** e **@Consumes**, indicando um tipo **MIME** que está **mapeado ao provedor**
- É possível criar provedores para representações não suportadas através da implementação de **MessageBodyWriter** (para produção) e **MessageBodyReader** (para consumo)
  - O provedor implementa a interface, é anotado com **@Provider** e mapeia o tipo que representa através de **@Produces** ou **@Consumes**

```
@Provider @Produces("application/msword")
public class WordDocWriter implements MessageBodyWriter<StreamingOutput> { ... }
```

```
@GET @Path("arquivo.doc") @Produces({"application/msword"})
public StreamingOutput getWordDocument() { ... }
```



# MediaType

- Tipos MIME suportados em HTTP podem ser representados por strings ou por constantes de **MediaType**
  - Vantagem: verificado em tempo de compilação
- Algumas constantes
  - APPLICATION\_JSON
  - APPLICATION\_FORM\_URL\_ENCODED
  - APPLICATION\_XML
  - TEXT\_HTML
  - TEXT\_PLAIN
  - MULTIPART\_FORM\_DATA
- As anotações abaixo são equivalentes
  - `@Consumes({"text/plain, text/html"})`
  - `@Consumes({MediaType.TEXT_PLAIN, MediaType.TEXT_HTML})`



# Provedores nativos

173

- Não é preciso criar **@Provider** para representações de vários tipos Java, mapeados automaticamente
- Suportam todos os tipos MIME `(*/*)`
  - `byte[]`, `String`
  - `InputStream`, `Reader`
  - `File`
  - `javax.activation.DataSource`
- Suportam `application/xml`, `application/json` e `text/xml`
  - `javax.xml.transform.Source`
  - `javax.xml.bind.JAXBElement` (classes geradas via JAXB)
- Suporta dados de formulário (`application/x-www-form-urlencoded`)
  - `MultivaluedMap<String, String>`
- Streaming de todos os tipos MIME na resposta (`MessageBodyWriter`)
  - `StreamingOutput` (streaming de imagens, videos, PDF, RTF, etc.)



# @Produces e @Consumes

- **@Produces** é usado para mapear tipos MIME de representações de resource **retornado** ao cliente

```
@GET @Produces({"application/xml"})
public List<Filme> findAll() {...}
```

- **Cliente** HTTP pode selecionar método com base nos tipos MIME que ele suporta através de um cabeçalho de requisição “**Accept:**”

```
GET /ctx/app/filmes
Accept: application/xml, application/json
```

- **@Consumes** é usado para mapear tipos MIME de representações de resource **enviado** ao serviço

```
@POST @Consumes({"application/xml", "application/json"})
public void create(Filme entity) { ... }
```

- **Cliente** HTTP pode enviar um tipo MIME compatível com o método desejado usando um cabeçalho de requisição “**Content-type:**”

```
POST /ctx/app/filmes
Content-type: application/xml
```



# @Produces e @Consumes

- As anotações podem ser colocadas no nível da classe, estabelecendo um valor **default** para todos os métodos
- Métodos individuais podem sobrepor valores herdados

```
@Path("/myResource")
@Produces("application/xml")
@Consumes("text/plain")
public class SomeResource {
    @GET
    public String doGetAsXML() { ... }

    @GET @Produces("text/html")
    public String doGetAsHtml() { ... }

    @PUT @Path("{text}")
    public String putPlainText(@PathParam("text") String txt) { ... }
}
```



# Response e ResponseBuilder

176

## ■ Builder para **respostas** HTTP

- Permite controlar a resposta HTTP enviada ao cliente
- **Response** cria um código de status que retorna **ResponseBuilder**
- ResponseBuilder possui diversos métodos para construir cabeçalhos, códigos de resposta, criar cookies, anexar dados, etc.
- Métodos devolvem ResponseBuilder permitindo uso em cascata
- Método **build()** retorna o Response finalizado

## ■ Exemplos

- `ResponseBuilder builder = Response.status(401);  
Response r1 = builder.build();`
- `Response r2 = Response.ok().entity(filme).build();`
- `Response r3 = Response.status(200)  
 .type("text/html")  
 .cookie(new NewCookie("u","1"))  
 .build();`



# java.net.URI e UriBuilders

- **URIs** são a interface mais importante em aplicações REST
- Um java.net.URI pode ser construído usando um **UriBuilder**
  - Permite trabalhar com os componentes individuais, segmentos e fragmentos da URI, concatenar e construir novas URIs
- Exemplos

```
// cria a URI res#anchor;a=x?open=yes
URI requestUri = UriBuilder.fromPath("{arg1}")
    .fragment("{arg2}")
    .matrixParam("a", "x")
    .queryParam("open", "{arg3}")
    .build("res", "anchor", "yes");
```



# Parâmetros do request

- Além de **@PathParam** há outras cinco anotações que permitem extrair informação de um request
- **@QueryParam** e **@DefaultValue**
  - Extraem dados de um query string (`?nome=valor&nome=valor`)
- **@FormParam**
  - Extrai dados de formulário (`application/x-www-form-urlencoded`)
- **@CookieParam**
  - Extrai dados de cookies (pares nome=valor)
- **@HeaderParam**
  - Extrai dados de cabeçalhos HTTP
- **@MatrixParam**
  - Extrai dados de segmentos de URL



# QueryParam

- Extrai dados passados no query-string de uma URI
- Parâmetros de query são anotados com
  - **@QueryParam("parametro")** – informando o nome do parâmetro HTTP correspondente ao parâmetro Java do método
  - **@DefaultValue("valor")** – informando o valor que será usado caso o parâmetro não tenha sido definido
- Exemplo de requisição  
GET /ctx/app/filme?maxAno=2010&minAno=1980
- Exemplo de método que lê os parâmetros

```
@GET @Path("filme")
public List<Filme> getFilmesPorAno(
    @DefaultValue("1900") @QueryParam("minAno") int min
    @DefaultValue("2013") @QueryParam("maxAno") int max)    {...}
```



# FormParam

- **@FormParam** extrai parâmetros de formulários HTML

- Exemplo de formulário

```
<FORM action="http://localhost:8080/festival/webapi/filme"
      method="post">

    Titulo: <INPUT type="text" name="titulo" tabindex="1">
    Diretor: <INPUT type="text" name="diretor" tabindex="2">
    Ano: <INPUT type="text" name="ano" tabindex="3">
</FORM>
```

- Exemplo de método que consome dados do formulário

```
@POST
@Consumes("application/x-www-form-urlencoded")
public void post(@FormParam("diretor") String diretor) {
    ...
}
```



# HeaderParam

- **@HeaderParam** permite obter dados que estão no cabeçalho da requisição (ex: Content-type, User-Agent, etc.)
- Requisição HTTP

```
GET /ctx/app/filme/echo HTTP/1.1  
Cookies: version=2.5, userid=9F3402
```

- Resource

```
You sent me cookies: userid=9F3402, version=2.5
```

```
@Path("/filme")  
public class Filme {  
  
    @GET  
    @Path("/echo")  
    public Response echo(@HeaderParam("Cookie") String cookieList) {  
        return Response.status(200)  
            .entity("You sent me cookies: " + cookieList)  
            .build();  
    }  
}
```



# CookieParam

- **@CookieParam** permite acesso a cookies individuais que o cliente está mandando para o servidor
- Requisição HTTP

```
GET /ctx/app/filme HTTP/1.1  
Cookies: version=2.5, userid=9F3402
```

- Resource

```
@GET  
@Produces({MediaType.TEXT_PLAIN})  
public Response getMyCookies(  
    @CookieParam(value = "userid") String userid,  
    @CookieParam(value = "version") String version) {  
  
    return Response.status(200)  
        .entity("Your cookies contain: "  
            + userid  
            + " and "  
            + version).build();  
}
```

Your cookies contain: 9F3402 and 2.5



# MatrixParam

- **@MatrixParam** extrai os pares nome=valor que são incluídos como anexo da URL (geralmente usado como alternativa a cookies)
- Requisição HTTP

```
GET /ctx/app/filme;version=2.5;userid=9F3402
```

- Resource

Your URL contains: 9F3402 and 2.5

```
@GET  
@Produces({MediaType.TEXT_PLAIN})  
public Response getMyCookies(  
    @MatrixParam(value = "userid") String userid,  
    @MatrixParam(value = "version") String version) {  
  
    return Response.status(200)  
        .entity("Your URLs contains: "  
            + userid  
            + " and "  
            + version).build();  
}
```



# UriInfo e HttpHeaders

- Pode-se ler vários parâmetros de uma vez usando os objetos UriInfo e HttpHeaders
  - A interface **UriInfo** contém informações sobre os componentes de uma requisição
  - A interface **HttpHeaders** contém informações sobre cabeçalhos e cookies
  - Ambas podem ser injetadas através de **@Context**

```
@GET  
public String getParams(@Context UriInfo ui,  
                        @Context HttpHeaders hh) {  
    MultivaluedMap<String, String> queryParams =  
        ui.getQueryParameters();  
    MultivaluedMap<String, String> pathParams =  
        ui.getPathParameters();  
    MultivaluedMap<String, String> headerParams =  
        hh.getRequestHeaders();  
    MultivaluedMap<String, Cookie> pathParams =  
        hh.getCookies();  
}
```



# Validação com Bean Validation

- Os dados enviados para métodos em classes de resource podem ser validados através da **API Bean Validation**, que é configurada via anotações

```
@POST  
@Path("/criar")  
@Consumes(MediaType.APPLICATION_FORM_URLENCODED)  
public void criarFilme(  
    @NotNull @FormParam("titulo") String titulo,  
    @NotNull @FormParam("diretor") String diretor,  
    @Min(1900) @FormParam("ano") int ano) { ... }  
  
@Pattern(regexp="tt[0-9]{5-7}")  
private String imdbCode;
```

- Violações na validação provocam **ValidationException**
  - Gera um erro **500** (Internal Server Error) se a execução ocorreu ao validar um tipo de retorno
  - Gera um erro **400** (Bad Request) se a execução ocorreu ao validar um parâmetro de entrada



# Autenticação HTTP

186

- A **autenticação** HTTP é realizada por cabeçalhos
- Exemplo (autenticação Basic)
  - Resposta do servidor a tentativa de acesso a dados protegidos
  - **HTTP/1.1 401 Unauthorized**  
**WWW-Authenticate:** Basic realm="localhost:8080"
  - Requisição do cliente enviando credenciais
  - GET /ctx/app/filmes HTTP/1.1  
**Authorization:** Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
  - Se a autenticação falhar, o servidor responde com **403 Forbidden**
- A configuração da autenticação deve ser realizada no servidor
  - Arquivo web.xml permite definir método de autenticação: **BASIC**, **DIGEST**, **CLIENT-CERT** (HTTPS)
  - Ferramentas do servidor definem perfis (**roles**) ou grupos mapeados a usuários autenticados
  - **Roles** são mapeados a URLs no **web.xml**



# JAAS no web.xml

- Autorização
  - A declaração de perfis de usuários é feita no web.xml usando **<security-role>**.
  - A associação desses perfis com usuários reais é dependente de servidor (ex: grupos no JBoss grupos = roles)

```
<security-role>
    <role-name>admin</role-name>
</security-role>

<security-role>
    <role-name>membro</role-name>
</security-role>

<security-role>
    <role-name>visitante</role-name>
</security-role>
```

- Autenticação
  - O método de autenticação é configurado usando o tag **<login-config>**

```
<login-config>
    <auth-method>
        BASIC
    </auth-method>
</login-config>
```



# Web-Resource Collection

- A coleção de recursos Web protegidos e métodos de acesso que podem ser usados para acessá-los é definido em um bloco **<web-resource-collection>** definido dentro de **<security-constraint>**
- Inclui URL-patterns **<url-pattern>** que indicam quais os mapeamentos que abrangem a coleção
- Inclui um **<http-method>** para cada método permitido

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Seção de Assinantes</web-resource-name>
        <url-pattern>/aplicacao/filme/*</url-pattern>
        <url-pattern>/aplicacao/sala/*</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    ...
</security-constraint>
```



# Security Constraint

- O Web Resource Collection faz parte do Security Constraint que associa perfis (roles) de usuário à coleção

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Administracao</web-resource-name>
        <url-pattern>/aplicacao/filme/*</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
        <http-method>PUT</http-method>
        <http-method>DELETE</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>admin</role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>
...
<security-constraint> outras coleções, ... </security-constraint>
```

Força uso de HTTPS



# Usuários e roles: configuração

190

- Dependente do servidor usado\*
  - diferente no Tomcat, JBoss, Glassfish
- Exemplo: JBoss AS 7
  - Crie usuário no ApplicationRealm e associe a um grupo

```
$ add-user.sh
What type of user do you wish to add?
  a) Management User (mgmt-users.properties)
  b) Application User (application-users.properties)
(a): b
Enter the details of the new user to add.
Realm (ApplicationRealm) :
Username : admin1234
Password :
Re-enter Password :
What roles do you want this user to belong to? (Please enter a comma separated
list, or leave blank for none)[  ]: admin, usuario
```

- Configure o uso de JAAS com o realm default (RealmUsersRoles) em um arquivo **WEB-INF/jboss-web.xml** contendo

```
<jboss-web>
  <security-domain>java:/jaas/other</security-domain>
</jboss-web>
```

\* Isto é um resumo: consulte a documentação do servidor



# @Context

- **@Context** pode ser usado para injetar diversos objetos contextuais disponíveis em uma requisição ou resposta HTTP
- Objetos da Servlet API:
  - `ServletConfig`,
  - `ServletContext`,
  - `HttpServletRequest`
  - `HttpServletResponse`
- Objetos da JAX-RS API:
  - `Application`
  - `UriInfo`
  - `Request`
  - `HttpHeaders`
  - `SecurityContext`
  - `Providers`

```
@Context
Request request;
@Context
UriInfo uriInfo;

@PUT
public metodo(@Context HttpHeaders headers) {
    String m = request.getMethod();
    URI ap   = uriInfo.getAbsolutePath();
    Map<String, Cookie> c = headers.getCookies();
}

@GET @Path("auth")
public login(@Context SecurityContext sc) {
    String userid =
        sc.getUserPrincipal().getName();
    (if sc.isUserInRole("admin")) { ... }
}
```



# Configuração do serviço

192

- Configuração da URI base pode ser no web.xml

```
<servlet-mapping>
    <servlet-name>javax.ws.rs.core.Application</servlet-name>
    <url-pattern>/webapi/*</url-pattern>
</servlet-mapping>
```

ou via anotação **@ApplicationPath** em subclasse de Application

```
@ApplicationPath("/webapi")
public class ApplicationConfig extends Application { ... }
```

- Application carrega todos os resources encontrados.

- Pode-se sobrepor getClasses() para selecionar quais disponibilizar

```
@javax.ws.rs.ApplicationPath("webapi")
public class ApplicationConfig extends Application {
    @Override
    public Set<Class<?>> getClasses() {
        Set<Class<?>> resources = new java.util.HashSet<>();
        resources.add(com.agonavis.festival.FilmeResource.class);
        resources.add(com.agonavis.festival.SalaResource.class);
        return resources;
    }
}
```



# Cliente REST

- Um cliente JAX-RS pode ser **qualquer cliente HTTP** (java.net.\* , Apache HTTP Client, cURL, etc.)
  - Pode ser escrito em Java, JavaScript, C#, Objective-C ou qualquer linguagens capaz de abrir **sockets** de rede
  - É preciso lidar com as **representações** recebidas: converter XML, JSON, encodings, etc.
  - É preciso gerar e interpretar **cabeçalhos HTTP** usados na comunicação (seleção de tipos MIME, autenticação, etc.)
- Existem frameworks que possuem APIs para escrever clientes
  - Jersey: <http://jersey.java.net>
  - RESTEasy: <http://www.jboss.org/resteasy>
- Existe uma **API padrão**, a partir do JAX-RS 2.0 (Java EE 7)



# Cliente usando java.net

194

- Qualquer API capaz de montar requisições HTTP pode ser usada para implementar clientes

```
URL url =
    new URL("http://localhost:8080/ctx/app/imdb/tt0066921");
HttpURLConnection conn =
    (HttpURLConnection) url.openConnection();
conn.setRequestMethod("GET");
conn.setRequestProperty("Accept", "application/xml");
if (conn.getResponseCode() != 200) {
    throw new RuntimeException("Erro : " + conn.getResponseCode());
}

BufferedReader br =
    new BufferedReader(new InputStreamReader((conn.getInputStream())));
String linha = br.readLine();
System.out.println("Dados recebidos: " + linha);
conn.disconnect();

JAXBContext jc = JAXBContext.newInstance(Filme.class);
Unmarshaller u = jc.createUnmarshaller();
Filme filme = (Filme) u.unmarshal(new StringReader(linha));

System.out.println(filme.getIMDB()); ...
```



# Cliente Apache HttpClient

- API do Apache HttpComponents HTTP Client é ainda mais simples

```
GetMethod method =
    new GetMethod("http://localhost:8080/ctx/app/filme/imdb/tt0066921");
method.addRequestHeader("Accept", "application/xml");
HttpClient client = new HttpClient();
int responseCode = client.executeMethod(method);
if (responseCode != 200) {
    throw new RuntimeException("Erro : " + responseCode);
}

String response = method.getResponseBodyAsString();

JAXBContext jc = JAXBContext.newInstance(Filme.class);
Unmarshaller u = jc.createUnmarshaller();
Filme filme = (Filme) u.unmarshal(new StringReader(linha));

System.out.println(filme.getIMDB()); ...
```



# Cliente Jersey\*

- Uma API de cliente específica para Web Services REST como o Jersey facilita o trabalho convertendo automaticamente representações em objetos

```
ClientConfig config =
    new DefaultClientConfig();
Client client = Client.create(config);
URI baseURI =
    UriBuilder.fromUri("http://localhost:8080/ctx").build();
WebResource service = client.resource(baseURI);

Filme filme = service.path("app")
    .path("filme/imdb/tt0066921")
    .accept(MediaType.APPLICATION_XML)
    .get(Filme.class);

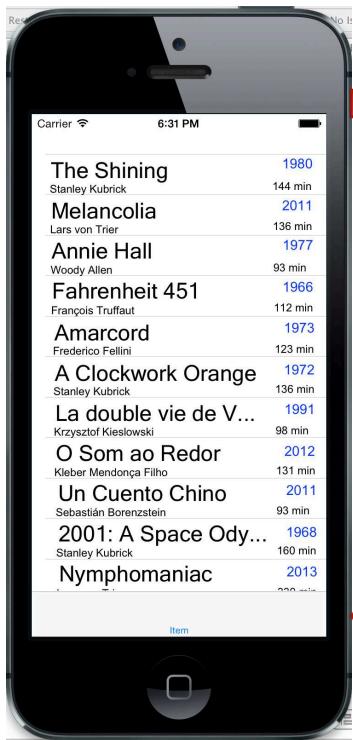
System.out.println(filme.getIMDB());
System.out.println(filme.getTitulo());
System.out.println(filme.getDiretor());
```

\* Outra alternativa  
é a API **RESTEasy**



# Cliente mobile (iPhone)

- REST é a melhor alternativa para Web Services que fazem **integração** como outras plataformas
- Exemplo: cliente em iPhone usando Objective-C



1) Cliente REST escrito em Objective-C rodando em iOS 7 gera requisição

GET `http://192.168.1.25/festival/webapi/filme`

2) Glassfish gera resposta HTTP com lista de filmes em **JSON**

Glassfish 4.0

3) iPhone extrai dados e preenche UITableView



- Para descrever serviços REST e permitir a geração automática de clientes, existe o padrão **WADL**
  - Web Application Description Language
- Tem função análoga ao WSDL (usada em SOAP Web Services) para descrição de Web Services RESTful
  - Obtido em `http://servidor/contexto/raiz/application.wadl`

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<application xmlns="http://wadl.dev.java.net/2009/02">
    <grammars/>
    <resources base="http://localhost:8080/festival/webapi">
        <resource path="filme/{imdb}">
            <param type="xs:string" style="template" name="imdbID"/>
            <method name="GET" id="getFilme">
                <response>
                    <representation mediaType="application/xml"/>
                    <representation mediaType="application/json"/>
                </response>
            </method>
        ...
    ...
</application>
```



# Cliente JAX-RS 2.0

199

- **Java EE 7** tem uma API padrão para clientes REST baseada na API do Jersey
- Exemplo de envio de uma requisição GET simples

```
Client client = ClientBuilder.newClient();
String nomeDoFestival =
    client.target("http://localhost:8080/festival/webapi/nome")
        .request(MediaType.TEXT_PLAIN)
        .get(String.class);
```

- Pode-se configurar com **WebTarget** para reusar o path base

```
Client client = ClientBuilder.newClient();
WebTarget baseResource = client.target("http://servidor/ctx/app");
WebTarget filmeResource = base.path("filme");
```

- A partir do resource pode-se construir um **request()** e enviar um método HTTP

```
Filme filme = filmeResource.queryParam("imdb", "tt0066921")
    .request(MediaType.APPLICATION_XML)
    .get(Filme.class)
```



# Geração de clientes

- JAX-RS 2.0 (Java EE 7) fornece uma ferramenta de linha de comando (e task Ant/Maven) que gera um cliente JAX-RS usando WADL
  - **wadl2java** -o diretorio  
-p pacote  
-s jaxrs20  
`http://localhost:8080/app/ctx/application.wadl`
- Cria uma classe (JAXBElement) para cada resource raiz
- Cria uma classe **Servidor\_ContextoAplicacao.class** (para um serviço em `http://servidor/contexto/aplicacao`).
  - Esta classe possui um método **createClient()** que cria um cliente JAX-RS 2.0 já configurado para usar o serviço
  - Também fornece métodos para retornar instâncias e representações dos objetos mapeados como resource



# Exercícios

201

1. Escreva um REST Web Service para acesso a uma lista de Produtos. Planeje a interface e ofereça operações para

- Listar todos os produtos
- Detalhar um produto
- Criar um produto
- Alterar o preço de um produto
- Pesquisar produtos por faixa de preço
- Remover um produto

Produto
<code>id: long</code>
<code>descricao: string</code>
<code>preco: double</code>

2. Escreva clientes REST que usem o serviço criado no exercício anterior para
  - Preencher o banco com 10 produtos
  - Listar os produtos disponíveis
  - Remover um produto
  - Alterar o preço de um produto
  - Listar produtos que custam mais de 1000

3. Configure o acesso à aplicação de forma que apenas usuários do grupo “admin” possam remover produtos ou alterar preços



# Referências

- Especificações
  - HTTP: <http://www.w3.org/Protocols/>
  - RFC 2616 (HTTP) <http://www.ietf.org/rfc/rfc2616.txt>
  - WADL <https://wadl.java.net/>
  - Arquiteturas de Web Services <http://www.w3.org/TR/ws-arch/>
- Artigos, documentação e tutoriais
  - Saldhana. "Understanding Web Security using web.xml via Use Cases" DZone. 2009. <http://java.dzone.com/articles/understanding-web-security>
  - Configuração de JAAS no Jboss 7 <https://community.jboss.org/wiki/JBossAS7SecurityDomainModel>
  - Java EE Tutorial sobre RESTful WebServices <http://docs.oracle.com/javaee/7/tutorial/doc/jaxrs.htm>
  - Documentação do Jersey (tutorial de JAX-RS): <https://jersey.java.net/documentation/latest>
  - Roy Thomas Fielding. "Architectural Styles and the Design of Network-based Software Architectures". University of California, 2000. [http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding\\_dissertation.pdf](http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf)
  - Alex Rodriguez. "RESTful Web Services: the basics". IBM Developerworks, 2008. <http://www.ibm.com/developerworks/webservices/library/ws-restful/>
  - Hadyel & Sandoz (editors). "JAX-RS: Java API for RESTful Web Services. Version 1.1. Oracle, 2009. <https://jax-rs-spec.java.net/>



5

Infraestutura de sistemas  
Java EE

## Cliente Web

Usando HTML, CSS, JavaScript e JQuery  
Comunicação assíncrona com Ajax



# Conteúdo

1. Introdução a interfaces Web estáticas
  - HTML e CSS essencial
  - Formulários do HTML
  - Document Object Model
  - JavaScript essencial
2. Interfaces Web dinâmicas
  - Manipulação da árvore DOM com JavaScript
  - Alteração de classes CSS
  - Recursos essenciais do HTML5 e CSS3
  - Ajax e comunicação com Web Services
3. JQuery essencial
  - Seletores e acesso a dados em HTML, XML e JSON
  - Ajax usando JQuery
  - Formulários dinâmicos
  - Boas práticas e técnicas essenciais para aplicações assíncronas



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

220

null



# java.lang.OutOfMemoryError

null



222

# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



225

# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



228

# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



230

# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



238

# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



240

# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



250

# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



256

# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



258

# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



260

java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null



# java.lang.OutOfMemoryError

null

