



TREINAMENTOS

Integração de Sistemas com Webservices, JMS e EJB

Integração de Sistemas com Webservices, JMS e EJB

20 de maio de 2011



Sumário

1	JMS	1
1.1	Middleware Orientado a Mensagens - MOM	1
1.2	Destinos: Filas e Tópicos	1
1.3	Exercícios	2
1.4	Fábricas de Conexões	4
1.5	Exercícios	5
1.6	Visão Geral	6
1.6.1	Obtendo as fábricas e os destinos	6
1.6.2	Criando Conexões e Sessões	6
1.6.3	Criando Emissores e Receptores	6
1.6.4	Criando Mensagens	7
1.6.5	Enviando Mensagens	7
1.6.6	Recebendo Mensagens	7
1.7	Exercícios	7
1.8	Modos de recebimento	16
1.9	Percorrendo uma fila	16
1.10	Exercícios	16
1.11	Selecionando mensagens de um tópico	17
1.12	Exercícios	17
1.13	Tópicos Duráveis	18
1.14	Exercícios	18
1.15	JMS e EJB	20
1.16	Exercícios	21
1.17	Projeto - Rede de Hotéis	22
1.18	Exercícios	22
2	JAX-WS	23
2.1	Web Services	23
2.2	JAXB	24
2.3	Exercícios	25
2.4	Criando um web service - Java SE	27
2.5	Consumindo um web service com JAX-WS	28
2.6	Exercícios	28
2.7	JAX-WS e EJB	32
2.8	Exercícios	32
2.9	Projeto - Táxi no Aeroporto	33

2.10 Exercícios	34
3 JAX-RS	35
3.1 REST vs Padrões W3C	35
3.2 Resources, URIs, Media Types e Operações	35
3.3 Web service com JAX-RS	36
3.4 Resources	36
3.5 Subresource	37
3.6 Exercícios	37
3.7 Parâmetros	39
3.7.1 PathParam	39
3.7.2 MatrixParam	40
3.7.3 QueryParam	40
3.7.4 FormParam	41
3.7.5 HeaderParam	41
3.7.6 CookieParam	41
3.8 Exercícios	41
3.9 Produzindo XML ou JSON	42
3.10 Consumindo XML ou JSON	43
3.11 Exercícios	43
3.12 Implementando um Cliente	46
3.13 Exercícios	46
3.14 Projeto	47

Capítulo 1

JMS

1.1 Middleware Orientado a Mensagens - MOM

Geralmente, em ambientes corporativos, existem diversos sistemas para implementar as inúmeras regras de negócio da empresa. É comum dividir esses sistemas por departamentos ou por regiões geográficas.

Muito provavelmente, em algum momento, os diversos sistemas de uma empresa devem trocar informações ou requisitar procedimentos entre si. Essa integração pode ser realizada através de intervenção humana. Contudo, quando o volume de comunicação entre os sistemas é muito grande, essa abordagem se torna inviável.

Daí surge a necessidade de automatizar a integração entre sistemas. A abordagem mais simples para implementar essa automatização é utilizar arquivos de texto contendo os dados que devem ser transmitidos de um sistema para outro. Normalmente, um sistema compartilhado de arquivos é utilizado para essa transmissão. Essa estratégia possui certas limitações principalmente em relação à integridade das informações armazenadas nos arquivos.

Uma abordagem mais robusta para implementar essa integração é utilizar um Middleware Orientado a Mensagens (MOM). Um Middleware Orientado a Mensagens permite que um sistema receba ou envie mensagens para outros sistemas de forma **assíncrona**. Além disso, o sistema que envia uma mensagem não precisa conhecer os sistemas que a receberão. Da mesma forma, que os sistemas que recebem uma mensagem não precisam conhecer o sistema que a enviou. Essas características permitem que os sistemas sejam integrados com baixo acoplamento.

A plataforma Java define o funcionamento de um Middleware Orientado a Mensagens através da especificação **Java Message Service - JMS**. Todo servidor de aplicação que segue a especificação **Java EE** deve oferecer uma implementação do MOM definido pela JMS. Especificações importantes da plataforma Java como Enterprise Java Beans (EJB), Java Transaction API (JTA) e Java Transaction Service (JTS) possuem um relacionamento forte com a especificação JMS. A seguir veremos a arquitetura do MOM definido pela JMS.

1.2 Destinos: Filas e Tópicos

Na arquitetura JMS, os sistemas que devem ser integrados podem enviar mensagens para **filas(queues)** ou **tópicos(topics)** cadastrados anteriormente no MOM. Na terminologia JMS,

as filas e os tópicos são chamados de **destinos(destinations)**.

Uma mensagem enviada para uma fila pode ser recebida por apenas **um** sistema (point-to-point). Uma mensagem enviada para um tópico pode ser recebida por **diversos** sistemas (publish-and-subscribe).

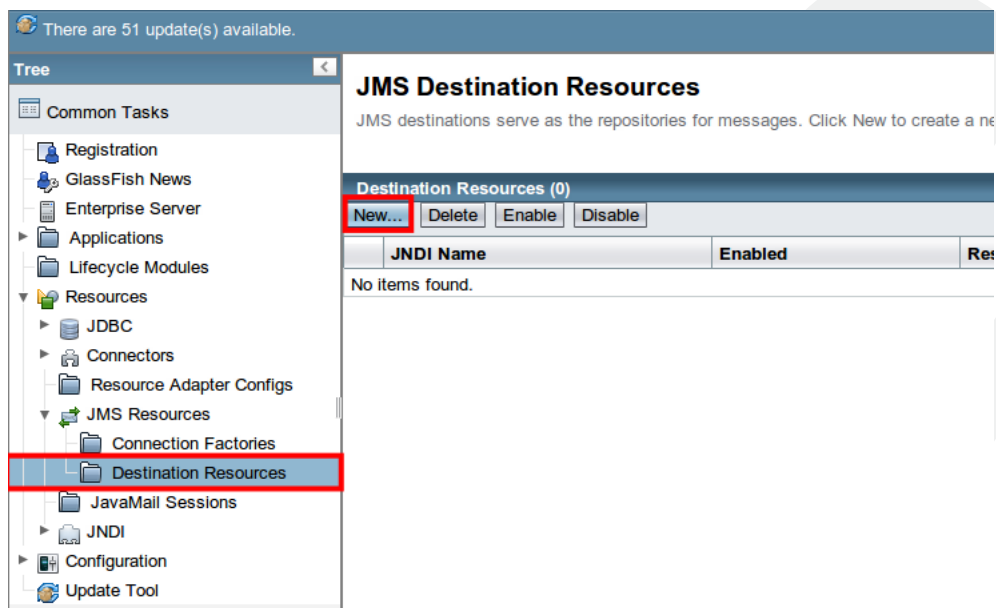
As filas e os tópicos são objetos criados pelos administradores do MOM. A especificação JMS não define uma forma padrão de criação desses objetos. Dessa maneira, cada implementação JMS possui os seus próprios procedimentos para esse processo.

No caso do Glassfish, as filas e os tópicos podem ser criados através da interface web de administração do servidor.

Toda fila ou tópico possui um nome único no MOM.

1.3 Exercícios

1. Na Área de Trabalho, entre na pasta **K19-Arquivos** e copie **glassfish-3.0.1-with-hibernate.zip** para o seu Desktop. Descompacte este arquivo na própria Área de Trabalho.
2. Ainda na Área de Trabalho, entre na pasta **glassfishv3/glassfish/bin** e execute o script **startserv** para executar o glassfish.
3. Verifique se o glassfish está executando através de um navegador acessando a url: **http://localhost:8080**.
4. Pare o glassfish executando o script **stopserv** que está na mesma pasta do script **startserv**.
5. No eclipse, abra a view **servers** e clique com o botão direito no corpo dela. Escolha a opção **new** e configure o glassfish.
6. Execute o glassfish pela view **servers** e verifique se ele está funcionando através de um navegador a url: **http://localhost:8080**.
7. Crie uma fila através da interface de administração do Glassfish seguindo exatamente os passos das imagens abaixo:



New JMS Destination Resource OK Cancel

The creation of a new Java Message Service (JMS) destination resource also creates an admin object resource.

JNDI Name: * A unique name; can be up to 255 characters, must contain only alphanumeric, underscore, dash, or dot characters

Physical Destination Name: * Destination name in the Message Queue broker. If the destination does not exist, it will be created automatically when needed.

Resource Type: * ▼

Description:

Status: ☒ Enabled

Additional Properties (0)

Add Property Delete Properties

Name	Value	Description
No items found.		

OK Cancel

8. Crie um tópico através da interface de administração do Glassfish seguindo exatamente os passos das imagens abaixo:

JMS Destination Resources

JMS destinations serve as the repositories for messages. Click New to create a new destination resource or click the name of a destination resource to modify its properties.

Destination Resources (1)

JNDI Name	Enabled	Resource Type
jms/pedidos	true	javax.jms.Queue

New JMS Destination Resource

OK Cancel

The creation of a new Java Message Service (JMS) destination resource also creates an admin object resource.

JNDI Name: *

A unique name; can be up to 255 characters, must contain only alphanumeric, underscore, dash, or dot characters

Physical Destination Name: *

Destination name in the Message Queue broker. If the destination does not exist, it will be created automatically when needed.

Resource Type: *

Description:

Status: ☒ Enabled

Additional Properties (0)

Name	Value	Description
No items found.		

OK Cancel

1.4 Fábricas de Conexões

Os sistemas que desejam trocar mensagens através de filas ou tópicos devem obter conexões JMS através das fábricas cadastradas no MOM. Assim como as filas e os tópicos, as fábricas de conexões JMS são objetos criados pelos administradores do MOM. A especificação JMS não define uma forma padrão para criar essas fábricas, então cada implementação define a sua própria forma de criação.

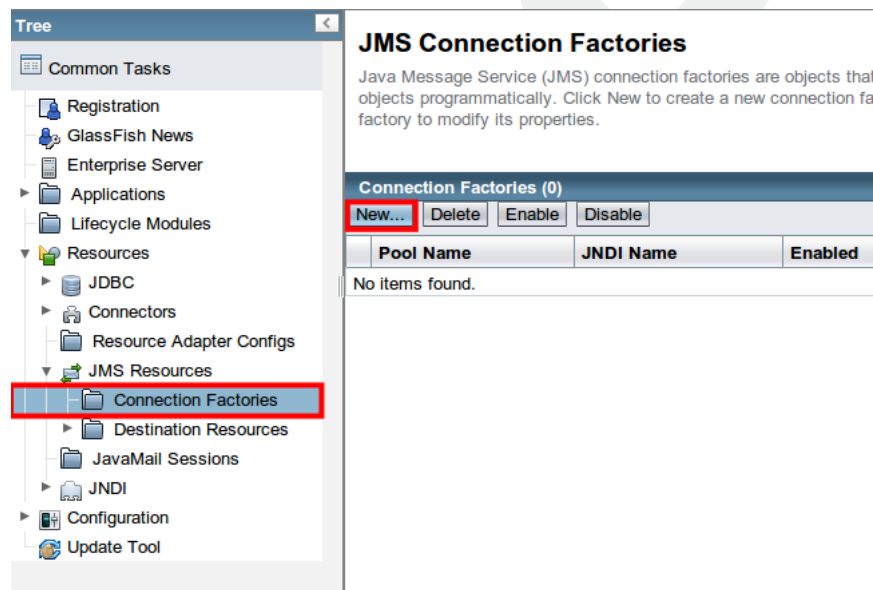
No Glassfish as fábricas podem ser criadas através da interface web de administração do servidor. É possível criar fábricas especificadas para filas ou para tópicos ou genéricas que

podem ser utilizadas para os dois tipos de destino.

Toda fábrica possui um nome único no MOM.

1.5 Exercícios

9. Crie uma fábrica de conexões JMS através da interface de administração do Glassfish seguindo exatamente os passos das imagens abaixo:



The screenshot shows the 'New JMS Connection Factory' dialog box. It includes a description of the creation process and a 'General Settings' section. The 'Pool Name' is set to 'jms/K19Factory' and the 'Resource Type' is set to 'javax.jms.ConnectionFactory'. The 'Status' is checked as 'Enabled'. The 'Pool Settings' section shows 'Initial and Minimum Pool Size' as 8, 'Maximum Pool Size' as 32, 'Pool Resize Quantity' as 2, and 'Idle Timeout' as 300 seconds. The 'OK' button is highlighted with a red box.

New JMS Connection Factory

The creation of a new Java Message Service (JMS) connection factory also creates a connector connection pool for the factory and a connector resource.

General Settings

Pool Name: * jms/K19Factory

Resource Type: * javax.jms.ConnectionFactory

Description:

Status: ☒ Enabled

Pool Settings

Initial and Minimum Pool Size: 8 Connections
Minimum and initial number of connections maintained in the pool

Maximum Pool Size: 32 Connections
Maximum number of connections that can be created to satisfy client requests

Pool Resize Quantity: 2 Connections
Number of connections to be removed when pool idle timeout expires

Idle Timeout: 300 Seconds
Maximum time that connection can remain idle in the pool

1.6 Visão Geral

Inicialmente, sem se ater a detalhes, vamos mostrar os passos necessários para enviar ou receber mensagens através da arquitetura JMS.

1.6.1 Obtendo as fábricas e os destinos

As filas, tópicos e fábricas são objetos criados pelos administradores do MOM. Quando uma aplicação deseja utilizar esses objetos, ela deve obtê-los através de pesquisas ao serviço de nomes do MOM. O serviço de nomes é definido pela especificação JNDI.

```
1 InitialContext ic = new InitialContext();
2
3 ConnectionFactory factory;
4 factory = (ConnectionFactory)ic.lookup("ConnectionFactory");
5
6 QueueConnectionFactory factory
7 factory = (QueueConnectionFactory)ic.lookup("QueueConnectionFactory");
8
9 TopicConnectionFactory factory;
10 factory = (TopicConnectionFactory)ic.lookup("TopicConnectionFactory");
11
12 Queue queue
13 queue = (Queue)ic.lookup("Queue");
14
15 Topic topic;
16 topic = (Topic)ic.lookup("Topic");
```

Normalmente, os servidores de aplicação Java EE oferecem o recurso de Injeção de Dependência para que as aplicações obtenham as fábricas, filas ou tópicos.

No Glassfish, é possível injetar esses objetos através da anotação **@Resource**. Inclusive em aplicações standalone (fora do servidor).

```
1 @Resource(mappedName = "jms/ConnectionFactory")
2 private ConnectionFactory connectionFactory;
```

```
1 @Resource(mappedName = "jms/Queue")
2 private Queue queue;
```

1.6.2 Criando Conexões e Sessões

As sessões JMS são responsáveis pela criação das mensagens JMS, dos emissores e dos receptores de mensagens. As sessões JMS são criadas pelas conexões JMS que por sua vez são obtidas através de uma fábrica de conexões.

```
1 Connection connection = factory.createConnection();
```

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

1.6.3 Criando Emissores e Receptores

Através de uma sessão JMS, podemos criar emissores e receptores de mensagens. Esses objetos são criados já ligados a um destino específico.

```
1 MessageProducer sender = session.createProducer(queue);
```

```
1 MessageConsumer receiver = session.createConsumer(queue);
```

1.6.4 Criando Mensagens

As mensagens são criadas pelas sessões JMS. O tipo mais simples de mensagem é o **Text-Message**.

```
1 TextMessage message = session.createTextMessage();  
2 message.setText("Olá");
```

1.6.5 Enviando Mensagens

As mensagens JMS são enviadas através do método **send()** de um emissor (MessageProducer).

```
1 sender.send(message);
```

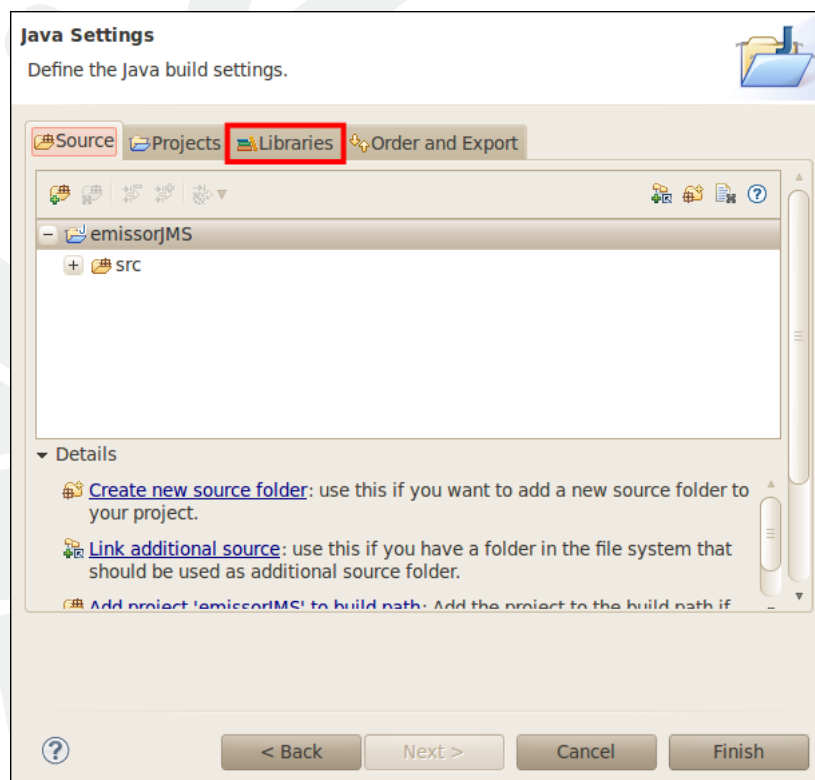
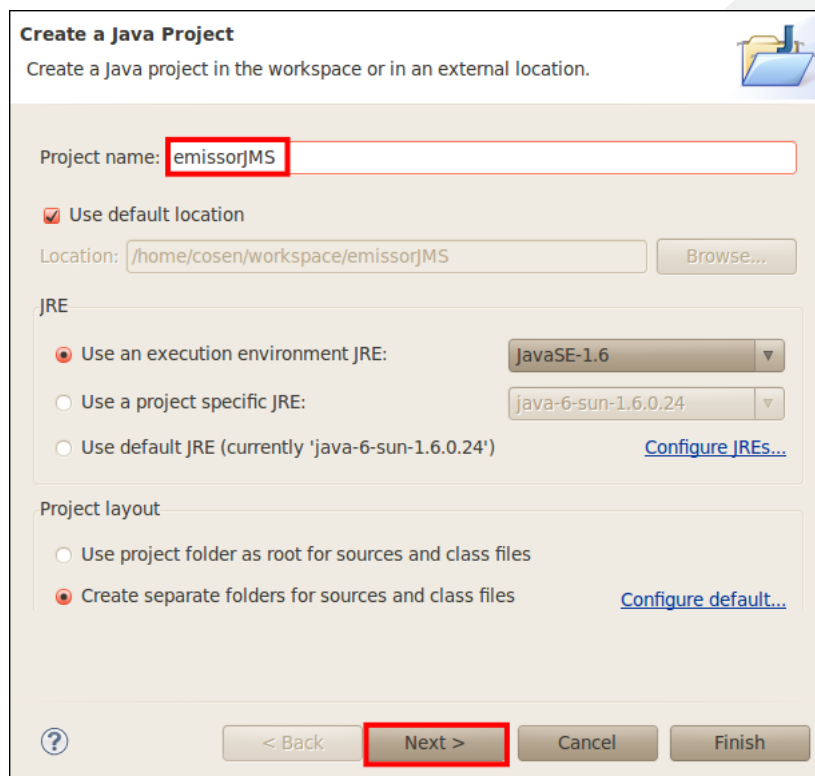
1.6.6 Recebendo Mensagens

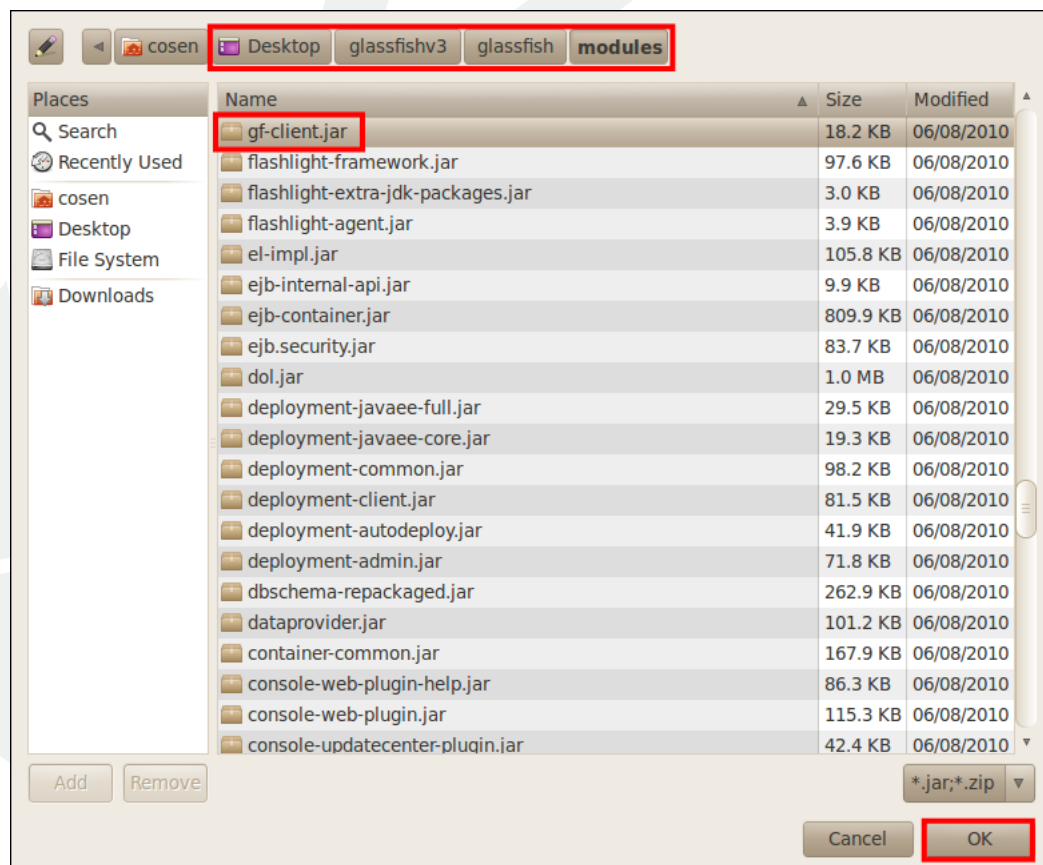
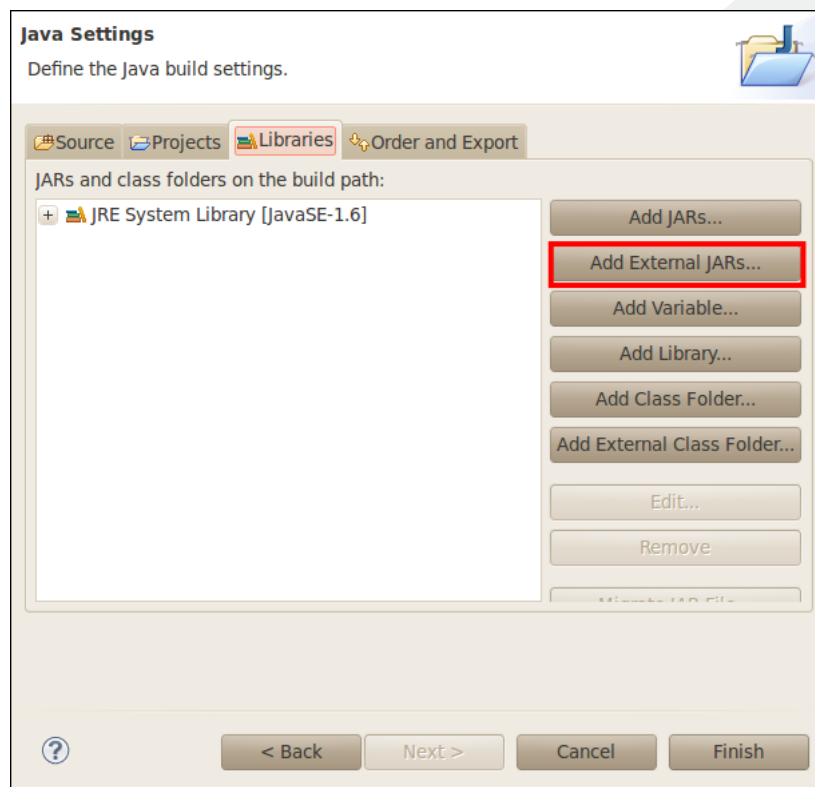
Para receber mensagens JMS, é necessário inicializar a conexão e depois utilizar o método **receive()** de um receptor (MessageConsumer).

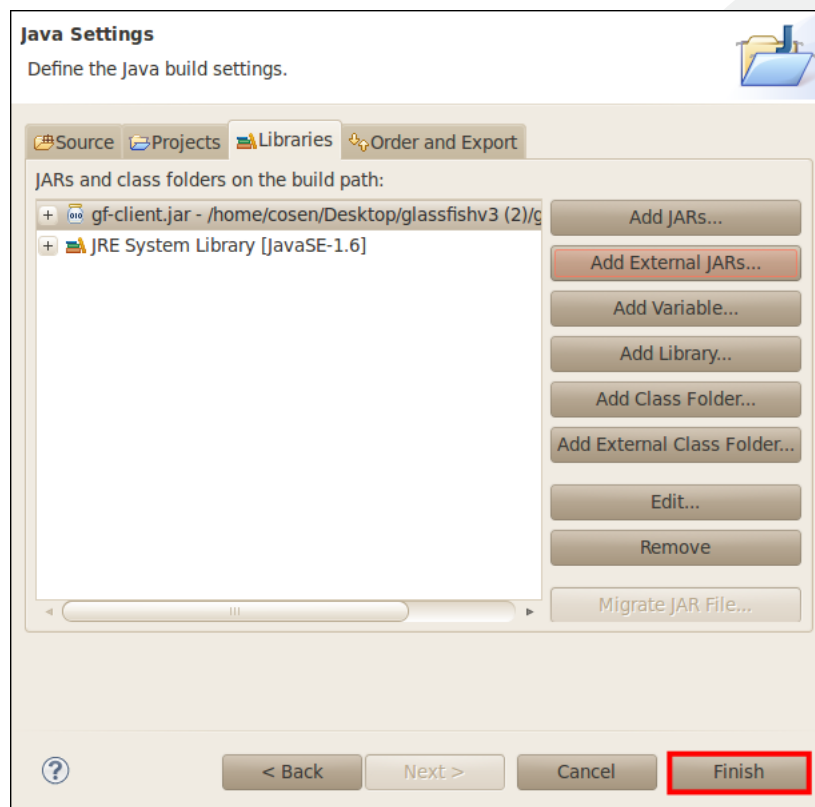
```
1 connection.start();  
2 TextMessage message = (TextMessage)receiver.receive();
```

1.7 Exercícios

10. Crie um Java Project no eclipse para implementar uma aplicação que possa enviar mensagens para a fila e o tópico criados anteriormente. Você pode digitar “CTRL+3” em seguida “new Java Project” e “ENTER”. Depois, siga exatamente as imagens abaixo.







11. Crie um pacote chamado **emissores** no projeto **emissorJMS**.
12. Adicione no pacote **emissores** uma classe com main para enviar uma mensagem JMS para a fila **pedidos**.

```

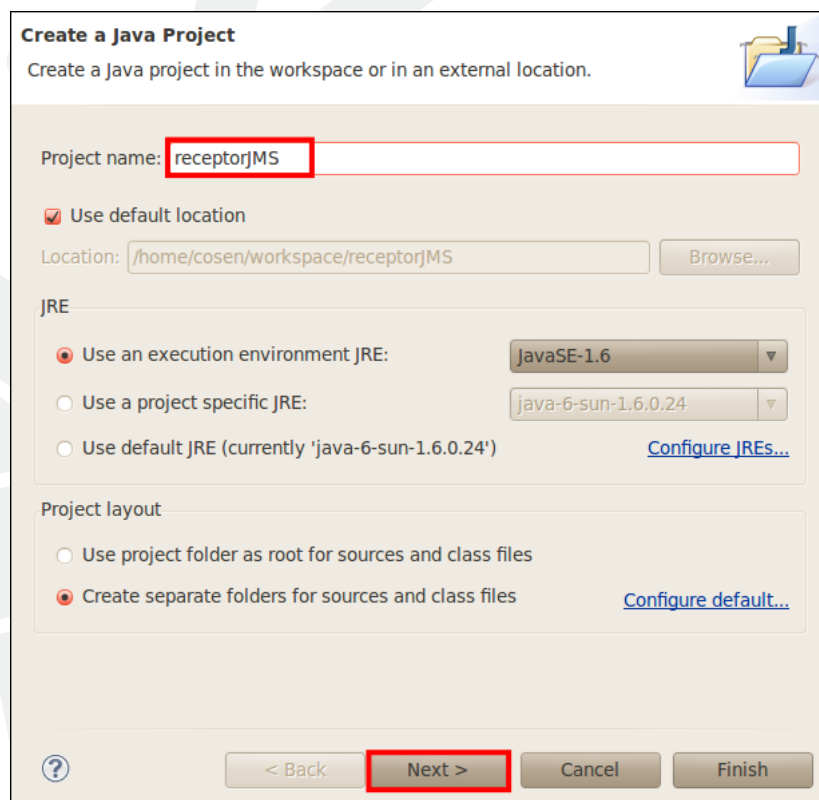
1 public class EnviaNovoPedido {
2     public static void main(String[] args) throws Exception {
3         // serviço de nomes - JNDI
4         InitialContext ic = new InitialContext();
5
6         // fábrica de conexões JMS
7         ConnectionFactory factory = (ConnectionFactory) ic
8             .lookup("jms/K19Factory");
9
10        // fila
11        Queue queue = (Queue) ic.lookup("jms/pedidos");
12
13        // conexão JMS
14        Connection connection = factory.createConnection();
15
16        // sessão JMS
17        Session session = connection.createSession(false,
18            Session.AUTO_ACKNOWLEDGE);
19
20        // emissor de mensagens
21        MessageProducer sender = session.createProducer(queue);
22
23        // mensagem
24        TextMessage message = session.createTextMessage();
25        message.setText("Uma pizza de cinco queijos e uma coca-cola 2l - " + System.↵
26            currentTimeMillis());
27
28        // enviando
29        sender.send(message);

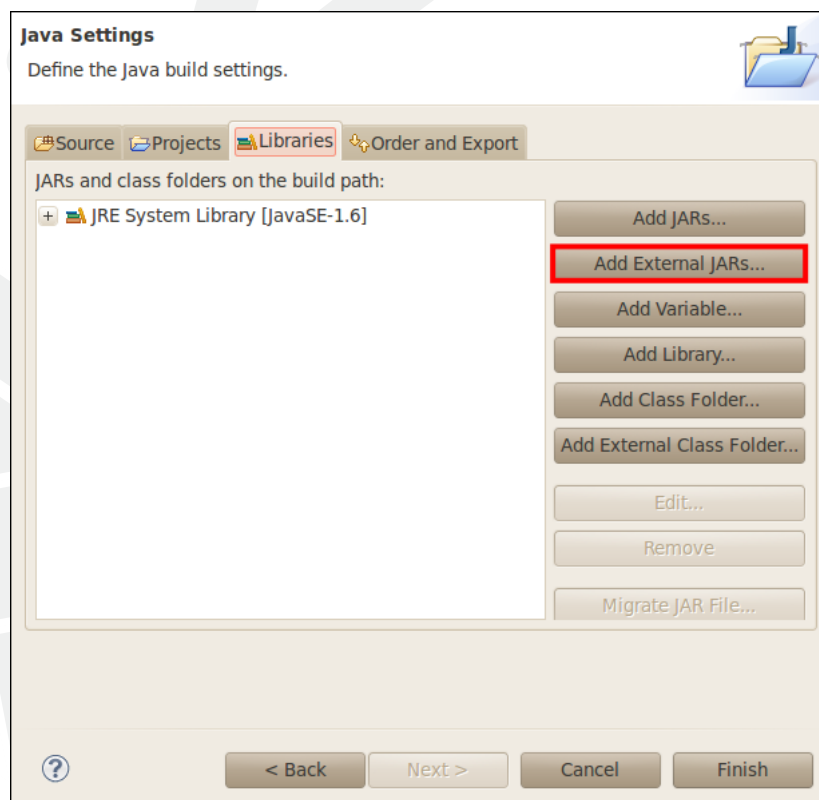
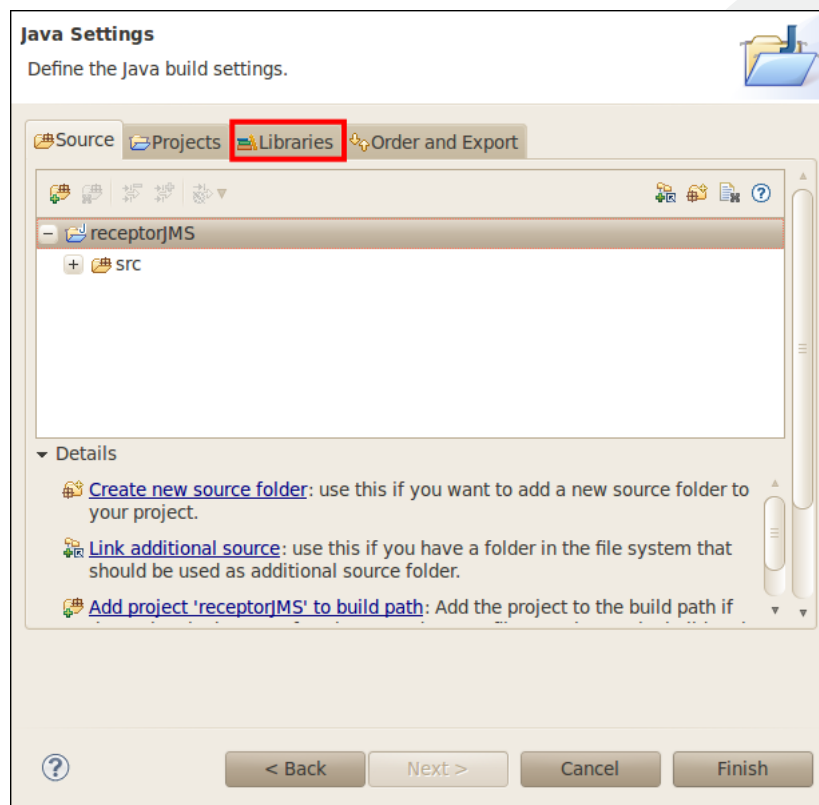
```

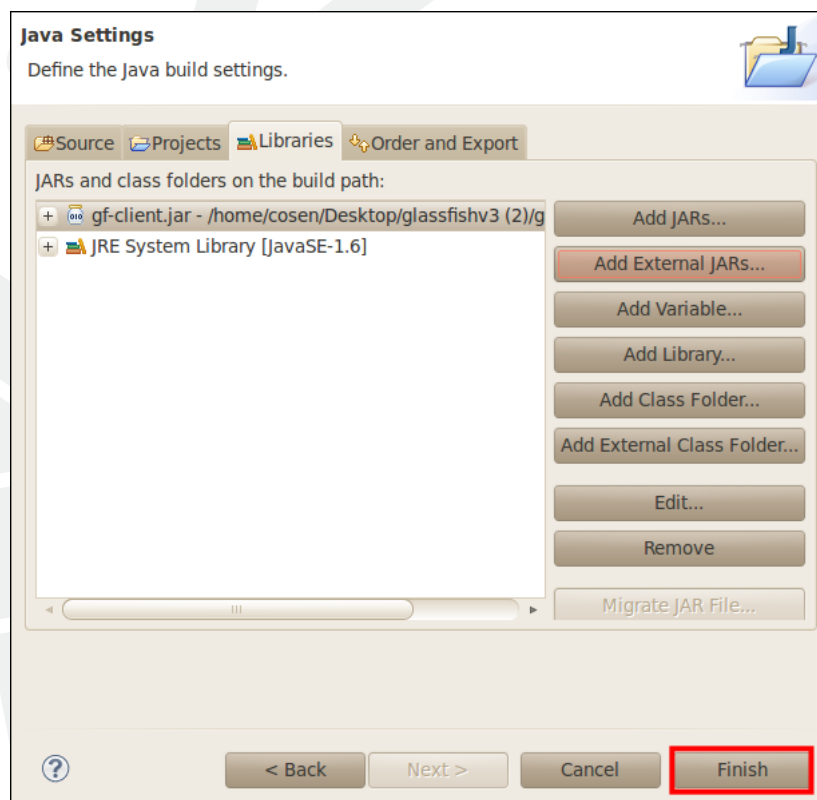
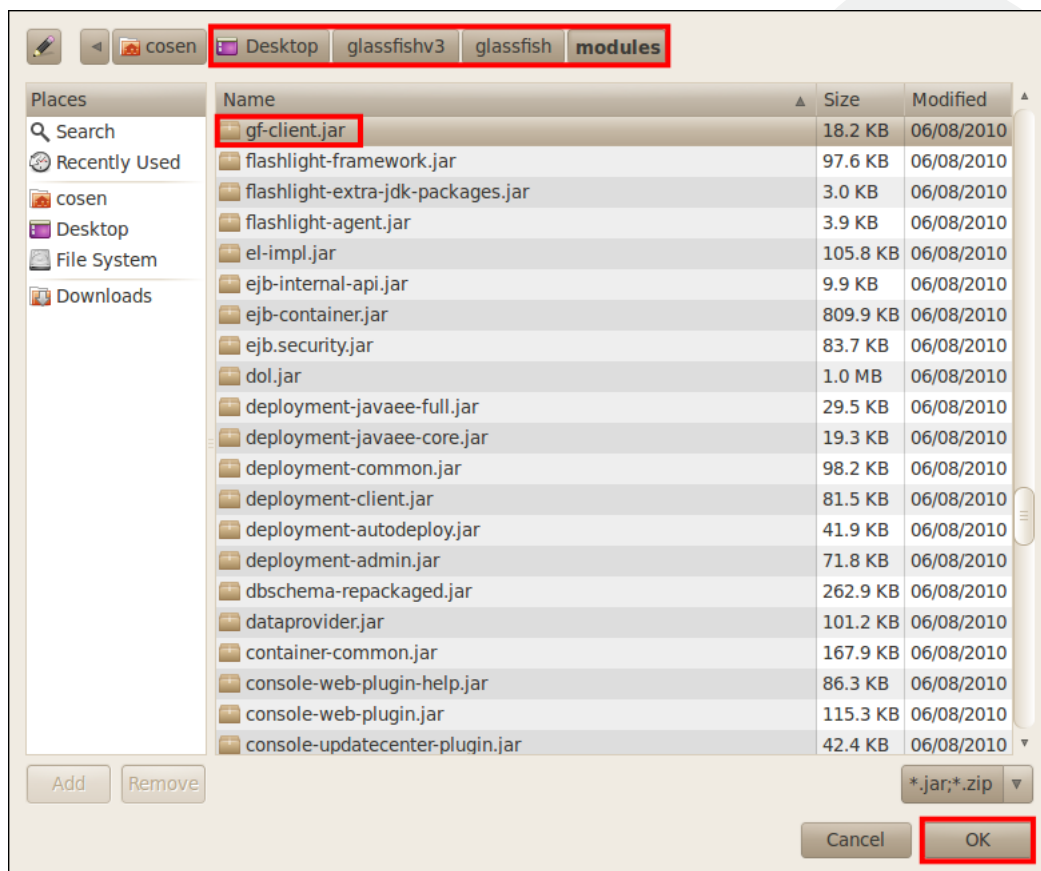
```
30 // fechando
31 sender.close();
32 session.close();
33 connection.close();
34
35 System.out.println("Mensagem Enviada");
36 System.exit(0);
37 }
38 }
```

13. Execute **uma** vez a classe **EnviaNovoPedido**.

14. Crie um Java Project no eclipse para implementar uma aplicação que possa receber as mensagens da fila e do tópico criados anteriormente. Você pode digitar “CTRL+3” em seguida “new Java Project” e “ENTER”. Depois, siga exatamente as imagens abaixo.







15. Crie um pacote chamado **receptores** no projeto **receptorJMS**.
16. Adicione no pacote **receptores** uma classe com main para receber uma mensagem JMS da fila **pedidos**.

```
1 public class RecebePedido {
2     public static void main(String[] args) throws Exception {
3         // serviço de nomes - JNDI
4         InitialContext ic = new InitialContext();
5
6         // fábrica de conexões JMS
7         ConnectionFactory factory = (ConnectionFactory) ic
8             .lookup("jms/K19Factory");
9
10        // fila
11        Queue queue = (Queue) ic.lookup("jms/pedidos");
12
13        // conexão JMS
14        Connection connection = factory.createConnection();
15
16        // sessão JMS
17        Session session = connection.createSession(false,
18            Session.AUTO_ACKNOWLEDGE);
19
20        // receptor de mensagens
21        MessageConsumer receiver = session.createConsumer(queue);
22
23        // inicializa conexão
24        connection.start();
25
26        // recebendo
27        TextMessage message = (TextMessage) receiver.receive();
28
29        System.out.println(message.getText());
30
31        // fechando
32        receiver.close();
33        session.close();
34        connection.close();
35
36        System.out.println("FIM");
37        System.exit(0);
38    }
39 }
```

17. Execute **uma** vez a classe **RecebePedido**.
18. Adicione no pacote **receptores** uma classe com main para receber uma mensagem JMS do tópico **noticias**.

```
1 public class AssinanteDeNoticias {
2     public static void main(String[] args) throws Exception {
3         // serviço de nomes - JNDI
4         InitialContext ic = new InitialContext();
5
6         // fábrica de conexões JMS
7         ConnectionFactory factory = (ConnectionFactory) ic
8             .lookup("jms/K19Factory");
9
10        // tópico
11        Topic topic = (Topic) ic.lookup("jms/noticias");
12
13        // conexão JMS
14        Connection connection = factory.createConnection();
15
16        // sessão JMS
17        Session session = connection.createSession(false,
```

```
18         Session.AUTO_ACKNOWLEDGE);
19
20         // receptor de mensagens
21         MessageConsumer receiver = session.createConsumer(topic);
22
23         // inicializa conexão
24         connection.start();
25
26         // recebendo
27         TextMessage message = (TextMessage) receiver.receive();
28
29         System.out.println(message.getText());
30
31         // fechando
32         receiver.close();
33         session.close();
34         connection.close();
35
36         System.out.println("FIM");
37         System.exit(0);
38     }
39 }
```

19. Execute **duas** vez a classe **AssinanteDeNoticias**.

20. Adicione no pacote **emissores** uma classe com main para enviar uma mensagem JMS para o tópico **noticias**.

```
1 public class EnviaNoticia {
2     public static void main(String[] args) throws Exception {
3         // serviço de nomes - JNDI
4         InitialContext ic = new InitialContext();
5
6         // fábrica de conexões JMS
7         ConnectionFactory factory = (ConnectionFactory)ic.lookup("jms/K19Factory");
8
9         // tópico
10        Topic topic = (Topic)ic.lookup("jms/noticias");
11
12        // conexão JMS
13        Connection connection = factory.createConnection();
14
15        // sessão JMS
16        Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
17
18        // emissor de mensagens
19        MessageProducer sender = session.createProducer(topic);
20
21        // mensagem
22        TextMessage message = session.createTextMessage();
23        message.setText("A copa do mundo de 2014 será no Brasil - " + System.↵
24                        currentTimeMillis());
25
26        // enviando
27        sender.send(message);
28
29        // fechando
30        sender.close();
31        session.close();
32        connection.close();
33
34        System.out.println("Mensagem Enviada");
35        System.exit(0);
36    }
37 }
```

21. Execute **uma** vez a classe **EnviaNoticia** e observe os consoles dos assinantes.

1.8 Modos de recebimento

As mensagens JMS podem ser recebidas através de um **MessageConsumer** de três maneiras diferentes:

Bloqueante: A execução não continua até o recebimento da mensagem.

```
1 receiver.receive();
```

Semi-Bloqueante: A execução não continua até o recebimento da mensagem ou até o término de um período estipulado.

```
1 // espera no máximo 5 segundos
2 receiver.receive(5000);
```

Não-Bloqueante: A execução não é interrompida se a mensagem não for recebida imediatamente.

```
1 receiver.receiveNoWait();
```

1.9 Percorrendo uma fila

Podemos percorrer as mensagens de uma fila sem retirá-las de lá. Para isso, devemos utilizar um **QueueBrowser**.

```
1 QueueBrowser queueBrowser = session.createBrowser(queue);
2
3 Enumeration<TextMessage> messages = queueBrowser.getEnumeration();
4 while (messages.hasMoreElements()) {
5     TextMessage message = messages.nextElement();
6 }
```

1.10 Exercícios

22. Adicione no pacote **receptores** uma classe com main para percorrer as mensagens da fila **pedidos**.

```
1 public class PercorrendoFila {
2     public static void main(String[] args) throws Exception {
3         // serviço de nomes - JNDI
4         InitialContext ic = new InitialContext();
5
6         // fábrica de conexões JMS
7         ConnectionFactory factory = (ConnectionFactory) ic
8             .lookup("jms/K19Factory");
9
10        // fila
11        Queue queue = (Queue) ic.lookup("jms/pedidos");
12
13        // conexão JMS
14        Connection connection = factory.createConnection();
15
16        // sessão JMS
17        Session session = connection.createSession(false,
18            Session.AUTO_ACKNOWLEDGE);
19    }
```

```

20 // queue browser
21 QueueBrowser queueBrowser = session.createBrowser(queue);
22
23 Enumeration<TextMessage> messages = queueBrowser.getEnumeration();
24 int count = 1;
25 while (messages.hasMoreElements()) {
26     TextMessage message = messages.nextElement();
27     System.out.println(count + " : " + message.getText());
28     count++;
29 }
30
31 // fechando
32 queueBrowser.close();
33 session.close();
34 connection.close();
35
36 System.out.println("FIM");
37 System.exit(0);
38 }
39 }

```

23. Execute algumas vezes a classe **EnviaNovoPedido** para popular a fila **pedidos** e depois execute a classe **PercorrendoFila**.

1.11 Selecionando mensagens de um tópico

Podemos anexar propriedades às mensagens enviadas a um tópico JMS. As propriedades podem servir como filtro para os assinantes do tópico selecionarem as mensagens que eles desejam receber.

O código abaixo acrescenta uma propriedade a uma mensagem JMS.

```

1 message.setStringProperty("categoria", "esporte");

```

Quando um receptor é criado associado a um tópico, podemos aplicar o critério de seleção das mensagens desejadas.

```

1 MessageConsumer receiver = session.createConsumer(topic, "(categoria = 'esporte')");

```

1.12 Exercícios

24. Altere a classe **AssinanteDeNoticias** para selecionar somente as mensagens de esporte. Observe o trecho de código que você deve alterar:

```

1 // receptor de mensagens
2 MessageConsumer receiver = session.createConsumer(topic, "(categoria = 'esporte')");

```

25. Altere a classe **EnviaNoticia** acrescentando uma propriedade à mensagem enviada. Observe o trecho de código que você deve alterar:

```

1 // mensagem
2 TextMessage message = session.createTextMessage();
3 message.setStringProperty("categoria", "esporte");
4 message.setText("A copa do mundo de 2014 será no Brasil - " + System.currentTimeMillis() + "\n");

```

26. Execute uma vez a classe **AssinanteDeNoticias** e depois a classe **EnviaNoticia**. Observe que a mensagem é recebida pelo assinante.

27. Altere o valor da propriedade da mensagem enviada.

```
1 // mensagem
2 TextMessage message = session.createTextMessage();
3 message.setStringProperty("categoria", "geral");
4 message.setText("A copa do mundo de 2014 será no Brasil - " + System.currentTimeMillis()
  ());
```

28. Execute uma vez a classe **EnviaNoticia**. Observe que agora a mensagem não é recebida pelo assinante.

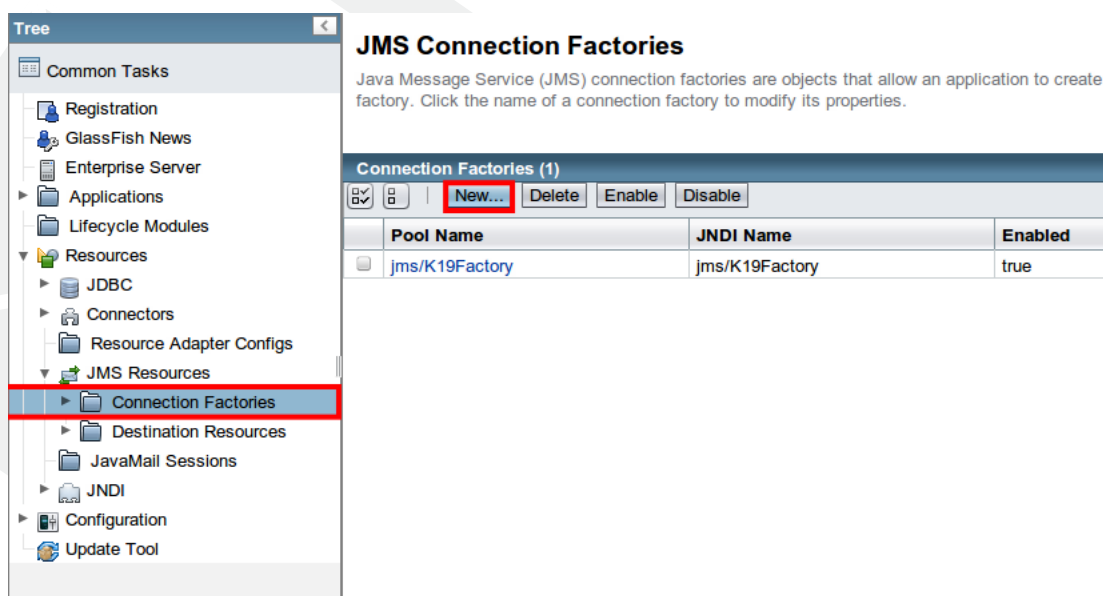
1.13 Tópicos Duráveis

As mensagens enviadas para tópicos JMS normais só podem ser recebidas pelos assinantes que estiverem conectados no momento do envio. Dessa forma, se um assinante estiver desconectado ele perderá as mensagens enviadas durante o período off-line.

A especificação JMS define um tipo especial de tópico que armazena as mensagens dos assinantes desconectados e as envia assim que eles se conectarem. Esses são os **tópicos duráveis**.

1.14 Exercícios

29. Crie uma fábrica para tópicos duráveis através da interface de administração do Glassfish seguindo exatamente os passos das imagens abaixo:



New JMS Connection Factory

The creation of a new Java Message Service (JMS) connection factory also creates a connector connection pool for the factory and a connector resource.

General Settings

Pool Name: *

Resource Type: *

Description:

Status: ☒ Enabled

Additional Properties (1)	
<input checked="" type="checkbox"/>	<input type="text" value="Clientid"/>
<input type="text" value="Id"/>	

30. Adicione no pacote **receptores** uma classe com main para receber uma mensagem JMS do tópico **noticias** inclusive as enviadas enquanto a aplicação estiver off-line.

```

1 public class AssinanteDuravel {
2     public static void main(String[] args) throws Exception {
3         // serviço de nomes - JNDI
4         InitialContext ic = new InitialContext();
5
6         // fábrica de conexões JMS
7         ConnectionFactory factory = (ConnectionFactory) ic
8             .lookup("jms/K19DurableFactory");
9
10        // tópico
11        Topic topic = (Topic) ic.lookup("jms/noticias");
12
13        // conexão JMS
14        Connection connection = factory.createConnection();
15
16        // sessão JMS
17        Session session = connection.createSession(false,
18            Session.AUTO_ACKNOWLEDGE);
19
20        // receptor de mensagens
21        MessageConsumer receiver = session.createDurableSubscriber(topic,
22            "Assinante1");
23
24        // inicializa conexão
25        connection.start();
26
27        // recebendo
28        TextMessage message = (TextMessage) receiver.receive(2000);
29
30        while(message != null){
31            System.out.println(message.getText());
32            message = (TextMessage) receiver.receive(2000);
33        }
34
35        // fechando
36        receiver.close();
37        session.close();
38        connection.close();
39
40        System.out.println("FIM");
41        System.exit(0);
42    }
43 }

```


31. Adicione no pacote **emissores** uma classe com main para enviar uma mensagem JMS durável para o tópico **noticias**.

```

1 public class EnviaNoticiaDuravel {
2     public static void main(String[] args) throws Exception {
3         // serviço de nomes - JNDI
4         InitialContext ic = new InitialContext();
5
6         // fábrica de conexões JMS
7         ConnectionFactory factory = (ConnectionFactory)ic.lookup("jms/↵
            K19DurableFactory");
8
9         // tópico
10        Topic topic = (Topic)ic.lookup("jms/noticias");
11
12        // conexão JMS
13        Connection connection = factory.createConnection();
14
15        // sessão JMS
16        Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
17
18        // emissor de mensagens
19        MessageProducer sender = session.createProducer(topic);
20
21        // mensagem
22        TextMessage message = session.createTextMessage();
23        message.setText("A copa do mundo de 2014 será no Brasil - " + System.↵
            currentTimeMillis());
24
25        // enviando
26        sender.send(message);
27
28        // fechando
29        sender.close();
30        session.close();
31        connection.close();
32
33        System.out.println("Mensagem Enviada");
34        System.exit(0);
35    }
36 }

```

32. Execute **uma** vez a classe **AssinanteDuravel** para cadastrar um assinante.
33. Execute algumas vezes a classe **EnviaNoticiaDuravel**. Perceba que o assinante está offline.
34. Execute a classe **AssinanteDuravel** para receber as mensagens enviadas enquanto o assinante estava offline.

1.15 JMS e EJB

A arquitetura JMS é intrinsecamente relacionada com a arquitetura EJB. Uma aplicação EJB pode receber e processar mensagens JMS de uma forma simples e eficiente.

A especificação EJB define um tipo de objeto especializado no recebimento de mensagens JMS. Esses objetos são os **Message Driven Beans (MDBs)**. Para implementar um MDB, devemos aplicar a anotação **@MessageDriven** e implementar a interface **MessageListener**.

```

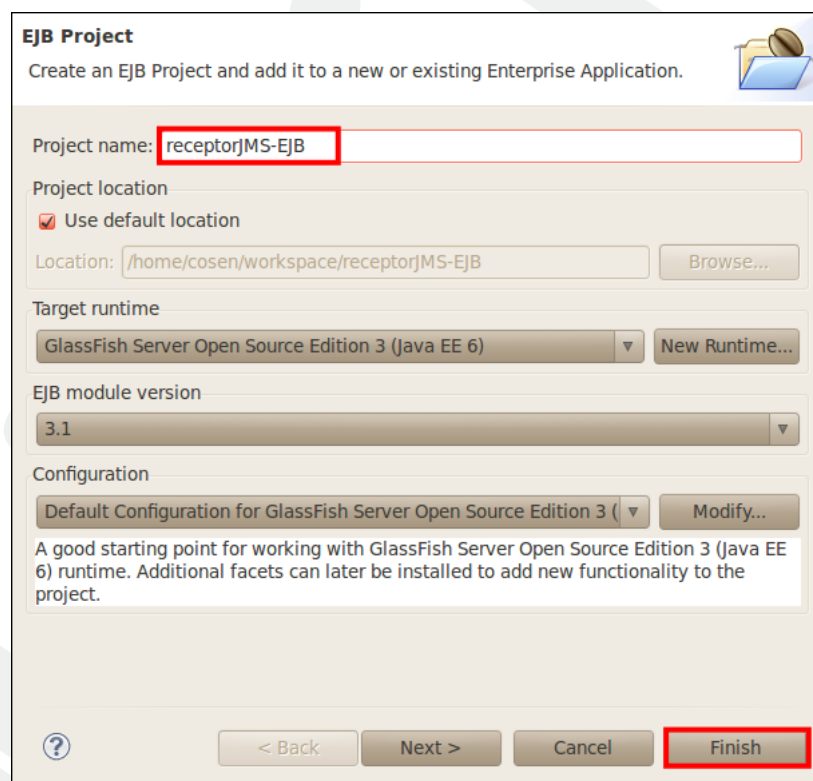
1 @MessageDriven(mappedName="jms/destination1")
2 public class TratadorDeMensagensMDB implements MessageListener{
3

```

```
4  @Override
5  public void onMessage(Message message) {
6
7      try {
8          TextMessage msg = (TextMessage) message;
9          System.out.println(msg.getText());
10     } catch (JMSEException e) {
11         System.out.println("erro");
12     }
13 }
14 }
```

1.16 Exercícios

35. Crie um EJB Project no eclipse para implementar uma aplicação que possa receber as mensagens do tópico **noticias** criado anteriormente. Você pode digitar “CTRL+3” em seguida “new EJB Project” e “ENTER”. Depois, siga exatamente a imagem abaixo.



36. Crie um pacote chamado **mdbs** no projeto **receptorJMS-EJB**.
37. Adicione no pacote **mdbs** um Message Driven Bean para receber e processar as mensagens do tópico **noticias**.

```
1  @MessageDriven(mappedName="jms/noticias")
2  public class TratadorDeMensagensMDB implements MessageListener{
3
4      @Override
5      public void onMessage(Message message) {
6
```

```
7      try {
8          TextMessage msg = (TextMessage) message;
9          System.out.println(msg.getText());
10     } catch (JMSException e) {
11         System.out.println("erro");
12     }
13 }
14 }
```

38. Implante o projeto **receptorJMS-EJB** no Glassfish através da view **Servers**.

39. Envie uma mensagem para o tópico **noticias** e observe o console do eclipse.

1.17 Projeto - Rede de Hotéis

Para praticar os conceitos da arquitetura JMS, vamos implementar aplicações que devem se comunicar de forma automatizada. Suponha que uma rede de hotéis possua uma aplicação central para administrar os pagamentos realizados pelos clientes em qualquer unidade da rede. Cada hotel é controlado por uma aplicação local que deve enviar os dados necessários de cada pagamento para aplicação central.

1.18 Exercícios

40. Adicione uma fila com o seguinte JNDI Name **jms/pagamentos** no Glassfish.

41. Implemente a aplicação local que deve executar em cada unidade da rede de hotéis através de uma aplicação Java SE. Essa aplicação deve obter os dados dos pagamentos através do Console do Eclipse e enviar uma mensagem JMS para a fila **jms/pagamentos** com esses dados. Utilize a classe **Scanner** para obter os dados digitados do Console.

```
1 // DICA
2 Scanner entrada = new Scanner(System.in);
3 String linha = entrada.nextLine();
```

42. Implemente a aplicação central da rede de hotéis que deve tratar todas as mensagens enviada à fila **jms/pagamentos** através de uma aplicação EJB. Essa aplicação deve utilizar Message Driven Beans. Simplesmente imprima no Console do Eclipse os dados extraídos das mensagens recebidas.

43. (Opcional) Faça a aplicação central enviar uma mensagem de confirmação para as aplicações locais a cada mensagem de pagamento processada. Dica: Utilize um tópico JMS e faça cada aplicação local filtrar as mensagens desse tópico que correspondem a pagamentos que ela enviou à aplicação central.

Capítulo 2

JAX-WS

2.1 Web Services

Muitas vezes, é necessário que os serviços oferecidos por uma organização sejam acessados diretamente pelos sistemas de outras organizações sem intervenção humana. Por exemplo, o Ministério da Fazenda do Brasil introduziu recentemente uma nova sistemática de emissão de notas fiscais. Hoje, as empresas podem descartar o método tradicional de emissão em papel e utilizar a emissão eletrônica. Inclusive, o serviço de emissão de nota fiscal eletrônica pode ser acessado diretamente pelos sistemas de cada empresa, automatizando o processo e consequentemente diminuindo gastos.

Diversos outros serviços possuem características semelhantes:

- Cotação de Moedas
- Previsão do Tempo
- Verificação de CPF ou CPNJ
- Cotação de Frete

Além da necessidade de serem acessados diretamente por sistemas e não por pessoas, geralmente, esses serviços devem ser disponibilizados através da internet para atingir um grande número de sistemas usuários. Nesse cenário, outra condição importante é que não exista nenhuma restrição quanto a plataforma utilizada em cada sistema.

Daí surge o conceito de **web service**. Resumidamente, um web service é um serviço oferecido por um sistema que pode ser acessado diretamente por outro sistema desenvolvido em qualquer tecnologia através de uma rede como a internet.

Cada plataforma oferece os recursos necessários para que os desenvolvedores possam disponibilizar ou acessar web services. A organização **W3C** define alguns padrões para definir o funcionamento de um web service. Em geral, as plataformas de maior uso comercial implementam a arquitetura definida pelos padrões da W3C.

Na plataforma Java, há especificações que definem a implementação Java dos padrões estabelecidos pelo W3C. A especificação java diretamente relacionada a Web Services que seguem os padrões da W3C é a **Java API for XML-Based Web Services - JAX-WS**. A JAX-WS depende fortemente de outra especificação Java, a **JAXB Java Architecture for XML Binding**.

A seguir mostraremos o funcionamento básico dos recursos definidos pela especificação JAXB e a arquitetura definida pela JAX-WS.

2.2 JAXB

A ideia principal da JAXB é definir o mapeamento e transformação dos dados de uma aplicação Java para XML e vice versa. Com os recursos do JAXB podemos transformar uma árvore de objetos Java em texto XML ou vice versa.

Suponha uma simples classe Java para modelar contas bancárias.

```
1 class Conta {  
2     private double saldo;  
3  
4     private double limite;  
5  
6     //GETTERS AND SETTERS  
7 }
```

Para poder transformar objetos da classe conta em texto XML, devemos aplicar a anotação **@XMLRootElement**.

```
1 @XMLRootElement  
2 class Conta {  
3     ...  
4 }
```

O contexto do JAXB deve ser criado para que as anotações de mapeamento possam ser processadas. O seguinte código cria o contexto do JAXB.

```
1 JAXBContext context = JAXBContext.newInstance(Conta.class);
```

O processo de transformar um objeto Java em texto XML é chamado de **marshal** ou **serialização**. Para serializar um objeto Java em XML, devemos obter através do contexto do JAXB um objeto da interface **Marshaller**.

```
1 Marshaller marshaller = context.createMarshaller();
```

Por fim, podemos criar um objeto da classe CONTA e utilizar um Marshaller para serializá-lo em XML e guardar o conteúdo em um arquivo.

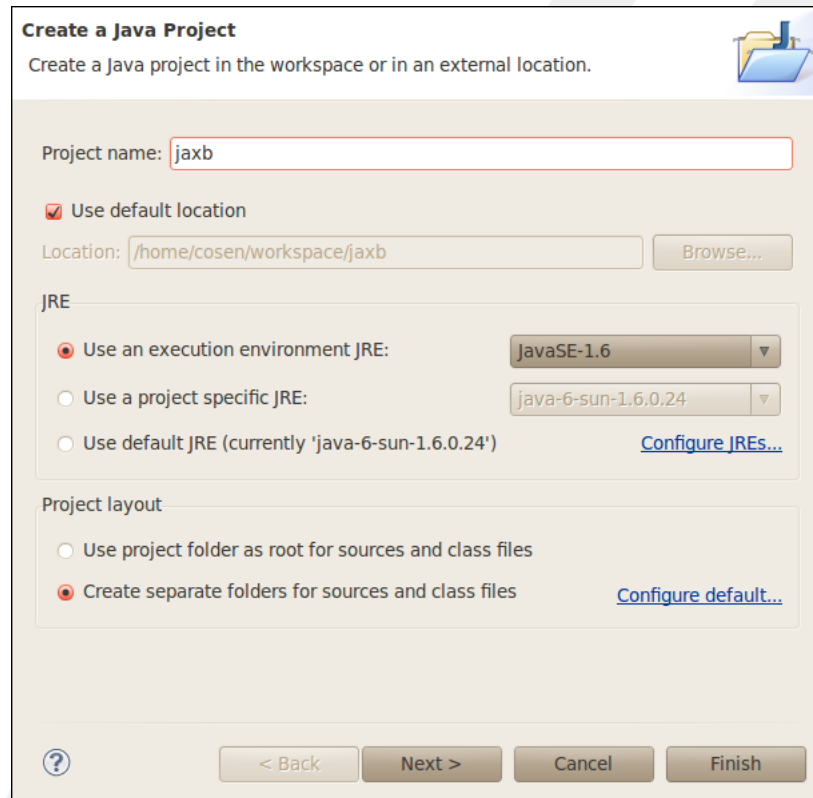
```
1 Conta conta = new Conta();  
2 conta.setLimite(2000);  
3 conta.setSaldo(2000);  
4  
5 marshaller.marshal(conta, new File("conta.xml"));
```

O processo inverso, ou seja, transformar um texto XML em um objeto da classe conta é denominado **unmarshal** ou **deserialização**. Para deserializar um XML, devemos obter através do contexto do JAXB um objeto da interface **Unmarshaller**.

```
1 Unmarshaller unmarshaller = context.createUnmarshaller();  
2  
3 Conta conta = (Conta) unmarshaller.unmarshal(new File("conta.xml"));
```

2.3 Exercícios

1. Crie um Java Project no eclipse para testar o funcionamento básico dos recursos definidos pela especificação JAXB. Você pode digitar “CTRL+3” em seguida “new Java Project” e “ENTER”. Depois, siga exatamente a imagem abaixo.



2. Crie um pacote chamado **jaxb** no projeto **jaxb**.
3. Adicione no pacote **jaxb** uma classe para modelar contas bancárias. Aplique a anotação **@XmlElement**.

```
1 @XmlElement
2 public class Conta {
3     private double saldo;
4
5     private double limite;
6
7     // GETTERS AND SETTERS
8 }
```

4. Adicione no pacote **jaxb** uma classe com main para serializar um objeto da classe **CONTA**.

```
1 public class Serializador {
2     public static void main(String[] args) throws JAXBException {
3
4         JAXBContext context = JAXBContext.newInstance(Conta.class);
5         Marshaller marshaller = context.createMarshaller();
6
7         Conta conta = new Conta();
8         conta.setLimite(2000);
```

```

9      conta.setSaldo(2000);
10     marshaller.marshal(conta, new File("conta.xml"));
11 }
12 }

```

5. Execute a classe **Serializador** e atualize o projeto para o arquivo **conta.xml** aparecer. Observe o conteúdo XML gerado.
6. Adicione no pacote **jaxb** uma classe com main para deserializar o XML criado anteriormente.

```

1 public class Deserializador {
2     public static void main(String[] args) throws JAXBException {
3
4         JAXBContext context = JAXBContext.newInstance(Conta.class);
5         Unmarshaller unmarshaller = context.createUnmarshaller();
6
7         Conta conta = (Conta) unmarshaller.unmarshal(new File("conta.xml"));
8
9         System.out.println(conta.getLimite());
10        System.out.println(conta.getSaldo());
11    }
12 }

```

7. Execute a classe **Deserializador**.
8. Adicione no pacote **jaxb** uma classe para modelar os clientes donos das contas bancárias.

```

1 public class Cliente {
2     private String nome;
3
4     // GETTERS AND SETTERS
5 }

```

9. Altere a classe **Conta** para estabelecer um vínculo com a classe **CLIENTE**.

```

1 @XmlRootElement
2 public class Conta {
3     private double saldo;
4
5     private double limite;
6
7     private Cliente cliente;
8
9     // GETTERS AND SETTERS
10 }

```

10. Altere a classe **Serializacao** e a teste novamente.

```

1 public class Serializador {
2     public static void main(String[] args) throws JAXBException {
3
4         JAXBContext context = JAXBContext.newInstance(Conta.class);
5         Marshaller marshaller = context.createMarshaller();
6
7         Cliente cliente = new Cliente();
8         cliente.setNome("Rafael Cosentino");
9
10        Conta conta = new Conta();
11        conta.setLimite(2000);
12        conta.setSaldo(2000);
13        conta.setCliente(cliente);
14    }

```

```
15     marshaller.marshal(conta, new File("conta.xml"));
16 }
17 }
```

11. Altere a classe **Deserializacao** e a teste novamente.

```
1 public class Deserializador {
2     public static void main(String[] args) throws JAXBException {
3
4         JAXBContext context = JAXBContext.newInstance(Conta.class);
5         Unmarshaller unmarshaller = context.createUnmarshaller();
6
7         Conta conta = (Conta) unmarshaller.unmarshal(new File("conta.xml"));
8
9         System.out.println(conta.getLimite());
10        System.out.println(conta.getSaldo());
11        System.out.println(conta.getCliente().getNome());
12    }
13 }
```

2.4 Criando um web service - Java SE

Para começar, implementaremos um serviço e o disponibilizaremos como Web Service através dos recursos definido pela JAX-WS. Lembrando que a especificação JAX-WS é compatível com os padrões do W3C. Essa implementação será realizada em ambiente Java SE.

Para exemplificar, suponha um serviço para gerar números aleatórios. A lógica desse serviço pode ser definida através de um método Java.

```
1 class Gerador {
2     public double geraNumero() {
3         return Math.random() * 200;
4     }
5 }
```

Para que essa classe seja interpretada como um web service devemos aplicar a anotação **@WebService**.

```
1 @WebService
2 class Gerador {
3     ...
4 }
```

Podemos publicar o serviço implementado pela classe **Gerador** através da classe **Endpoint**.

```
1 public class Publicador {
2     public static void main(String[] args) {
3         System.out.println("web service - Gerador Inicializado");
4         Gerador gerador = new Gerador();
5         Endpoint.publish("http://localhost:8080/gerador", geradorDeNumeros);
6     }
7 }
```

A definição do web service em **WSDL** pode ser consultada através da url <http://localhost:8080/gerador?wsdl>.

2.5 Consumindo um web service com JAX-WS

Agora, implementaremos um cliente de um web service que seja compatível com os padrões W3C utilizando os recursos do JAX-WS. O primeiro passo é utilizar a ferramenta **wsimport** para gerar as classes necessárias para acessar o web service a partir da definição do mesmo em WSDL. As classes geradas pelo wsimport não devem ser alteradas.

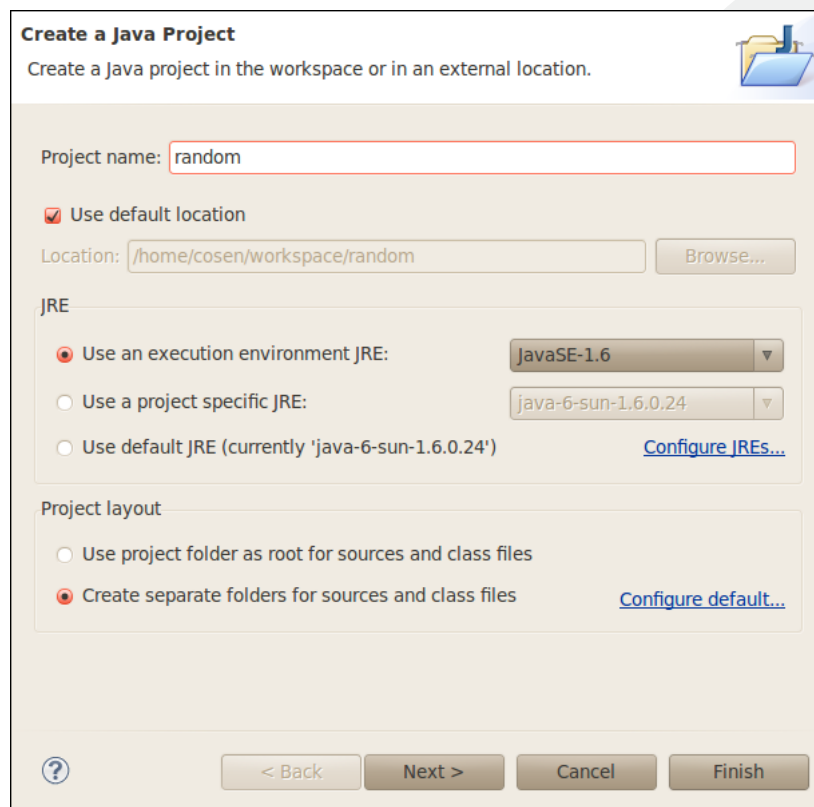
```
1 wsimport -keep http://localhost:8080/gerador?wsdl
```

Com o apoio das classes geradas pelo wsimport podemos consumir o web service.

```
1 public class Consumidor {  
2     public static void main(String[] args) {  
3         // Serviço  
4         GeradorDeNumerosService service = new GeradorDeNumerosService();  
5  
6         // Proxy  
7         GeradorDeNumeros proxy = service.getGeradorDeNumerosPort();  
8  
9         // Consumindo  
10        double numero = proxy.geraNumero();  
11  
12        System.out.println(numero);  
13    }  
14 }
```

2.6 Exercícios

12. Crie um Java Project no eclipse para implementar um web service que gere números aleatórios. Você pode digitar “CTRL+3” em seguida “new Java Project” e “ENTER”. Depois, siga exatamente a imagem abaixo.



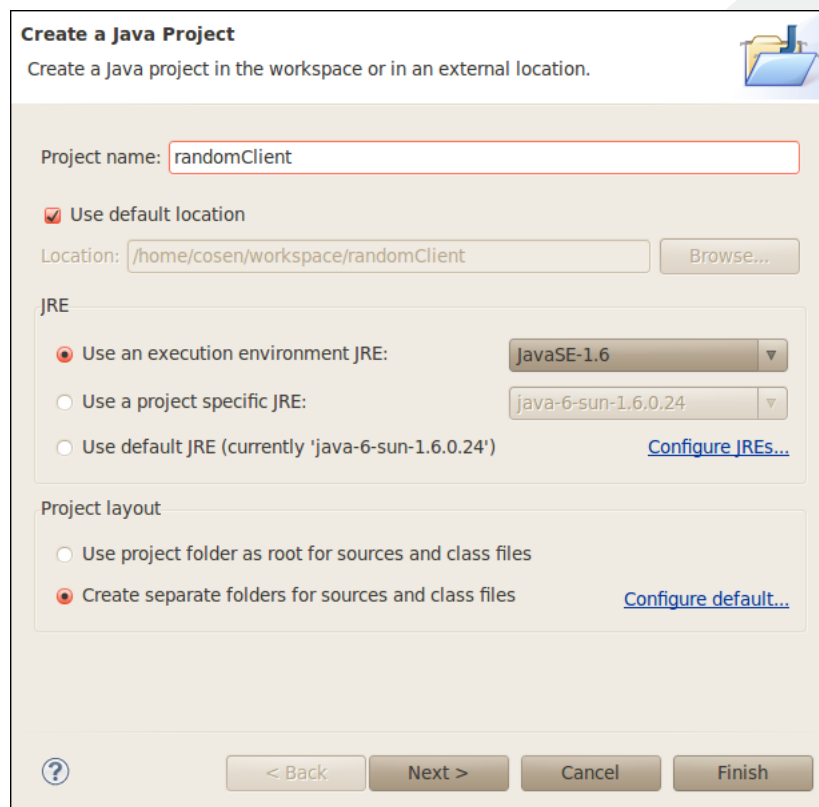
13. Crie um pacote chamado **webservices** no projeto **random**.
14. Adicione no pacote **webservices** uma classe para implementar o serviço de gerar números aleatórios.

```
1 @WebService
2 public class Random {
3     public double next(double max){
4         return Math.random() * max;
5     }
6 }
```

15. Adicione no pacote **webservices** uma classe com main para publicar o web service.

```
1 public class RandomPublisher {
2     public static void main(String[] args) {
3         System.out.println("Random web service start...");
4         Random random = new Random();
5         Endpoint.publish("http://localhost:8080/random", random);
6     }
7 }
```

16. Execute a classe **RandomPublisher**.
17. Consulte através de um navegador a url <http://localhost:8080/random?wsdl> para conferir a definição do web service em WSDL.
18. Crie um Java Project no eclipse para consumir o web service que gera números aleatórios implementado anteriormente. Você pode digitar “CTRL+3” em seguida “new Java Project” e “ENTER”. Depois, siga exatamente a imagem abaixo.



19. Gere as classes necessárias para consumir o web service através da ferramenta **wsimport**. Abra um terminal e siga os passos abaixo.

```
File Edit View Terminal Help
cosen@cosen-laptop:~$ cd workspace/randomClient/src/
cosen@cosen-laptop:~/workspace/randomClient/src$
(End)java)

Item Execute a classe \destaque(RandomPublisher).

Item Consulte através de um navegador a url
(url:http://localhost:8080/random?wsdl) para conferir a definição do web
service em WSDL.

Item Crie um Java Project no eclipse para consumir o web service que gera
números aleatórios implementado anteriormente. Você pode digitar 'CTRL+3' em
seguida 'new Java Project' e 'ENTER'. Depois, siga exatamente a imagem abaixo.
```

```
File Edit View Terminal Help
cosen@cosen-laptop:~/workspace/randomClient/src$ wsimport -keep http://localhost:8080/random?wsdl
parsing WSDL...
(End)java)

generating code...lasse \destaque(RandomPublisher).

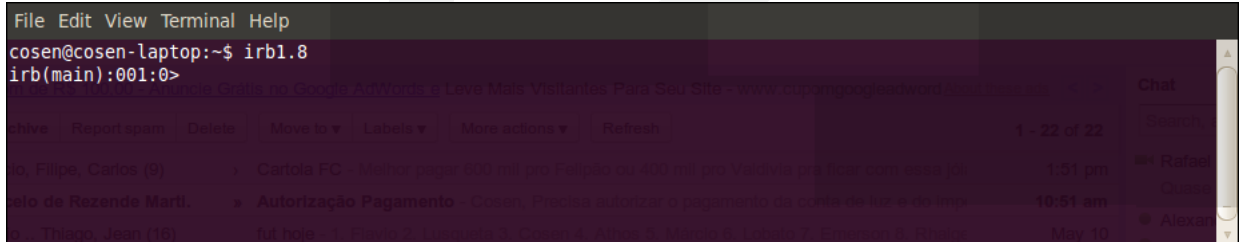
Item Consulte através de um navegador a url
compiling code...alhost:8080/random?wsdl) para conferir a definição do web
service em WSDL.
cosen@cosen-laptop:~/workspace/randomClient/src$
(End)java)

Item Crie um Java Project no eclipse para consumir o web service que gera
números aleatórios implementado anteriormente. Você pode digitar 'CTRL+3' em
seguida 'new Java Project' e 'ENTER'. Depois, siga exatamente a imagem abaixo.
```

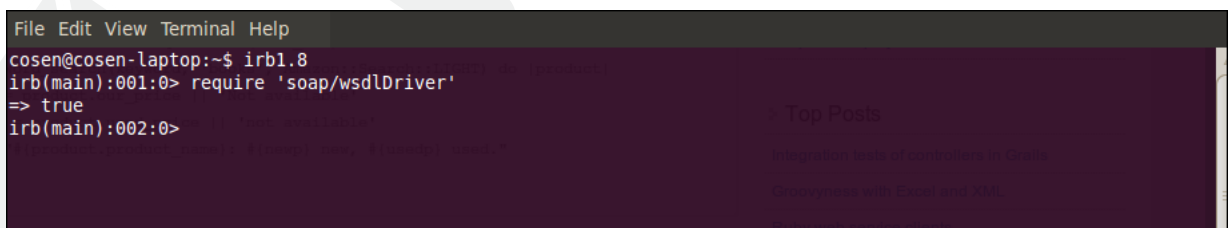
20. Atualize o projeto **randomClient**. Clique com o botão direito em cima desse projeto e depois clique com o esquerdo em **refresh**.
21. Adicione no pacote **webservices** do projeto **randomClient** uma classe para consumir o serviço de gerar números aleatórios.

```
1 public class Consumer {
2     public static void main(String[] args) {
3         // service
4         RandomService randomService = new RandomService();
5
6         // proxy
7         Random proxy = randomService.getRandomPort();
8
9         // operation
10        double next = proxy.next(50);
11        System.out.println(next);
12    }
13 }
```

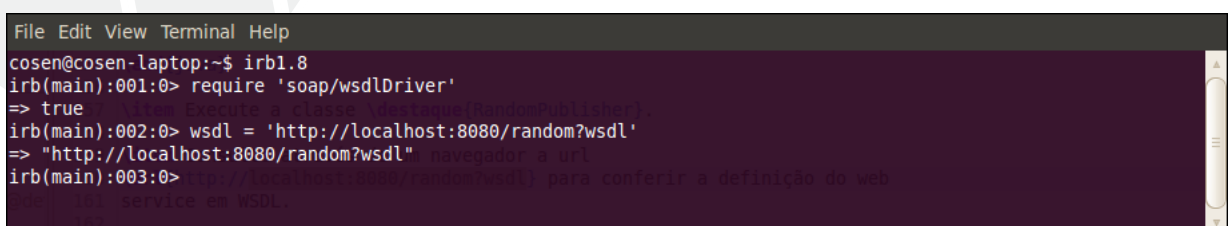
22. Implemente outro consumidor só que agora em Ruby. Abra um terminal e siga os passos abaixo.



```
File Edit View Terminal Help
cosen@cosen-laptop:~$ irb1.8
irb(main):001:0>
```



```
File Edit View Terminal Help
cosen@cosen-laptop:~$ irb1.8
irb(main):001:0> require 'soap/wSDLDriver'
=> true
irb(main):002:0>
```



```
File Edit View Terminal Help
cosen@cosen-laptop:~$ irb1.8
irb(main):001:0> require 'soap/wSDLDriver'
=> true
irb(main):002:0> wsdl = 'http://localhost:8080/random?wsdl'
=> "http://localhost:8080/random?wsdl"
irb(main):003:0>
```

```
File Edit View Terminal Help
cosen@cosen-laptop:~$ irb1.8
irb(main):001:0> require 'soap/wsdlDriver'
=> true
irb(main):002:0> wsdl = 'http://localhost:8080/random?wsdl'
=> "http://localhost:8080/random?wsdl"
irb(main):003:0> proxy = SOAP::WSDLDriverFactory.new(wsdl).create_rpc_driver
ignored attr: {}version
=> #<SOAP::RPC::Driver:#<SOAP::RPC::Proxy:http://localhost:8080/random>>
irb(main):004:0>
```

```
File Edit View Terminal Help
cosen@cosen-laptop:~$ irb1.8
irb(main):001:0> require 'soap/wsdlDriver'
=> true
irb(main):002:0> wsdl = 'http://localhost:8080/random?wsdl'
=> "http://localhost:8080/random?wsdl"
irb(main):003:0> proxy = SOAP::WSDLDriverFactory.new(wsdl).create_rpc_driver
ignored attr: {}version
=> #<SOAP::RPC::Driver:#<SOAP::RPC::Proxy:http://localhost:8080/random>>
irb(main):004:0> proxy.next(:arg0 => 50).return
=> "22.9980548624178"
irb(main):005:0>
```

2.7 JAX-WS e EJB

Os recursos da arquitetura EJB podem ser utilizados juntamente com os recursos definidos pela especificação JAX-WS. Podemos expor um Stateless Session Bean ou um Singleton Session Bean como um Web Service que segue os padrões da W3C pois as duas especificações, EJB e JAX-WS, estão relacionadas.

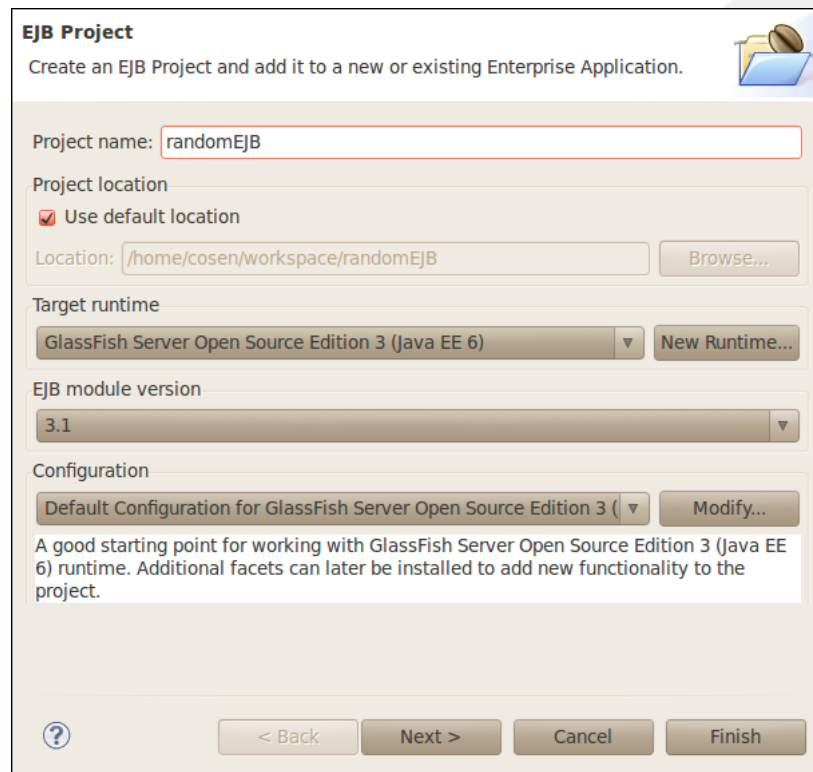
Do ponto de vista da aplicação, basta aplicar a anotação **@WebService** na classe que implementa um Stateless Session Bean ou um Singleton Session Bean.

```
1 @WebService
2 @Stateless
3 public class Random {
4     public double next(double max) {
5         return Math.random() * max;
6     }
7 }
```

O container EJB publicará o web service ao inicializar a aplicação.

2.8 Exercícios

23. Crie um EJB Project no eclipse para implementar um Stateless Session Bean que gere números aleatórios e exponha o como um web service. Você pode digitar “CTRL+3” em seguida “new EJB Project” e “ENTER”. Depois, siga exatamente a imagem abaixo.



24. Crie um pacote chamado **webservices** no projeto **randomEJB**.
25. Adicione no pacote **webservices** uma classe para implementar o Stateless Session Bean.

```
1 @WebService
2 @Stateless
3 public class Random {
4     public double next(double max) {
5         return Math.random() * max;
6     }
7 }
```

26. Implante o projeto **randomEJB** no Glassfish através da view **Servers**.
27. Acesse a url <http://localhost:8080/RandomService/Random?WSDL> para obter o WSDL que define o web service.
28. Acesse a url <http://localhost:8080/RandomService/Random?Tester> para testar o web service através de uma página criada automaticamente pelo Glassfish.
29. Analogamente, implemente um Stateless Session Bean que ofereça as operações fundamentais da matemática.
30. (Opcional) Implemente um cliente Java SE para consumir esse web service.

2.9 Projeto - Táxi no Aeroporto

Para melhorar o atendimento aos seus clientes, a diretoria de um aeroporto decidiu implantar um serviço próprio de táxi. Ela acredita que dessa forma conseguirá oferecer preços

menores e maior qualidade para os seus passageiros. O aeroporto funcionará como um intermediário entre os passageiros e os taxistas. Os pagamentos serão realizados para o aeroporto e depois repassados para o proprietário do táxi. A contratação desse serviço pode ser realizada na chegada do passageiro ao aeroporto ou através da internet.

Como o aeroporto recebe pessoas de todas as partes do mundo, a diretoria quer oferecer a possibilidade dos seus passageiros efetuarem o pagamento do serviço de táxi com dinheiro de qualquer país. Daí surge a necessidade de obter a cotação das moedas. A equipe de TI decidiu obter as cotações através de um web service.

Para a emitir nota fiscal para brasileiros, o serviço de táxi deve verificar a validade do CPF do passageiro ou do CNPJ da empresa que ficará responsável pelo pagamento. Essa verificação deve ser realizada através um web service.

O valor cobrado dos passageiros dependerá do preço atual da gasolina e da distância a ser percorrida. Para obter o preço do combustível e calcular a distância, o serviço de táxi deverá consultar web services.

2.10 Exercícios

31. Implemente com dados fictícios um web service em JAX-WS que informe a cotação das moedas.
32. Implemente com lógica fictícia um web service em JAX-WS que realize a validação de CPF ou CNPJ.
33. Implemente com dados fictícios um web service em JAX-WS que informe o preço da gasolina.
34. Implemente com dados fictícios um web service em JAX-WS que calcule a distância entre dois locais.
35. Crie uma aplicação Java SE para implementar o serviço de táxi do aeroporto. Obtenha da entrada padrão as seguintes informações:
 - (a) Moeda utilizada para o pagamento
 - (b) CPF ou CNPJ
 - (c) Endereço de destino

Acesse os web services criados anteriormente para obter as informações necessárias e imprima na saída padrão o valor a ser pago pelo passageiro.

36. (Opcional) Substitua o web service de cotação de moeda implementado anteriormente pelo web service da **WebServiceX.NET**. O WSDL desse web service pode ser obtido através da <http://www.webs servicex.net/CurrencyConvertor.asmx?WSDL>.
37. (Opcional) Substitua a lógica fictícia de validação de CPF ou CNPJ pela lógica real.
38. (Opcional) Utilize o serviço do google para calcular a distância entre duas localidades.

Capítulo 3

JAX-RS

3.1 REST vs Padrões W3C

No capítulo 2, vimos como implementar web services seguindo os padrões da W3C (WSDL, SOAP e XML). Em geral, a complexidade de evoluir um web service que segue os padrões W3C é alta pois qualquer alteração no serviço implica em uma nova definição em WSDL. Consequentemente, os proxies dos clientes devem ser atualizados.

Como alternativa, podemos desenvolver web services seguindo apenas os princípios do estilo arquitetural **REST** (Representational State Transfer - http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm). Em geral, web services que seguem os princípios REST são mais fáceis de implementar e evoluir.

Na plataforma Java, há especificações que definem o modelo de programação de web services Java que seguem os princípios REST. A principal especificação para esse tipo de web service é a **API for RESTful Web Services JAX-RS**.

A seguir mostraremos os conceitos principais do estilo arquitetural REST funcionamento básico dos recursos definidos pela especificação JAX-RS.

3.2 Resources, URIs, Media Types e Operações

No estilo arquitetural REST, qualquer informação disponível é um **Resource**. O cadastro de uma pessoa, uma imagem, um documento e a cotação de uma moeda são exemplos de resources.

Cada resource deve possuir um identificador único. Esse identificador será utilizado para que o resource possa ser acessado. Na internet ou em uma intranet um web resources é identificado por uma **URI** (Uniform Resource Identifier - <http://tools.ietf.org/html/rfc3986>). Por exemplo, a URI www.k19.com.br/cursos identifica na internet a página com os cursos da K19.

Os resources também podem ser representados em diversos formatos (**Media Type**). Normalmente, na internet ou em uma intranet, seria normal que o cadastro de uma pessoa pudesse ser obtido em **html**, **xml** e **json**.


```

1 <html>
2   <head>
3     <title>Rafael Cosentino</title>
4   </head>
5
6   <body>
7     <h1>Rafael Cosentino</h1>
8     <p>Líder de treinamentos da K19</p>
9   </body>
10 </html>

```

```

1 < Pessoa>
2   < nome>Rafael Cosentino</ nome>
3   < descricao>Líder de treinamentos da K19</ descricao>
4 </ Pessoa>

```

```

1 {"nome": "Rafael Cosentino", "descricao": "Líder de treinamentos da K19"}

```

Em uma arquitetura REST, um conjunto pequeno e fixo de operações deve ser definido previamente. As operações são utilizadas para manipular os recursos de alguma forma.

Por exemplo, na internet ou em uma intranet, os recursos são manipulados pelos métodos do protocolo HTTP. Podemos atribuir uma semântica diferente para cada método HTTP.

Resource	Método HTTP	Semântica
www.k19.com.br/cursos	GET	pega a lista de cursos
www.k19.com.br/cursos	POST	adiciona um curso na lista

3.3 Web service com JAX-RS

A especificação JAX-RS define um modelo de programação para a criação de web services restful (web service que seguem os princípios do estilo arquitetural REST).

3.4 Resources

De acordo com a JAX-RS, os web resources são implementados por classes Java (resource classes). Todo web resource deve possuir uma URI que é definida parcialmente pela anotação **@Path**.

```

1 @Path("/Cotacao")
2 class CotacaoResource {
3   ...
4 }

```

Os métodos HTTP podem ser mapeados para métodos Java de uma resource class. As anotações **@GET**, **@PUT**, **@POST**, **@DELETE** e **@HEAD** são utilizadas para realizar esse

mapeamento.

```
1 @Path("/Cotacao")
2 class CotacaoResource {
3
4     @GET
5     public String getCotacao() {
6         // implementação
7     }
8 }
```

O Media Type que será utilizado para a representação do resource pode ser definido através da anotação **@Produces** e o do enum **MediaType**.

```
1 @Path("/Cotacao")
2 class CotacaoResource {
3
4     @GET
5     @Produces(MediaType.TEXT_PLAIN)
6     public String getCotacao() {
7         // implementação
8     }
9 }
```

3.5 Subresource

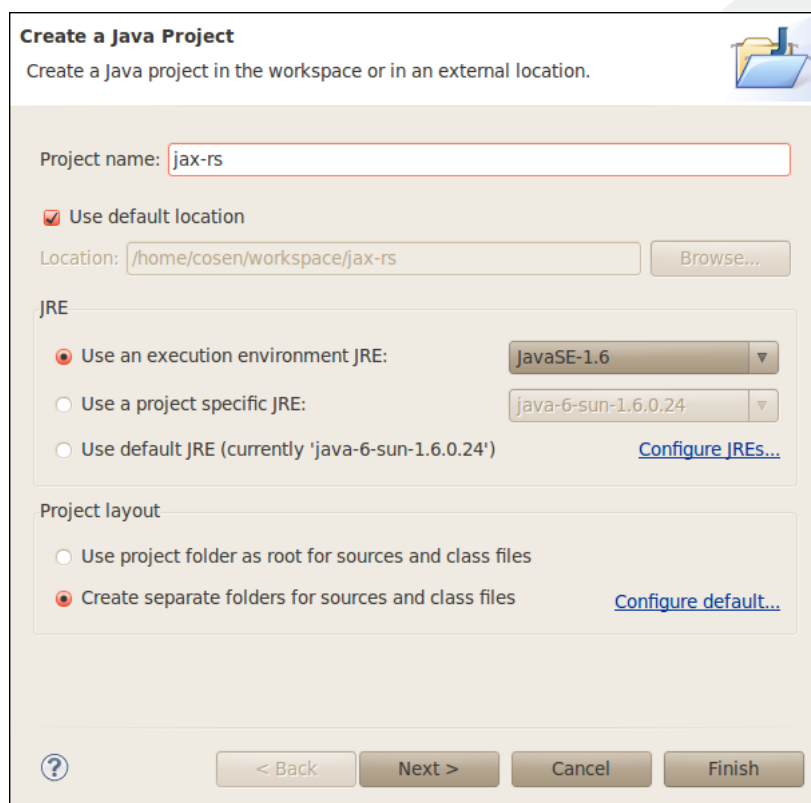
A princípio, uma resource class define apenas um resource. Porém, podemos definir subresources dentro de uma resource class através de métodos anotados com **@Path**.

```
1 @Path("/Cotacao")
2 class CotacaoResource {
3
4     @GET
5     @Path("/DollarToReal")
6     @Produces(MediaType.TEXT_PLAIN)
7     public String getCotacaoDollarToReal() {
8         // implementação
9     }
10
11     @GET
12     @Path("/EuroToReal")
13     @Produces(MediaType.TEXT_PLAIN)
14     public String getCotacaoEuroToReal() {
15         // implementação
16     }
17 }
```

O sufixo da URI de um subresource é definido pela concatenação do valor da anotação **@PATH** aplicada na resource class com o valor da anotação **@PATH** aplicada no método correspondente ao subresource. No exemplo acima, temos dois subresources com URI que possuem os seguintes sufixos: */Cotacao/DollarToReal* e */Cotacao/EuroToReal*.

3.6 Exercícios

1. Crie um Java Project no eclipse para testar o funcionamento básico dos recursos definidos pela especificação JAX-RS. Você pode digitar “CTRL+3” em seguida “new Java Project” e “ENTER”. Depois, siga exatamente a imagem abaixo.



2. JAX-RS é uma especificação. Para testar os recursos definidos nessa especificação temos que escolher uma implementação. Aqui, utilizaremos o projeto Jersey que implementa a JAX-RS. Crie uma pasta chamada **lib** no projeto **jax-rs**.
3. Copie os arquivos **asm-3.1.jar**, **jersey-bundle-1.6.jar** e **jsr311-api-1.1.jar** para a pasta **lib**. Esses arquivos podem ser encontrados na pasta **K19-Arquivos** na sua Área de Trabalho.
4. Adicione esses arquivos no classpath do projeto.
5. Crie um pacote chamado **resources** no projeto **jax-rs**.
6. Adicione no pacote **resources** uma classe para implementar um resource de cotação de moeda.

```

1  @Path("/Cotacao")
2  class CotacaoResource {
3
4      @GET
5      @Path("/DollarToReal")
6      @Produces(MediaType.TEXT_PLAIN)
7      public String getCotacaoDollarToReal() {
8          return "1.6";
9      }
10
11     @GET
12     @Path("/EuroToReal")
13     @Produces(MediaType.TEXT_PLAIN)
14     public String getCotacaoEuroToReal() {
15         return "3.1";
16     }
17 }

```

7. Crie um pacote chamado **main** no projeto **jax-rs**.
8. Adicione no pacote **main** uma classe para publicar o web service.

```
1 public class Publicador {
2     public static void main(String[] args) throws IllegalArgumentException, ↵
3         IOException {
4         HttpServer server = HttpServerFactory.create("http://localhost:8080/");
5         server.start();
6     }
7 }
```

9. Execute a classe **Publicador**.
10. Abra o firefox e acesse as seguintes urls: <http://localhost:8080/Cotacao/DollarToReal> e <http://localhost:8080/Cotacao/EuroToReal>.
11. No firefox abra o Poster (clique no link no canto inferior direito) e faça uma requisição para cada uma das seguintes urls: <http://localhost:8080/Cotacao/DollarToReal> e <http://localhost:8080/Cotacao/EuroToReal>.

3.7 Parâmetros

A especificação JAX-RS define um mecanismo bem simples para injetar os dados contidos nas requisições HTTP nos métodos Java das resource classes. Veremos a seguir que tipo de dados podem ser injetados e como injetá-los.

3.7.1 PathParam

Suponha que desejamos utilizar uris com o seguinte formato para realizar a cotação de moedas.

/Cotacao/Dollar/Real: Valor do Dollar em Real.

/Cotacao/Euro/Real: Valor do Euro em Real.

/Cotacao/Moeda1/Moeda2: Valor da Moeda1 na Moeda2.

Para trabalhar com uris com esse formato, podemos definir parâmetros na URI de um resource através da anotação **@PATHPARAM**.

```
1 @Path("/{M1}/{M2}")
```

Os parâmetros definidos através da anotação **@PATH** podem ser recuperados através da anotação **@PathParam** que deve ser aplicada nos argumentos dos métodos de uma resource class.

```
1 @Path("/Cotacao")
2 public class CotacaoResource {
3
4     @GET
5     @Path("/{M1}/{M2}")
6     @Produces(MediaType.TEXT_PLAIN)
7     public String cotacao(@PathParam("M1") String m1, @PathParam("M2") String m2){
```

```
8      // implementacao
9    }
10 }
```

3.7.2 MatrixParam

Suponha que desejamos utilizar uris com o seguinte formato para realizar a cotação de moedas.

/Cotacao;M1=dollar;M2=real: Valor do Dollar em Real.

/Cotacao;M1=euro;M2=real: Valor do Euro em Real.

/Cotacao;M1=moeda1;M2=moeda2: Valor da Moeda1 na Moeda2.

As URIs acima possuem dois **matrix params**: M1 e M2. Esses parâmetros podem ser recuperados através da anotação **@MatrixParam** que deve ser aplicada nos argumentos dos métodos de uma resource class.

```
1 @Path("/Cotacao")
2 public class CotacaoResource {
3
4     @GET
5     @Produces(MediaType.TEXT_PLAIN)
6     public String cotacao(@MatrixParam("M1") String m1, @MatrixParam("M2") String m2){
7         // implementacao
8     }
9 }
```

3.7.3 QueryParam

Suponha que desejamos utilizar uris com o seguinte formato para realizar a cotação de moedas.

/Cotacao?M1=dollar&M2=real: Valor do Dollar em Real.

/Cotacao?M1=euro&M2=real: Valor do Euro em Real.

/Cotacao?M1=moeda1&M2=moeda2: Valor da Moeda1 na Moeda2.

As URIs acima possuem dois **query params**: M1 e M2. Esses parâmetros podem ser recuperados através da anotação **@QueryParam** que deve ser aplicada nos argumentos dos métodos de uma resource class.

```
1 @Path("/Cotacao")
2 public class CotacaoResource {
3
4     @GET
5     @Produces(MediaType.TEXT_PLAIN)
6     public String cotacao(@QueryParam("M1") String m1, @QueryParam("M2") String m2){
7         // implementacao
8     }
9 }
```

3.7.4 FormParam

Parâmetros enviados através de formulários HTML que utilizam o método POST do HTTP podem ser recuperados através da anotação **@FormParam**.

```
1 <form action="http://www.k19.com.br/Cotacao" method="POST">
2   Moeda1: <input type="text" name="M1">
3   Moeda2: <input type="text" name="M2">
4 </form>
```

```
1 @Path("/Cotacao")
2 public class CotacaoResource {
3
4     @GET
5     @Produces(MediaType.TEXT_PLAIN)
6     public String cotacao(@FormParam("M1") String m1, @FormParam("M2") String m2){
7         // implementacao
8     }
9 }
```

3.7.5 HeaderParam

Os headers HTTP podem ser recuperados através da anotação **@HeaderParam**.

```
1 @Path("/User-Agent")
2 public class UserAgentResource {
3
4     @GET
5     @Produces(MediaType.TEXT_PLAIN)
6     public String userAgent(@HeaderParam("User-Agent") String userAgent){
7         return userAgent;
8     }
9 }
```

3.7.6 CookieParam

Os valores dos cookies enviados nas requisições HTTP podem ser recuperados através da anotação **@CookieParam**.

```
1 @Path("/cookie")
2 public class CookieResource {
3
4     @GET
5     @Produces(MediaType.TEXT_PLAIN)
6     public String userAgent(@CookieParam("clienteID") String clienteID){
7         return clienteID;
8     }
9 }
```

3.8 Exercícios

12. Adicione no pacote **resources** do projeto **jax-rs** uma classe para testar os path params.

```
1 @Path("/path-param")
2 class TestaPathParamResource {
3
```

```

4  @GET
5  @Path("/{p1}/{p2}")
6  @Produces(MediaType.TEXT_PLAIN)
7  public String pathParam(@PathParam("p1") String p1, @PathParam("p2") String p2) {
8      return "P1 = " + p1 + ", P2 = " + p2;
9  }
10 }

```

13. Execute a classe **Publicador**.

14. Acesse as seguintes URIs para testar:

- <http://localhost:8080/path-param/1/2>
- <http://localhost:8080/path-param/java/csharp>

15. (Opcional) Analogamente aos exercícios anteriores teste os matrix params e os query params.

3.9 Produzindo XML ou JSON

A especificação JAX-RS utiliza como base os recursos definidos pela especificação JAXB para produzir XML e JSON. A princípio os recursos do JAXB possibilitam apenas a produção de XML. Contudo, a arquitetura JAXB é flexível e pode ser facilmente estendida para produzir JSON também.

Suponha que seja necessário implementar um web service que manipule a seguinte entidade:

```

1  class Produto {
2      private String nome;
3
4      private Double preco;
5
6      private Long id;
7
8      // GETTERS AND SETTERS
9  }

```

Adicionando a anotação **@XMLRootElement** da especificação JAXB na classe PRODUTO, podemos gerar produtos em XML ou JSON.

```

1  @XMLRootElement
2  class Produto {
3      ...
4  }

```

Agora, basta definir o Media Type nos métodos de uma resource class de acordo com o formato que desejamos utilizar, XML ou JSON.

```

1  @Path("/produtos")
2  class ProdutoResource {
3
4      @GET
5      @Path("/{id}/xml")
6      @Produces(MediaType.APPLICATION_XML)
7      public Produto getProdutoAsXML(@PathParam("id") int id) {
8          // implementacao
9      }
10 }

```

```
11 @GET
12 @Path("/{id}/json")
13 @Produces(MediaType.APPLICATION_JSON)
14 public Produto getProdutoAsJSON(@PathParam("id") int id) {
15     // implementacao
16 }
17 }
```

3.10 Consumindo XML ou JSON

Os recursos do JAXB também são utilizados para consumir XML ou JSON. Novamente suponha a seguinte entidade anotada com `@XMLRootElement`:

```
1 @XMLRootElement
2 class Produto {
3     private String nome;
4
5     private Double preco;
6
7     private Long id;
8
9     // GETTERS AND SETTERS
10 }
```

Nos métodos da resource class, devemos aplicar a anotação `@Consumes` nos métodos.

```
1 @Path("/produtos")
2 public class ProdutoResource {
3     @POST
4     @Consumes(MediaType.APPLICATION_XML)
5     public void adiciona(Produto p) {
6         // implementacao
7     }
8 }
```

```
1 @Path("/produtos")
2 public class ProdutoResource {
3     @POST
4     @Consumes(MediaType.APPLICATION_JSON)
5     public void adiciona(Produto p) {
6         // implementacao
7     }
8 }
```

3.11 Exercícios

16. Crie um pacote chamado **entities** no projeto **jax-rs**.
17. Adicione no pacote **entities** uma classe para modelar produtos.

```
1 @XmlRootElement
2 public class Produto {
3     private String nome;
4
5     private Double preco;
6
7     private Long id;
8
9     // GETTERS AND SETTERS
10 }
```


18. Adicione no pacote **resources** uma classe para produzir produtos em XML e JSON.

```

1  @Path("/produtos")
2  public class ProdutoResource {
3
4      @GET
5      @Path("/{id}/xml")
6      @Produces(MediaType.APPLICATION_XML)
7      public Produto getProdutoAsXML(@PathParam("id") long id) {
8          return this.geraProdutoFalso(id);
9      }
10
11     @GET
12     @Path("/{id}/json")
13     @Produces(MediaType.APPLICATION_JSON)
14     public Produto getProdutoAsJSON(@PathParam("id") long id) {
15         return this.geraProdutoFalso(id);
16     }
17
18     public Produto geraProdutoFalso(long id){
19         Produto p = new Produto();
20         p.setNome("produto" + id);
21         p.setPreco(50.0*id);
22         p.setId(id);
23
24         return p;
25     }
26 }

```

19. Execute a classe **Publicador**.

20. Acesse as seguintes URIs para testar:

- <http://localhost:8080/produtos/1/xml>
- <http://localhost:8080/produtos/1/json>

21. Adicione no pacote **resources** uma classe converter produtos de XML para JSON e vice versa.

```

1  @Path("/produtos/converte")
2  public class ConversorDeProdutoResource {
3
4      @POST
5      @Path("/json/xml")
6      @Consumes(MediaType.APPLICATION_JSON)
7      @Produces(MediaType.APPLICATION_XML)
8      public Produto transformToXML(Produto p) {
9          return p;
10     }
11
12     @POST
13     @Path("/xml/json")
14     @Consumes(MediaType.APPLICATION_XML)
15     @Produces(MediaType.APPLICATION_JSON)
16     public Produto transformToJSON(Produto p) {
17         return p;
18     }
19 }

```

22. Execute a classe **Publicador**.

23. Abra o firefox e depois o Poster (clique no canto inferior direito).

24. Faça uma requisição através do Poster como mostra a imagem abaixo:

The screenshot shows the Poster application interface. The **Request** section has the URL `http://localhost:8080/produtos/converte/xml/json` highlighted in red. The **Actions** section has the **POST** button highlighted in red. The **Content to Send** tab is selected, showing a **Content Type** of `application/xml` (highlighted in red) and **Content Options** set to **Base64 Encode**. The request body, also highlighted in red, contains the following XML:

```
<produto>
<id>1</id>
<nome>produto1</nome>
<preco>50.0</preco>
</produto>
```

25. Agora, faça outra requisição através do Poster como mostra a imagem abaixo:

The screenshot shows the Poster application interface. The **Request** section has the URL `http://localhost:8080/produtos/converte/json/xml` highlighted in red. The **Actions** section has the **POST** button highlighted in red. The **Content to Send** tab is selected, showing a **Content Type** of `application/json` (highlighted in red) and **Content Options** set to **Base64 Encode**. The request body, also highlighted in red, contains the following JSON:

```
{"id": "1", "nome": "produto1", "preco": "50.0"}
```

3.12 Implementando um Cliente

A especificação JAX-RS não define uma API para padronizar o desenvolvimento de clientes Java de web services restful. Algumas implementações JAX-RS definem uma API não padronizada para a criação desses clientes. Veremos a seguir um pouco da API para implementação de consumidores Java de web services restful do projeto Jersey.

O primeiro passo para utilizar a Client API do Jersey é criar um objeto da classe **Client** através do método estático **create()** dessa mesma classe.

```
1 Client client = Client.create();
```

Depois, é necessário definir com qual resource queremos interagir através da URI do mesmo. Um objeto do tipo **WebResource** deve ser obtido através do método **resource()** do cliente.

```
1 WebResource resource = client.resource("http://www.k19.com.br/cursos/k23");
```

Por fim, podemos executar uma operação HTTP no resource através dos métodos da classe **WebResource**. O tipo do resultado da operação é definido previamente através do parâmetro da operação escolhida.

```
1 Curso curso = resource.get(Curso.class);
```

3.13 Exercícios

26. Crie um pacote chamado **client** no projeto **jax-rs**.
27. Adicione no pacote **client** uma classe para interagir com o resource de cotação de moeda.

```
1 public class TesteCotacaoResource {
2     public static void main(String[] args) {
3         Client client = Client.create();
4         WebResource resource = client.resource("http://localhost:8080/Cotacao/↵
5             DollarToReal");
6         String cotacao = resource.get(String.class);
7         System.out.println(cotacao);
8     }
9 }
```

28. Adicione no pacote **client** uma classe para interagir com o resource de produtos.

```
1 public class TesteProdutoResource {
2     public static void main(String[] args) {
3         Client client = Client.create();
4
5         WebResource resourceXML = client.resource("http://localhost:8080/produtos/1/↵
6             xml");
7
8         System.out.println("TESTANDO COM XML");
9
10        String xml = resourceXML.get(String.class);
11        System.out.println(xml);
12
13        Produto produto1 = resourceXML.get(Produto.class);
14        System.out.println(produto1.getId());
15        System.out.println(produto1.getNome());
16        System.out.println(produto1.getPreco());
17
18        WebResource resourceJSON = client.resource("http://localhost:8080/produtos/1/↵
19            json");
```

```

18         System.out.println("TESTANDO COM JSON");
19
20
21         String json = resourceJSON.get(String.class);
22         System.out.println(json);
23
24         Produto produto2 = resourceJSON.get(Produto.class);
25         System.out.println(produto2.getId());
26         System.out.println(produto2.getNome());
27         System.out.println(produto2.getPreco());
28     }
29 }

```

29. Adicione no pacote **client** uma classe para interagir com o resource de conversão de formato dos produtos.

```

1 public class TesteConversorDeProdutoResource {
2     public static void main(String[] args) {
3         Produto p = new Produto();
4         p.setId(1L);
5         p.setNome("Bola");
6         p.setPreco(45.67);
7
8         Client client = Client.create();
9
10        System.out.println("Convertendo para XML");
11        WebResource resource1 = client.resource("http://localhost:8080/produtos/↵
        converte/json/xml");
12        String xml = resource1.type("application/json").post(String.class, p);
13        System.out.println(xml);
14
15        System.out.println("Convertendo para JSON");
16        WebResource resource2 = client.resource("http://localhost:8080/produtos/↵
        converte/xml/json");
17        String json = resource2.type("application/xml").post(String.class, p);
18        System.out.println(json);
19    }
20 }

```

3.14 Projeto

30. Para consolidar os recursos da JAX-RS e do projeto Jersey, implemente um web service restful para funcionar como CRUD de clientes. Os dados podem ser mantidos apenas em memória.

Utilize o seguinte esquema de URIs e operações para os resources do seu web service:

Resource	Método HTTP	Semântica
localhost:8080/clientes	GET	lista dos clientes
localhost:8080/clientes	POST	adiciona um cliente
localhost:8080/clientes/id	PUT	atualiza um cliente
localhost:8080/clientes/id	DELETE	remove um cliente

Consulte o artigo sobre web services restful da K19. <http://www.k19.com.br/artigos/criando-um-webservice-restful-em-java/>

31. Implementes clientes através da API do projeto Jersey para testar o web service.

