

# Caelum

*"Mata o tempo e matarás a tua carreira"*

Bryan Forbes

## Sobre a empresa

A Caelum atua no mercado desde 2002, desenvolvendo sistemas e prestando consultoria em diversas áreas, sempre à luz da plataforma Java. Foi fundada por profissionais que se encontraram no Brasil após experiências na Alemanha e Itália, desenvolvendo sistemas de grande porte, com integração aos mais variados ERPs. Seus profissionais já publicaram diversos artigos nas revistas brasileiras sobre Java, artigos acadêmicos e são presença constante nos eventos de tecnologia.

Em 2004, a Caelum criou uma gama de cursos que rapidamente ganharam grande reconhecimento no mercado. Os cursos foram elaborados por ex-instrutores da Sun que queriam trazer mais dinamismo e aplicar as ferramentas e bibliotecas utilizadas no mercado, tais como Eclipse, Hibernate, Struts, e outras tecnologias *open source* que não são abordadas pela Sun. O material utilizado foi elaborado durante os cursos de verão de Java na Universidade de São Paulo, em janeiro de 2004, pelos instrutores da Caelum.

Em 2008, a empresa foca seus esforços em duas grandes áreas:

- Consultoria, mentoring e coaching de desenvolvimento
- Treinamento com intuito de formação

## Sobre a apostila

Esta apostila da Caelum visa ensinar Java de uma maneira elegante, mostrando apenas o que é necessário e quando é necessário, no momento certo, poupar o leitor de assuntos que não costumam ser de seu interesse em determinadas fases do aprendizado.

A Caelum espera que você aproveite esse material e que ele possa ser de grande valia para auto-didatas e estudantes. Todos os comentários, críticas e sugestões serão muito bem-vindos.

O material aqui contido pode ser publicamente distribuído, desde que seu conteúdo não seja alterado e que seus créditos sejam mantidos. Ele não pode ser usado para ministrar qualquer outro curso, porém pode ser utilizado como referência e material de apoio. Caso você esteja interessado em usá-lo para fins comerciais, entre em contato com a empresa.

**Atenção:** Você pode verificar a versão da apostila no fim do índice. Não recomendamos imprimir a apostila que você receber de um amigo ou pegar por e-mail, pois atualizamos constantemente esse material, quase que mensalmente. Vá até nosso site e veja a última versão.

---

# Índice

<b>1</b>	<b>Como Aprender Java</b>	<b>1</b>
1.1	O que é realmente importante? . . . . .	1
1.2	Sobre os exercícios . . . . .	2
1.3	Tirando dúvidas . . . . .	2
1.4	Bibliografia . . . . .	2
<b>2</b>	<b>JDBC - java.sql</b>	<b>3</b>
2.1	Executando o Eclipse pela primeira vez . . . . .	3
2.2	O banco . . . . .	4
2.3	Sockets: uma idéia inocente . . . . .	5
2.4	A conexão em Java . . . . .	5
2.5	Fábrica de Conexões . . . . .	8
2.6	Soluções para viagem – Design Patterns . . . . .	8
2.7	Exercícios - ConnectionFactory . . . . .	9
2.8	A tabela de exemplo . . . . .	15
2.9	Javabeans . . . . .	16
2.10	Exercícios - Contato . . . . .	17
2.11	Inserindo dados . . . . .	18
2.12	DAO – Data Access Object . . . . .	20
2.13	Exercícios - ContatoDAO . . . . .	22
2.14	Exercícios Opcionais . . . . .	23
2.15	Pesquisando . . . . .	23

2.16 Exercícios - Listagem . . . . .	25
2.17 Um pouco mais... . . . . .	26
2.18 Exercícios opcionais . . . . .	26
2.19 Outros métodos para o seu DAO . . . . .	27
2.20 Exercícios opcionais - Alterar e remover . . . . .	27
<b>3 O que é J2EE? . . . . .</b>	<b>29</b>
3.1 Java EE . . . . .	29
3.2 APIs . . . . .	30
3.3 Implementações compatíveis com a especificação . . . . .	30
<b>4 Servlet Container . . . . .</b>	<b>31</b>
4.1 Introdução . . . . .	31
4.2 Aplicações web no Java EE e Servlet Container . . . . .	32
4.3 Instalando o Tomcat . . . . .	32
4.4 Sobre o Tomcat . . . . .	33
4.5 Instalando o Tomcat em casa . . . . .	34
4.6 Outra opção: Jetty . . . . .	34
<b>5 O eclipse e seus plugins . . . . .</b>	<b>35</b>
5.1 O plugin WTP . . . . .	35
5.2 Configurando o Tomcat no WTP . . . . .	36
<b>6 Novo projeto web . . . . .</b>	<b>39</b>
6.1 Novo projeto . . . . .	39
6.2 Análise do resultado final . . . . .	41
6.3 Exercícios: primeira página . . . . .	44
6.4 Para saber mais: configurando o Tomcat sem o plugin . . . . .	44
<b>7 JSP - JavaServer Pages . . . . .</b>	<b>45</b>
7.1 O que é uma página JSP . . . . .	45
7.2 Exercícios: Primeiro JSP . . . . .	46
7.3 Listando os contatos . . . . .	47
7.4 Exercícios: Lista de contatos com scriptlet . . . . .	48

7.5	HTML e Java: eu não quero código Java no meu JSP!	49
7.6	EL: Expression language	50
7.7	Exercícios: parâmetros com EL	50
7.8	Exercícios opcionais	50
7.9	Erros comuns	51
7.10	Exercício opcional	52
7.11	Instanciando POJOs	52
7.12	Para saber mais: Compilando os arquivos JSP	53
<b>8</b>	<b>JSTL - JavaServer Pages Tag Library</b>	<b>54</b>
8.1	JSTL	54
8.2	As empresas hoje em dia	54
8.3	Instalação	55
8.4	Cabeçalho para a JSTL core	55
8.5	For	55
8.6	Exercícios: forEach	57
8.7	Exercício opcional	58
8.8	c:out e c:set	58
8.9	Mas quais são as tags da taglib core?	58
8.10	Import: trabalhando com cabeçalhos e rodapés	59
8.11	Exercícios	59
8.12	Erros Comuns	60
8.13	Inclusão estática de arquivos	60
8.14	Exercícios	61
8.15	Exercícios opcionais	61
8.16	Trabalhando com links	62
8.17	Exercícios opcionais	62
8.18	c;if	63
8.19	Exercícios	63

<b>9 Controle de erro</b>	<b>65</b>
9.1 Exceptions . . . . .	65
9.2 JSTL é a solução? . . . . .	66
9.3 Exercício opcional . . . . .	66
9.4 Quando acontece um erro em uma página JSP . . . . .	67
9.5 Página de erro . . . . .	67
9.6 Exercícios . . . . .	68
9.7 Erros comuns . . . . .	69
<b>10 Servlets</b>	<b>70</b>
10.1 Servlet . . . . .	70
10.2 A estrutura de diretórios . . . . .	72
10.3 Mapeando uma servlet no web.xml . . . . .	72
10.4 Exercícios: Servlet OiMundo . . . . .	72
10.5 Erros comuns . . . . .	74
10.6 Init e Destroy . . . . .	75
10.7 Curiosidades do mapeamento de uma servlet . . . . .	76
10.8 OutputStream x PrintWriter . . . . .	77
10.9 Parâmetros . . . . .	77
10.10 Exercícios: TestaParametros . . . . .	78
10.11 Exercício opcional . . . . .	80
10.12 doGet, doPost e outros . . . . .	80
10.13 Conversão de parâmetros . . . . .	80
10.14 Exercícios opcionais . . . . .	81
10.15 Variáveis membro . . . . .	83
10.16 Exercícios: Contador . . . . .	84
10.17 HTML e Java: eu não quero código html na minha servlet! . . . . .	85
10.18 Como funciona uma página JSP . . . . .	86
10.19 Web archive (.war) . . . . .	86
10.20 Exercícios: Deploy com war . . . . .	86
10.21 Quando acontece um erro em uma servlet . . . . .	89
10.22 O try e catch . . . . .	90

10.23	Tratamento padrão de erros – modo declarativo . . . . .	91
10.24	Configurando a página de erro . . . . .	91
10.25	Exercícios . . . . .	91
10.26	Erros comuns . . . . .	93
10.27	Tratamento de outros erros . . . . .	93
10.28	Exercício opcional . . . . .	94
10.29	Outro erro comum . . . . .	94
10.30	Servlet para adicionar contatos no banco . . . . .	95
10.31	Exercício . . . . .	95
10.32	Exercícios Opcionais . . . . .	97
<b>11</b>	<b>Servlet e JSP API</b>	<b>98</b>
11.1	Propriedades de páginas JSP . . . . .	98
11.2	Exercícios . . . . .	99
11.3	Filtros . . . . .	99
11.4	Configuração de filtros . . . . .	100
11.5	Exercícios . . . . .	100
11.6	Entendendo os filtros . . . . .	101
11.7	Exercício Opcional . . . . .	101
<b>12</b>	<b>Model View Controller</b>	<b>102</b>
12.1	Servlet ou JSP? . . . . .	102
12.2	Request dispatchers . . . . .	103
12.3	Exercícios . . . . .	104
12.4	Melhorando o processo . . . . .	105
12.5	Retomando o <i>design pattern Factory</i> . . . . .	107
<b>13</b>	<b>Construindo um Framework MVC</b>	<b>108</b>
13.1	Nossa interface de execução . . . . .	108
13.2	Exercícios . . . . .	108
13.3	Criando um controlador e um pouco mais de reflection . . . . .	109
13.4	Configurando o web.xml . . . . .	111
13.5	Exercícios . . . . .	111

13.6 Erros comuns . . . . .	112
13.7 Exercícios . . . . .	112
13.8 Exercícios opcionais . . . . .	114
13.9 Model View Controller . . . . .	114
13.10Lista de tecnologias: camada de controle . . . . .	115
13.11Lista de tecnologias: camada de visualização . . . . .	116
13.12MVC 2 . . . . .	116
<b>14 Struts Framework</b>	<b>117</b>
14.1 Struts . . . . .	117
14.2 Configurando o Struts . . . . .	118
14.3 Exercícios . . . . .	118
14.4 Uma ação Struts . . . . .	121
14.5 Configurando a ação no struts-config.xml . . . . .	122
14.6 Exercícios: TesteSimplesAction . . . . .	123
14.7 Erros comuns . . . . .	124
14.8 Pesquisando um banco de dados . . . . .	126
14.9 Criando a ação . . . . .	126
14.10O arquivo WebContent/lista.jsp . . . . .	127
14.11ListaContatos no struts-config.xml . . . . .	128
14.12Exercício: ListaContatosAction . . . . .	128
14.13Resultado condicional com o Struts . . . . .	130
14.14Exercícios: listagem vazia . . . . .	130
14.15Resultado do struts-config.xml . . . . .	131
14.16Novos contatos . . . . .	131
14.17Formulário . . . . .	132
14.18Mapeando o formulário no arquivo struts-config.xml . . . . .	132
14.19Exercício: ContatoForm . . . . .	133
14.20Erro comum . . . . .	133
14.21Lógica de Negócios . . . . .	133
14.22Exercício: AdicionaContatoAction . . . . .	134
14.23Erros comuns . . . . .	136

14.24	Arquivo de mensagens . . . . .	136
14.25	Exercícios: Mensagens . . . . .	137
14.26	Erros comuns . . . . .	138
14.27	Validando os campos . . . . .	140
14.28	Exercício: validação . . . . .	141
14.29	Erros comuns . . . . .	142
14.30	Exercícios opcionais . . . . .	142
14.31	Limpando o formulário . . . . .	142
14.32	Exercícios: scope . . . . .	143
14.33	Exercícios opcionais . . . . .	143
14.34	O mesmo formulário para duas ações . . . . .	144
14.35	Exercícios opcionais . . . . .	144
14.36	Struts-logic taglib: um exemplo antigo de for . . . . .	146
14.37	Para saber mais . . . . .	146
<b>15</b>	<b>Autenticação, cookies e sessão</b> . . . . .	<b>148</b>
15.1	Preparando um sistema de login . . . . .	148
15.2	Nossas classes: Funcionario e FuncionarioDAO . . . . .	148
15.3	Passo 1: Formbean . . . . .	150
15.4	Passo 2: A página de login: formulario-login.jsp . . . . .	150
15.5	Exercícios: formulário de login . . . . .	151
15.6	A ação . . . . .	152
15.7	A ação no struts-config.xml . . . . .	152
15.8	ok.jsp e erro.jsp . . . . .	153
15.9	Exercícios: LoginAction . . . . .	153
15.10	Erro comum . . . . .	155
15.11	Exercícios opcionais . . . . .	155
15.12	Cookies . . . . .	155
15.13	Sessão . . . . .	156
15.14	Configurando o tempo limite . . . . .	156
15.15	Registrando o usuário logado na sessão . . . . .	156
15.16	Exercícios: autorização . . . . .	157
15.17	Exercícios opcionais . . . . .	158

<b>16 Hibernate</b>	<b>160</b>
16.1 Vantagens . . . . .	160
16.2 Criando seu projeto . . . . .	160
16.3 Modelo . . . . .	161
16.4 Configurando a classe/tabela Produto . . . . .	161
16.5 Exercícios . . . . .	162
16.6 Propriedades do banco . . . . .	168
16.7 Exercícios . . . . .	169
16.8 Configurando . . . . .	169
16.9 Criando as tabelas . . . . .	170
16.10 Exercícios . . . . .	170
16.11 Dica: log do Hibernate . . . . .	171
16.12 HibernateUtil . . . . .	172
16.13 Exercícios . . . . .	172
16.14 Erros comuns . . . . .	173
16.15 Salvando novos objetos . . . . .	173
16.16 Exercícios . . . . .	173
16.17 Buscando pelo id . . . . .	174
16.18 Criando o ProdutoDAO . . . . .	174
16.19 Exercícios . . . . .	175
16.20 Buscando com uma cláusula where . . . . .	176
16.21 ProdutoDAO: Listar tudo e fazer paginação . . . . .	176
16.22 Exercícios . . . . .	177
16.23 Exercícios para o preguiçoso . . . . .	178
16.24 Exercício opcional . . . . .	179
<b>17 VRaptor</b>	<b>180</b>
17.1 Eu não quero o que eu não conheço . . . . .	180
17.2 Vantagens . . . . .	183
17.3 Vraptor 2 . . . . .	183
17.4 Exercícios . . . . .	183
17.5 Internacionalização . . . . .	184

17.6 A classe de modelo . . . . .	184
17.7 Minha primeira lógica de negócios . . . . .	185
17.8 Como configurar a minha lógica? . . . . .	185
17.9 E o JSP com o formulário? . . . . .	186
17.10 E a página final? . . . . .	186
17.11 Exercícios . . . . .	187
17.12 A lista de produtos . . . . .	188
17.13 Exercícios opcionais . . . . .	189
17.14 Velocity, Freemarker e Sitemesh . . . . .	189
17.15 Configurações . . . . .	190
17.16 Um pouco mais... . . . . .	190
17.17 Plugin para o Eclipse . . . . .	190
17.18 Pequenos exemplos de simplicidade . . . . .	190
<b>18 E agora?</b>	<b>192</b>
18.1 Certificação . . . . .	192
18.2 Frameworks . . . . .	192
18.3 Revistas . . . . .	192
18.4 Grupo de Usuários . . . . .	192
18.5 Falando em Java - Próximos módulos . . . . .	193
<b>19 Apêndice A - Servlet e JSP API</b>	<b>194</b>
19.1 Início e término da sua aplicação . . . . .	194
19.2 Exercícios . . . . .	195
19.3 getServletContext() . . . . .	196
19.4 Exercícios opcionais . . . . .	196
19.5 Acessando a aplicação no JSP . . . . .	197
19.6 Exercícios . . . . .	197
19.7 Configuração de uma servlet . . . . .	198
19.8 Exercícios . . . . .	199
19.9 Descobrindo todos os parâmetros do request . . . . .	199

<b>20 Apêndice B - Design Patterns</b>	<b>200</b>
20.1 Factory – exemplo de cache de objetos . . . . .	200
20.2 Singleton . . . . .	202
20.3 Exercícios . . . . .	203
20.4 Um pouco mais... . . . . .	204
<b>21 Respostas dos Exercícios</b>	<b>205</b>

**Versão: 10.9.18**

# Como Aprender Java

*"Homens sábios fazem provérbios, tolos os repetem"*  
— Samuel Palmer

## 1.1 - O que é realmente importante?

Muitos livros, ao passar dos capítulos, mencionam todos os detalhes da linguagem juntamente com seus princípios básicos. Isso acaba criando muita confusão, em especial pois o estudante não consegue distinguir exatamente o que é importante aprender e reter naquele momento daquilo que será necessário mais tempo e principalmente experiência para dominar.

Se uma classe abstrata deve ou não ter ao menos um método abstrato, se o if só a ceitar argumentos booleanos e todos os detalhes de classes internas realmente não devem ser preocupações para aquele que possui como objetivo primário aprender Java. Esse tipo de informação será adquirida com o tempo, e não é necessário até um segundo momento.

Neste curso, separamos essas informações em quadros especiais, já que são informações extras. Ou então, apenas citamos num exercício e deixamos para o leitor procurar informações se for de seu interesse.

Algumas informações não são mostradas e podem ser adquiridas em tutoriais ou guias de referência. São detalhes que para um programador experiente em Java podem ser importantes, mas não para quem está começando.

Por fim, falta mencionar sobre a prática, que deve ser tratada seriamente: todos os exercícios são muito importantes e os desafios podem ser feitos quando o curso acabar. De qualquer maneira, recomendamos aos alunos estudar em casa, principalmente àqueles que fazem os cursos intensivos.

### O curso

Para aqueles que estão fazendo o curso Java para Desenvolvimento Web, é recomendado estudar em casa aquilo que foi visto durante a aula, tentando resolver os exercícios que não foram feitos e os desafios que estão lá para envolver mais o leitor no mundo do Java.

### Convenções de Código

Para mais informações sobre as convenções de código-fonte Java, acesse: <http://java.sun.com/docs/codeconv/>

## 1.2 - Sobre os exercícios

Os exercícios do curso variam de práticos até pesquisas na Internet, ou mesmo consultas sobre assuntos avançados em determinados tópicos para incitar a curiosidade do aprendiz na tecnologia.

## 1.3 - Tirando dúvidas

Para tirar dúvidas, tanto dos exercícios quanto de Java em geral, recomendamos o fórum do GUJ (<http://www.guj.com.br/>), onde sua dúvida será respondida prontamente.

Se você já participa de um grupo de usuários Java ou alguma lista de discussão, incentivamos tirar suas dúvidas nos dois lugares.

Fora isso, sinta-se à vontade para entrar em contato conosco e tirar todas as dúvidas que tiver durante o curso.

## 1.4 - Bibliografia

É possível aprender muitos dos detalhes e pontos não cobertos no treinamento em tutoriais na Internet em portais como o GUJ, em blogs (como o da Caelum: <http://blog.caelum.com.br>) e em muitos Sites especializados.

Mas se você deseja algum livro para expandir seus conhecimentos ou ter como guia de referência, temos algumas indicações dentre várias possíveis:

### Sobre Java para Web

- Use a cabeça! JSP e Servlets
- Struts em ação - Ted Husted

### Sobre Java e melhores práticas

- Refactoring, Martin Fowler
- Effective Java - 2nd edition, Joshua Bloch
- Design Patterns, Erich Gamma et al

### Para iniciantes na plataforma Java:

- Java - Como programar, de Harvey M. Deitel
- Use a cabeça! - Java, de Bert Bates e Kathy Sierra

# JDBC - java.sql

*“O medo é o pai da moralidade”*  
— Friedrich Wilhelm Nietzsche

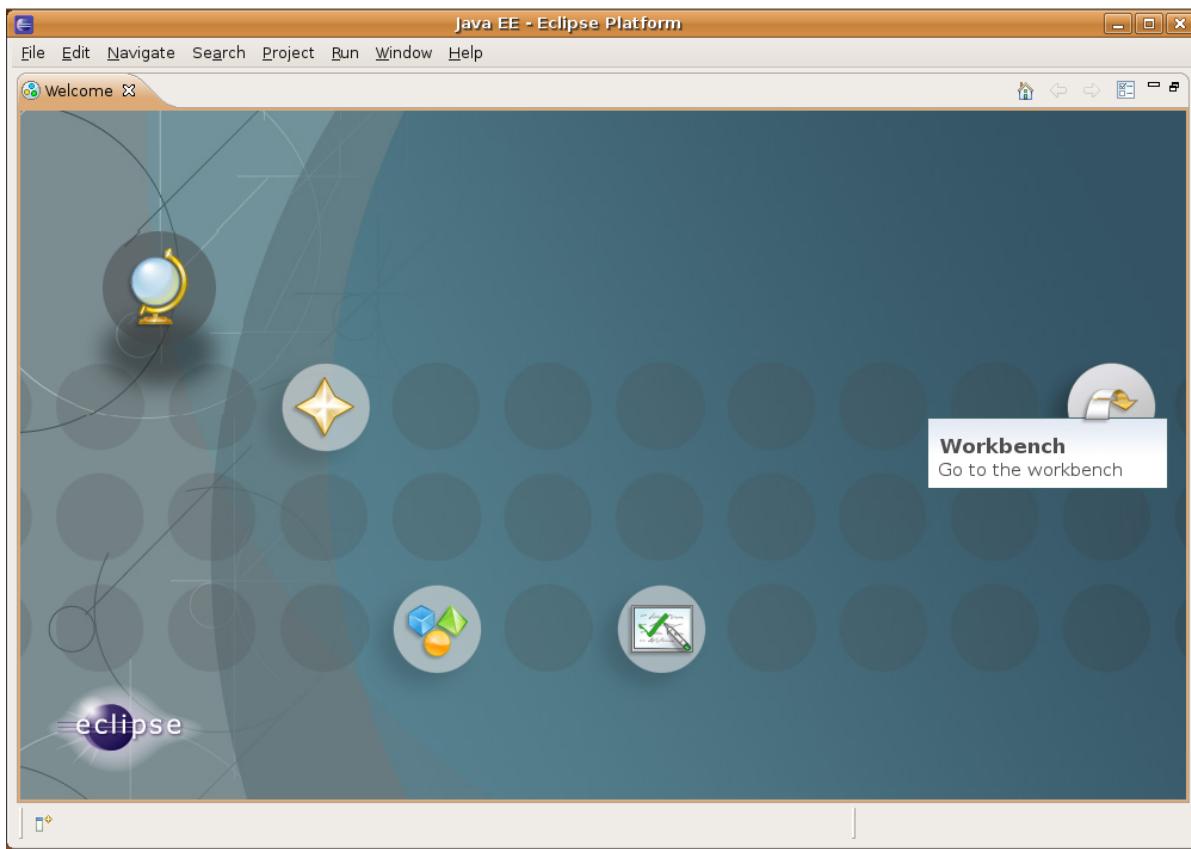
Ao término desse capítulo, você será capaz de:

- conectar-se a um banco de dados qualquer através da API `java.sql`;
- criar uma fábrica de conexões usando o design pattern Factory;
- pesquisar dados através de queries;
- DAO – Data Access Object.

## 2.1 - Executando o Eclipse pela primeira vez

1) Abra o terminal e digite `eclipsejee`

2) Escolha Workbench.



#### Baixando em o Eclipse em casa

Estamos usando o Eclipse Java EE 3.4. Você pode obtê-lo direto no site do Eclipse em [www.eclipse.org](http://www.eclipse.org)

## 2.2 - O banco

O banco de dados é onde guardamos os dados que pertencem ao nosso sistema. A maioria dos bancos de dados comerciais hoje em dia são relacionais e derivam de uma estrutura diferente daquela orientada a objetos.

O **MYSQL** é o banco de dados que usamos para nossos exemplos, portanto iremos utilizar o seguinte comando no seu terminal para acessar o mesmo:

```
mysql -u root
```

#### Banco de dados

Para aqueles que não conhecem um banco de dados, é recomendado ler mais sobre o assunto e SQL para começar a usar a API JDBC.

O processo de armazenamento e captura de dados em um banco é chamado de **persistência**. A biblioteca padrão de persistência em banco de dados em Java é a JDBC, mas já existem diversos projetos do tipo **ORM** (Object Relational Mapping) que solucionam muitos dos problemas que a estrutura da API JDBC (e ODBC) gerou. Veremos um pouco de Hibernate no final do curso.

## 2.3 - Sockets: uma idéia inocente

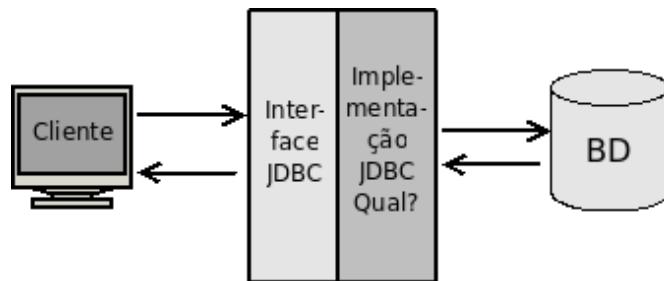
Para conectar-se a um banco de dados, a primeira idéia, simples em sua forma mas complexa em sua implementação, é a de abrir sockets diretamente com o banco de dados desejado, por exemplo um Oracle, e se comunicar com ele através de seu protocolo proprietário – e não só SQL.

Mas você conhece o protocolo proprietário de algum banco de dados?

Devido à natureza complexa desses protocolos, seria muito mais simples se existisse algo em Java com quem nosso programa fosse capaz de se comunicar em Java e que se comunicasse com o banco em um protocolo qualquer, alheio ao nosso programa.

## 2.4 - A conexão em Java

O sistema desenvolvido em Java abstrai o método através do qual é possível fazer uma conexão pois as conexões são feitas através de uma ponte que implementa todas as funcionalidades que um banco de dados padrão deve nos fornecer.



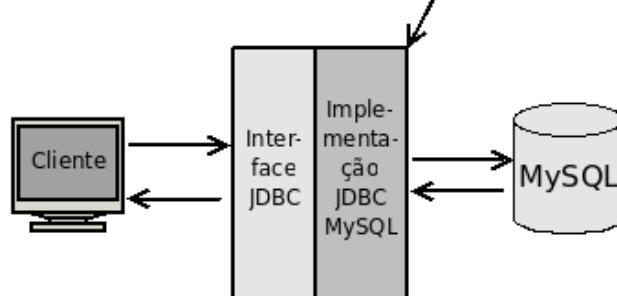
Por exemplo, toda conexão deve permitir executar código de atualização, pesquisa, etc.

Essa implementação precisa ser escolhida. Essa escolha não é feita programaticamente, e sim basta usar uma ponte.

Veja no esquema ao lado a ponte (implementação) entre o programa (cliente) e o banco de dados.

O serviço de encontrar uma ponte, um **driver**, certa é delegado para um controlador de drivers. Um **gerente de drivers**. Nada mais normal que ele se chame `DriverManager`.

```
DriverManager.getConnection("jdbc:mysql://localhost/teste");
```



Através dele, é possível chamar um método `getConnection` com uma url que indica qual o banco que desejo abrir.

O padrão da url para o driver do mysql que iremos utilizar é:

```
jdbc:mysql://ip/banco
```

Devemos substituir `ip` pelo IP da máquina e `banco` pelo nome do banco a ser utilizado.

Seguindo o exemplo da linha acima e tudo que foi dito até agora, é possível rodar o exemplo abaixo e receber uma conexão para um banco MySQL na própria máquina:

```
package br.com.caelum.jdbc;

// imports aqui (ctrl + shift + o)

public class JDBCExemplo {

    public static void main(String[] args) {

        try {
            Connection con = DriverManager.getConnection("jdbc:mysql://localhost/teste");
            System.out.println("Conectado!");
            con.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

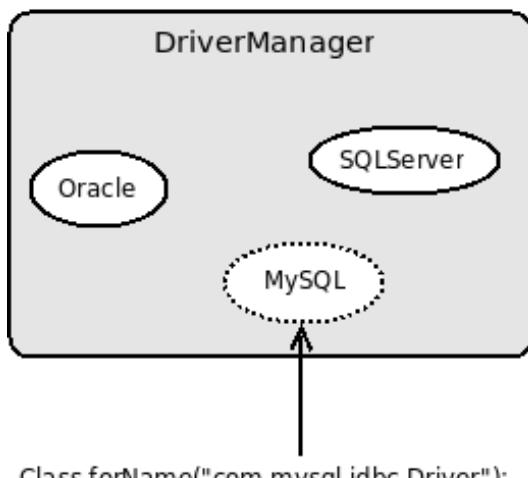
Mas, ao testar o código acima, nada funciona. A conexão não pode ser aberta. Por quê?

O sistema ainda não consegue descobrir qual implementação do **JDBC** deve ser usado para a URL mencionada.

O primeiro passo é adicionar a implementação ao `classpath`: o arquivo `jar` contendo a implementação do mysql (mysql connector) precisa ser colocado em um lugar visível ou adicionado à variável de ambiente `classpath`.

Ainda falta registrar o driver do mysql no sistema. Para isso basta carregar ele através do método `Class.forName()`. Esse método abre uma classe que se registra com o `DriverManager.getConnection()`.

Avisando o DriverManager sobre  
a existência de um Driver para o MySQL



```

package br.com.caelum.jdbc;

// imports aqui (ctrl + shift + o)

public class JDBCExemplo {

    public static void main(String[] args) {
        try {

            Class.forName("com.mysql.jdbc.Driver");
            Connection con = DriverManager.getConnection("jdbc:mysql://localhost/teste", "root", "");
            System.out.println("Conectado!");
            con.close();

        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

```

#### Alterando o banco de dados

Teoricamente, basta alterar as duas *Strings* que escrevemos para mudar de um banco para outro. Porém, não é tudo tão simples assim! O código SQL que veremos a seguir pode funcionar em um banco e não em outros. Depende de qual padrão SQL o banco suporta. Isso só causa dor de cabeça e existem certos arcabouços que resolvem isso facilmente, como é o caso do Hibernate ([www.hibernate.org](http://www.hibernate.org)) e da JPA (Java Persistence API).

### Lista de drivers

Os drivers podem ser baixados normalmente no site do fabricante do banco de dados. A Sun possui um sistema de busca de drivers em seu site: <http://developers.sun.com/product/jdbc/drivers>. Alguns casos, como no MSSQL, existem outros grupos que desenvolvem o driver em <http://jtds.sourceforge.net>. Enquanto isso, você pode achar o driver do MYSQL (chamado de mysql connector) no site <http://www.mysql.org>.

## 2.5 - Fábrica de Conexões

Em determinado momento de nossa aplicação, gostaríamos de ter o controle sobre a construção dos objetos da nossa classe. Muita coisa pode ser feita através do construtor, como saber quantos objetos foram instanciados ou fazer o log sobre essas instanciações.

Às vezes, também queremos controlar um processo muito repetitivo e trabalhoso, como abrir uma conexão com o banco de dados.

[subsection Fábrica de Conexões – facilitando o acesso ao banco]

Tomemos como exemplo a classe a seguir que seria responsável por abrir uma conexão com o banco:

```
package br.com.caelum.jdbc;

// imports aqui (ctrl + shift + o)

public class ConnectionFactory {

    public static Connection getConnection() throws SQLException {
        try {
            Class.forName("com.mysql.jdbc.Driver");
            return DriverManager.getConnection("jdbc:mysql://localhost/teste","root","");
        } catch (ClassNotFoundException e) {
            throw new SQLException(e.getMessage());
        }
    }
}
```

Poderíamos colocar um aviso na nossa aplicação, notificando todos os programadores a adquirir uma conexão:

```
Connection con = ConnectionFactory.getConnection();
```

Note que o método `getConnection()` é uma fábrica de conexões, isto é, ele fabrica conexões para nós, não importando de onde elas vieram. Portanto, vamos chamar a classe de `ConnectionFactory` e o método de `getConnection`.

## 2.6 - Soluções para viagem – Design Patterns

Orientação a objetos resolve as grandes dores de cabeças que tínhamos na programação procedural, restringindo e centralizando responsabilidades.

Mas algumas coisas não podemos simplesmente resolver com orientação a objetos, pois não existe palavra chave para uma funcionalidade tão específica.

Alguns desses pequenos problemas aparecem com tamanha freqüência que as pessoas desenvolvem uma solução padrão para ele. Com isso, ao nos defrontarmos com um desses problemas clássicos, podemos rapidamente implementar essa solução genérica com uma ou outra modificação. Essa solução padrão tem o nome de **Design Pattern (Padrão de Projeto)**.

A melhor maneira para aprender o que é um Design Pattern é vendo como surgiu sua necessidade.

A nossa `ConnectionFactory` implementa o design pattern Factory que prega o encapsulamento da construção (fabricação) de objetos complicados.

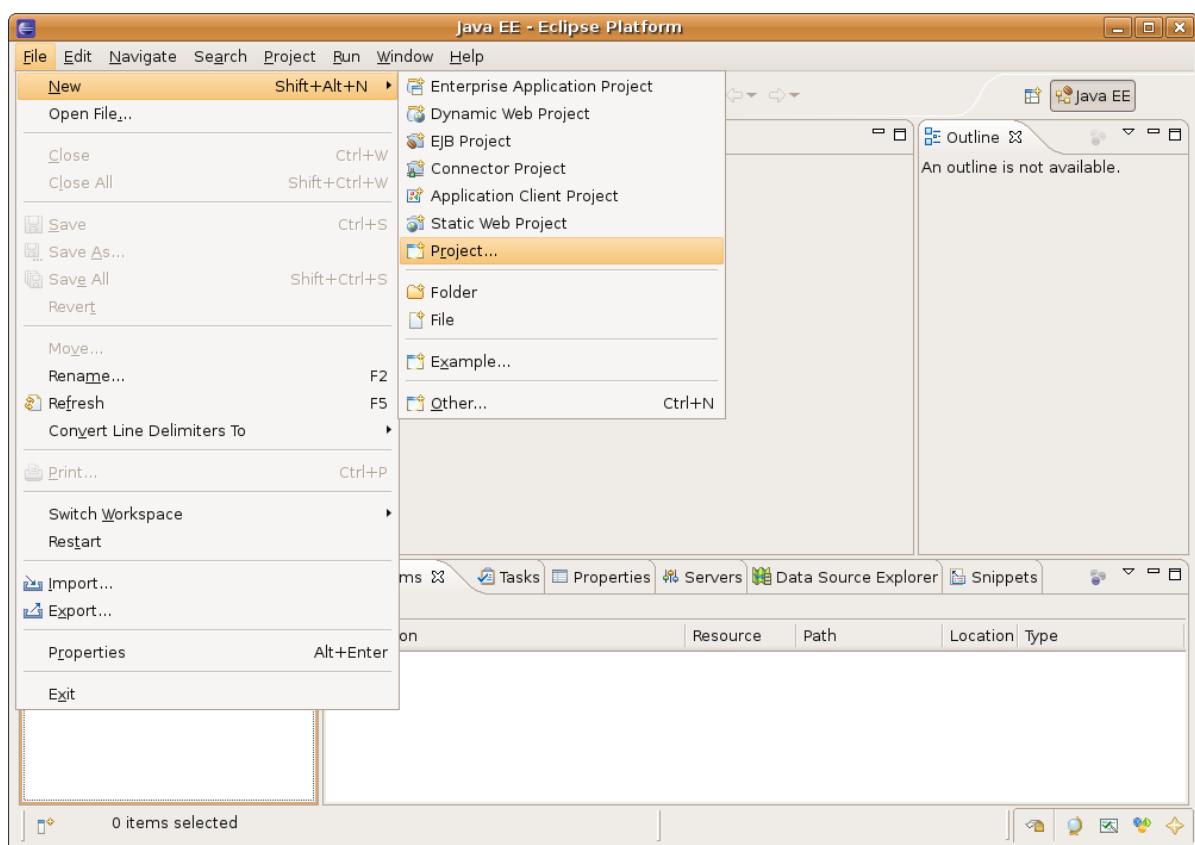
### A bíblia

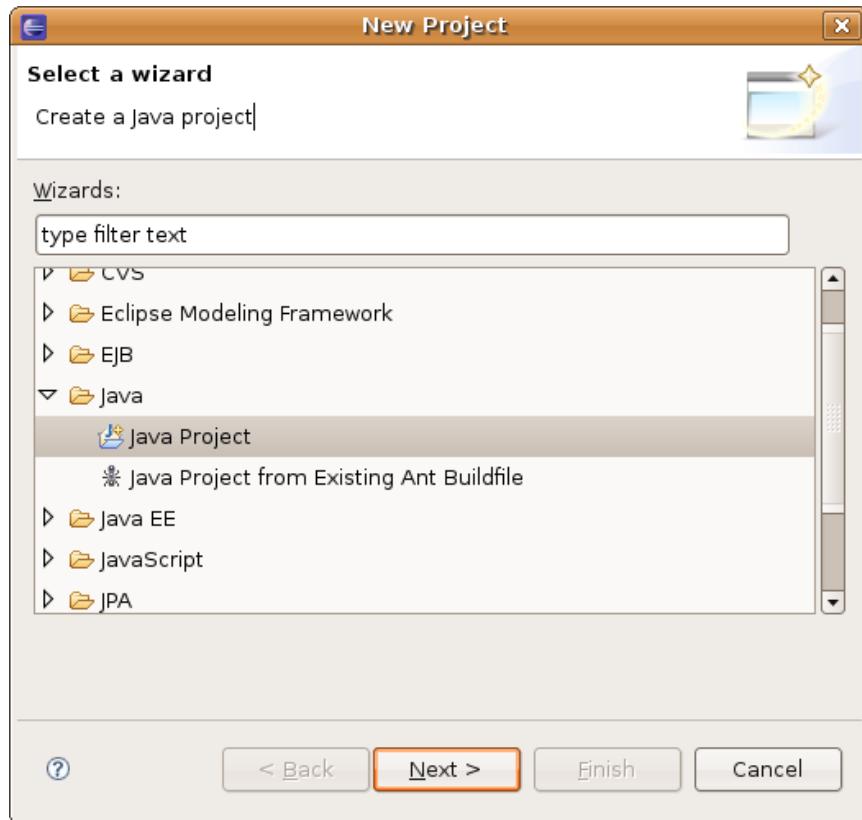
O livro mais conhecido de Design Patterns foi escrito em 1995 e tem trechos de código em C++ e Smalltalk. Mas o que realmente importa são os conceitos e os diagramas que fazem desse livro independente de qualquer linguagem. Além de tudo, o livro é de leitura agradável.

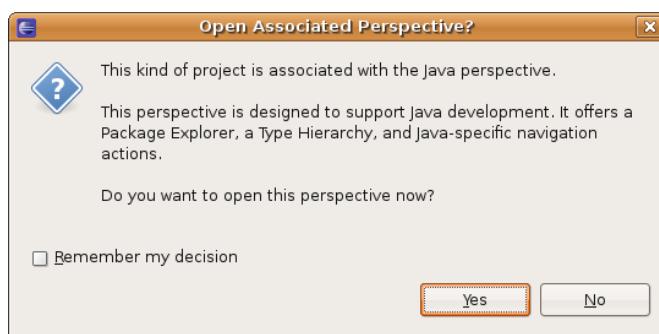
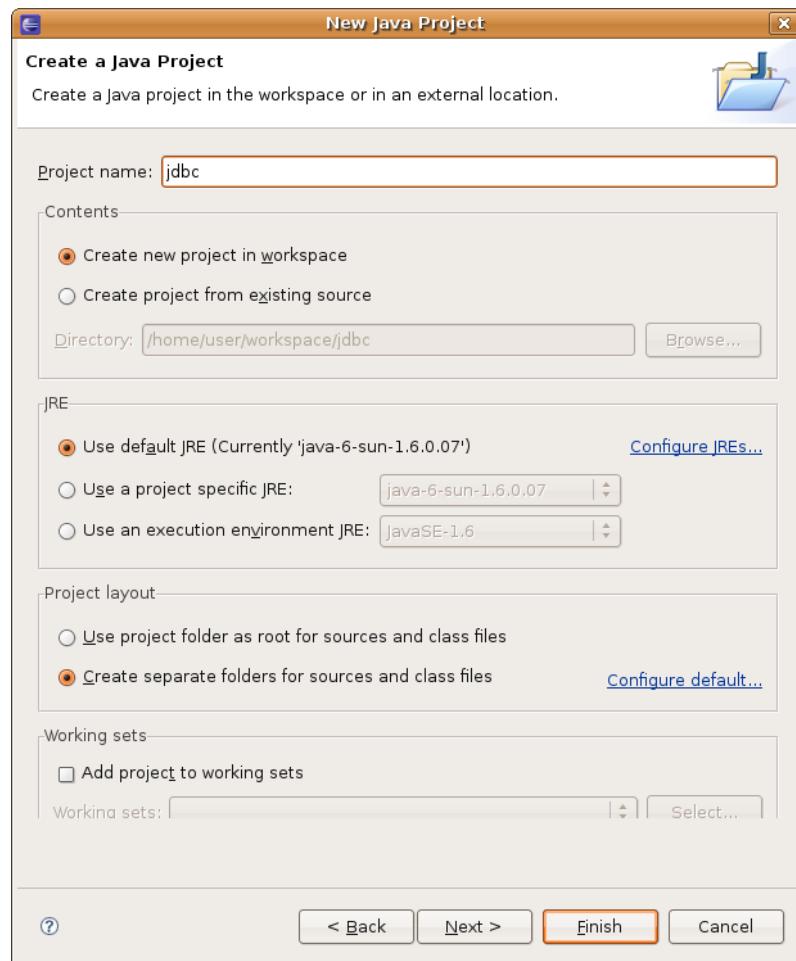
**Design Patterns, Erich Gamma et al.**

## 2.7 - Exercícios - ConnectionFactory

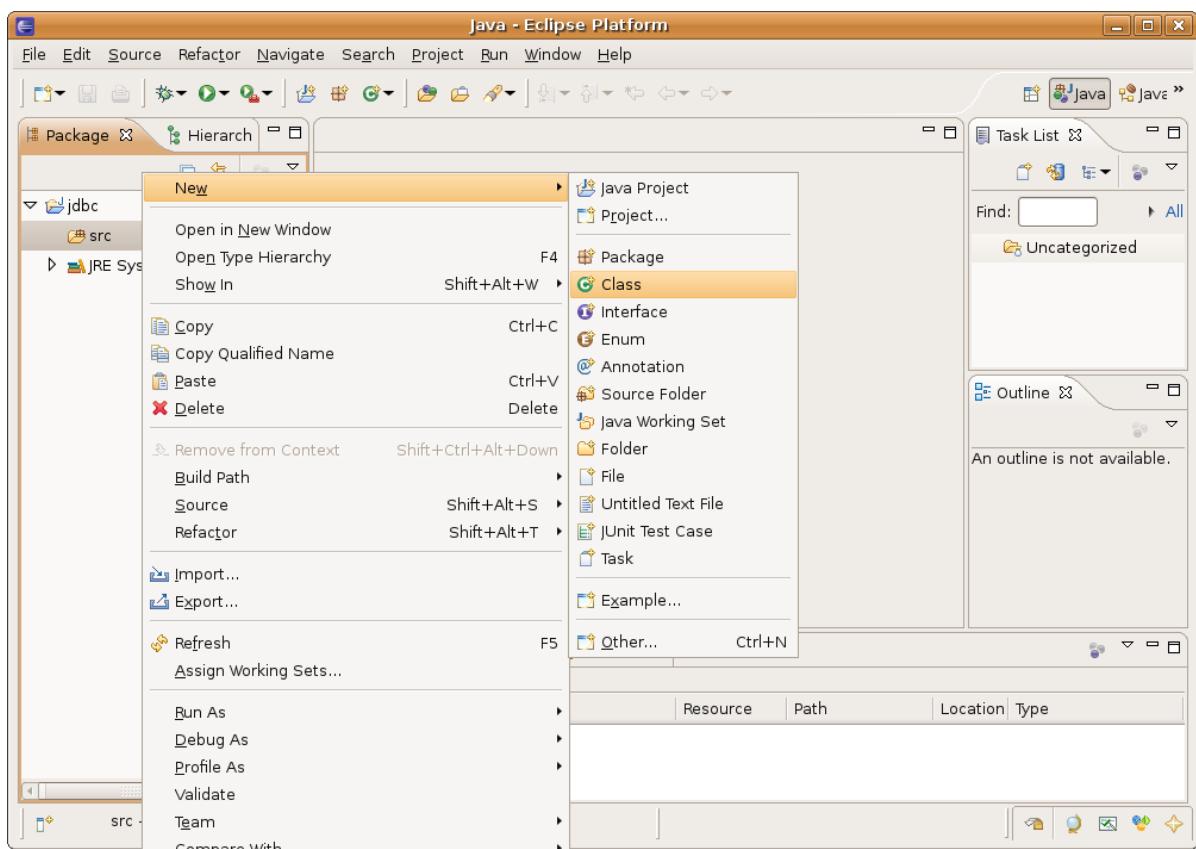
1) a) Crie um projeto no Eclipse chamado `jdbc` e confirme a mudança de perspectiva.



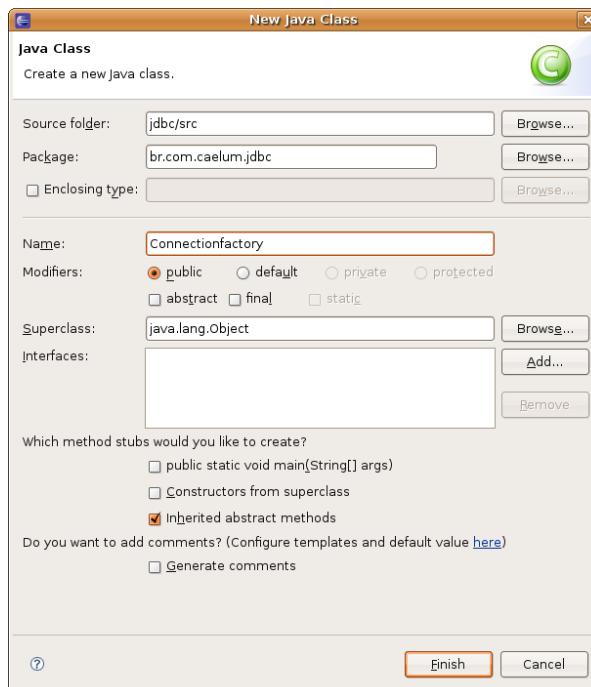




- 2) Copie o driver do mysql para o seu projeto.
  - a) no seu Desktop, clique na pasta caelum;
  - b) clique da direita no driver do mysql mais novo, escolha Copy;
  - c) vá para sua pasta principal (webXXX) na coluna da direita do FileBrowser;
  - d) entre no diretório workspace, jdbc;
  - e) clique da direita e escolha Paste: você acaba de colocar o arquivo ".jar" no seu projeto.
- 3) Vamos criar a classe que fabrica conexões:



- a) Crie-a no pacote br.com.caelum.jdbc e nomeie-a ConnectionFactory.



- b) Crie o método estático `getConnection` que retorna uma nova conexão. Quando perguntado, importe as classes do pacote `java.sql` (cuidado para não importar de `com.mysql`).

```
1 public static Connection getConnection() throws SQLException {
```

```
2     try {
3         Class.forName("com.mysql.jdbc.Driver");
4         System.out.println("Conectando ao banco");
5         return DriverManager.getConnection("jdbc:mysql://localhost/teste", "root", "");
6     } catch (ClassNotFoundException e) {
7         throw new SQLException(e.getMessage());
8     }
9 }
```

- 4) Crie uma classe chamada TestaConexao no pacote br.com.caelum.jdbc.teste. Todas as nossas classes de teste deverão ficar nesse pacote.

- a) coloque o método main

```
public static void main (String[] args) { ... }
```

- b) fabrique uma conexão:

```
Connection connection = ConnectionFactory.getConnection();
```

- c) feche a conexão

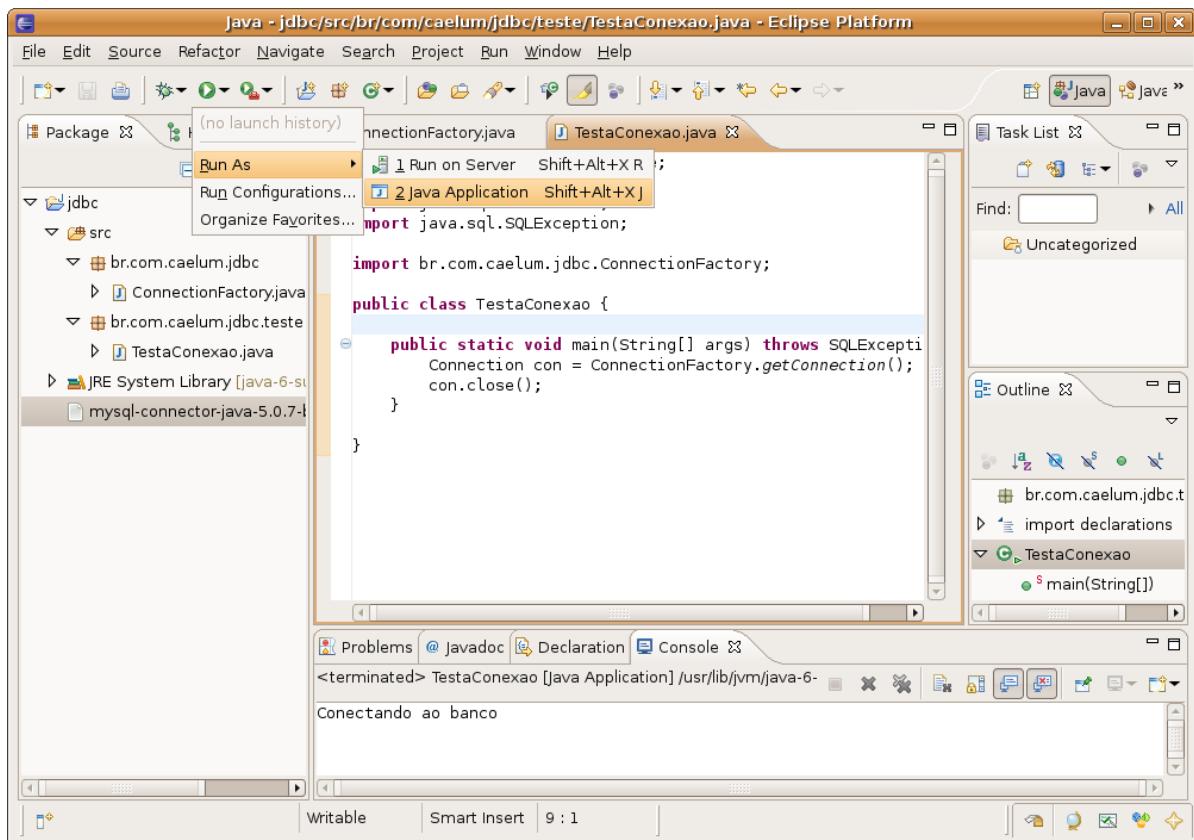
```
connection.close();
```

- d) Trate os erros com throws. ( Use: Ctrl + 1 e escolha “add throws declaration”).

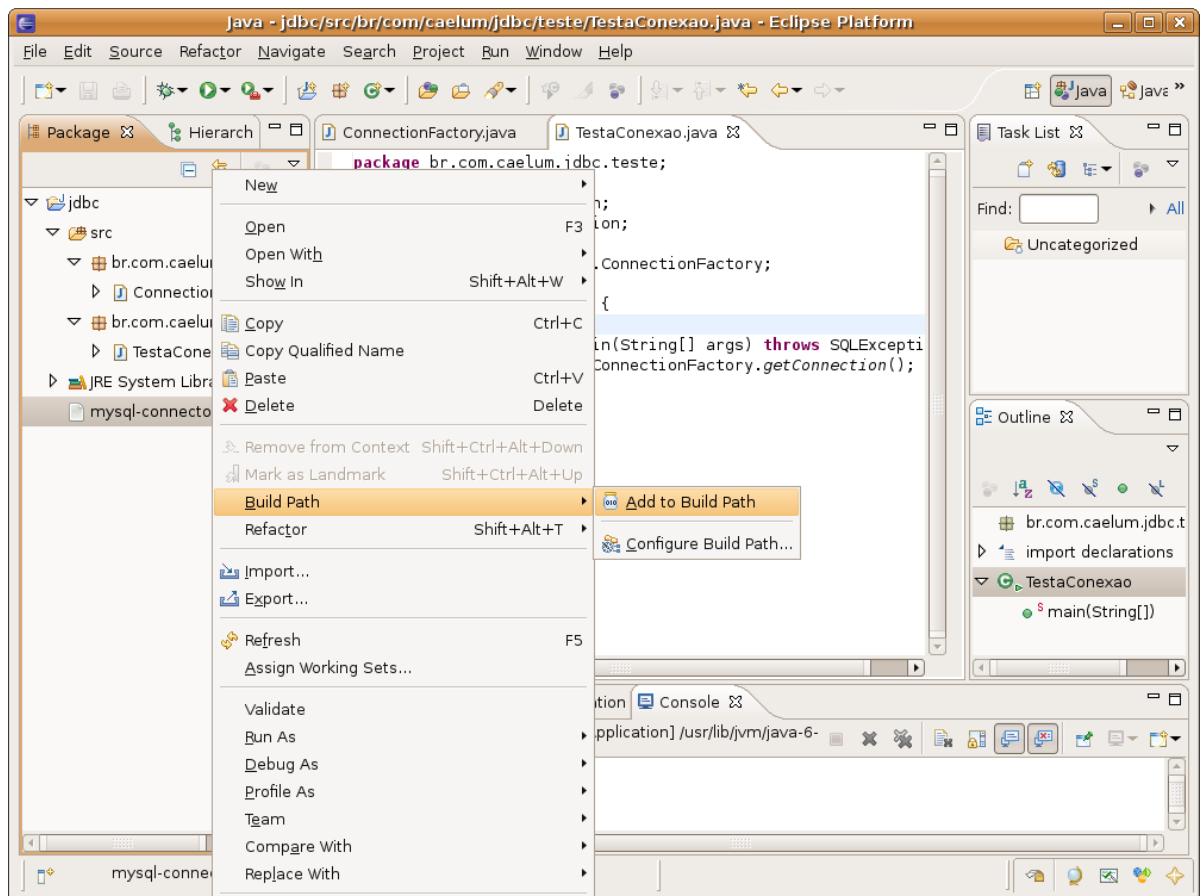
- 5) Rode a sua classe TestaConexao

- a) Clique da direita na sua classe TestaConexao

- b) Escolha Run as, Java Application



- 6) Parece que sua aplicação não funciona pois o driver não foi encontrado? Esquecemos de colocar o jar no classpath! (buildpath no eclipse)
- Clique no seu projeto, pressione F5 para executar um Refresh.
  - Selecione o seu driver do mysql, clique da direita e escolha Build Path, Add to Build Path.



c) Rode novamente sua aplicação TestaConexao agora que colocamos o driver no classpath.

## 2.8 - A tabela de exemplo

Para criar uma tabela nova, devemos rodar o comando `mysql` para entrar no mesmo.

```
mysql -u root
```

Agora nos preparamos para usar o banco de dados **teste**:

```
use teste;
```

A seguinte tabela será usada nos exemplos desse capítulo:

```
create table contatos (
    id BIGINT NOT NULL AUTO_INCREMENT,
    nome VARCHAR(255),
    email VARCHAR(255),
    endereco VARCHAR(255),
    primary key (id)
);
```

No banco de dados relacional, é comum representar um contato (entidade) em uma tabela de contatos.

## 2.9 - Javabeans

O que são Javabeans? A pergunta que não quer se calar pode ser respondida muito facilmente uma vez que a maior confusão feita aí fora é entre Javabeans e Enterprise Java Beans (EJB).

Javabeans são classes que possuem o construtor sem argumentos e métodos de acesso do tipo get e set! Mais nada! Simples, não?

Já os EJBs são javabeans com características mais avançadas e são o assunto principal do curso FJ-31 da Caelum.

Podemos usar beans por diversos motivos, normalmente as classes de modelo da nossa aplicação costumam ser javabeans.

Agora utilizaremos:

- uma classe com métodos do tipo get e set para cada um de seus parâmetros, que representa algum objeto;
- uma classe com construtor sem argumentos que representa uma coleção de objetos.

A seguir, você vê um exemplo de uma classe javabean que seria equivalente ao nosso modelo de entidade do banco de dados:

```
package br.com.caelum.jdbc.modelo;

public class Contato {

    private Long id;
    private String nome;
    private String email;
    private String endereço;

    // métodos get e set para id, nome, email e endereço

    public String getNome() {
        return this.nome;
    }
    public void setNome(String novo) {
        this.nome = novo;
    }

    public String getEmail() {
        return this.email;
    }
    public void setEmail(String novo) {
        this.email = novo;
    }

    public String getEndereço() {
        return this.endereço;
    }
    public void setEndereço(String novo) {
        this.endereço = novo;
    }
}
```

```
public Long getId() {  
    return this.id;  
}  
public void setId(Long novo) {  
    this.id = novo;  
}  
}
```

A tecnologia javabeans é muito grande e mais informações sobre essa vasta área que é a base dos componentes escritos em java pode ser encontrada em:

<http://java.sun.com/products/javabeans>

Se você quer saber mais sobre Enterprise Java Beans (EJB), a Caelum oferece o curso FJ-31. Não os confunda com Java Beans!

## 2.10 - Exercícios - Contato

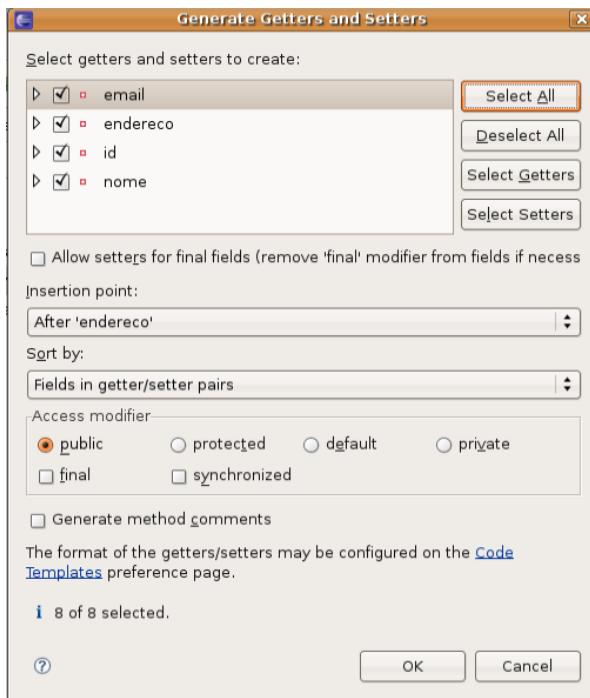
1) Crie a classe de Contato.

a) No pacote `br.com.caelum.jdbc.modelo`, crie uma classe chamada `Contato`.

```
1 package br.com.caelum.jdbc.modelo;  
2  
3 public class Contato {  
4     private Long id;  
5     private String nome;  
6     private String email;  
7     private String endereco;  
8 }
```

b) Aperte `Ctrl + 3` e digite `ggas` ou `Generate getters and setters` e selecione todos os getters e setters.





## 2.11 - Inserindo dados

Para inserir dados em uma tabela de um banco de dados entidade-relacional basta usar a cláusula **INSERT**. Precisamos especificar quais os campos que desejamos atualizar e os valores.

Primeiro o código SQL:

```
String sql = "insert into contatos (nome,email,endereco)
values ('" + nome + "', '" + email + "', '" + endereco + "')";
```

O exemplo acima possui dois pontos negativos que são importantíssimos. O primeiro é que o programador que não escreveu o código original não consegue bater o olho e entender o que está escrito. O que o código acima faz? Lendo rapidamente fica difícil. Mais difícil ainda é saber se faltou uma vírgula, um fecha parênteses talvez?

Outro problema é o clássico “preconceito contra Joana D’arc”, formalmente chamado de **SQL Injection**. O que acontece quando o contato a ser adicionado possui no nome uma aspas simples? O código sql se quebra todo e pára de funcionar ou, pior ainda, o usuário final é capaz de alterar seu código sql para executar aquilo que ele deseja (SQL injection)... tudo isso porque escolhemos aquela linha de código e não fizemos o escape de caracteres especiais.

Por esses dois motivos não usaremos código sql como mostrado anteriormente... vamos imaginar algo mais genérico e um pouco mais interessante:

```
String sql = "insert into contatos (nome,email,endereco) values (?,?,?,?)";
```

Existe uma maneira em Java de escrever o código sql como no primeiro exemplo dessa seção (com concatenações de strings). Essa maneira não será ensinada durante o curso pois é uma péssima prática que dificulta a manutenção do seu projeto.

Perceba que não colocamos os pontos de interrogação de brincadeira, e sim porque realmente não sabemos o que desejamos inserir. Estamos interessados em executar aquele código mas não sabemos ainda quais são os **parâmetros** que iremos utilizar nesse código sql que será executado, chamado de `statement`.

As cláusulas são executadas em um banco de dados através da interface `PreparedStatement`. Para receber um `PreparedStatement` relativo à conexão, basta chamar o método `prepareStatement`, passando como argumento o comando SQL com os valores vindos de variáveis preenchidos com uma interrogação.

```
String sql = "insert into contatos (nome,email,endereco) values (?,?,?)";
PreparedStatement stmt = con.prepareStatement(sql);
```

Logo em seguida, chamamos o método `setString` do `PreparedStatement` para preencher os valores, passando a posição (começando em 1) da interrogação no SQL e o valor que deve ser colocado.

```
// preenche os valores
stmt.setString(1, "Caelum");
stmt.setString(2, "contato@caelum.com.br");
stmt.setString(3, "R. Vergueiro 3185 cj57");
```

Por fim, uma chamada a `execute()` executa o comando SQL.

```
stmt.execute();
```

Agora imagine todo esse processo sendo escrito toda vez que desejar inserir algo no banco? Ainda não consegue visualizar o quanto destrutivo isso pode ser?

Veja o exemplo abaixo, que abre uma conexão e insere um contato no banco:

```
public class JDBCInsere {

    public static void main(String[] args) throws SQLException {

        // conectando
        Connection con = ConnectionFactory.getConnection();

        // cria um preparedStatement
        String sql = "insert into contatos (nome,email,endereco) values (?,?,?)";
        PreparedStatement stmt = con.prepareStatement(sql);

        // preenche os valores
        stmt.setString(1, "Caelum");
        stmt.setString(2, "contato@caelum.com.br");
        stmt.setString(3, "R. Vergueiro 3185 cj57");

        // executa
        stmt.execute();
        stmt.close();

        System.out.println("Gravado!");

        con.close();
    }
}
```

### Fechando a conexão

Não é comum utilizar JDBC hoje em dia. O mais praticado é o uso de alguma api de ORM como o **Hibernate** ou **EJB**, porém aqueles que ainda insistem no uso de JDBC devem prestar atenção no momento de fechar a conexão.

O exemplo dado acima não fecha caso algum erro ocorra no momento de inserir um dado no banco de dados.

O comum é fechar a conexão em um bloco `finally`.

### Má prática: Statement

Em vez de usar o `PreparedStatement`, você pode usar uma interface mais simples chamada `Statement`, que simplesmente executa uma cláusula SQL no método `execute`:

```
Statement stmt = con.createStatement();
stmt.execute("INSERT INTO ...");
stmt.close();
```

Mas prefira a classe `PreparedStatement` que é mais rápida que `Statement` e deixa seu código muito mais limpo.

Geralmente, seus comandos SQL conterão valores vindos de variáveis do programa Java; usando `Statements`, você terá que fazer muitas concatenações, mas usando `PreparedStatements`, isso fica mais limpo e fácil.

## 2.12 - DAO – Data Access Object

Já foi possível sentir que colocar código SQL dentro de suas classes de lógica é algo nem um pouco elegante e muito menos viável quando você precisa manter o seu código.

Quantas vezes você não ficou bravo com o programador responsável por aquele código ilegível?

A idéia a seguir é remover o código de acesso ao banco de dados de suas classes de lógica e colocá-lo em uma classe responsável pelo acesso aos dados. Assim o código de acesso ao banco de dados fica em um lugar só, tornando mais fácil a manutenção.

Que tal seria se pudéssemos chamar um método `adiciona` que adiciona um `Contato` ao banco?

Em outras palavras quero que o código a seguir funcione:

```
// adiciona os dados no banco
Misterio bd = new Misterio();
bd.adiciona("meu nome", "meu email", "meu endereço");
```

Mas... Java é orientado a `Strings`? Vamos tentar novamente: em outras palavras quero que o código a seguir funcione:

```
// adiciona um contato no banco
Misterio bd = new Misterio();

// método muito mais elegante
bd.adiciona(contato);
```

Tentaremos chegar ao código anterior: seria muito melhor e mais elegante poder chamar um único método responsável pela inclusão, certo?

```
public class TestaInsere {  
  
    public static void main(String[] args) throws SQLException {  
  
        // pronto para gravar  
        Contato contato = new Contato();  
        contato.setNome("Caelum");  
        contato.setEmail("contato@caelum.com.br");  
        contato.setEndereco("R. Vergueiro 3185 cj87");  
  
        // grava nessa conexão!!!  
        Misterio bd = new Misterio();  
  
        // método elegante  
        bd.adiciona(contato);  
  
        System.out.println("Gravado!");  
  
    }  
}
```

O código anterior já mostra o poder que alcançaremos: através de uma única classe seremos capazes de acessar o banco de dados e, mais ainda, somente através dessa classe será possível acessar os dados.

Esta idéia, inocente à primeira vista, é capaz de isolar todo o acesso a banco em classes bem simples, cuja instância é um **objeto** responsável por **acessar os dados**. Da responsabilidade deste objeto surgiu o nome de **Data Access Object** ou simplesmente **DAO**, um dos mais famosos padrões de desenvolvimento (design pattern).

O que falta para o código acima funcionar é uma classe chamada `ContatoDAO` com um método chamado `adiciona`. Vamos criar uma que se conecta ao banco ao construirmos uma instância dela:

```
public class ContatoDAO {  
  
    // a conexão com o banco de dados  
    private Connection connection;  
  
    public ContatoDAO() throws SQLException {  
        this.connection = ConnectionFactory.getConnection();  
    }  
}
```

Agora que todo `ContatoDAO` possui uma conexão com o banco podemos focar no método `adiciona`, que recebe um `Contato` como argumento e é responsável por adicioná-lo através de código sql.

```
public void adiciona(Contato contato) throws SQLException {  
  
    // prepared statement para inserção  
    String sql = "insert into contatos (nome,email,endereco) values (?,?,?)";  
    PreparedStatement stmt = con.prepareStatement(sql);  
  
    // seta os valores  
    stmt.setString(1, contato.getNome());  
    stmt.setString(2, contato.getEmail());
```

```
stmt.setString(3, contato.getEndereco());  
  
        // executa  
        stmt.execute();  
        stmt.close();  
    }  

```

## 2.13 - Exercícios - ContatoDAO

- 1) Crie a classe br.com.caelum.jdbc.dao.ContatoDAO

```
1 package br.com.caelum.jdbc.dao;  
2  
3 // imports aqui (CTRL+SHIFT+0)  
4  
5 public class ContatoDAO {  
6  
7     // a conexão com o banco de dados  
8     private Connection connection;  
9  
10    public ContatoDAO() throws SQLException {  
11        this.connection = ConnectionFactory.getConnection();  
12    }  
13  
14    public void adiciona(Contato contato) throws SQLException {  
15  
16        // prepared statement para inserção  
17        String sql = "insert into contatos (nome,email,endereco) values (?,?,?)";  
18        PreparedStatement stmt = connection.prepareStatement(sql);  
19  
20        // seta os valores  
21        stmt.setString(1,contato.getNome());  
22        stmt.setString(2,contato.getEmail());  
23        stmt.setString(3,contato.getEndereco());  
24  
25        // executa  
26        stmt.execute();  
27        stmt.close();  
28    }  
29 }
```

Lembre-se de importar as classes de SQL do pacote java.sql!

- 2) Crie, no pacote br.com.caelum.jdbc.teste, uma classe chamada TestaInsere com um método main:

```
package br.com.caelum.jdbc.teste;  
  
// imports aqui (CTRL+SHIFT+0)  
  
public class TestaInsere {  
  
    public static void main(String[] args) throws SQLException {
```

```
// pronto para gravar
Contato contato = new Contato();
contato.setNome("Caelum");
contato.setEmail("contato@caelum.com.br");
contato.setEndereco("R. Vergueiro 3185 cj57");

// grave nessa conexão!!!
ContatoDAO dao = new ContatoDAO();

// método elegante
dao.adiciona(contato);

System.out.println("Gravado!");
}
}
```

- 3) Teste seu programa.
- 4) Verifique se o contato foi adicionado. Abra o terminal e digite:

```
mysql -h localhost -u root
use teste;
select * from contatos;

exit para sair do console do MySQL.
```

## 2.14 - Exercícios Opcionais

- 1) Altere seu programa e use a classe `java.util.Scanner` do Java 5 para ler os dados através do teclado:

```
// cria o Scanner
Scanner teclado = new Scanner(System.in);

// pronto para gravar
Contato contato = new Contato();

System.out.print("Nome: ");
contato.setNome(teclado.nextLine());

System.out.print("E-mail: ");
contato.setEmail(teclado.nextLine());

System.out.print("Endereço: ");
contato.setEndereco(teclado.nextLine());
```

## 2.15 - Pesquisando

Para pesquisar também utilizamos a interface `PreparedStatement`, de forma que o método `executeQuery` retorna todos os contatos no exemplo a seguir.

O objeto retornado é do tipo `ResultSet` que permite navegar por seus registros através do método `next`.

Esse método irá retornar false quando chegar ao fim da pesquisa, portanto ele é normalmente utilizado para fazer um loop nos registros como no exemplo a seguir:

```
// pega a conexão e o Statement
Connection con = ConnectionFactory.getConnection();
PreparedStatement stmt = con.prepareStatement("select * from contatos");

// executa um select
ResultSet rs = stmt.executeQuery();

// itera no ResultSet
while (rs.next()) {
}

rs.close();
stmt.close();
con.close();
```

Para retornar o valor de uma coluna no banco de dados basta chamar um dos métodos get do ResultSet, dentre os quais, o mais comum: getString.

```
// pega a conexão e o Statement
Connection con = ConnectionFactory.getConnection();
PreparedStatement stmt = con.prepareStatement("select * from contatos");

// executa um select
ResultSet rs = stmt.executeQuery();

// itera no ResultSet
while (rs.next()) {
    System.out.println(
        rs.getString("nome") + " :: " + rs.getString("email")
    );
}

stmt.close();
con.close();
```

#### Recurso Avançado: O cursor

Assim como o cursor do banco de dados, só é possível mover para o próximo registro. Para permitir um processo de leitura para trás é necessário especificar na abertura do ResultSet que tal cursor deve ser utilizado.

Mas, novamente, podemos aplicar as idéias de **DAO** e criar um método `getLista()` no nosso `ContatoDAO`. Mas o que esse método retornaria? Um `ResultSet`? E teríamos o código de manipulação de `ResultSet` espalhado por todo o código? Vamos fazer nosso `getLista()` devolver algo mais interessante, uma lista de `Contato`:

```
PreparedStatement stmt = this.connection.prepareStatement("select * from contatos");
ResultSet rs = stmt.executeQuery();

List<Contato> contatos = new ArrayList<Contato>();
```

```
while (rs.next()) {  
  
    // criando o objeto Contato  
    Contato contato = new Contato();  
    contato.setNome(rs.getString("nome"));  
    contato.setEmail(rs.getString("email"));  
    contato.setEndereco(rs.getString("endereco"));  
  
    // adicionando o objeto à lista  
    contatos.add(contato);  
}  
  
rs.close();  
stmt.close();  
  
return contatos;
```

## 2.16 - Exercícios - Listagem

- 1) Crie o método `getLista` na classe `ContatoDAO`. Importe `List` da `java.util`.

```
1 public List<Contato> getLista() throws SQLException {  
2  
3     PreparedStatement stmt = this.connection.prepareStatement("select * from contatos");  
4     ResultSet rs = stmt.executeQuery();  
5  
6     List<Contato> contatos = new ArrayList<Contato>();  
7  
8     while (rs.next()) {  
9         // criando o objeto Contato  
10        Contato contato = new Contato();  
11        contato.setNome(rs.getString("nome"));  
12        contato.setEmail(rs.getString("email"));  
13        contato.setEndereco(rs.getString("endereco"));  
14  
15        // adicionando o objeto à lista  
16        contatos.add(contato);  
17    }  
18  
19    rs.close();  
20    stmt.close();  
21  
22    return contatos;  
23 }
```

- 2) Vamos usar o método `getLista` agora para listar todos os contatos do nosso banco de dados.

- 3) Crie uma classe chamada `TestaListaDAO` com um método `main`:

- a) Crie um `ContatoDAO`:

```
ContatoDAO dao = new ContatoDAO();
```

- b) Liste os contatos com o DAO:

```
List<Contato> contatos = dao.getLista();
```

- c) Itere nessa lista e imprima as informações dos contatos:

```
for (Contato contato : contatos) {  
    System.out.println("Nome: " + contato.getNome());  
    System.out.println("Email: " + contato.getEmail());  
    System.out.println("Endereço: " + contato.getEndereco() + "\n");  
}
```

- 4) Rode o programa anterior clicando da direita no mesmo, Run, Run as Java Application.

## 2.17 - Um pouco mais...

- 1) Assim como o MYSQL existem outros bancos de dados gratuitos e opensource na internet. O HSQLDB é um banco desenvolvido em Java que pode ser acoplado a qualquer aplicação e libera o cliente da necessidade de baixar qualquer banco de dados antes da instalação de um produto Java!
- 2) O Hibernate tomou conta do mercado e virou febre mundial pois não se faz necessário escrever uma linha de código SQL!
- 3) Outros projetos como o Hibernate são bastante famosos como o OJB e o Torque, cada um com suas próprias características implementando a funcionalidade de ORM.
- 4) O Prevayler, iniciativa brasileira, funciona como uma camada de prevalência de seus objetos, de uma maneira muito diferente de uma ferramenta de ORM convencional.
- 5) Se um projeto não usa nenhuma das tecnologias de ORM disponíveis, o mínimo a ser feito é seguir o DAO.

## 2.18 - Exercícios opcionais

- 1) Use cláusulas where para refinar sua pesquisa no banco de dados. Por exemplo: where nome like 'C%'
- 2) Crie o método pesquisar que recebe um id (int) e retorna um objeto do tipo Contato.

### Desafios

- 1) Faça conexões para outros tipos de banco de dados disponíveis.

## 2.19 - Outros métodos para o seu DAO

Agora que você já sabe usar o PreparedStatement para executar qualquer tipo de código SQL e ResultSet para receber os dados retornados da sua pesquisa fica simples, porém maçante, escrever o código de diferentes métodos de uma classe típica de DAO.

Veja primeiro o método altera, que recebe um contato cujos valores devem ser alterados:

```
1 public void altera(Contato contato) throws SQLException {
2     PreparedStatement stmt = connection.prepareStatement("update contatos " +
3                                         "set nome=?, email=?, endereco=? where id=?");
4     stmt.setString(1, contato.getNome());
5     stmt.setString(2, contato.getEmail());
6     stmt.setString(3, contato.getEndereco());
7     stmt.setLong(4, contato.getId());
8     stmt.execute();
9     stmt.close();
10 }
```

Não existe nada de novo nas linhas acima. Uma execução de query! Simples, não?

Agora o código para remoção: começa com uma query baseada em um contato, mas usa somente o id dele para executar a query do tipo delete:

```
1 public void remove(Contato contato) throws SQLException {
2     PreparedStatement stmt = connection.prepareStatement("delete from contatos where id=?");
3     stmt.setLong(1, contato.getId());
4     stmt.execute();
5     stmt.close();
6 }
```

## 2.20 - Exercícios opcionais - Alterar e remover

- 1) Adicione o método para alterar contato no seu ContatoDAO.

```
1 public void altera(Contato contato) throws SQLException {
2     PreparedStatement stmt = connection.prepareStatement("update " +
3                                         "contatos set nome=?, email=?, endereco=? where id=?");
4     stmt.setString(1, contato.getNome());
5     stmt.setString(2, contato.getEmail());
6     stmt.setString(3, contato.getEndereco());
7     stmt.setLong(4, contato.getId());
8     stmt.execute();
9     stmt.close();
10 }
```

- 2) Adicione o método para remover contato no seu ContatoDAO

```
public void remove(Contato contato) throws SQLException {
    PreparedStatement stmt = connection.prepareStatement("delete from contatos where id=?");
    stmt.setLong(1, contato.getId());
    stmt.execute();
```

```
    stmt.close();  
}
```

- 3) Use os métodos criados anteriormente para fazer testes com o seu banco de dados: atualize e remova um contato.
- 4) Crie uma classe chamada Funcionario com os campos id (Long), nome, usuario e senha (String).
- 5) Crie uma tabela no banco de dados.
- 6) Crie uma classe DAO.
- 7) Use-a para instanciar novos funcionários e colocá-los no seu banco.

# O que é J2EE?

*“Ensinar é aprender duas vezes.”*

— Joseph Joubert

- O que é o Java Enterprise Edition?
- Servidor de aplicação
- Servlet Contêiner
- Implementação de referência

## 3.1 - Java EE

O **Java EE** (Java Enterprise Edition ou Java EE) não passa de uma série de especificações bem detalhadas, dando uma receita de como deve ser implementado um software que faz um determinado serviço.

Veremos no curso os vários serviços que um software deve implementar para seguir as especificações do Java EE. Veremos também conceitos muito importantes, para depois firmar jargões como **servidor de aplicação e contêiners**.

Esses serviços variam desde envio de emails, até complexos serviços de transação.

Porque a Sun faz isso? A idéia é que você possa criar uma aplicação que utilize esses serviços. Como esses serviços são bem complicados, você não perderá tempo implementando essa parte do sistema, porém terá de comprar de alguém (existem implementações gratuitas de excelente qualidade).

Algum dia, você poderá querer trocar essa implementação atual por uma que é mais rápida em determinados pontos (e consequentemente mais cara). Porém continuará utilizando a mesma interface, isto é, como você chama aquelas funcionalidades do Java EE. O que muda é a implementação da especificação: você tem essa liberdade, não está preso a um código e a especificação garante que sua aplicação funcionará com a implementação de outra empresa.

### Onde encontrar as especificações

O grupo responsável por gerir as especificações usa o site do Java Community Process: <http://www.jcp.org/>

Lá você pode encontrar tudo sobre as Java Specification Requests, isto é, os novos pedidos de bibliotecas e especificações para o Java, tanto para JSE, quanto EE e outros.

Sobre o Java EE, você pode encontrar em: <http://java.sun.com/javaee/>

**J2EE**

O nome J2EE era usado nas versões mais antigas, até a 1.4. Hoje, o nome correto é Java EE, mas você ainda vai ver muitas referências a J2EE.

**3.2 - APIs**

As APIs a seguir são as principais dentre as disponibilizadas pelo Java Enterprise Edition através de sua especificação em sua versão 5:

- 1) JavaServer Pages (JSP), Java Servlets, Java Server Faces (JSF) (trabalhar para a web)
- 2) Enterprise JavaBeans Components (EJB) e Java Persistence API (objetos distribuídos, clusters, acesso remoto a objetos etc)
- 3) Java API for XML Web Services (JAX-WS), Java API for XML Binding (JAX-B) (trabalhar com arquivos xml)
- 4) Java Authentication and Authorization Service (JAAS) (API padrão do Java para segurança)
- 5) Java Transaction API (JTA) (controle de transação no contêiner)
- 6) Java Message Service (JMS) (troca de mensagens síncronas ou não)
- 7) Java Naming and Directory Interface (JNDI) (espaço de nomes e objetos)
- 8) Java Management Extensions (JMX) (administração da sua aplicação e estatísticas sobre a mesma)

Entre outras para trabalhar com Webservices e outros tipos de acesso remoto ou invocação remota de métodos (RMI).

**3.3 - Implementações compatíveis com a especificação**

Existem diversos servidores de aplicação famosos compatíveis com a especificação do J2EE 1.4, abaixo listamos alguns. A versão 5 já vem sendo utilizada em projetos mais recentes, mas empresas usualmente resistem em migrar projetos já antigos para uma nova versão da tecnologia.

Do grupo a seguir, o JBoss se destaca como o primeiro a implementar essas novas especificações.

- 1) RedHat/Jboss, JBoss Application Server, gratuito, Java EE 5;
- 2) Sun, GlassFish, gratuito, Java EE 5.
- 3) Apache, Apache Geronimo, gratuito, Java EE 5;
- 4) Oracle/BEA, WebLogic Application Server, Java EE 5;
- 5) IBM, IBM Websphere Application Server, Java EE 5;
- 6) Sun, Sun Java System Application Server (baseado no GlassFish), pago, Java EE 5;
- 7) SAP, SAP Application Server, pago, Java EE 5;
- 8) Objectweb, Objectweb Jonas, gratuito, J2EE 1.4;
- 9) (outros)

# Servlet Container

*“Que ninguém se engane: só se consegue a simplicidade através de muito trabalho.”*  
– Clarisse Lispector

O que é e como funciona um Servlet Container?

## 4.1 - Introdução

No começo, a Internet era uma dúzia de páginas estáticas contendo pesquisas acadêmicas de diversas instituições.

Da necessidade de gerar conteúdo dinâmico, como os primeiros contadores, uma idéia considerada bem simples hoje em dia, surgiram os primeiros programas de CGI (Common Gateway Interface).

Através de linguagens como **C**, **C++**, **Perl**, **ASP**, **PHP**, **Cobol**, **Delphi**, **Shell**, etc, foi possível gerar conteúdo que permite ao usuário acesso a diversas funcionalidades através de páginas **HTML**, como quando você deseja comprar produtos em uma loja virtual.

Para melhorar o desempenho do último, inventaram o que viria a ser uma **servlet**, uma nova forma de trabalhar com requisições de clientes via web que economiza o tempo de processamento de uma chamada e a memória que seria gasta para tal processo, além de ser em Java e possuir todas as vantagens e facilidades da orientação a objetos.

Além do mais, servlets são tão portáveis quanto qualquer programa escrito em Java, e aqueles que programam servlets não precisam mais se preocupar com a funcionalidade do servidor, que já foi escrita para nós e não precisa ser alterada.

### HTML

Este curso tem como pré-requisito o conhecimento de HTML: saber utilizar as tags principais para a construção de páginas dinâmicas (html, body, form, input, textarea e select).

Caso não esteja acostumado com páginas HTML, recomenda-se que tente ler algum tutorial para que não apareçam dificuldades durante o curso.

Em breve estudaremos as servlets, mas antes veremos o JSP (Java Server Pages), que é como escrevemos a maior parte de nossas páginas dinâmicas em Java.

## 4.2 - Aplicações web no Java EE e Servlet Container

O Java EE 5 é composto pelas seguintes especificações ligadas a uma aplicação web:

- JSP
- Servlets
- JSTL
- JSF

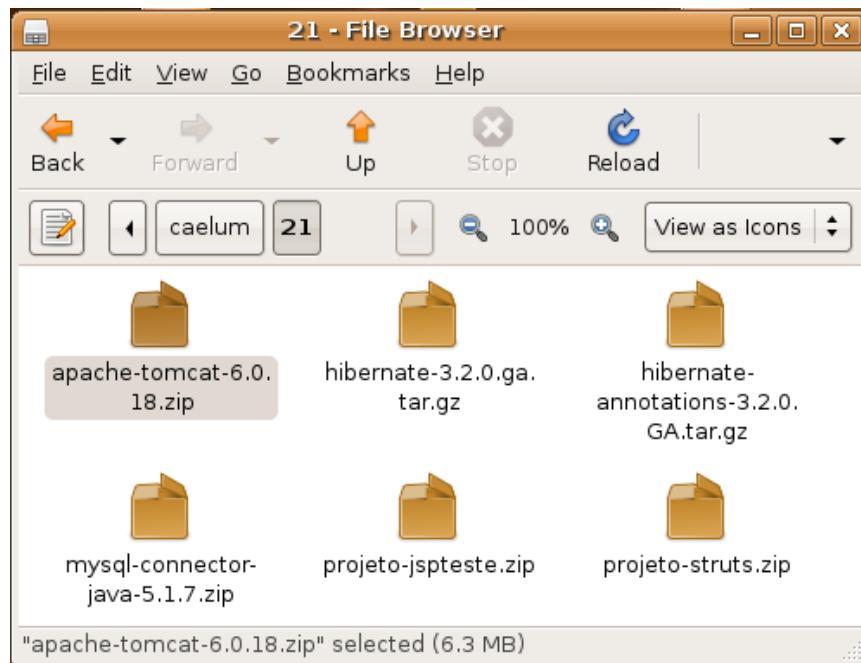
Um Servlet Container é um servidor que suporta essas funcionalidades mas não o Java EE completo. É indicado a quem não precisa de tudo do Java EE e está interessado apenas na parte web (boa parte do mercado se encaixa nessa categoria).

Há alguns servlet containers famosos no mercado. O mais famoso é o Apache Tomcat, mas há outros como o Jetty da Mortbay.

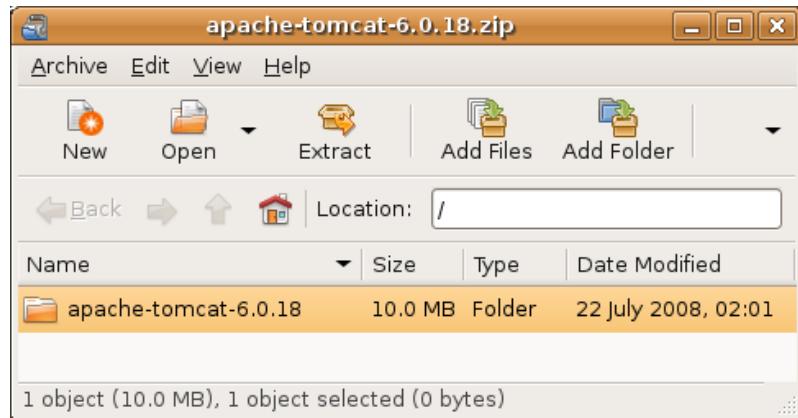
## 4.3 - Instalando o Tomcat

Para instalar o Tomcat na Caelum, siga os seguintes passos:

- 1) Entre no atalho **caelum** na sua Área de Trabalho;
- 2) Entre na pasta **21** e selecione o arquivo do **apache-tomcat**;



- 3) Dê dois cliques para abrir o Archive Manager do Linux;



- 4) Clique em **Extract**;
- 5) Escolha o seu **Desktop** e clique em extract;



- 6) O resultado é uma pasta chamada **apache-tomcat**: o tomcat já está instalado.



#### 4.4 - Sobre o Tomcat

Baixe o Tomcat em <http://tomcat.apache.org> na página de downloads da versão que escolher, você precisa de uma "Binary Distribution".

O Tomcat é considerado implementação padrão e referência de novas APIs de servlets, isto é, quando uma nova especificação surge, o Tomcat costuma ser o primeiro servlet contêiner a implementar a nova API. Ele também é o servlet contêiner padrão utilizado pelo JBoss.

## 4.5 - Instalando o Tomcat em casa

### Iniciando o Tomcat

Entre no diretório de instalação e rode o programa `startup.sh`:

```
cd apache-tomcat<TAB>/bin  
.startup.sh
```

### Parando o tomcat

Entre no diretório de instalação do tomcat e rode o programa `shutdown.sh`:

```
cd apache-tomcat<TAB>/bin  
.shutdown.sh
```

### Tomcat no Windows

Para instalar o Tomcat no Windows basta executar o arquivo `.exe` que pode ser baixado no site do Tomcat. Depois disso, você pode usar os scripts `startup.bat` e `shutdown.bat`, analogamente aos scripts do Linux.

Tudo o que vamos desenvolver neste curso funciona em qualquer ambiente compatível com o Java Enterprise Edition.

## 4.6 - Outra opção: Jetty

O Jetty é uma outra implementação criada pela MortBay (<http://jetty.mortbay.org>) de Servlet Container e HTTP Server.

Pequeno e eficiente, ele é uma opção ao Tomcat bastante utilizada devido a algumas de suas características. Especialmente:

- facilmente embarcável;
- escalável;
- “plugabilidade”, isto é, é fácil trocar para diferentes implementações dos principais componentes da API.

O Jetty também costuma implementar, antes do Tomcat, idéias diferentes que ainda não estão na API de servlets do Java EE. Uma dessas implementações pioneiras foi do uso de conectores NIO, por exemplo.

# O eclipse e seus plugins

*“O repouso é uma boa coisa mas o tédio é seu irmão.”*  
— Voltaire (François-Marie Arouet)

Neste capítulo, você aprenderá a:

- instalar o Eclipse com WTP;
- configurar o Tomcat dentro do WTP;
- conhecer outros plugins.

## 5.1 - O plugin WTP

O **WTP**, *Web Tools Platform*, é um conjunto de plugins que auxilia o desenvolvimento de aplicações Java EE, em particular, de aplicações Web. Contém desde editores para JSP, JS e HTML até perspectivas e jeitos de rodar servidores de dentro do Eclipse.

Este plugin vai nos ajudar bastante com content-assists e atalhos para tornar o desenvolvimento Web mais eficiente.

Para instalar o eclipse com WTP basta ir no site do Eclipse e:

- 1) Abra a página [www.eclipse.org/downloads](http://www.eclipse.org/downloads) ;
- 2) Baixe o Eclipse IDE for Java EE Developers;
- 3) Descompacte o arquivo e pronto.

### Para saber mais: outros plugins

O WTP é o plugin oficial do eclipse para desenvolvimento Java EE. Mas há muitas outras possibilidades de plugins.

Antes do WTP, era muito comum o uso do plugin **Sysdeo Tomcat** para gerenciar o tomcat e o **Amateras HTMLEditor** para suporte a JSPs. São alternativas ainda disponíveis e com a vantagem de serem mais leves que o WTP, embora com muito menos recursos.

O **MyEclipse** é outro plugin muito famoso. Ele é bem completo, com suporte a Java EE, Spring, Struts, Desktop, Mobile e outros. É uma ferramenta paga para o Eclipse, mas sua anuidade é barata.

Há outras possibilidades. No curso, usaremos o WTP por ser oficial, gratuito e bastante completo.

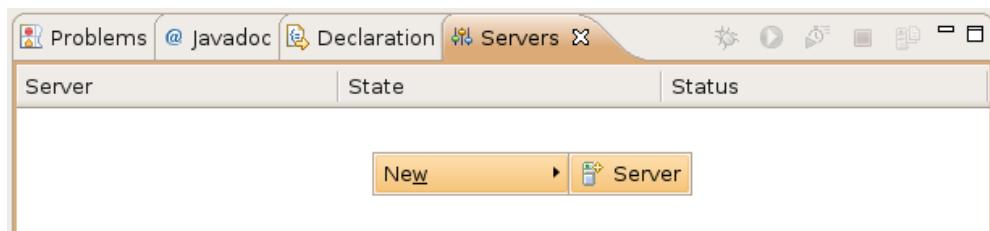
## 5.2 - Configurando o Tomcat no WTP

Vamos primeiro configurar no WTP o servidor Tomcat que acabamos de descompactar.

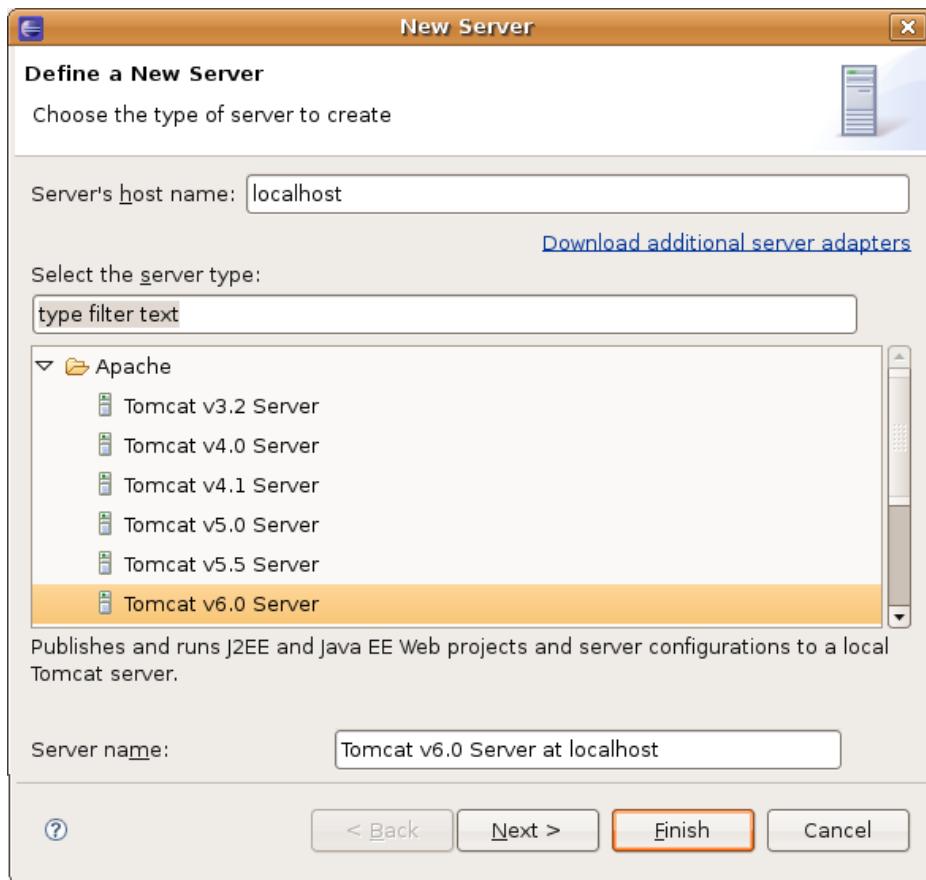
- 1) Mude a perspectiva do Eclipse para **Java** (e não Java EE, por enquanto). Para isso, vá no canto direito superior e selecione **Java**;
- 2) Abra a *View* de **Servers** na perspectiva atual. Aperte **Ctrl + 3** e digite **Servers**:



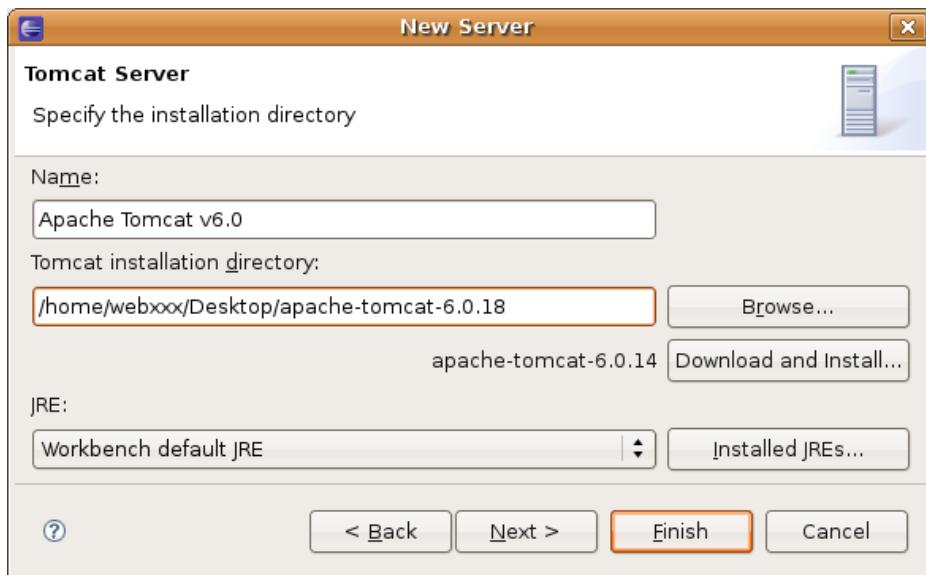
- 3) Clique com o botão direito dentro da aba Servers e vá em **New > Server**:



- 4) Selecione o **Apache Tomcat 6.0** e clique em **Next**:

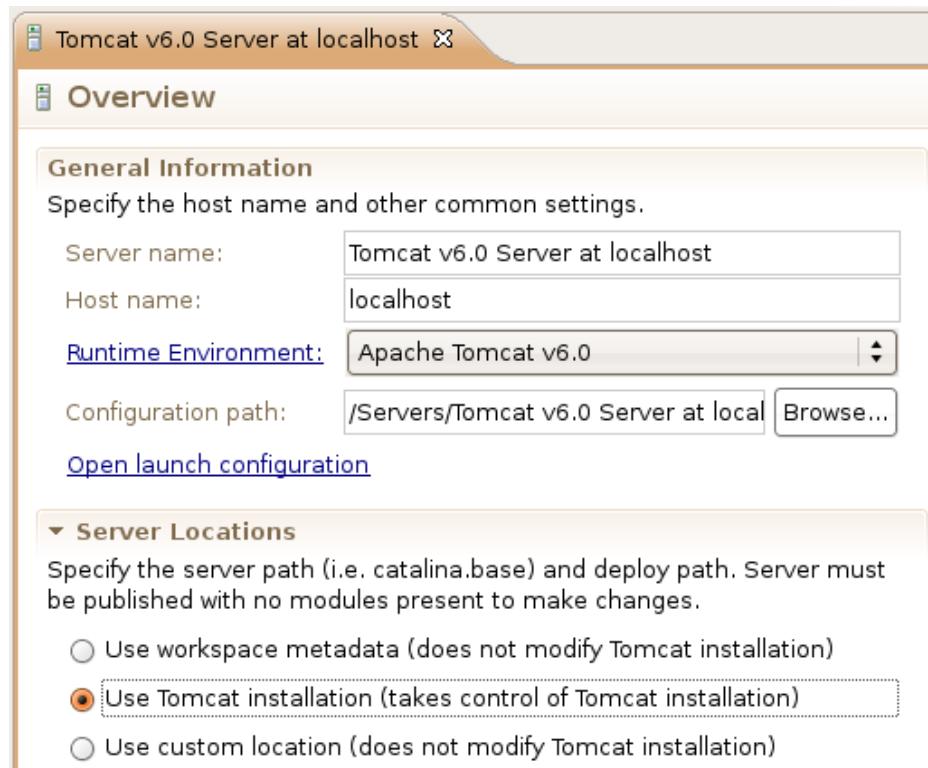


- 5) Na próxima tela, selecione o diretório onde você descompactou o Tomcat e clique em **Finish**:



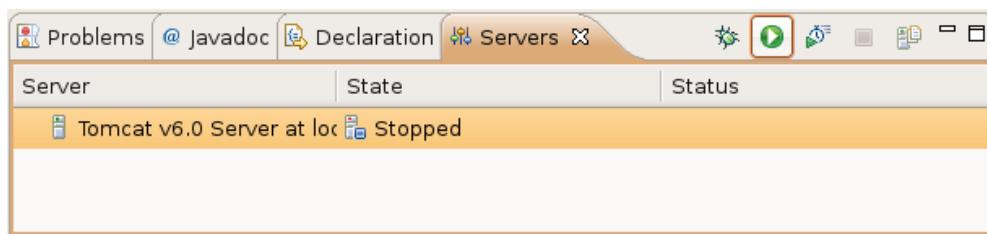
- 6) Por padrão, o WTP gerencia todo o Tomcat para nós e não permite que configurações sejam feitas por fora do Eclipse. Para simplificar, vamos desabilitar isso e deixar o Tomcat no modo padrão do próprio Tomcat.

Na aba Servers, dê dois cliques no servidor Tomcat que uma tela de configuração se abrirá. Localize a seção **Server Locations**. Repare que a opção *use workspace metadata* está marcada. Marque a opção **Use Tomcat installation:**



Salve e feche essa tela.

- 7) Selecione o servidor que acabamos de adicionar e clique em **Start** (ícone play verde na view servers):



- 8) Abra o navegador e acesse a URL `http://localhost:8080/` Deve aparecer uma tela de mensagem do Tomcat e a versão do mesmo.

Pronto! O WTP está configurado para rodar com o Tomcat!

# Novo projeto web

*“São muitos os que usam a régua, mas poucos os inspirados.”*  
– Platão

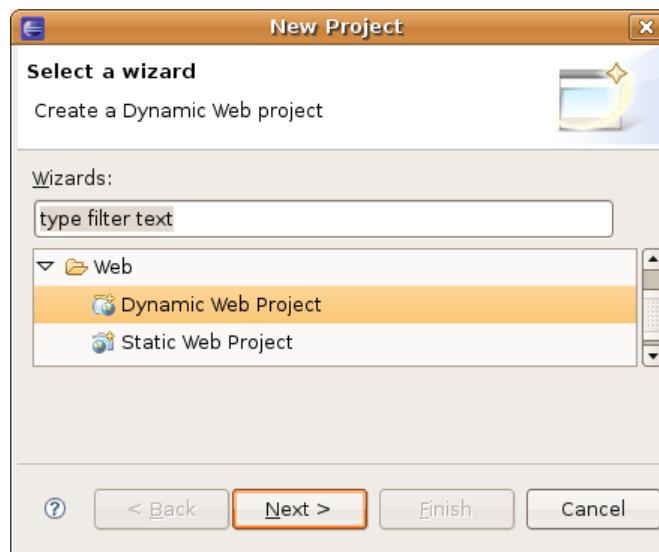
Nesse capítulo, você aprenderá:

- a criar um novo projeto web no eclipse;
- quais são os diretórios importantes de uma aplicação web;
- quais são os arquivos importantes de uma aplicação web;
- onde colocar suas páginas e arquivos estáticos;
- rodar a aplicação no Tomcat de dentro do Eclipse.

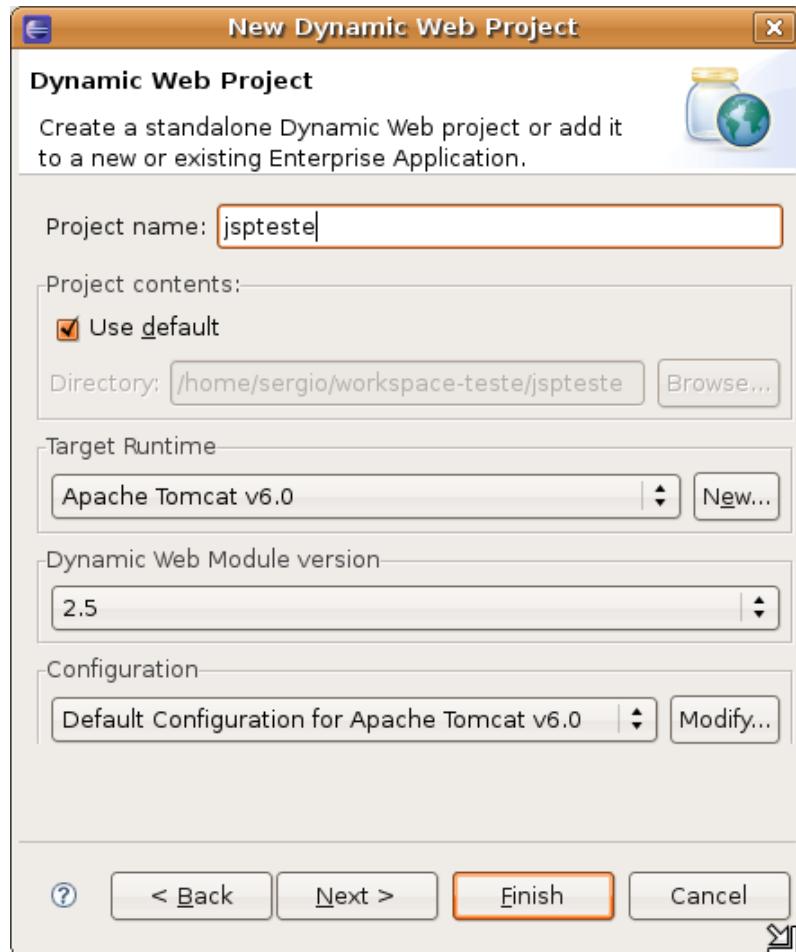
## 6.1 - Novo projeto

Primeiro, vamos criar um novo projeto web no Eclipse usando os recursos do Eclipse WTP.

1) Vá em **New > Project** e selecione **Dynamic Web Project** e clique **Next**:



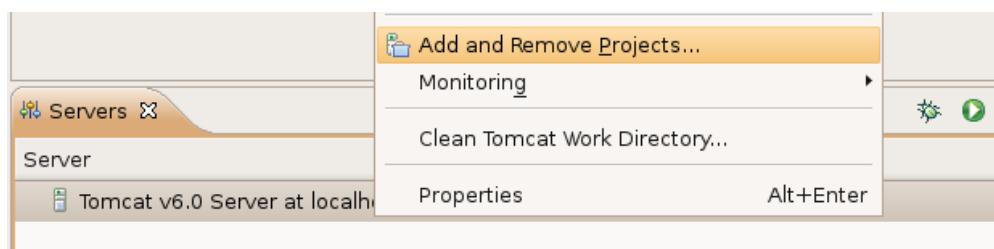
2) Coloque o nome do projeto como **jspteste** e selecione a versão do Tomcat que acabamos de configurar como Runtime Environment:



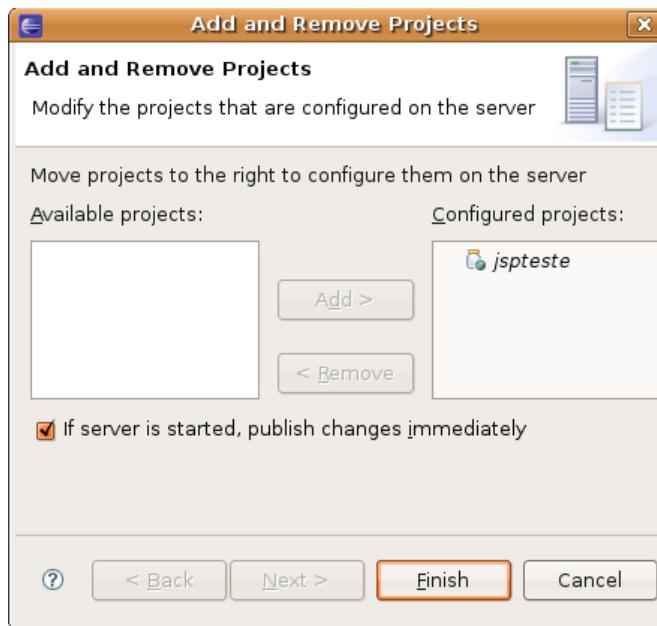
3) Clique em **Finish**. Se for perguntado sobre a mudança para a perspectiva Java EE, selecione **Não**.

O último passo é configurar o projeto para rodar no Tomcat que configuramos:

1) Na aba **Servers**, clique com o botão direito no Tomcat e vá em **Add and remove projects**:



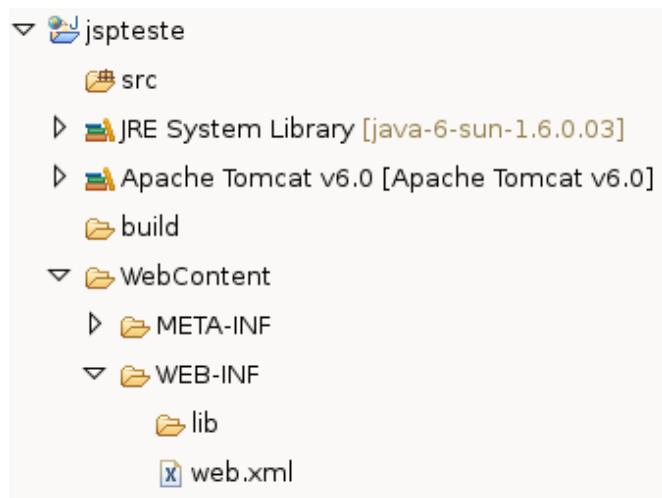
2) Agora basta selecionar o nosso projeto jspteste e clicar em **Add**:



Dê uma olhada nas pastas que foram criadas e na estrutura do nosso projeto nesse instante. Vamos analisá-la em detalhes!

## 6.2 - Análise do resultado final

Olhe bem a estrutura de pastas e verá algo parecido com isso:



A pasta **src** já é velha conhecida. É onde vamos colocar nosso código fonte Java. Em um projeto normal do Eclipse, essas classes seriam compiladas para a pasta *bin*, mas no WTP é costume se usar a pasta **build** que vemos no nosso projeto.

Nosso projeto será composto de muitas páginas web e vamos querer acessá-lo no navegador web. Já sabemos que o servidor é acessado pelo <http://localhost:8080>, mas como será que dizemos que queremos acessar o nosso projeto e não eventuais outros projetos?

No Java EE, trabalhamos com o conceito de diferentes **contextos web** para diferenciar sites ou projetos distintos em um mesmo servidor. Na prática, é como uma *pasta virtual* que, quando acessada, remete a algum projeto em questão.

Por padrão, o WTP gera o **context name** com o mesmo nome do projeto; no nosso caso, **jspteste**. Podemos mudar isso na hora de criar o projeto ou posteriormente indo em **Project > Properties > Web Project Settings** e mudando a opção **Context Root**. Repare que *não* é necessário que o nome do contexto seja o mesmo nome do projeto.

Agora sabemos que, para acessar o projeto na URL, usaremos: <http://localhost:8080/jspteste/>

Mas o que será que vai aparecer quando abrirmos essa URL? Será que veremos todo o conteúdo do projeto? Por exemplo, será possível acessar a pasta **src** que está dentro do nosso projeto? Tomara que não.

Para solucionar isso, uma outra configuração é importante no WTP: o **Content Directory**. Ao invés de abrir acesso a tudo, criamos uma pasta dentro do projeto e dizemos que ela é a raiz (root) do conteúdo a ser exibido no navegador. No WTP, por padrão, é criada a pasta **WebContent**, mas poderia ser qualquer outra pasta configurada na criação do projeto (outro nome comum de se usar é **web**).

Tudo que colocarmos na pasta **WebContent** será acessível na URL do projeto. Por exemplo, se queremos uma página de boas vindas <http://localhost:8080/jspteste/bemvindo.html> criamos o arquivo **jspteste/WebContent/bemvindo.html**

## WEB-INF

Mas repare que dentro da **WebContent** já há uma pasta chamada **WEB-INF**. Essa pasta é extremamente importante para qualquer projeto web Java EE. Ela contém *configurações* e *recursos* necessários para nosso projeto rodar no servidor.

Boa parte das configurações fica no arquivo **web.xml** dentro dessa pasta. Abra-o e veja sua estrutura, por enquanto bem simples:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  id="WebApp_ID" version="2.5">
  <display-name>jspteste</display-name>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.htm</welcome-file>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

É o básico gerado pelo próprio WTP. Tudo o que ele faz é definir o nome da aplicação e a lista de arquivos acessados que vão ser procurados por padrão. Todas essas configurações são opcionais.

Repare ainda que há uma pasta chamada **lib** dentro da WEB-INF. Muitos projetos web que vamos escrever vão precisar usar bibliotecas externas, como o driver do MySQL por exemplo. Copiamos todas elas para essa pasta **lib**. Cuidado que podemos copiar apenas arquivos *.jar* para essa pasta.

#### WEB-INF/lib

O diretório lib dentro do WEB-INF pode conter todas as bibliotecas necessárias para a aplicação web, evitando assim que o classpath da máquina que roda a aplicação precise ser alterado.

Além do mais, cada aplicação web poderá usar suas próprias bibliotecas com suas versões específicas! Você vai encontrar projetos open source que somente fornecem suporte e respondem perguntas aqueles usuários que utilizam tal diretório para suas bibliotecas, portanto evite ao máximo o uso do classpath global.

Há ainda um último diretório, oculto no Eclipse, o **WEB-INF/classes**. Para rodarmos nossa aplicação no servidor, precisamos ter acesso às classes compiladas (não necessariamente ao código fonte). Por isso, nossos **.class** são colocados nessa pasta dentro do projeto. Repare que o WTP compila nossas classes na pasta **build** e depois *automaticamente* copia as coisas para o **WEB-INF/classes**.

Note que a pasta **WEB-INF** é muito importante e contém recursos vitais para o funcionamento do projeto. Imagine se o usuário tiver acesso a essa pasta! Códigos compilados (facilmente descompiláveis), bibliotecas potencialmente sigilosas, arquivos de configuração internos etc.

Para que isso não aconteça, a pasta **WEB-INF** com esse nome especial é uma **pasta invisível ao usuário final**. Isso quer dizer que se alguém acessar a URL <http://localhost:8080/jspteste/WEB-INF> verá apenas uma página de erro (404).

### Resumo final das pastas

- **src** - código fonte Java (.java)
- **build** - onde o WTP compila as coisas (.class)
- **WebContent** - content directory (páginas, imagens, css etc vão aqui)
- **WebContent/WEB-INF/** - pasta oculta com configurações e recursos do projeto
- **WebContent/WEB-INF/lib/** - bibliotecas .jar
- **WebContent/WEB-INF/classes/** - arquivos compilados são copiados para cá

#### META-INF

A pasta META-INF é opcional mas é gerada pelo WTP. É onde fica o arquivo de manifesto como usado em arquivos *.jar*.

## 6.3 - Exercícios: primeira página

Vamos testar nossas configurações criando um arquivo HTML de teste.

- 1) Crie o arquivo **WebContent/index.html** com o seguinte conteúdo:

```
<html>
    <h1>Novo projeto jspteste</h1>
</html>
```

- 2) Inicie (ou reinicie) o Tomcat clicando no botão de play verde na aba Servers.

- 3) Acesse no navegador: <http://localhost:8080/jspteste/index.html>

Teste também a configuração do welcome-file: <http://localhost:8080/jspteste/>

## 6.4 - Para saber mais: configurando o Tomcat sem o plugin

Se fosse o caso de criar uma aplicação web sem utilizar o plugin do tomcat deveríamos criar um arquivo de extensão xml com o nome de sua aplicação no diretório **tomcat/conf/Catalina/localhost**.

Para isso teríamos que configurar a url **/jspteste** para o diretório **/home/usuario/workspace/jspteste/web/**. Queremos também permitir que o Tomcat faça o *restart* de sua aplicação sempre que julgar necessário.

- 1) Abra os seus diretórios;
- 2) Vá para o diretório **tomcat**;
- 3) Escolha o diretório **conf/Catalina/localhost**;
- 4) Crie um arquivo chamado **jspteste.xml**;
- 5) Escreva o código a seguir no seu arquivo:

```
<Context path="/jspteste"
        docBase="/home/usuario/workspace/jspteste/web/" reloadable="true" />
```

**Importante!** Não esqueça de trocar a palavra “usuario” pelo nome do seu usuário.

### O arquivo xml de configuração do Tomcat

Em processos de *build* mais desenvolvidos, não existe configuração a ser feita nem mesmo na máquina de desenvolvimento, sendo tudo automatizado por processos de *build* e *deploy*.

# JSP - JavaServer Pages

*“O maior prazer é esperar pelo prazer.”*

– Gotthold Lessing

Nesse capítulo, você aprenderá:

- O que é JSP;
- Suas vantagens e desvantagens.
- Escrever arquivos JSP com scriptlets;
- Usar Expression Language;
- O que são taglibs.

## 7.1 - O que é uma página JSP

O primeiro arquivo JSP que vamos criar é chamado **bemvindo.jsp**. Esse arquivo poderia conter simplesmente código HTML, como o código a seguir:

```
<html>Bem vindo</html>
```

Afinal, **JSP é uma página HTML comum que contém também código Java** – e possui extensão `jsp`, claro.

Assim, fica claro que uma página JSP nada mais é que um arquivo baseado em HTML. Sejamos elegantes a ponto de escrever um código Java na nossa primeira página. Que tal declarar uma variável do tipo `String`:

```
<%  
String mensagem = "Bem vindo!";  
%>
```

Simples! Para escrever código Java na sua página basta escrevê-lo entre as tags `<%` e `%>`. Esse código é chamado de **scriptlet**.

Essa idéia de colocar código de uma linguagem de programação junto com HTML não é tão nova. A Netscape possuía o **SSJS** (Server-Side Javascript) por exemplo, usando código baseado em JavaScript.

### Scriptlet

Scriptlet é o código escrito entre `<%` e `%>`. Esse nome é composto da palavra *script* (linguagem de script) com o sufixo *let*, que indica algo pequeno.

A Sun possui essa mania de colocar o sufixo *let* em muitas coisas como os *scriptlets*, *servlets*, *portlets*, *midlets*, *applets* etc...

Podemos avançar mais um pouco e utilizar uma das variáveis já implícitas no JSP: todo arquivo JSP já possui uma variável chamada `out` (do tipo `JspWriter`) que permite imprimir objetos através do método `println`:

```
<% out.println(nome); %>
```

A variável `out` é um objeto ímplicito na nossa página JSP e existem outras de acordo com a especificação.

Existem ainda outras possibilidades para imprimir o conteúdo da nossa variável: podemos utilizar um atalho (muito parecido, ou igual, a outras linguagens de *script* para a web):

```
<%= nome %><br>
```

Agora, já estamos aptos a escrever o nosso primeiro JSP.

#### Comentários

Os comentários em uma página JSP devem ser feitos como o exemplo a seguir:

```
<%-- comentário em jsp --%>
```

## 7.2 - Exercícios: Primeiro JSP

- 1) Crie o arquivo **WebContent/bemvindo.jsp** com o seguinte conteúdo:

```
<html>

<%-- comentário em jsp aqui: nossa primeira página jsp --%>

<%
String mensagem = "Bem vindo!";
%>

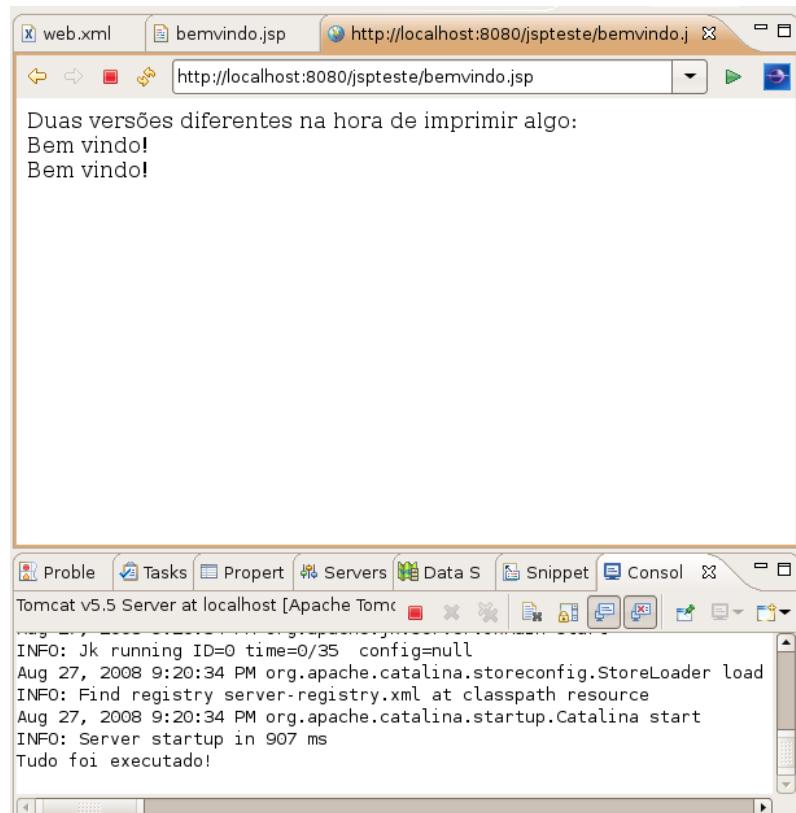
Duas maneiras diferentes na hora de imprimir algo:<br>

<% out.println(mensagem); %><br>
<%= mensagem %><br>

<%
System.out.println("Tudo foi executado!");
%>

</html>
```

- 2) Clique com o botão direito no projeto e então em Run As -> Run On Server.
- 3) Clique em **Finish** e então um Web Browser irá aparecer no seu Eclipse.
- 4) Teste a URL `http://localhost:8080/jspteste/bemvindo.jsp`



- 5) Onde apareceu a mensagem "Tudo foi executado!"?

Lembre-se que o código java é interpretado no servidor, portanto apareceu no console do seu Tomcat.

- 6) Se você possuísse um arquivo chamado database.properties, em que diretório você o colocaria?  
7) Em qual diretório você deve compilar suas classes? Por que esse diretório está dentro do diretório WEB-INF?

### 7.3 - Listando os contatos

Uma vez que podemos escrever qualquer código Java como scriptlet, não fica difícil criar uma listagem de todos os contatos do banco de dados.

Temos todas as ferramentas necessárias para fazer tal listagem uma vez que já fizemos isso no capítulo de JDBC.

Basicamente, o código utilizará o ContatoDAO que criamos anteriormente para imprimir a lista de Contato:

```

<%
ContatoDAO dao = new ContatoDAO();
List<Contato> contatos = dao.getLista();

for (Contato contato : contatos ) {
%>
<li><%=contato.getNome()%>, <%=contato.getEmail()%>:
<%=contato.getEndereco()%></li>

<%

```

```
}
```

```
%>
```

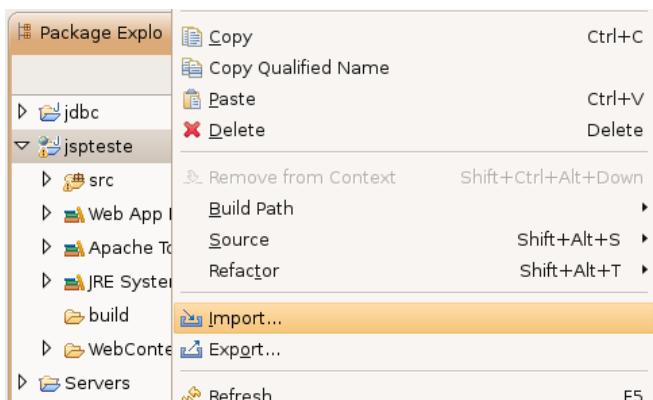
Ainda falta importar as classes dos pacotes corretos.

No JSP, usamos a tag `<%@ page import="" %>` para importar aquilo que será usado no nosso código scriptlet. O atributo `import` permite que seja especificado qual o pacote a ser importado. Esse atributo é o único que pode aparecer várias vezes. Nesse caso, importaremos diversos pacotes separados por vírgulas.

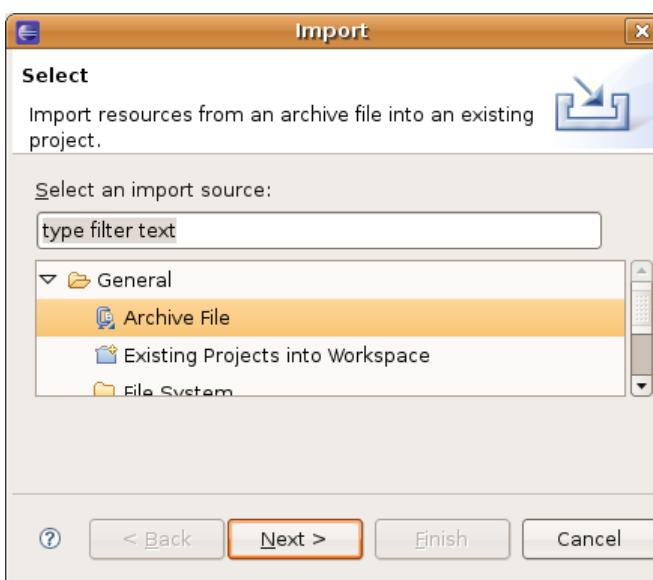
## 7.4 - Exercícios: Lista de contatos com scriptlet

1) Precisamos das classes do projeto anterior de JDBC para prosseguir. Para facilitar, vamos importar um arquivo ZIP que já tem tudo preparado corretamente.

a) No Eclipse, clique com o botão direito em cima do nome do projeto **jspteste** e vá em **Import**:



b) Selecione a opção **General, Archive File** e clique Next:



c) Clique em **Browse**, selecione o arquivo **caelum/21/projeto-jspteste.zip** dentro do seu **Desktop** e clique em **Finish**

Analise o resultado final. Veja que as classes do projeto anterior agora estão dentro do Eclipse. Repare também que há três JARs dentro de *WebApp Libraries* (é aí que o WTP mostra os JARs do WEB-INF/lib). Um dos JARs é o driver do MySQL que já conhecemos. Os outros dois veremos logo adiante.

- 2) Crie o arquivo **WebContent/lista-scriptlet.jsp** e siga:

- a) Importe os pacotes necessários. Use o Ctrl+Espaço do WTP para ajudar a escrever os pacotes.

```
<%@ page import="java.util.* , br.com.caelum.jdbc.* ,  
br.com.caelum.jdbc.dao.* , br.com.caelum.jdbc.modelo.*" %>
```

- b) Coloque o código para fazer a listagem. Use bastante o Ctrl+Espaço do WTP.

```
<html><ul>  
  
<%  
ContatoDAO dao = new ContatoDAO();  
List<Contato> contatos = dao.getLista();  
  
for (Contato contato : contatos) {  
%>  
  
<li><%=contato.getNome()%> , <%=contato.getEmail()%>:  
<%=contato.getEndereco()%></li>  
  
<%  
}  
%>  
  
</ul></html>
```

- c) Teste a url <http://localhost:8080/jsp teste/lista-scriptlet.jsp>

#### Para saber mais: new JSP

No WTP, você pode criar novos arquivos JSP mais facilmente. Basta clicar na pasta WebContent com o botão direito e então ir em New -> JSP. Um arquivo JSP vai ser gerado a partir de um template, com o esqueleto html pronto e algumas configurações básicas.

## 7.5 - HTML e Java: eu não quero código Java no meu JSP!

É complicado ficar escrevendo Java em seu arquivo JSP, não é?

Primeiro, fica tudo mal escrito e difícil de ler. O Java passa a atrapalhar o código HTML em vez de ajudar.

Depois, quando o responsável pelo design gráfico da página quiser alterar algo, terá que conhecer Java para entender o que está escrito lá dentro. Hmm... não parece uma boa solução.

Uma idéia boa é o MVC, que será visto mais adiante neste curso.

## 7.6 - EL: Expression language

Para remover um pouco do código Java que fica na página JSP, a Sun desenvolveu uma linguagem chamada **Expression Language** que é interpretada pelo servlet contêiner.

Nosso primeiro exemplo com essa linguagem é utilizá-la para mostrar parâmetros que o cliente envia através de sua requisição.

Por exemplo, se o cliente chama a página **testaparam.jsp?idade=24**, o programa deve mostrar a mensagem que o cliente tem 24 anos.

Como fazer isso? Simples, existe uma variável chamada **param** que, na expression language, é responsável pelos parâmetros enviados pelo cliente. Para ler o parâmetro chamado **idade** basta usar  **\${param.idade}**. Para ler o parâmetro chamado dia devemos usar  **\${param.dia}**.

## 7.7 - Exercícios: parâmetros com EL

- 1) Crie uma página chamada **WebContent/testaidade.jsp** com o conteúdo:

```
<html>
    Digite sua idade e pressione o botão:<br/>

    <form action="testaparametro.jsp">
        Idade: <input name="idade"/> <input type="submit"/>
    </form>
</html>
```

- 2) Crie um arquivo chamado **WebContent/testaparametro.jsp** e coloque o código de expression language que mostra a idade que foi enviada como parâmetro para essa página:

```
<html>
    Testando seus parametros:<br/>
    A idade é ${param.idade}.
</html>
```

- 3) Teste o sistema acessando a página <http://localhost:8080/jspteste/testaidade.jsp>.



## 7.8 - Exercícios opcionais

- 1) Tente utilizar o quadro a seguir para definir a página padrão de seu site.

### welcome-file-list

O arquivo web.xml abaixo diz que os arquivos chamados “bemvindo.jsp” devem ser chamados quando um cliente tenta acessar um diretório web qualquer.

O valor desse campo costuma ser “index.html” em outras linguagens de programação.

Como você pode ver pelo arquivo gerado automaticamente pelo WTP, é possível indicar mais de um arquivo para ser o seu welcome-file! Mude-o para:

```
<welcome-file-list>
    <welcome-file>bemvindo.jsp</welcome-file>
</welcome-file-list>
Reinic peace o tomcat e acesse a URL: http://localhost:8080/jspteste/
```

## 7.9 - Erros comuns

- Depois que você digitou a idade no formulário você obteve a página abaixo?



Verifique no seu arquivo testaidade.jsp no action dentro do formulário se o nome da página para a qual deveria ser redirecionada está digitado corretamente.

- A idade não apareceu e sua tela ficou assim?



Verifique se no seu arquivo testaparametro.jsp o parâmetro `idade` está digitado corretamente.

## 7.10 - Exercício opcional

1) Crie outro exemplo com dois campos em vez de um só. Mostre os dois parâmetros na segunda página.

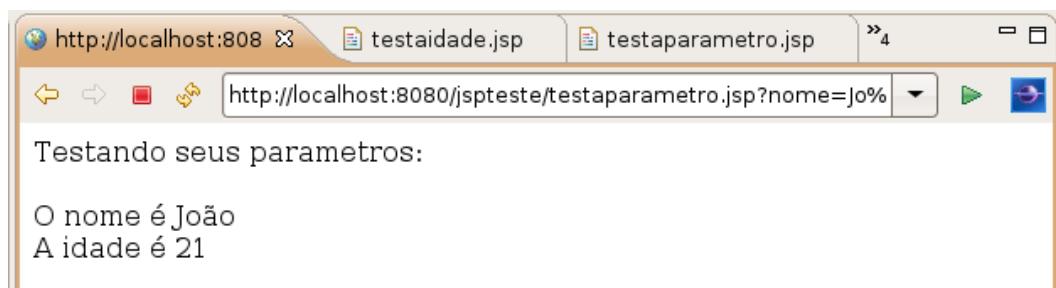
- Exemplo com dois campos:

```
<html>
    Digite sua idade e pressione o botão:<br/>
    <form action="testaparametro.jsp">
        Nome: <input name="nome"/> <br/>
        Idade: <input name="idade"/> <input type="submit"/>
    </form>
</html>
```

- Mostrando os dois parâmetros:

```
<html>
    Testando seus parametros:<br/><br/>
    O nome é ${param.nome} <br/>
    A idade é ${param.idade}
</html>
```

- Exemplo de um resultado final:



## 7.11 - Instanciando POJOs

Como já foi comentado anteriormente, os Javabeans devem possuir o construtor público sem argumentos (um típico Plain Old Java Object: POJO), getters e setters.

Se desejarmos instanciar um objeto desse tipo em nossa página JSP podemos fazer uso de uma tag simples.

Isso mesmo! Uma **tag**. A Sun percebeu que os programadores estavam abusando do código Java no JSP e tentou criar algo mais “natural” (um ponto um tanto quanto questionável da maneira que foi apresentada no início), sugerindo o uso de tags para substituir trechos de código.

O resultado final é um conjunto de tags (uma **tag library**, ou **taglib**) padrão, que possui, entre outras tags, a funcionalidade de instanciar objetos através do construtor sem argumentos.

Isso não é tão difícil. Dê uma olhada na tag a seguir:

```
<jsp:useBean id="contato" class="br.com.caelum.jdbc.modelo.Contato"/>
```

Agora, podemos imprimir o nome do contato (que está em branco, claro...):

---

```
 ${contato.nome}
```

Mas, onde está o `getNome()`? A expression language é capaz de perceber sozinha a necessidade de chamar um método do tipo *getter*, por isso o padrão getter/setter do pojo é tão importante hoje em dia.

Desta maneira, classes como `Contato` são ferramentas poderosas por seguir esse padrão pois diversas bibliotecas importantes estão baseadas nele: Hibernate, Struts, VRaptor, JXPath, EJB etc.

**Atenção**

Na Expression Language  `${contato.nome}` chamará o método `getNome` por padrão. Para que isso sempre funcione, devemos colocar o parâmetro em letra minúscula. Ou seja,  `${contato.Nome}` não funciona.

---

## 7.12 - Para saber mais: Compilando os arquivos JSP

Os arquivos JSPs não são compilados dentro do Eclipse, por esse motivo na hora que estamos escrevendo o JSP no Eclipse não precisamos das classes do driver.

Os JSPs são transformados em uma servlet, que veremos adiante, por um compilador JSP (o Tomcat contém um compilador embutido). Esse compilador JSP pode gerar uma código Java que é então compilado para gerar bytecode diretamente para a servlet.

Então, somente durante a execução de uma página JSP, quando ele é transformado em uma servlet, que seu código Java é compilado e necessitamos das classes do driver que são procuradas no diretório lib.

# JSTL - JavaServer Pages Tag Library

*“Saber é compreendermos as coisas que mais nos convém.”*  
– Friedrich Nietzsche

Nesse capítulo, você aprenderá o que é a JSTL e terá a chance de utilizar diversas das principais tags do grupo chamado core.

## 8.1 - JSTL

Seguindo a idéia de melhorar o código Java que precisa de uma maneira ou outra ser escrito na página JSP, a Sun sugeriu o uso da **JavaServer Pages Standard Tag Library**, a **JSTL**.

### Observação

Antes de 2005, JSTL significava *JavaServer Pages Standard Template Library*.

A **JSTL** é a API que encapsulou em tags simples toda a funcionalidade que diversas páginas web precisam, como controle de laços (`fors`), controle de fluxo do tipo `if else`, manipulação de dados xml e a internacionalização de sua aplicação.

Antigamente, diversas bibliotecas foram criadas por vários grupos com funcionalidades similares ao JSTL (principalmente ao Core), culminando com a aparição da mesma, numa tentativa da Sun de padronizar algo que o mercado vê como útil.

Existem ainda outras partes da JSTL, por exemplo aquela que acessa banco de dados e permite escrever códigos SQL na nossa página, mas se o designer não comprehende Java o que diremos de SQL??? O uso de tal parte da JSTL é desencorajado exceto em casos muito especiais.

A JSTL foi a forma encontrada de padronizar o trabalho de milhares de programadores de páginas JSP.

Antes disso, muita gente programava como nos exemplos que vimos anteriormente, somente com JSPs e Javabeans, o chamado Modelo 1, que na época fazia parte dos Blueprints de J2EE da Sun (boas práticas).

## 8.2 - As empresas hoje em dia

Muitas páginas JSP no Brasil ainda possuem grandes pedaços de scriptlets espalhados dentro delas.

Recomendamos a todos os nossos alunos que optarem pelo JSP como camada de visualização, que utilizem a JSTL e outras bibliotecas de tag para evitar o código incompreensível que pode ser gerado com scriptlets.

O código das scriptlets mais confunde do que ajuda, tornando a manutenção da página JSP cada vez mais custosa para o programador e para a empresa.

## 8.3 - Instalação

Para instalar a implementação mais famosa da **JSTL** basta baixar a mesma no site <https://jstl.dev.java.net/>.

Você encontrará lá os arquivos **.jar** que estão no diretório lib do seu projeto. Eles são a implementação padrão da JSTL feita pela própria Sun através do GlassFish.

Ao usar o JSTL em alguma página precisamos primeiro definir o cabeçalho. Existem quatro APIs básicas e iremos aprender primeiro a utilizar a biblioteca chamada de **core**.

## 8.4 - Cabeçalho para a JSTL core

Sempre que vamos utilizar uma taglib devemos primeiro escrever um cabeçalho através de uma tag JSP que define qual taglib iremos utilizar e um nome, chamado *prefixo*.

Esse prefixo pode ter qualquer valor mas no caso da taglib core da JSTL o padrão da Sun é a letra **c**. Já a URI (que não deve ser decorada) é mostrada a seguir e não implica em uma requisição pelo protocolo http e sim uma busca entre os arquivos .jar no diretório lib.

Ao descompactarmos o arquivo projeto-jspteste.zip no capítulo anterior, colocamos também tais arquivos .jar, de tal modo que podemos incluir a taglib core como no exemplo a seguir:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

## 8.5 - For

Usando a JSTL core, vamos reescrever o arquivo que lista todos contatos.

O cabeçalho já é conhecido da seção anterior:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

```
<html>
```

Depois, precisamos instanciar e declarar nosso DAO. Ao revisar o exemplo da lista através de scriptlets, queremos executar o seguinte:

- classe: br.com.caelum.jdbc.dao.ContatoDAO;
- construtor: sem argumentos;
- variável: DAO.

Já vimos a tag **jsp:useBean**, capaz de instanciar determinada classe através do construtor sem argumentos e dar um nome (id) para essa variável. Na realidade essa tag faz muito mais do que isso, mas para nossos exemplos que serão descartados mais pra frente quando aprendermos o MVC isso é mais que suficiente.

Portanto vamos utilizar a tag **useBean** para instanciar nosso ContatoDAO:

```
<jsp:useBean id="dao" class="br.com.caelum.jdbc.dao.ContatoDAO"/>
```

A partir deste momento, temos a variável **dao** no escopo de página, esse é o escopo chamado **pageContext**, onde os beans ficam armazenados. Podemos mostrar o nome do primeiro contato usando a JSTL core. Para isso usaremos o prefixo configurado no cabeçalho: **c**.

```
<c:out value="${dao.lista[0].nome}" />
```

Ou o seu e-mail:

```
<c:out value="${dao.lista[0].email}" />
```

Muito complexo? Com o tempo fica, felizmente, mais legível. No primeiro exemplo, é chamado o método `getLista`, o primeiro item, e então o método `getNome`. O resultado é enviado para a variável `out`.

Que tal? Ainda não é tão elegante quanto queríamos, certo? O código dentro do atributo `value` é chamado de **Expression Language (EL)**, e é parte de uma linguagem que utilizaremos durante esse curso.

Agora que temos a variável `dao` na “mão” desejamos chamar o método `getList`a e podemos fazer isso através da EL:

```
 ${dao.lista}
```

E agora desejamos executar um loop para cada contato dentro da coleção retornada por esse método:

- array ou coleção: `dao.lista`;
- variável temporária: `contato`.

No nosso exemplo com scriptlets, o que falta é a chamada do método `getList`a e a iteração:

```
<%
// ...
List<Contato> contatos = dao.getList();
for (Contato contato : contatos) {
%>
<li><%=contato.getNome()%>, <%=contato.getEmail()%>:
<%=contato.getEndereco()%></li>

<%
}
%>
```

A JSTL core disponibiliza uma tag chamada `c:forEach` capaz de iterar por uma coleção, exatamente o que precisamos. O exemplo a seguir mostra o uso de *expression language* de uma maneira muito mais elegante:

```
<c:forEach var="contato" items="${dao.lista}">
<li>${contato.nome}, ${contato.email}: ${contato.endereco}</li>
</c:forEach>
```

Mais elegante que o código que foi apresentado usando scriptlets, não?

**forEach e varStatus**

É possível criar um contador do tipo `int` dentro do seu laço `forEach`. Para isso basta definir o atributo chamado `varStatus` para a variável desejada e utilizar a propriedade `count` dessa variável.

```
<table border="1">
    <c:forEach var="contato" items="${dao.lista}" varStatus="id">
        <tr bgcolor="#${id.count % 2 == 0 ? 'aaee88' : 'ffffff'}">
            <td>${id.count}</td><td>${contato.nome}</td>
        </tr>
    </c:forEach>
</table>
```

## 8.6 - Exercícios: forEach

1) Liste os contatos de ContatoDAO usando `jsp:useBean` e JSTL.

a) Crie o arquivo **WebContent/lista-elegante.jsp** com o conteúdo que vimos:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>

<!-- cria a lista -->
<jsp:useBean id="dao" class="br.com.caelum.jdbc.dao.ContatoDAO"/>

<!-- for -->
<c:forEach var="contato" items="${dao.lista}">
    <li>

        nome: ${contato.nome},
        email ${contato.email},
        endereço ${contato.endereco}

    </li>
</c:forEach>
</html>
```

b) Acesse <http://localhost:8080/jspteste/lista-elegante.jsp>



**Repare que após criar uma nova página JSP não precisamos reiniciar o nosso container!**

2) Scriptlets ou JSTL? Qual dos dois é mais fácil para o designer entender?

## 8.7 - Exercício opcional

1) Utilize uma variável de status no seu c:forEach para colocar duas cores diferentes em linhas pares e ímpares.

## 8.8 - c:out e c:set

Tente substituir \${contato.nome} por <c:out value="\${contato.nome}" />. Qual a diferença?

A tag c:out aceita também um atributo chamado default, que indica o valor padrão caso o valor mencionado seja null (por padrão vazio). Seria impossível fazer isso somente com a *expression language* (sem nenhuma gambiarra).

A tag c:set permite armazenar o resultado da expressão contida no atributo value em outra variável, para algum tipo de manipulação futura.

Teste, por exemplo:

```
<c:set var="nome" value="${contato.nome}" />
<c:out value="${nome}" />
```

Como você pode perceber, é muito simples aprender a utilizar uma taglib, basta ler o que ela faz, passar os argumentos corretos, e pronto.

Sugerimos a leitura completa da especificação da JSTL no site da Sun:

<http://java.sun.com/products/jsp/jstl/> <http://java.sun.com/products/jsp/jstl/1.1/docs/tlddocs/index.html>

Não precisa decorar tudo, basta ler por cima e saber o que existe e o que não existe.

Quando surgir a necessidade do uso de uma dessas tags você terá ela disponível em suas mãos.

## 8.9 - Mas quais são as tags da taglib core?

A lista completa das tags da versão 1.2 da JSTL core pode ser facilmente compreendida:

- **c:catch** - bloco do tipo try/catch
- **c:choose** - bloco do tipo switch
- **c:forEach** - for
- **c:forTokens** - for em tokens (ex: "a,b,c" separados por vírgula)
- **c:if** - if
- **c:import** - import
- **c:otherwise** - default do switch
- **c:out** - saída
- **c:param** - parâmetro

- **c:redirect** - redirecionamento
- **c:remove** - remoção de variável
- **c:set** - criação de variável
- **c:url** - veja adiante
- **c:when** - teste para o switch

## 8.10 - Import: trabalhando com cabeçalhos e rodapés

Uma pergunta que sempre aparece na vida dos programadores é a de como executar o código de outro arquivo jsp dentro de um primeiro arquivo jsp, isto é, você quer colocar um cabeçalho? Um rodapé?

Existe uma tag da JSTL core que faz isso para você:

```
<c:import url="outrapagina.jsp"/>
```

## 8.11 - Exercícios

1) Crie uma página chamada **WebContent/jstl-import.jsp**

a) Defina a JSTL core:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

b) Importe cabecalho.jsp:

```
<c:import url="cabecalho.jsp"/>
```

c) Escreva alguma mensagem de texto:

d) Importe rodape.jsp:

```
<c:import url="rodape.jsp"/>
```

2) Crie a página **WebContent/cabecalho.jsp** e escreva:

```
<html>
  <head><h2>Aplicacao web basica</h2><br/></head>
```

3) Crie a página **WebContent/rodape.jsp** e escreva:

```
<br/><hr/>Copyright Caelum
</html>
```

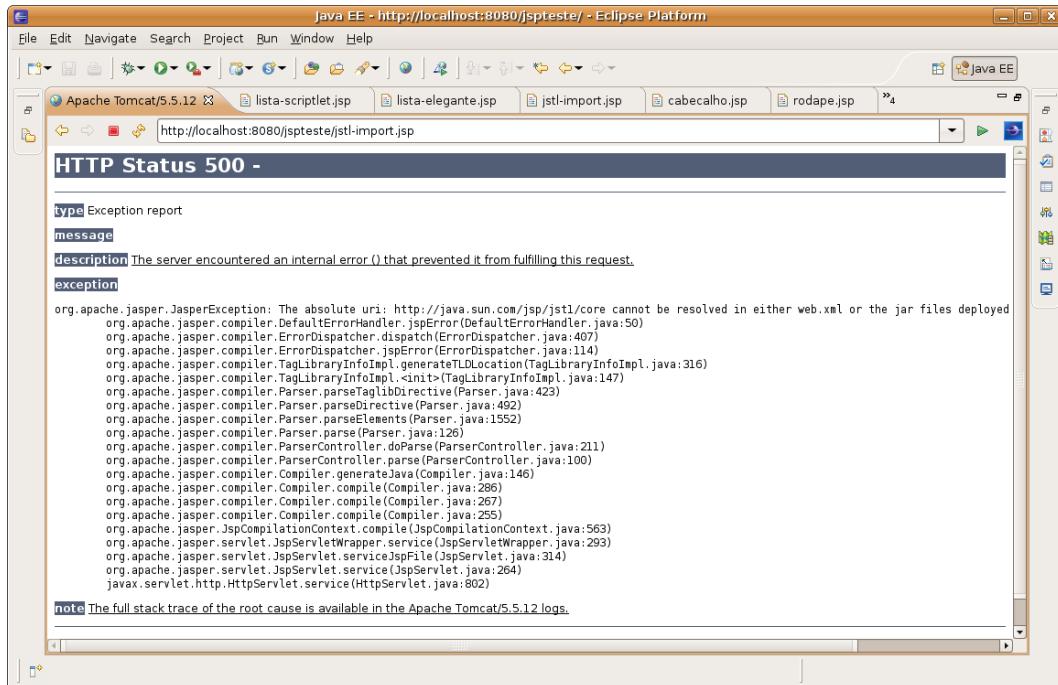
4) Teste no browser abrindo o endereço: <http://localhost:8080/jspteste/jstl-import.jsp>



A inclusão feita nesse exercício é dinâmica, ou seja, é feita uma requisição para a página incluída a cada acesso e o resultado é adicionado na página atual.

## 8.12 - Erros Comuns

Você obteve essa página ao invés da de cima?



Verifique se a sua taglib URI está digitada corretamente.

## 8.13 - Inclusão estática de arquivos

Existe uma maneira em um arquivo JSP de incluir um outro arquivo estaticamente. Isto faz com que o arquivo a ser incluído seja literalmente copiado e colado dentro do seu arquivo antes da primeira interpretação (compilação) do seu jsp.

A vantagem é que como a inclusão é feita uma única vez antes do arquivo ser compilado, essa inclusão é extremamente rápida, porém vale lembrar que o arquivo incluído pode ou não funcionar separadamente.

```
<%@ include file="outra_pagina.jsp" %>
```

## 8.14 - Exercícios

- 1) Crie uma página chamada **WebContent/titulo.jsp**. Esse arquivo irá mostrar o conteúdo da variável `titulo`:

```
<h1><%= titulo %></h1>
```

- 2) Crie uma página chamada **WebContent/testa-titulo.jsp**. Esse JSP vai definir uma variável chamada `titulo` e incluir o arquivo `titulo` estaticamente:

```
<html>
<%
String titulo = "Teste de titulo";
%>

<%@ include file="titulo.jsp" %>
<br/>
<font color="blue">Conteúdo da sua página aqui...</font>

</html>
```

- 3) Teste a url `http://localhost:8080/jspteste/testa-titulo.jsp`



- 4) Teste a url `http://localhost:8080/jspteste/titulo.jsp`.

Por que ela não funciona? Talvez não fizesse sentido esse arquivo `titulo.jsp` ficar no diretório web. Que tal tentar movê-lo para o diretório WEB-INF?

## 8.15 - Exercícios opcionais

- 1) Tente utilizar a tag `c:import` para importar a página `titulo`, qual o resultado?

```
<%
String titulo = "Teste de c:import";
%>

<c:import url="titulo.jsp"/>
```

- 2) Altere seu arquivo `lista-scriptlet.jsp` e inclua a página `titulo.jsp`:

```
<%
```

```
String titulo = "Lista de contatos via scriptlets";
%>
```

```
<%@ include file="titulo.jsp" %>
```



## 8.16 - Trabalhando com links

Às vezes não é simples trabalhar com links pois temos que pensar na URL que o cliente acessa ao visualizar a nossa página.

A JSTL resolve esse problema: supondo que a sua aplicação se chame jspteste, o código abaixo gera a string /jspteste/imagem/banner.jpg.

```
<c:url value="/imagem/banner.jpg"/>
```

## 8.17 - Exercícios opcionais

1) Crie uma página chamada **WebContent/jstl-url.jsp**.

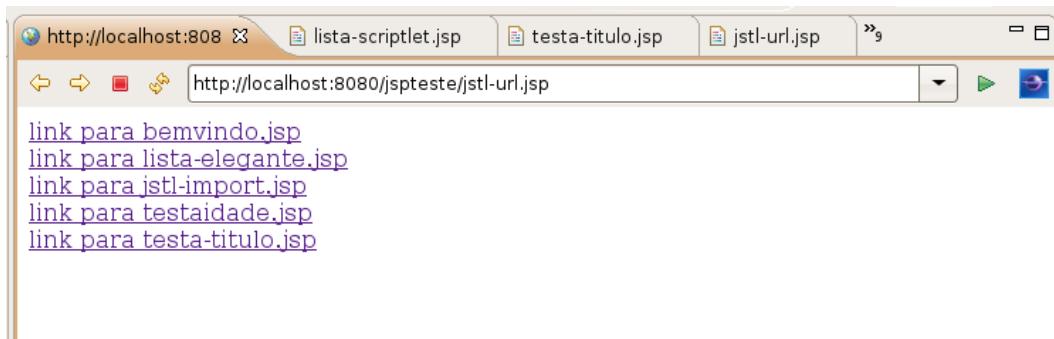
a) Defina a JSTL core:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

b) Crie um link utilizando a tag `c:url` para todas as páginas JSP que você já criou, por exemplo:

```
<a href=<c:url value="/bemvindo.jsp"/>>link para bemvindo.jsp</a>
<a href=<c:url value="/lista-elegante.jsp"/>>link para lista-elegante.jsp</a>
<a href=<c:url value="/jstl-import.jsp"/>>link para jstl-import.jsp</a>
<a href=<c:url value="/testaidade.jsp"/>>link para testaidade.jsp</a>
<a href=<c:url value="/testatitulo.jsp"/>>link para testatitulo.jsp</a>
```

2) Teste a sua página e veja o resultado (código fonte HTML).



## 8.18 - c:if

Ao usar a tag `c:if` é possível construir expressões condicionais simples. Por exemplo:

```
<c:if test="${empty param.nome}">
    Voce nao preencheu o campo nome.
</c:if>
```

A tag `c:if` tem uma condição e um pedaço de código. Caso a condição da tag for satisfeita o pedaço de código é executado.

No JSTL, **não existe** a tag `c:else` por questões estruturais do XML.

## 8.19 - Exercícios

- 1) Crie um arquivo `jsp` `WebContent/preenchenome.jsp`:

```
<html>
    Digite seu nome e pressione o botão:<br/>

    <form action="testapreencheu.jsp">
        Nome: <input name="nome"/> <input type="submit"/>
    </form>
</html>
```



- 2) Crie um arquivo `WebContent/testapreencheu.jsp`, ele vai checar se no formulário anterior a pessoa preencheu ou não o nome.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<c:if test="${empty param.nome}">
```

```
Voce nao preencheu o campo nome.  
</c:if>  
<c:if test="${not empty param.nome}">  
    Voce preencheu ${param.nome}.  
</c:if>  
</html>
```

- a) Exemplo caso a pessoa tenha preenchido:



- b) Caso a pessoa não tenha preenchido:



# Controle de erro

*“Divide as dificuldades que tenhas de examinar em tantas partes quantas for possível, para uma melhor solução.”*  
– René Descartes

Nesse capítulo, você aprenderá:

- Desvantagens na utilização de blocos try/catch em um arquivo jsp;
- Desvantagens na utilização da tag c:catch em um arquivo jsp;
- Vantagens no controle de erros através de configuração declarativa;
- Controle de erros através de exceptions;
- Controle de erros através de error-codes.

## 9.1 - Exceptions

O que acontece quando diversos pontos da nossa aplicação precisam tratar seus erros?

O que acontece quando um tipo de erro que ocorre em diversos pontos deve ser alterado? Devemos passar por todos os servlets para tratar isso? Por todos os arquivos jsp?

Uma idéia bem simples seria colocar em toda página jsp um código imenso do tipo try/catch como no exemplo a seguir do nosso já conhecido lista-scriptlet.jsp:

```
<%@ page import="java.util.* , br.com.caelum.jdbc.* , br.com.caelum.jdbc.dao.* ,  
br.com.caelum.jdbc.modelo.* , java.sql.*" %>  
<html>  
  <ul>  
    <%  
    try {  
      ContatoDAO dao = new ContatoDAO();  
      List<Contato> contatos = dao.getLista();  
  
      for (Contato contato : contatos) {  
        %>  
        <li><%=contato.getNome()%> , <%=contato.getEmail()%>:  
          <%=contato.getEndereco()%></li>  
    %>  
    }  
  %>
```

```
    } catch(SQLException ex) {
    %>
        Ocorreu algum erro ao acessar o banco de dados.
    <%
    }
%>
</ul>
</html>
```

Basta olhar o código acima para perceber que não é o melhor caminho. Imagina como seria tratar os erros dessa maneira em **toda** a sua aplicação? E se a mensagem de erro mudasse? Teríamos que mudar **todas** as páginas?

## 9.2 - JSTL é a solução?

Poderíamos usar a tag `c:catch`, com o mesmo tipo de problema da solução anterior:

```
<c:catch var="error">
<jsp:useBean id="dao" class="br.com.caelum.jdbc.dao.ContatoDAO"/>
<c:forEach var="contato" items="${dao.lista}">
    <li>
        nome: ${contato.nome},
        email ${contato.email},
        endereço ${contato.endereco}
    </li>
</c:forEach>
</c:catch>
<c:if test="${not empty error}">
    Ocorreu algum erro ao acessar o banco de dados.
</c:if>
```

Repare que a própria JSTL nos apresenta uma solução que não se mostra boa para esse tipo de erro que queremos tratar. É importante deixar claro que desejamos tratar o tipo de erro que não tem volta, devemos mostrar uma mensagem de erro para o cliente e pronto, por exemplo quando a conexão com o banco cai ou quando ocorre algum erro no servidor.

Esses são erros do servidor (ou do programador) e não erros do cliente, como problemas de validação de campo, que trataremos no capítulo do Struts.

## 9.3 - Exercício opcional

- 1) Trate o erro da listagem de contatos com `c:catch` criando um arquivo chamado `testa-catch.jsp`.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
    <h1>Tratando erro com c:catch </h1>
    <br>
```

```
<c:catch var="error">
    <jsp:useBean id="dao" class="br.com.caelum.jdbc.dao.ContatoDAO"/>
    <c:forEach var="contato" items="${dao.lista}">
        <li>

            nome: ${contato.nome},
            email ${contato.email},
            endereço ${contato.endereco}

        </li>
    </c:forEach>
</c:catch>
<c:if test="${not empty error}">
    Ocorreu algum erro ao acessar o banco de dados.
</c:if>
</html>
```

## 9.4 - Quando acontece um erro em uma página JSP

É muito simples controlar erros em páginas JSP. Imagine a página a seguir que simula um erro de conexão ao banco de dados:

```
<%@ page errorPage="/erro.jsp" %>
<html>
<%
java.sql.DriverManager.getConnection("jdbc:teste:invalido","usuario","senha");
%>
</html>
```

Agora que já configuramos, quando ocorrer uma `SQLException`, a página `erro.jsp` será mostrada se nada for feito pelo programador: não precisamos fazer **nada** na página `jsp` que pode gerar um erro, exceto dizer qual a página que controla os erros dela. Simples, não?

Quem trata o erro é o servlet container, que lê o `jsp` e envia o fluxo da requisição para a página de erro indicada na tag `page`.

## 9.5 - Página de erro

Começamos criando uma página JSP chamada `erro.jsp` que utiliza uma diretiva para indicar que é uma página de controle de erro, isto é, quando erros ocorrem, o fluxo será redirecionado para essa página.

A diretiva a ser utilizada é a mesma que usamos para importar classes e pacotes. Nesse caso utilizamos o atributo `isErrorPage` para disponibilizar o erro (`exception`) que ocorreu para o nosso código `jsp`.

```
<%@ page isErrorPage="true"%>
```

Agora podemos mostrar a mensagem de erro usando EL. Lembre-se que a variável a seguir só será criada se o erro for uma exception!

```
 ${pageContext.errorData.throwable}
```

Portanto a página chamada `erro.jsp` acaba sendo o cabeçalho com a mensagem de erro:

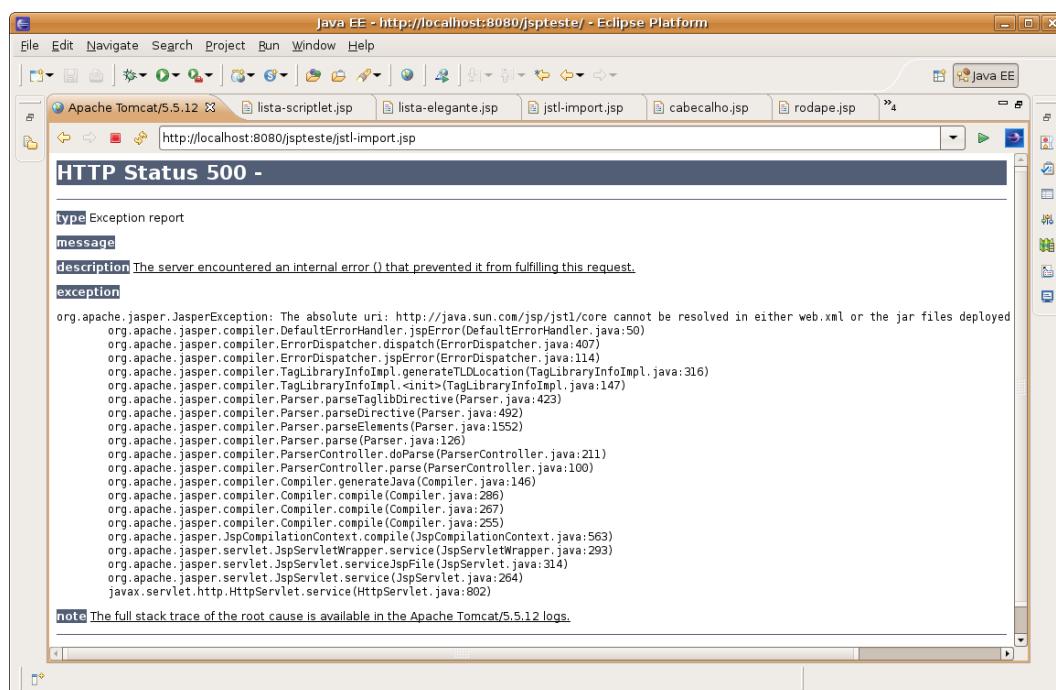
```
<%@ page isErrorPage="true" %>
<html>
    Um erro ocorreu.<br/>
    ${pageContext.errorData.throwable}
</html>
```

## 9.6 - Exercícios

- 1) Crie o arquivo **WebContent/testa-erro.jsp**:

```
<html>
<%
java.sql.DriverManager.getConnection("jdbc:teste:invalido");
%>
</html>
```

- a) Teste a url <http://localhost:8080/jsp teste/testa-erro.jsp>



- b) Altere sua página, adicionando o cabeçalho:

```
<%@ page isErrorPage="/erro.jsp" %>
```

- 2) Crie o arquivo **WebContent/erro.jsp**.

```
<%@ page isErrorPage="true" %>
<html>
    <h1>Um erro ocorreu.</h1><br/>
    ${pageContext.errorData.throwable}
</html>
```

3) Teste a url <http://localhost:8080/jspteste/testa-erro.jsp>



## 9.7 - Erros comuns

No exercício anterior, é muito comum errar no seguinte detalhe: errar na página de erro (como o texto da Expression Language), e visualizar a página de erro normal do Tomcat.

# Servlets

*“Vivemos todos sob o mesmo céu, mas nem todos temos o mesmo horizonte.”*  
– Konrad Adenauer

Neste capítulo, você aprenderá a criar pequenos objetos que funcionam como aplicações web.

## 10.1 - Servlet

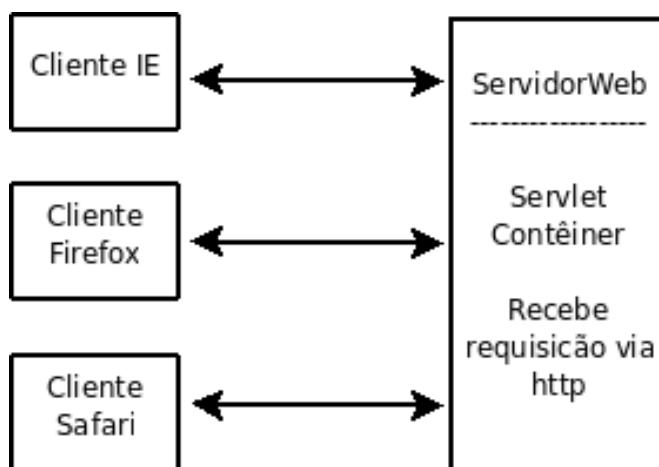
Uma **servlet** funciona como um pequeno servidor (servidorzinho em inglês) que recebe chamadas de diversos clientes.

Uma primeira idéia da servlet seria que cada uma delas é responsável por uma página, sendo que ela lê dados da requisição do cliente e responde com outros dados (html, gif etc). Como no Java tentamos sempre que possível trabalhar orientado a objetos, nada mais natural que uma servlet seja representada como um objeto.

Resumindo, cada servlet é um objeto java que recebe tais requisições (**request**) e retorna algo (**response**), como por exemplo uma página html ou uma imagem do formato jpg.

Diversas requisições podem ser feitas à uma mesma servlet ao mesmo tempo em um único servidor, por isso ela é mais rápida que um programa CGI comum. A especificação da servlet cita algumas vantagens que possui sobre o antigo CGI.

O diagrama abaixo mostra três clientes acessando o mesmo servidor web/container de servlets através do protocolo http.



A página a ser retornada pela servlet pode ser um jpg, um gif, um arquivo html etc: arquivos de texto ou simplesmente binários.

O comportamento das servlets que iremos ver neste capítulo foi definido na classe `HttpServlet` do pacote `javax.servlet`. Eles se aplicam às servlets que trabalham através do protocolo `Http`.

A interface `Servlet` é a que define exatamente como uma servlet funciona, mas não é necessariamente o que vamos utilizar neste capítulo uma vez que ela possibilita o uso de qualquer protocolo baseado em requisições e respostas.

É importante frisar que a mesma instância de uma servlet (o mesmo objeto) pode ser chamada mais de uma vez para diferentes requisições ao mesmo tempo, justamente para obter as vantagens mencionadas anteriormente contra o uso de `CGI`.

O funcionamento básico de uma servlet compreende:

- sua inicialização (veremos com mais detalhes mais adiante)
- chamadas a métodos de serviço, essas chamadas passam dois argumentos para o método `service`, a requisição que o cliente faz e a resposta que permite enviar dados para o mesmo:

```
void service(HttpServletRequest req, HttpServletResponse res);
```

- finalização (veremos com detalhes mais adiante),

O exemplo a seguir mostra uma servlet implementando o método de `service`.

Um primeiro exemplo de método `service` seria aquele que não executa nada e mostra uma mensagem de bem vindo para o usuário. Para isso precisamos “alterar” a resposta que a servlet enviará para o cliente.

O writer de saída do cliente pode ser obtido através do método `getWriter` da variável `response` e então fica simples utilizar um `PrintWriter` para imprimir algo como resposta para o cliente:

```
protected void service(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    // recebe o writer
    PrintWriter out = response.getWriter();

    // escreve o texto
    out.println("<html>");
    out.println("Caelum explica");
    out.println("</html>");
}
```

### Servlet x CGI

- Fica na memória entre requisições, não precisa ser reiniciada;
- O nível de segurança e permissão de acesso pode ser controlado;
- em `CGI`, cada cliente é representado por um processo, enquanto que com `Servlets`, cada cliente é representado por uma linha de execução.

Esse capítulo está focado na `HttpServlet`, um tipo que gera aplicações web baseadas no protocolo `HTTP`, mas vale lembrar que a api não foi criada somente para este protocolo, podendo ser facilmente estendida para outros.

## 10.2 - A estrutura de diretórios

A estrutura de diretórios é a mesma utilizada em uma aplicação que usava páginas JSP e, portanto, usaremos o mesmo projeto.

## 10.3 - Mapeando uma servlet no web.xml

Para fazer um mapeamento de uma URL específica para uma servlet é necessário usar o arquivo `web.xml`.

Uma vez que chamar a servlet pelo pacote e nome da classe acabaria criando URLs estranhas e complexas, é comum mapear, por exemplo, uma servlet como no exemplo, chamada `OiMundo` para o nome `servletDeTeste`:

Começamos com a definição da servlet em si, dentro da tag `<servlet>`:

```
<servlet>
    <servlet-name>servletDeTeste</servlet-name>
    <servlet-class>br.com.caelum.servlet.OiMundo</servlet-class>
</servlet>
```

Em seguida, mapeie nossa servlet para a URL `/oi`. Perceba que isso acontece dentro da tag `<servlet-mapping>` (mapeamento de servlets) e que você tem que indicar que está falando daquela servlet que definimos logo acima: passamos o mesmo `servlet-name` para o mapeamento.

```
<servlet-mapping>
    <servlet-name>servletDeTeste</servlet-name>
    <url-pattern>/oi</url-pattern>
</servlet-mapping>
```

Portanto, são necessários dois passos para mapear uma servlet para uma url:

- 1) Definir o nome e classe da servlet;
- 2) Usando o nome da servlet, definir a url;

Agora a servlet pode ser acessada através da seguinte URL:

`http://localhost:8080/jspteste/oi`

Assim que o arquivo `web.xml` e a classe de servlet de exemplo forem colocados nos diretórios corretos basta configurar o Tomcat para utilizar o diretório de base como padrão para uma aplicação web.

## 10.4 - Exercícios: Servlet OiMundo

- 1) Crie a servlet `OiMundo` no pacote `br.com.caelum.servlet`. Escolha o menu **File, New, Other, Class**.



a) Estenda HttpServlet:

```
public class OiMundo extends HttpServlet {  
}
```

b) Utilize o CTRL+SHIFT+O para importar HttpServlet.

c) Para escrever a estrutura do método service, dentro da classe, escreva apenas **service** e dê **Ctrl+espaço**: o Eclipse gera pra você o método.

```
package br.com.caelum.servlet;  
  
import javax.servlet.http.HttpServlet;  
  
public class OiMundo extends HttpServlet {  
  
    service  
    }  
    □ service(HttpServletRequest arg0, HttpServletResponse arg1)  
    ● service(ServletRequest arg0, ServletResponse arg1) : void
```

**Cuidado para escolher a versão de service que recebe HttpServletRequest/Response.**

A anotação @Override serve para notificar o compilador que estamos sobrescrevendo o método service da classe pai, se por algum acaso errarmos o nome do método ou trocarmos a ordem dos parâmetros, o compilador irá reclamar e você vai perceber o erro ainda em tempo de compilação.

O método gerado deve ser esse. **Troque os nomes dos parâmetros como abaixo.**

```
@Override  
protected void service(HttpServletRequest request, HttpServletResponse response)  
    throws ServletException, IOException {  
  
}
```

d) Escreva a implementação do método service **dentro** dele. Cuidado em tirar a chamada ao super.service antes e repare que a declaração do método já foi feita no passo anterior.

```

protected void service(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
    // recebe o writer
    PrintWriter out = response.getWriter();

    // escreve o texto
    out.println("<html>");
    out.println("Caelum explica");
    out.println("</html>");
}

```

- 2) Abra o arquivo **web.xml** e mapeie a URL **/oi** para a servlet **OiMundo**. Aproveite o auto-completar do WTP e cuidado ao escrever o nome da classe e do pacote.

```

<servlet>
    <servlet-name>servletDeTeste</servlet-name>
    <servlet-class>br.com.caelum.servlet.OiMundo</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>servletDeTeste</servlet-name>
    <url-pattern>/oi</url-pattern>
</servlet-mapping>

```

- 3) Reinicie o Tomcat clicando no botão verde na aba Servers.

- 4) Teste a url <http://localhost:8080/jspteste/oi>



## 10.5 - Erros comuns

Existem diversos erros comuns nos exercícios anteriores. Aqui vão alguns deles:

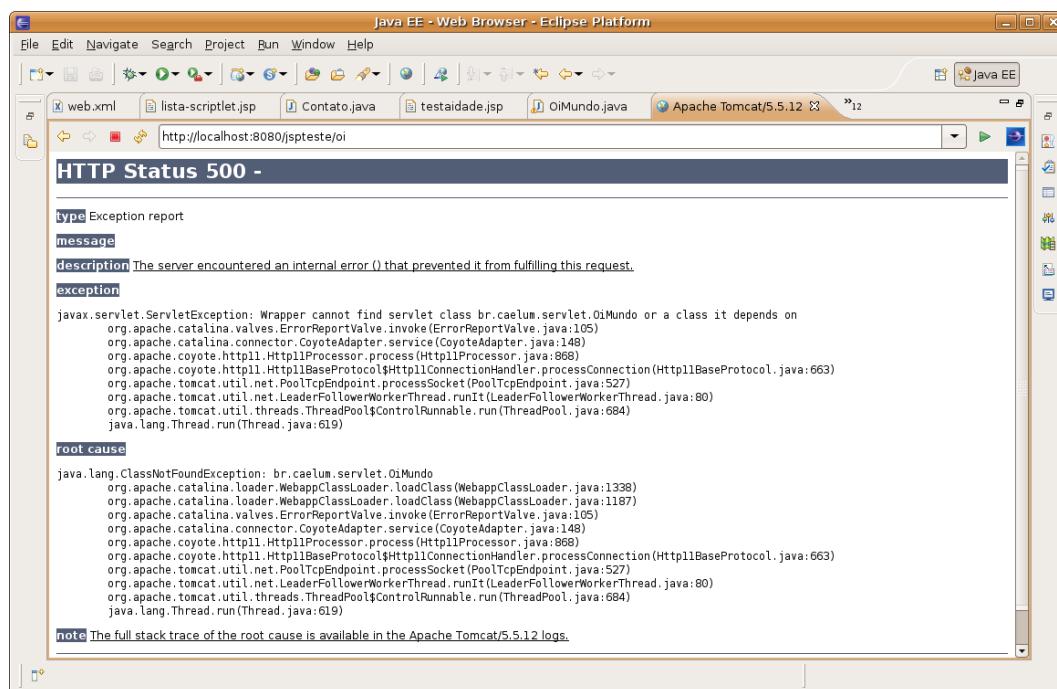
- 1) Esquecer da barra inicial no URL pattern:

```
<url-pattern>oi</url-pattern>
```



2) Digitar errado o nome do pacote da sua servlet:

```
<servlet-class>br.caelum.servlet.OiMundo</servlet-class>
```



3) Esquecer de colocar o nome da classe no mapeamento da servlet:

```
<servlet-class>br.com.caelum.servlet</servlet-class>
```

## 10.6 - Init e Destroy

Toda servlet deve possuir um construtor sem argumentos para que o container possa criá-la. O servlet container inicializa a servlet com o método `init(ServletConfig config)` e a usa durante todo o seu período ativo, até que irá desativá-la através do método `destroy()`, para então liberar o objeto.

Na inicialização de uma servlet, quando parâmetros podem ser lidos e variáveis comuns a todas as requisições devem ser inicializadas. Por exemplo, conexões ao banco de dados são estabelecidas nesse momento:

```
void init (ServletConfig config);
```

Na finalização, quando os recursos devem ser liberados:

```
void destroy();
```

Os métodos `init` e `destroy`, quando reescritos, são obrigados a chamar o `super.init()` e `super.destroy()` respectivamente. Isso acontece pois um método é diferente de um construtor, quando estendemos uma classe e criamos o nosso próprio construtor da classe filha, ela chama o construtor da classe pai sem argumentos, preservando a garantia da chamada de um construtor.

Supondo que o método `init` (ou `destroy`) executa alguma tarefa fundamental em sua classe pai, se você esquecer de chamar o `super` terá problemas.

O exemplo a seguir mostra uma servlet implementando os método de inicialização e finalização.

Os métodos `init` e `destroy` podem ser bem simples (lembre-se que são opcionais):

```
package br.com.caelum.servlet;

//imports aqui

public class OiMundo extends HttpServlet {

    public void destroy() {
        super.destroy();
        log("Destruindo a servlet");
    }

    public void init(ServletConfig config) throws ServletException {
        super.init(config);
        log("Iniciando a servlet");
    }

    // método service aqui
}
```

## 10.7 - Curiosidades do mapeamento de uma servlet

Existe outra maneira de configurar servlets no `web.xml`. O segundo tipo de mapping é o que especifica diversas URLs para apontar para a mesma a servlet.

Se marcarmos o `url-pattern` como `/teste/*`, toda URL que acessar o padrão `http://localhost:PORTA/jsp teste/teste/*` irá acessar nossa servlet:

```
<servlet-mapping>
    <servlet-name>servletDeTeste</servlet-name>
    <url-pattern>/teste/*</url-pattern>
</servlet-mapping>
```

Outra opção que o `web.xml` nos dá é a de marcar a servlet para inicialização junto com a aplicação web. Para isto basta usar uma tag chamada `load-on-startup` e atribuir um valor não negativo.

```
<servlet>
    <servlet-name>servletDeTeste</servlet-name>
    <servlet-class>br.com.caelum.servlet.OiMundo</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
```

**Recurso Avançado: load-on-startup**

As servlets marcadas com números menores serão inicializadas antes que as de números maiores. Servlets marcadas com o mesmo número não possuem uma ordem de inicialização que seja definida pela especificação.

## 10.8 - OutputStream x PrintWriter

No nosso primeiro exemplo de servlet usamos o método `getWriter` para acessar um `PrintWriter` do Java.

Para retornar algo ao cliente, podemos usar a `OutputStream` ou o `PrintWriter` que é retornado através do objeto `response`.

```
PrintWriter writer = response.getWriter();
OutputStream stream = response.getOutputStream();
```

Também é possível redirecionar o usuário para outra página através do método `sendRedirect(String)`:

```
response.sendRedirect(novaURL);
```

O importante aqui é que só se deve chamar um dos três métodos acima. Se você escreve algo através do `writer`, o cabeçalho é enviado ao cliente e impede o redirecionamento, enquanto que se você chamar o método `getWriter` e depois o `getOutputStream` ocorrerá uma exception pois não se deve enviar bytes depois de ter aberto um fluxo de caracteres (o encoding já foi definido, não faz mais sentido enviar bytes que não representem caracteres).

Para mais informações sobre bytes e caracteres, `Writers` e `OutputStreams`, confira a nossa apostila de Java e Orientação a Objetos.

## 10.9 - Parâmetros

Toda requisição pode vir acompanhada de parâmetros que costumam ser de extrema importância no desenvolvimento para a web.

No método GET é comum ter uma URL que termine com "?parametro=valor" enquanto no método POST podemos enviar todos os parâmetros através de um formulário ou simplesmente escondidos da URL.

Independentemente do método chamado, os valores dos parâmetros podem ser lidos com o seguinte código, que lê o valor da "idade":

```
request.getParameter("idade");
```

Sendo assim, a servlet a seguir recebe um parâmetro chamado `idade` e o imprime como resposta:

```

protected void service(HttpServletRequest request, HttpServletResponse response)
                      throws ServletException, IOException {
    // recebe o writer
    PrintWriter writer = response.getWriter();

    // escreve o texto
    writer.println("<html>");
    writer.println("Caelum explica o parametro: " + request.getParameter("idade"));
    writer.println("</html>");
}

```

O método `getParameter` retorna uma `String`, portanto todo tipo de tratamento deve ser feito manualmente. Caso o parâmetro não tenha sido passado pela requisição esse método retorna `null`.

Para enviar tais dados podemos utilizar um link com parâmetro através de uma requisição do tipo `get`, ou um formulário com campos de texto (método `get` ou `post`).

Repare que em um arquivo `jsp`, as variáveis `request` e `response` também existem, portanto você pode escrever o mesmo código do `request.getParameter` dentro de um `jsp`. Faz sentido fazer isso? Scriptlet? Ler parâmetro em `jsp`? Código java no meio do código html?

## 10.10 - Exercícios: TestaParametros

- 1) Crie a servlet `TestaParametros` no pacote `br.com.caelum.servlet`.
- 2) Estenda `HttpServlet`. Utilize o `CTRL+SHIFT+O` para importar `HttpServlet`.
- 3) Escreva o código do método `service`. Lembre de usar o truque do `Ctrl+espaço` para gerar a assinatura do método.

```

protected void service(HttpServletRequest request, HttpServletResponse response)
                      throws ServletException, IOException {
    // recebe o writer
    PrintWriter writer = response.getWriter();

    // escreve o texto
    writer.println("<html>");
    writer.println("Caelum explica o parametro: " + request.getParameter("idade"));
    writer.println("</html>");
}

```

- 4) Abra o arquivo `web.xml` e mapeie a servlet `TestaParametros` para a URL `/testa-idade`. **Aumente** o mapeamento abaixo logo após o mapeamento da outra servlet que fizemos.

```

<servlet>
    <servlet-name>idade</servlet-name>
    <servlet-class>br.com.caelum.servlet.TestaParametros</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>idade</servlet-name>

```

```

<url-pattern>/testa-idade</url-pattern>
</servlet-mapping>
```

- 5) Crie um arquivo chamado testa-get.jsp:

```

<html>
<body>
    <a href="/jspteste/testa-idade?idade=24">Testa Parâmetros</a>
</body>
</html>
```

- 6) Crie um arquivo chamado testa-post.jsp:

```

<html>
<body>
    <form action="/jspteste/testa-idade" method="POST">
        <input type="text" name="idade" value="24"/>
        <input type="submit" value="Enviar"/>
    </form>
</body>
</html>
```

- 7) Teste a URL <http://localhost:8080/jspteste/testa-get.jsp>



- 8) Teste a URL <http://localhost:8080/jspteste/testa-post.jsp>



Para saber mais: new Servlet

Na perspectiva Java EE, é possível criar uma nova servlet usando um wizard. É só clicar com o botão direito na pasta src, e então em New -> Servlet. Depois de escolher o nome da classe e o pacote, podemos clicar em Next e configurar o nome e o mapeamento da servlet (o equivalente ao servlet mapping do xml), e ao clicar em Next de novo podemos escolher quais métodos do HttpServlet serão sobrescritos. Ao clicar em Finish a classe da Servlet já vai estar criada, com os métodos selecionados em branco, e a servlet já vai estar configurada no web.xml.

## 10.11 - Exercício opcional

- 1) Crie uma nova servlet que utiliza o método `getParameterNames` para mostrar todos os parâmetros que foram enviados.

Todos os parâmetros

O método `request.getParameterNames()` retorna uma `Enumeration` com todos os nomes de parâmetros enviados.

## 10.12 - doGet, doPost e outros

Por causa da arquitetura da API das servlets, o método `service` é o ponto inicial de uma nova requisição e delega o processo para o representante adequado, de acordo com a requisição.

A implementação de nenhum dos métodos abaixo é obrigatória se a sua servlet estender a classe `HttpServlet`.

Outros dois métodos comuns na API de servlets e no protocolo HTTP são os que tratam requisições específicas de um método como o GET ou o POST:

- `doGet` – responsável pelos métodos **GET**

```
protected void doGet(HttpServletRequest, HttpServletResponse)
                      throws ServletException, IOException {
}
```

- `doPost` – responsável pelo **POST**

```
protected void doPost(HttpServletRequest, HttpServletResponse)
                      throws ServletException, IOException{
}
```

## 10.13 - Conversão de parâmetros

A maior complicação (e chatice) dos parâmetros enviados através do protocolo HTTP é que eles estão limitados a `Strings`, portanto devem ser convertidos para o tipo desejado.

Essa tarefa fica repetitiva e chata se feita manualmente conforme mostramos no código a seguir. Utilizando bibliotecas famosas como Struts, Webwork etc essa tradução fica transparente (ou separada), mais simples e menos perigosa.

```
package br.com.caelum.servlet;

//imports aqui

public class TestaConversaoParametros extends HttpServlet {

    protected void service(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        // recebe o writer
        PrintWriter writer = response.getWriter();

        // escreve o texto
        writer.println("<html>");
        int idade = Integer.parseInt(request.getParameter("idade"));
        writer.println("Caelum explica o parametro: " + idade);
        writer.println("</html>");
    }
}
```

## 10.14 - Exercícios opcionais

- 1) Crie a classe TestaConversaoParametros no pacote br.com.caelum.servlet.
- 2) Escreva o código que vimos anteriormente:

```
package br.com.caelum.servlet;

//faça imports aqui, use CTRL+SHIFT+O

public class TestaConversaoParametros extends HttpServlet {

    protected void service(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        // recebe o writer
        PrintWriter writer = response.getWriter();

        // escreve o texto
        writer.println("<html>");
        int idade = Integer.parseInt(request.getParameter("idade"));
        writer.println("Caelum explica o parametro: " + idade);
        writer.println("</html>");
    }
}
```

- 3) Abra o arquivo web.xml e mapeie a servlet TestaConversaoParametros para a URL conversor-idade.

```
<servlet>
    <servlet-name>conversor</servlet-name>
```

```

<servlet-class>br.com.caelum.servlet.TestaConversaoParametros
</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>conversor</servlet-name>
    <url-pattern>/conversor-idade</url-pattern>
</servlet-mapping>

```

- 4) Crie um arquivo chamado converte-idade.jsp

```

<html>
    <body>
        <form action="/jspteste/converte-idade" method="POST">
            <input type="text" name="idade" value="32"/>
            <input type="submit" value="Enviar"/>
        </form>
    </body>
</html>

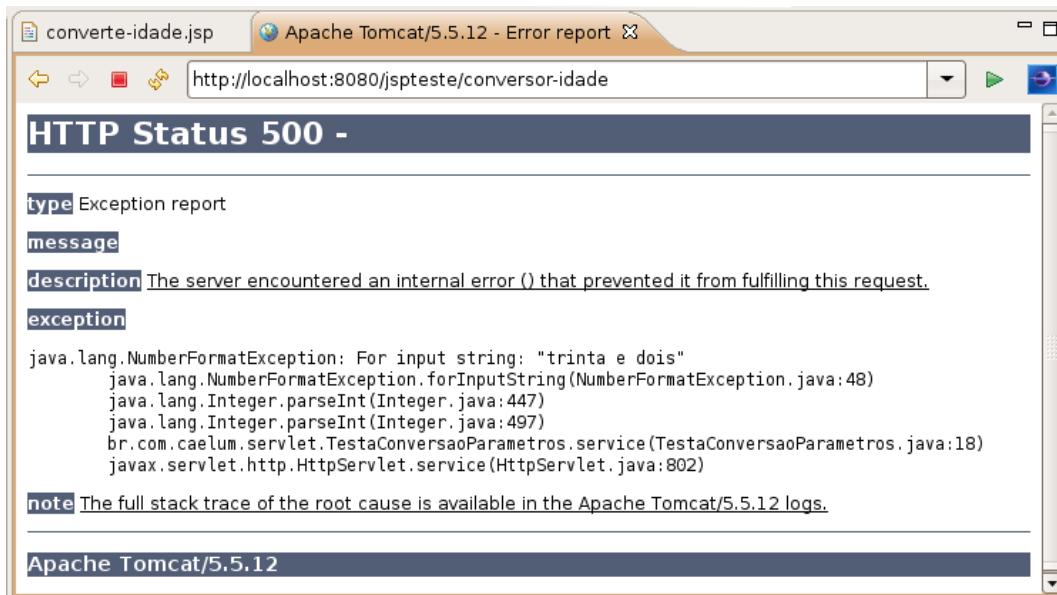
```

- 5) Teste a URL <http://localhost:8080/jspteste/converte-idade.jsp>



- 6) O que acontece ao passar um número inválido?

Um exemplo de número inválido:



7) Tente utilizar a classe `SimpleDateFormat` para fazer o parsing de datas.

## 10.15 - Variáveis membro

Nosso próximo exemplo ilustra um simples contador utilizando uma variável membro de uma servlet.

Da maneira que estamos trabalhando, a especificação da servlet garante que só existirá uma instância dessa servlet por máquina, portanto somente uma instância no total a não ser que você rode sua aplicação em um cluster.

Baseado nesse fato vamos criar uma variável membro chamada `contador` para fixar a idéia de que somente uma instância dessa servlet existirá na memória e que ela sobrevive a requisições: suas variáveis membro não são “limpadas” após o término das requisições.

Escolhendo uma variável do tipo `int` para nosso contador, nossa servlet deve:

- Estender `HttpServlet`;
- Conter uma variável membro do tipo `int`, chamada `contador`;
- A cada requisição somar um no contador e imprimir o número do visitante;

E, a partir dos três pontos acima, devemos testar as nossas URLs diversas vezes, fechar o browser e tentar novamente etc.

Baseando-se no nosso primeiro exemplo vejamos o método `service`:

```
protected void service(HttpServletRequest request, HttpServletResponse response)
                      throws ServletException, IOException {
    contador++; // algum problema aqui?

    // recebe o writer
    PrintWriter out = response.getWriter();
```

```
// escreve o texto
out.println("<html>");
out.println("Caelum explica: " + contador + " visita.");
out.println("</html>");
}
```

## 10.16 - Exercícios: Contador

- 1) Crie a servlet Contador no pacote correto. Escolha o menu File, New, Class.

- a) Estenda HttpServlet.

```
public class Contador extends HttpServlet {  
}
```

- b) Utilize o CTRL+SHIFT+O para importar **HttpServlet**.

- c) Adicione uma variável membro do tipo int, chamada contador.

```
private int contador = 0;
```

- d) Escreva a estrutura do método service. Muito cuidado com o nome dos argumentos e etc.

```
@Override  
protected void service(HttpServletRequest request, HttpServletResponse response)  
    throws ServletException, IOException {  
}
```

- e) Escreva o código do método service.

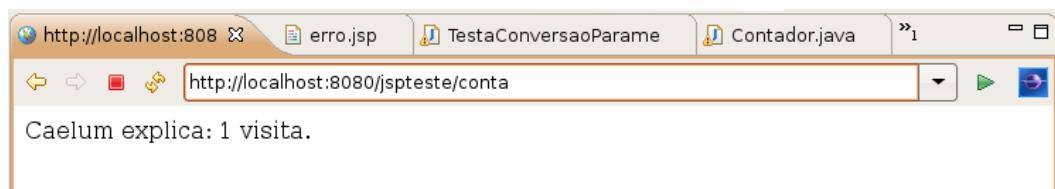
```
protected void service(HttpServletRequest request, HttpServletResponse response)  
    throws ServletException, IOException {  
    contador++; // algum problema aqui?  
  
    // recebe o writer  
    PrintWriter out = response.getWriter();  
  
    // escreve o texto  
    out.println("<html>");  
    out.println("Caelum explica: " + contador + " visita.");  
    out.println("</html>");  
}
```

- 2) Abra o arquivo web.xml e mapeie a url /conta para a servlet Contador.

```
<servlet>  
    <servlet-name>nossoContador</servlet-name>  
    <servlet-class>br.com.caelum.servlet.Contador</servlet-class>  
</servlet>  
  
<servlet-mapping>  
    <servlet-name>nossoContador</servlet-name>
```

```
<url-pattern>/conta</url-pattern>
</servlet-mapping>
```

- 3) Teste a URL a seguir duas vezes <http://localhost:8080/jspteste/conta>



- 4) Passado algum tempo, feche o browser. Abra novamente. Lembre-se que o browser é o cliente que não mandou em nenhum momento o server executar um destroy em sua servlet... portanto a servlet ainda está lá viva: acesse a mesma URL. Qual o resultado?



### Servlets e concorrência

Muitos programadores que começam a usar servlets acabam esquecendo dos problemas que a concorrência gera.... membros de uma classe que represente uma servlet devem ser tratados com muito cuidado para evitar situações onde a perda de dados irá ocorrer.

Implementar a interface `SingleThreadModel` na servlet é uma saída altamente desaconselhada, uma vez que poderá perder a grande vantagem das servlets de processarem mais de uma requisição ao mesmo tempo.

A partir da versão 2.4 da API de servlets, a interface `SingleThreadModel` foi depreciada, isto é, não deve ser utilizada.

Portanto, jamais utilize variáveis membros em uma servlet, use argumentos de métodos para passar valores entre os mesmos.

## 10.17 - HTML e Java: eu não quero código html na minha servlet!

Foi possível inferir dos exemplos anteriores que é complicado escrever html dentro de uma servlet o tempo todo.

Veja só como foi difícil ler um parâmetro dentro de uma servlet e retornar o html para o cliente com diversos `out.println`.

Primeiro fica tudo mal escrito e difícil de ler. O **html passa a atrapalhar o código Java**. Depois, quando o responsável pelo design gráfico da página quiser alterar algo terá que conhecer java para entender o que está escrito lá dentro... hmm... não parece uma boa solução.

Mudou o css, mudou o campo, recompila a servlet e envia novamente para o cliente... é uma **solução inviável**.

## 10.18 - Como funciona uma página JSP

O web container interpreta o arquivo JSP, o compila e transforma em uma servlet! Assim sendo, logo que o arquivo JSP é chamado pela primeira vez por um cliente, uma servlet que o representa é criada, aplicando todos os benefícios da mesma para uma página JSP.

### Vantagens

O benefício mais claro é não colocar uma série imensa de código html dentro de uma classe em Java, o que dificulta muito a alteração da página por um designer. Compare o código da listagem do OiMundo como servlet. Muito mais simples de editar o html: mas o designer não comprehende o código Java.

Duas coisas que ajudam a combater esse problema são os JavaBeans e padrões de arquitetura variados existentes no mercado.

**JSP:** fica mais fácil para o designer alterar o código html **Servlets:** fica mais fácil de programar orientado a objetos e em Java

## 10.19 - Web archive (.war)

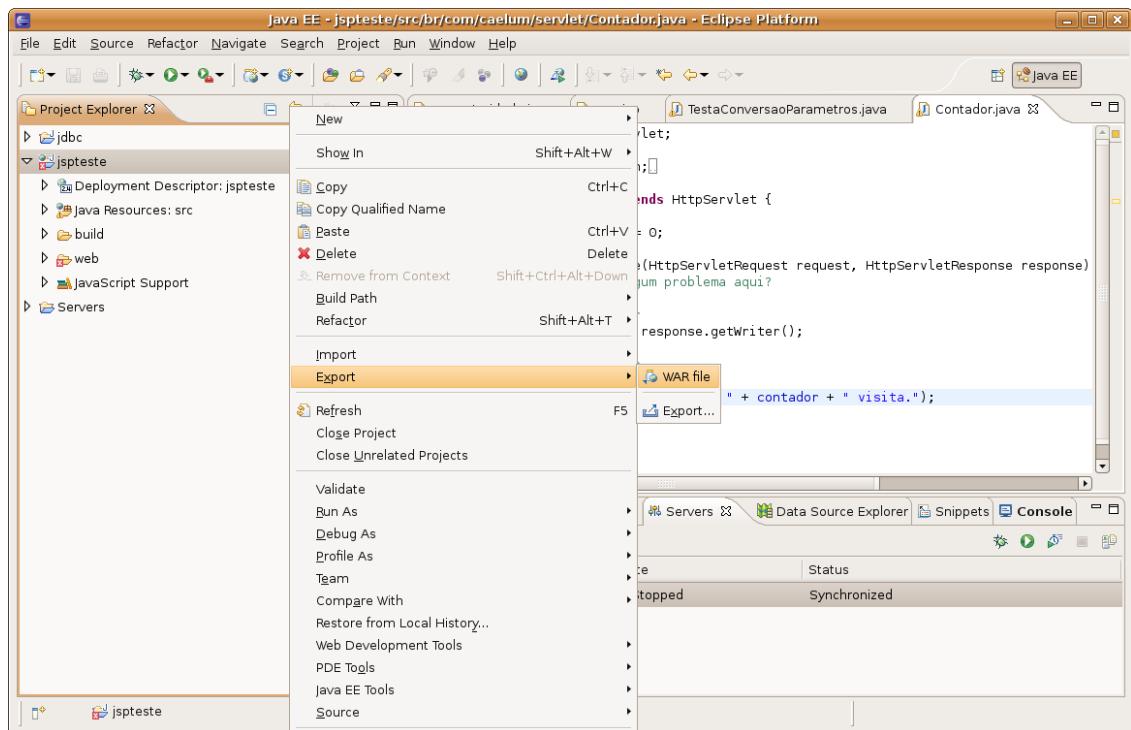
O processo padrão de deploy (envio, submissão) de uma aplicação web é o de criar um arquivo de extensão war, que é um arquivo zip com o diretório base da aplicação sendo a raiz do zip.

No nosso exemplo, todo o conteúdo do diretório web deveria ser incluído em um arquivo teste.war. Após compactar o diretório web com esse nome, efetuaremos o *deploy*.

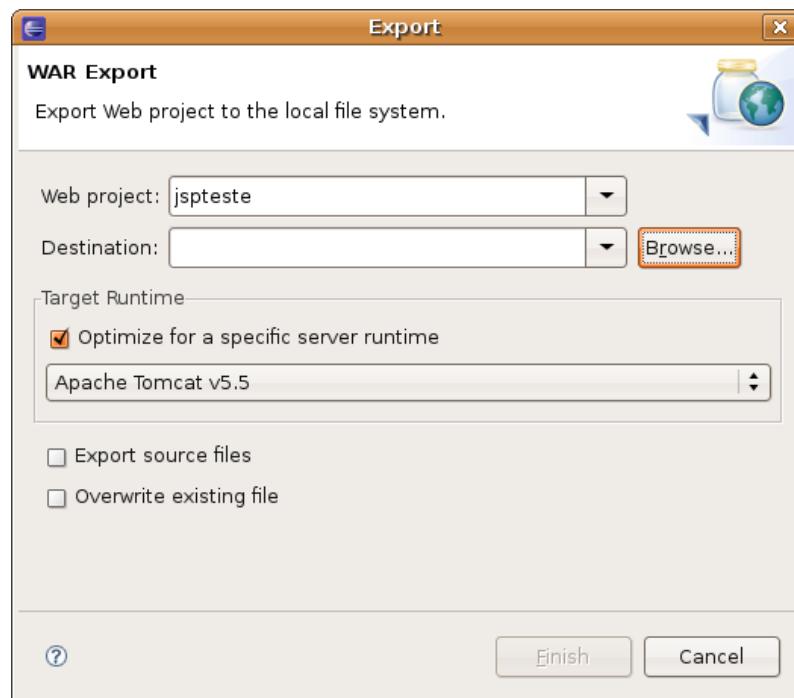
No Tomcat, basta copiar o arquivo .war no diretório TOMCAT/webapps/ e ele será descompactado, por fim o novo contexto chamado teste estará disponível.

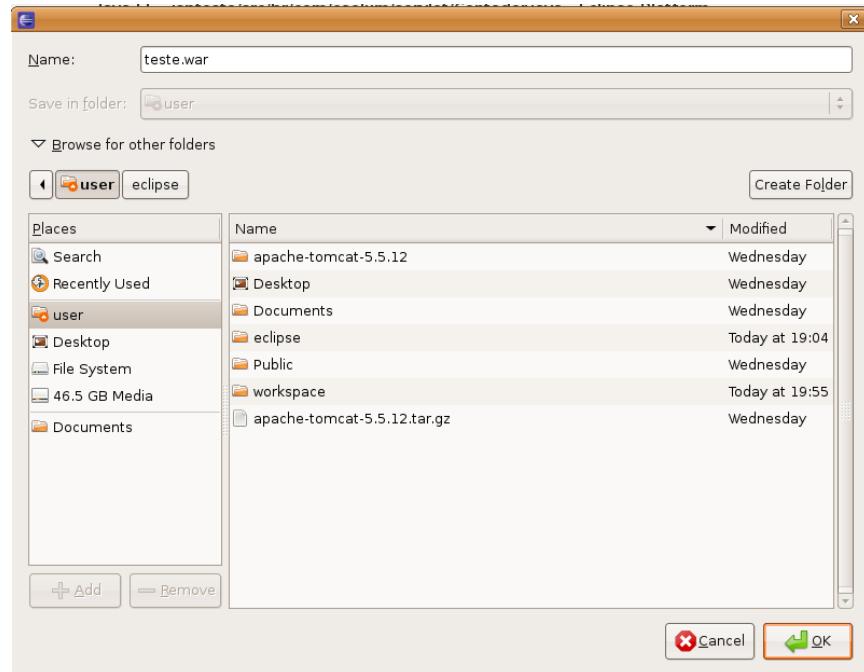
## 10.20 - Exercícios: Deploy com war

- 1) Vamos praticar criando um war e utilizando-o.
  - a) Clique com o botão direito no projeto e vá em Export -> WAR file



- b) Clique no botão Browse e escolha a pasta do seu usuário e o nome **teste.war**.





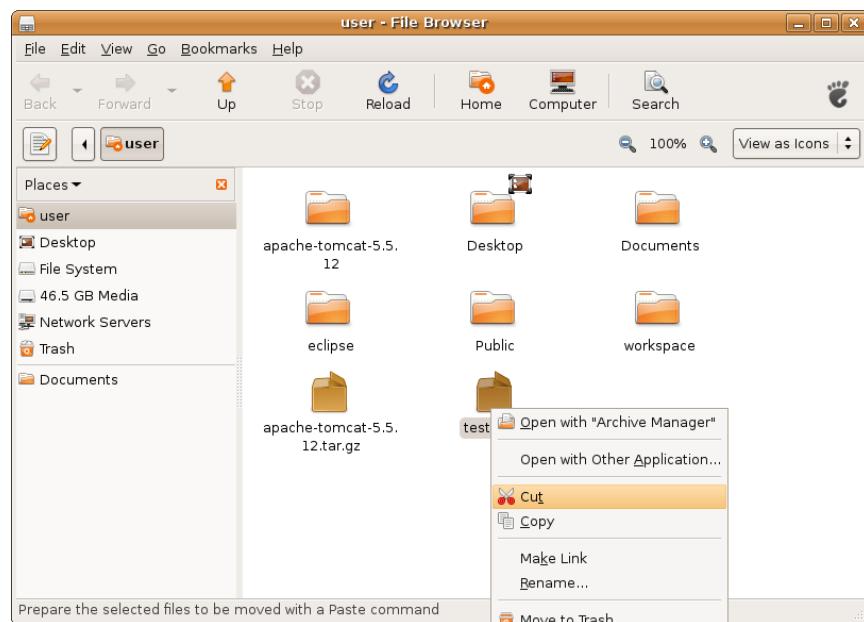
c) Clique em Finish

Pronto, nosso war está criado!

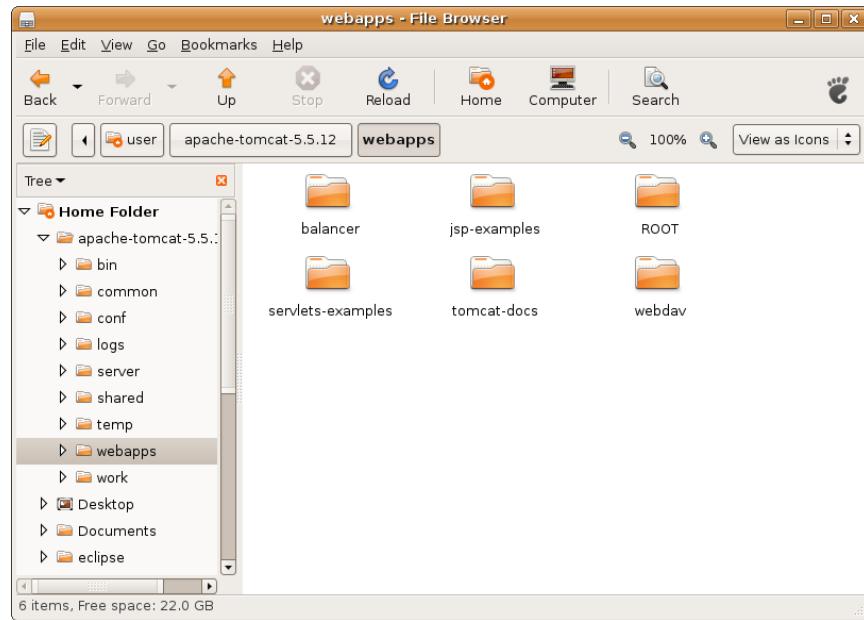
2) Vamos instalar nosso war!

a) Abra o File Browser

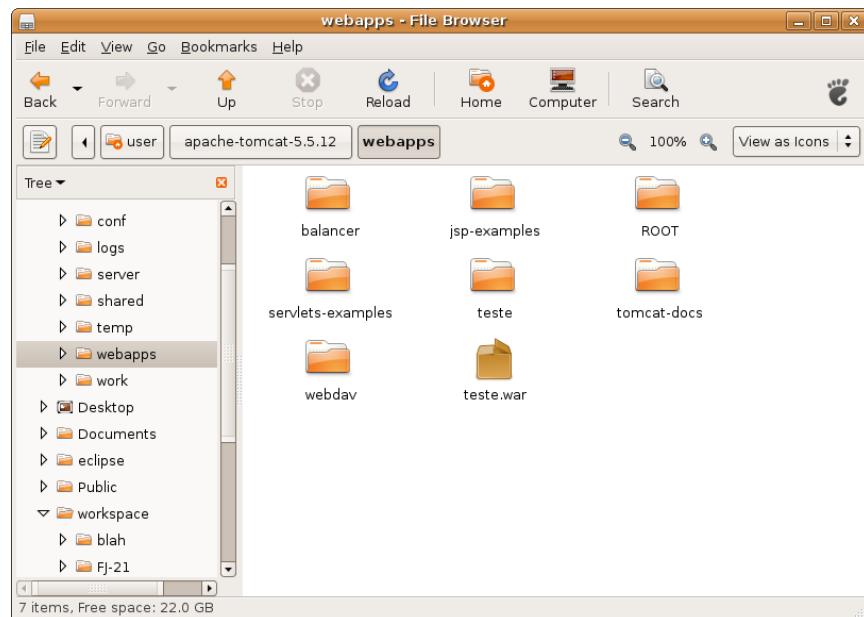
b) Clique da direita no arquivo teste.war e escolha Cut.



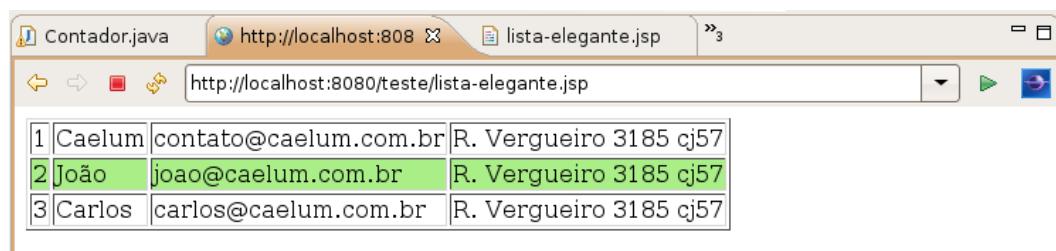
c) Vá para o diretório **apache-tomcat, webapps**. Certifique-se que seu tomcat está rodando.



- d) Cole o seu arquivo aqui (**Edit, Paste**). Repare que o diretório teste foi criado.



- e) Agora podemos acessar o projeto através da URL: <http://localhost:8080/teste/lista-elegante.jsp>



## 10.21 - Quando acontece um erro em uma servlet

Imagine fazer o tratamento de erro em nossas servlets assim como fizemos com nossos JSPs: quase nada. Seria possível?

Retomando o curso FJ-11 e os fundamentos da linguagem e tecnologia Java, o método `service` foi definido na classe `HttpServlet` e, portanto, só podemos jogar as exceptions que tal método definiu.

Olhando de perto a assinatura desse método, encontramos `IOException` e `ServletException`. Lembramos também que todos os erros do tipo *unchecked* (que estendem `RuntimeException`) não precisam ser declarados no `throws` portanto não precisamos tratá-los.

Mas o que fazer com erros do tipo `SQLException`? Ou qualquer outro erro que estenda `Exception` não através de `RuntimeException` (seja um *checked exception*) diferente de `IOException` e `ServletException`? Não podemos jogar. Por quê? Pois a classe pai definiu o método sem esse `throws`.

Como no Java só temos duas opções - ou jogamos a exception ou usamos um `try` e `catch` -, nos resta uma única opção...

## 10.22 - O try e catch

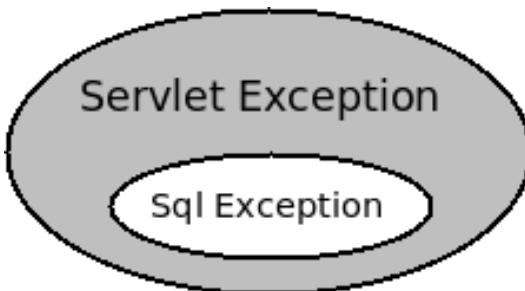
Portanto, com servlets a história já é outra. Temos que ‘tratar’ os erros antes de jogá-los adiante.

Aqui basta lembrar da assinatura do método `service` (ou `doGet`, `doPost`, etc), ele só permite fazer o throw de `ServletException`, `IOException` e (claro) `RuntimeException`.

A classe `ServletException` foi criada para ser utilizada como *wrapper* para as nossas exceptions, isto é, devemos ‘embrulhar’ nossa exceção em uma `ServletException` antes de sair do nosso método. Por exemplo:

```
try {
    java.sql.DriverManager.getConnection("jdbc:teste:invalido", "usuario", "senha");
} catch (SQLException e) {
    throw new ServletException(e);
}
```

Neste caso, `SQLException` está dentro de `ServletException`. Repare que essa técnica de injetar uma causa de uma exception dentro de outra é amplamente utilizada na biblioteca padrão Java e em muitas extensões.



O que acontece quando o servlet container pega uma `ServletException`?

Primeiro, ele verifica se existiu alguma causa (*root cause*) para essa exception, se sim, ele usa essa causa para procurar o mapeamento no `web.xml`. Se não existir nenhuma causa ele procura `ServletException` no arquivo `web.xml`.

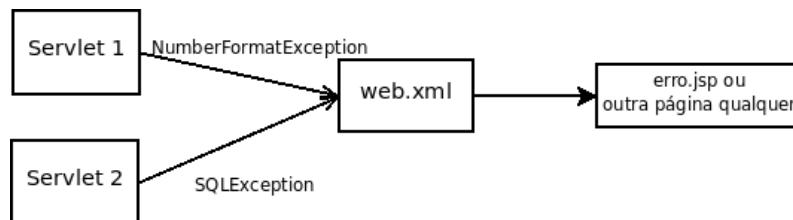
Portanto no caso que definimos acima, o servlet container procurará o mapeamento de `SQLException` e, repare, vamos fazer isso a seguir.

## 10.23 - Tratamento padrão de erros – modo declarativo

O processo declarativo para controle de erros é o mais fácil de usar pois não impõe mudanças nas páginas servlets quando surge a necessidade de alterar tal processo.

No do arquivo `web.xml` é possível configurar, para cada tipo de erro (através de código ou exception), qual a página HTML, JSP, servlet etc, deve ser utilizada.

O diagrama abaixo mostra o que acontece se uma exception ocorrer em algum ponto de sua aplicação web através do processo declarativo: todas as exceções de um tipo ou de um código definido vai para uma certa página de erro.



## 10.24 - Configurando a página de erro

Agora, precisamos configurar no arquivo `web.xml` qual tipo de exception vai para qual página JSP. Esse é um mapeamento simples, basta adicionar o nome da classe da exception (`exception-type`) e o nome da página alvo (`location`).

```

<error-page>
  <exception-type>classe</exception-type>
  <location>/página.jsp</location>
</error-page>
  
```

No final, o controle de erro centralizado no servlet container ajuda muito pois basta mapear uma ou mais páginas de erro no `web.xml` e pronto, todo erro desse tipo que ocorrer em suas servlets será redirecionado para nossa página de erro.

## 10.25 - Exercícios

1) Crie uma servlet para testar o controle de exceptions:

- Pacote: `br.com.caelum.servlet`, classe `TestaErro`;
- Estenda `HttpServlet`;
- Método `service`:

```

protected void service(HttpServletRequest request, HttpServletResponse response)
                      throws ServletException, IOException {
    try {
        java.sql.DriverManager.getConnection("jdbc:teste:invalido", "usuario", "senha");
    } catch (SQLException e) {
        e.printStackTrace();
        request.setAttribute("msg", "Ocorreu um erro ao tentar conectar ao banco de dados.");
        RequestDispatcher dispatcher = request.getRequestDispatcher("erro.jsp");
        dispatcher.forward(request, response);
    }
}
  
```

**Alavanque sua carreira com um de nossos treinamentos Java**

```
    } catch (SQLException e) {
        throw new ServletException(e);
    }
}
```

- 2) Crie a servlet chamada TestaErro no seu web.xml:

```
<servlet>
    <servlet-name>TestaErro</servlet-name>
    <servlet-class>br.com.caelum.servlet.TestaErro</servlet-class>
</servlet>
```

- 3) Crie a url /testaeerroservlet para a servlet TestaErro no seu web.xml:

```
<servlet-mapping>
    <servlet-name>TestaErro</servlet-name>
    <url-pattern>/testa-errosservlet</url-pattern>
</servlet-mapping>
```

- 4) Altere o arquivo web.xml e adicione uma error-page. Para generalizar os erros que são direcionados para erro.jsp mapeie java.lang.Exception para a sua página de erro.

- a) exception-type: java.lang.Exception
  - b) location: /erro.jsp

```
<error-page>
    <exception-type>java.lang.Exception</exception-type>
    <location>/erro.jsp</location>
</error-page>
```

- 5) Acesse a URL <http://localhost:8080/jspteste/testa-erroserlylet>



`response.sendError()` e `response.setStatus()`

Existem dois métodos `sendError` que podemos utilizar e estão na interface `HttpServletResponse`. Eles são os métodos responsáveis por lidar com as exceções em estilo programático.

Esses são os métodos responsáveis por lidar com as exceções em código programático. Os dois métodos recebem um `int` com o código do erro que ocorreu (constantes estão disponíveis através da interface `HttpServletResponse`). O segundo método recebe também uma variável do tipo `String` que representa a mensagem de erro.

O método `setStatus` que recebe um `int` funciona do mesmo jeito mas não gera o processo de erro correspondente, somente marca no cabeçalho o que ocorreu.

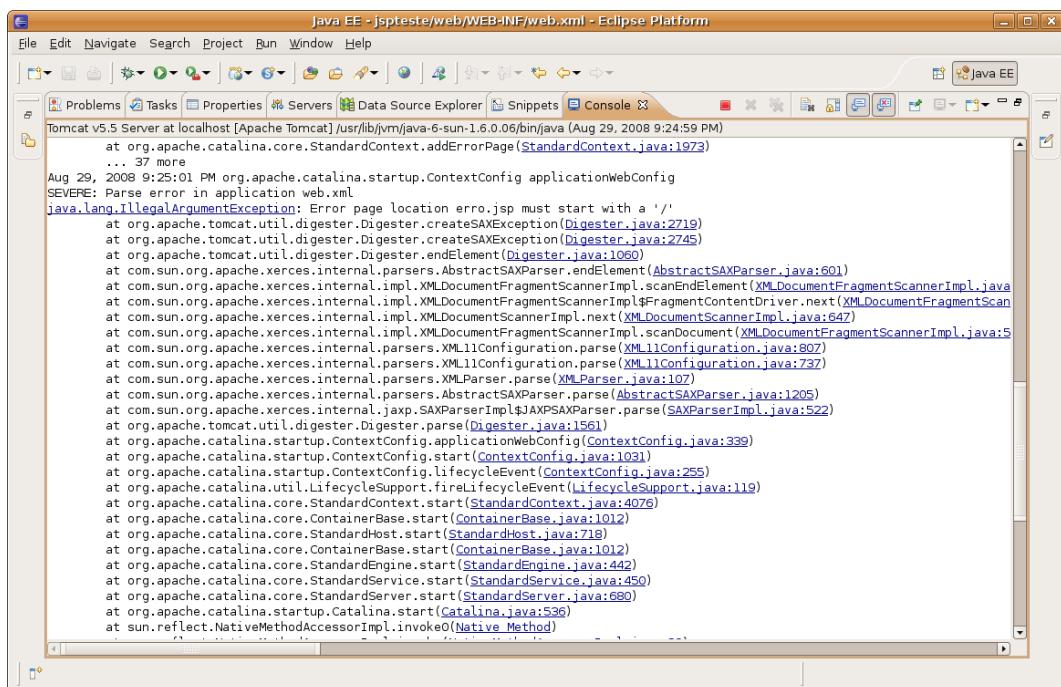
## 10.26 - Erros comuns

Esquecer a barra “/” no mapeamento da página de erro e ver uma página do tipo 404. Por quê?

Acontece que quando você iniciou o Tomcat, ele percebeu que seu xml é inválido e não inicializou seu contexto. Este é um bom momento para você pegar a prática de, sempre que reiniciar o seu Tomcat, verificar se aparece alguma exception no console e, se aparecer, lê-la.

Mais importante ainda é aprender a ler exceptions sem a ajuda de terceiros. O nome do erro, a mensagem, e as linhas onde ela ocorreu na pilha de execução entregam todas as dicas para você descobrir o que aconteceu – na maior parte das vezes.

Verifique seu arquivo de console:



```
Java EE - jspteste/web/WEB-INF/web.xml - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Problems Tasks Properties Servers Data Source Explorer Snippets Console
Tomcat v5.5 Server at localhost [Apache Tomcat] /usr/lib/jvm/java-6-sun-1.6.0_06/bin/java (Aug 29, 2008 9:24:59 PM)
    at org.apache.catalina.core.StandardContext.addErrorPage(StandardContext.java:1973)
    ... 37 more
Aug 29, 2008 9:25:01 PM org.apache.catalina.startup.ContextConfig applicationWebConfig
SEVERE: Parse error in application web.xml
java.lang.IllegalArgumentException: Error page location erro.jsp must start with a '/'
    at org.apache.tomcat.util.digester.createSAXException(Digester.java:2719)
    at org.apache.tomcat.util.digester.createSAXException(Digester.java:2745)
    at org.apache.tomcat.util.digester.Digester.endElement(Digester.java:1060)
    at com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser.endElement(AbstractSAXParser.java:601)
    at com.sun.org.apache.xerces.internal.impl.XMLDocumentFragmentScannerImpl.scanEndElement(XMLDocumentFragmentScannerImpl.java)
    at com.sun.org.apache.xerces.internal.impl.XMLDocumentFragmentScannerImpl$FragmentContentDriver.next(XMLDocumentFragmentScannerImpl.java)
    at com.sun.org.apache.xerces.internal.impl.XMLDocumentScannerImpl.next(XMLDocumentScannerImpl.java:647)
    at com.sun.org.apache.xerces.internal.impl.XMLDocumentFragmentScannerImpl.scanDocument(XMLDocumentFragmentScannerImpl.java:58)
    at com.sun.org.apache.xerces.internal.parsers.XML11Configuration.parse(XML11Configuration.java:807)
    at com.sun.org.apache.xerces.internal.parsers.XML11Configuration.parse(XML11Configuration.java:737)
    at com.sun.org.apache.xerces.internal.parsers.XMLParser.parse(XMLParser.java:107)
    at com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser.parse(AbstractSAXParser.java:1205)
    at com.sun.org.apache.xerces.internal.jaxp.SAXParserImpl$JAXPSAXParser.parse(SAXParserImpl.java:522)
    at org.apache.tomcat.util.digester.Digester.parse(Digester.java:1561)
    at org.apache.catalina.startup.ContextConfig.applicationWebConfig(ContextConfig.java:339)
    at org.apache.catalina.startup.ContextConfig.start(ContextConfig.java:103)
    at org.apache.catalina.startup.ContextConfig.lifecycleEvent(ContextConfig.java:255)
    at org.apache.catalina.util.LifecycleSupport.fireLifecycleEvent(LifecycleSupport.java:119)
    at org.apache.catalina.core.StandardContext.start(StandardContext.java:4076)
    at org.apache.catalina.core.ContainerBase.start(ContainerBase.java:102)
    at org.apache.catalina.core.StandardHost.start(StandardHost.java:719)
    at org.apache.catalina.core.ContainerBase.start(ContainerBase.java:102)
    at org.apache.catalina.core.StandardEngine.start(StandardEngine.java:442)
    at org.apache.catalina.core.StandardService.start(StandardService.java:450)
    at org.apache.catalina.core.StandardServer.start(StandardServer.java:680)
    at org.apache.catalina.startup.Catalina.start(Catalina.java:536)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
```

Página de erro erro.jsp não começa com uma “/”, isto é, ele está dizendo exatamente o seu erro!

## 10.27 - Tratamento de outros erros

Para muitas aplicações, é importante mostrar mensagens de erro padronizadas. Por exemplo, quando o usuário tenta acessar uma página que não existe, ele deve receber como resposta uma página elegante indicando que não foi possível encontrar a página requisitada, incluindo o erro 404 (que faz parte do protocolo HTTP).

Para fazer isso, definiremos uma nova `error-page` no arquivo `web.xml` mas em vez de colocar o `exception-type` utilizaremos a tag `error-code`. Será necessário criar uma página separada para o erro 404 pois a que criamos anteriormente assumia a existência de uma exception, que não é esse caso:

```
<error-page>
<error-code>404</error-code>
```

```
<location>/pagina-nao-encontrada.jsp</location>
</error-page>
```

## 10.28 - Exercício opcional

- 1) Trate o erro tipo 404 na sua aplicação.

- Teste a URL `http://localhost:8080/jspteste/qualquer_coisa.jsp` e veja qual a mensagem padrão do Tomcat.



- Crie uma pagina de erro para páginas não encontradas: `WebContent/pagina-nao-encontrada.jsp`

```
<%@ page isErrorPage="true" %>
<html>
    <h1>404 - Página não encontrada.</h1>
</html>
```

- Mapeie o erro 404 para a página `/pagina-nao-encontrada.jsp`

```
<error-page>
    <error-code>404</error-code>
    <location>/pagina-nao-encontrada.jsp</location>
</error-page>
```

- Teste a url `http://localhost:8080/jspteste/qualquer_coisa.jsp`



## 10.29 - Outro erro comum

Mesmo depois que você mapeou suas páginas de erro parece que o `web.xml` não existe, não houve redirecionamento nenhum?

**Solução:** Reinicie o Tomcat.

## 10.30 - Servlet para adicionar contatos no banco

Imagine um método `service` que recebe três parâmetros: nome, email e endereço.

Sendo extremamente lógicos basta:

- Ler os parâmetros e preencher um objeto do tipo Contato;
- Instanciar ContatoDAO e adicionar tal objeto no banco;
- Mostrar uma mensagem de ok para o cliente.

Portanto vamos ao exercício...

## 10.31 - Exercício

- 1) Crie uma servlet chamada `AdicionaContatoServlet` no pacote `br.com.caelum.servlet`
- 2) Não se esqueça de estender a classe `HttpServlet`.

```
public class AdicionaContatoServlet extends HttpServlet {
```

- 3) Coloque o método `service`.

```
protected void service(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
```

- 4) Use CTRL+SHIFT+O para fazer os imports necessários.

- 5) Crie um objeto do tipo Contato, chame-o de contato.

```
Contato contato = new Contato();
```

- 6) Através da variável `request` leia o parâmetro `nome` e use seu valor para chamar o setter do `contato`. Faça o mesmo para os campos `email` e `endereço`.

```
String nome = request.getParameter("nome");
String endereco = request.getParameter("endereco");
String email = request.getParameter("email");

contato.setNome(nome);
contato.setEndereco(endereco);
contato.setEmail(email);
```

- 7) Crie um objeto do tipo `ContatoDAO`, chame-o de `dao` e utilize as variáveis `contato` e `dao` para adicionar o contato ao banco de dados.

```
try{
    ContatoDAO dao = new ContatoDAO();
    dao.adiciona(contato);
} catch (SQLException e) {
```

```

        throw new ServletException(e);
}

```

- 8) Através do response.getWriter() mostre uma mensagem de ok para o cliente.

```

PrintWriter writer = response.getWriter();
writer.println("<html>");
writer.println("Contato Adicionado");
writer.println("</html>");

```

- 9) Mapeie a classe AdicionaContatoServlet no web.xml.

```

<servlet>
    <servlet-name>adicionaContato</servlet-name>
    <servlet-class>br.com.caelum.servlet.AdicionaContatoServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>adicionaContato</servlet-name>
    <url-pattern>/testa-adiciona</url-pattern>
</servlet-mapping>

```

- 10) Faça um arquivo WebContent/testa-adiciona-contato.jsp.

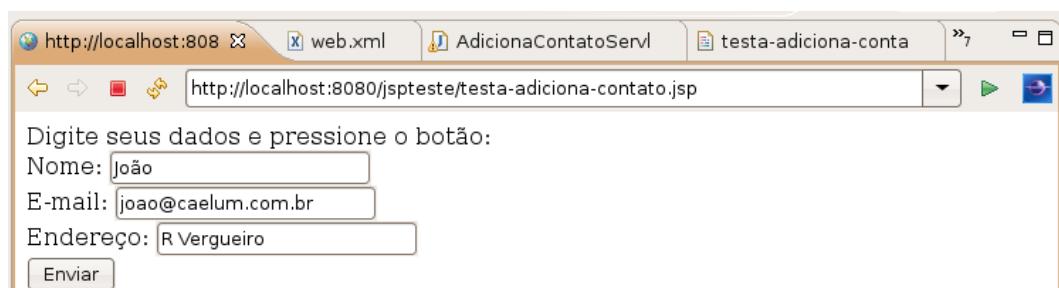
```

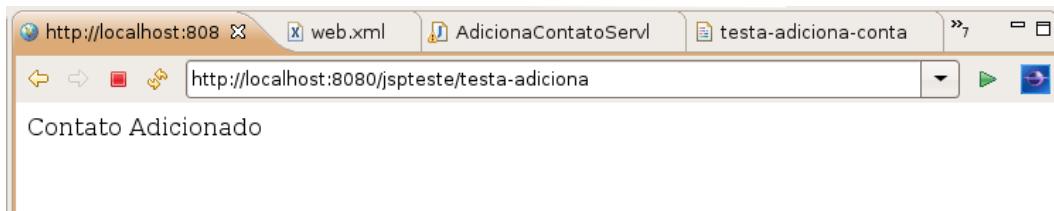
<html>
    <body>
        Digite seus dados e pressione o botão:<br/>

        <form action="testa-adiciona" method="POST">
            Nome:      <input type="text" name="nome"/><br/>
            E-mail:    <input type="text" name="email"/><br/>
            Endereço: <input type="text" name="endereco"/><br/>
            <input type="submit" value="Enviar"/>
        </form>
    </body>
</html>

```

- 11) Teste a URL http://localhost:8080/jsp teste/testa-adiciona-contato.jsp





## 10.32 - Exercícios Opcionais

- 1) Faça agora uma Servlet para mostrar todos os contatos do banco e uma para remover os contatos do banco.

# Servlet e JSP API

*"Vivemos todos sob o mesmo céu, mas nem todos temos o mesmo horizonte."*  
– Konrad Adenauer

Neste capítulo, você aprenderá a utilizar outros recursos avançados da API de servlets.

## 11.1 - Propriedades de páginas JSP

Como dizer qual o encoding de nossos arquivos jsp de uma maneira global? Como nos proteger de programadores iniciantes em nossa equipe e desabilitar o código scriptlet? Como adicionar um arquivo antes e/ou depois de todos os arquivos JSPs? Ou de todos os JSPs dentro de determinado diretório?

Para responder essas e outras perguntas, a API de jsp resolveu possibilitar definir algumas tags no nosso arquivo `web.xml`.

Por exemplo, para desativar *scripting* (os scriptlets):

```
<scripting-invalid>true</scripting-invalid>
```

Ativar *expression language* (que já vem ativado):

```
<el-ignored>false</el-ignored>
```

Determinar o encoding dos arquivos de uma maneira genérica:

```
<page-encoding>UTF-8</page-encoding>
```

Incluir arquivos estaticamente antes e depois de seus JSPs:

```
<include-prelude>/antes.jspf</include-prelude>
<include-coda>/depois.jspf</include-coda>
```

O código a seguir mostra como aplicar tais características para todos os JSPs, repare que a tag `url-pattern` determina o grupo de arquivos cujos atributos serão alterados:

```
<jsp-config>
  <jsp-property-group>
    <display-name>todos os jsps</display-name>
    <description>configurações de todos os jsps</description>
    <url-pattern>*.jsp</url-pattern>
    <scripting-invalid>true</scripting-invalid>
    <el-ignored>false</el-ignored>
    <page-encoding>UTF-8</page-encoding>
    <include-prelude>/antes.jspf</include-prelude>
```

```
<include-coda>/depois.jspf</include-coda>
</jsp-property-group>
</jsp-config>
```

## 11.2 - Exercícios

- 1) Configure o seu arquivo `web.xml` para que todos os JSPs não aceitem scriptlet, por exemplo:

```
<jsp-config>
  <jsp-property-group>
    <display-name>jsp sem script</display-name>
    <url-pattern>*.jsp</url-pattern>
    <scripting-invalid>true</scripting-invalid>
  </jsp-property-group>
</jsp-config>
```

Além das tags usadas nesse exemplo, você pode adicionar qualquer uma das comentadas anteriormente.

- 2) Para que os `jsp-config` tenham efeito, precisamos reiniciar o Tomcat e limpar a pasta `work` do servidor. Essa pasta é onde o Tomcat coloca os arquivos JSP compilados para melhor performance; apagá-la força o servidor a recompilar tudo e aplicar as novas configurações.

Para limpar essa pasta, basta clicar com botão direito no Tomcat na aba `Servers` e escolher **Clean Tomcat work directory....**



## 11.3 - Filtros

Um dos conceitos mais interessantes que existem na API de servlets é um *design pattern* que permite executar tarefas independentemente do que aconteceu e do que vai acontecer, um **filtro**.

A idéia consiste em um método de interceptação chamado `doFilter`. Esse método é responsável por executar o que desejar e, a qualquer instante, pedir para a execução continuar, algo como:

```
public void doFilter(ServletRequest request, ServletResponse response,
                      FilterChain chain) throws ServletException, IOException {
    // executa o que desejar antes de continuar

    // continua
    chain.doFilter(request, response);

    // executa o que desejar depois de continuar
}
```

Note que o código acima não sabe o que será executado ao chamar o método `doFilter` do `FilterChain`. Pode ser que exista outro filtro para ser executado, pode ser que uma servlet seja executada, ou até mesmo um

JSP. Mas esse filtro não tem conhecimento e - acima de tudo - não está interessado no que vai acontecer ao chamar esse método.

O uso clássico utilizado para filtros, e mostrado em exemplos de Java no mundo todo, é para logar informações referentes a requisição, abrir (antes) e fechar (depois) transações, descriptografar informações que foram enviadas (antes) e criptografar informações antes de enviar (depois), ou ainda compactar os dados a serem enviados ao cliente (depois).

No nosso exercício, utilizaremos o filtro para mostrar uma mensagem de log no sistema.

## 11.4 - Configuração de filtros

A configuração de filtros é feita de maneira semelhante a das servlets, com a principal diferença sendo que eles vão ser executados antes daquilo que foi mapeado.

O exemplo a seguir mostra um filtro mapeado para todas as imagens de formato gif, isto é, sempre que uma URL terminando em .gif for acessada, o filtro será executado antes.

```
<filter>
    <filter-name>FiltrarGif</filter-name>
    <filter-class>br.com.caelum.servlet.filtro.FiltrarGif</filter-class>
</filter>

<filter-mapping>
    <filter-name>FiltrarGif</filter-name>
    <url-pattern>*.gif</url-pattern>
</filter-mapping>
```

Note a similaridade do mapeamento de uma servlet com o de um filtro.

## 11.5 - Exercícios

- 1) Crie uma classe chamada LogFiltro no pacote br.com.caelum.servlet.filtro.
- 2) Faça sua classe **implementar** javax.servlet.Filter. O Eclipse vai reclamar que faltam alguns métodos. Após o eclipse mostrar o erro de compilação acima, utilize a tecla de atalho **CTRL+1** para resolver o erro: escreva a definição dos três métodos (**add unimplemented methods**).
- Os métodos init e destroy já são conhecidos pois funcionam exatamente como definimos para as servlets.
- 3) Implemente o *conteúdo* método doFilter:

```
public void doFilter(ServletRequest request, ServletResponse response,
                      FilterChain chain) throws ServletException, IOException {
    // executa o que desejar antes de continuar
    System.out.println("Nova requisição para :"
                        + ((HttpServletRequest) request).getRequestURI());
    // continua
    chain.doFilter(request, response);
    // executa o que desejar depois de continuar
    System.out.println("Requisição terminada");
}
```

- 4) Agora, vamos mapear todos os JSPs para executarem primeiro esse filtro:

```
<filter>
    <filter-name>LogFiltro</filter-name>
    <filter-class>br.com.caelum.servlet.filtro.LogFiltro</filter-class>
</filter>

<filter-mapping>
    <filter-name>LogFiltro</filter-name>
    <url-pattern>*.jsp</url-pattern>
</filter-mapping>
```

- 5) Teste a URL <http://localhost:8080/jsp teste/bemvindo.jsp>



## 11.6 - Entendendo os filtros

A maneira mais simples de entender os filtros é visualizá-los como camadas que são executadas uma depois das outras, sem saber o que acontece.

Durante o curso, comentaremos sobre alguns controladores, e a idéia de filtros é normalmente chamada de **Interceptadores** neles.

Tais interceptadores ganharam fama primeiro no Jboss e depois no Tomcat, neste segundo sob o nome de válvulas, e são tão importantes que podem ser utilizados para controle de autorização e autenticação, upload, compactação de arquivos a serem baixados etc, sempre sem mudar uma linha de código da sua lógica de negócios.

## 11.7 - Exercício Opcional

- 1) Faça um Filtro que diga quanto tempo a sua servlet demorou para executar

# Model View Controller

*“Ensinar é aprender duas vezes.”*  
— Joseph Joubert

O padrão arquitetural MVC e o *request dispatcher*.

## 12.1 - Servlet ou JSP?

Colocar todo HTML dentro de uma Servlet realmente não parece a melhor idéia. O que acontece quando precisamos mudar o design da página? O seu designer não vai ter tempo de editar sua Servlet, recompilá-la e colocá-la no servidor.

Uma idéia mais interessante é usar o que é bom de cada um dos dois.

O JSP foi feito apenas para apresentar o resultado, e ele não deveria fazer acessos a bancos e outros. Isso deveria estar na Servlet.

O ideal então é que a Servlet faça o trabalho sujo e árduo, e o JSP apenas apresente esses resultados. A Servlet possui a lógica de negócios (ou regras de negócio) e o JSP tem a lógica de apresentação.

Imagine o código do método da servlet AdicionaContatoServlet que fizemos antes:

```
protected void service(
    HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

    // log
    System.out.println("Tentando criar um novo contato...");

    // acessa o bean
    Contato contato = new Contato();
    // chama os setters
    ...

    // adiciona ao banco de dados
    ContatoDAO dao = new ContatoDAO();
    dao.adiciona(contato);

    // ok.... visualização
    request.getWriter().println("<html>Ok</html>");

}
```

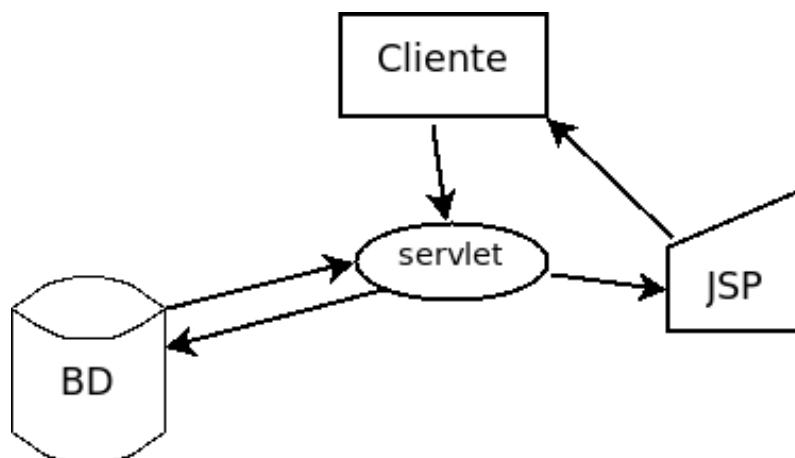
Repare que, no final do nosso método, misturamos o código HTML com Java. O que queremos extrair do código acima é justamente essa última linha (ou mais linhas em um caso de HTML maior).

Seria muito mais interessante para o programador e para o designer ter um arquivo JSP chamado contato-adicionado.jsp apenas com o HTML:

```
<html>
    Ok
</html>
```

A ferramenta que buscamos é um redirecionador de requisições, capaz de enviar uma requisição para um novo recurso do servidor: um jsp, uma servlet, uma imagem etc.

Podemos usar o recurso de **dispatch das requisições** para que o JSP só seja chamado depois de que suas regras foram executadas.



## 12.2 - Request dispatchers

Poderíamos melhorar a nossa aplicação se trabalhássemos com o código Java na servlet e depois o código HTML em uma página JSP.

A API da servlet nos permite fazer tal redirecionamento. Basta conhecermos a URL que queremos acessar e podemos usar o que fora chamado de RequestDispatcher para acessar outro recurso web, seja esse recurso uma página jsp ou uma servlet:

```
RequestDispatcher rd = request.getRequestDispatcher("/contato-adicionado.jsp");
rd.forward(request, response);
```

Agora podemos facilmente executar a lógica de nossa aplicação web em uma servlet e então redirecionar para uma página jsp, onde você possui seu código html.

### Forward e include

O método `forward` só pode ser chamado quando nada for escrito para a saída. No momento que algo for escrito fica impossível redirecionar o usuário pois o protocolo http não possui meios de voltar atrás naquilo que já foi enviado ao cliente.

Existe outro método da classe `RequestDispatcher` que representa a inclusão de página e não o redirecionamento. Esse método se chama `include` e pode ser chamado a qualquer instante para acrescentar ao resultado de uma página os dados de outra.

Apesar dos dois métodos parecerem úteis eles não costumam ser usados a não ser na parte de controlador de uma aplicação web utilizando o padrão MVC.

## 12.3 - Exercícios

- 1) Altere sua servlet `AdicionaContatoServlet` para que, após a execução da lógica de negócios, o fluxo da requisição seja redirecionado para um JSP chamado `contato-adicionado.jsp`:

- a) Substitua as linhas:

```
PrintWriter writer = response.getWriter();
writer.println("<html>");
writer.println("Contato Adicionado");
writer.println("</html>");
```

por:

```
RequestDispatcher rd = request.getRequestDispatcher("/contato-adicionado.jsp");
rd.forward(request, response);
```

- b) Faça o arquivo `contato-adicionado.jsp` na pasta WebContent.

```
<html>
Contato Adicionado
</html>
```

- 2) Teste a url: `http://localhost:8080/jspteste/testa-adiciona-contato.jsp`



The figure consists of two screenshots of a web browser window. Both screenshots show the same URL: `http://localhost:8080/jspteste/testa-adiciona-contato.jsp`.  
The top screenshot shows a form with three text input fields and one button:  
- Nome: João  
- E-mail: joao@caelum.com.br  
- Endereço: R Vergueiro  
- Enviar button  
The bottom screenshot shows the result of the form submission, displaying the text "Contato Adicionado".

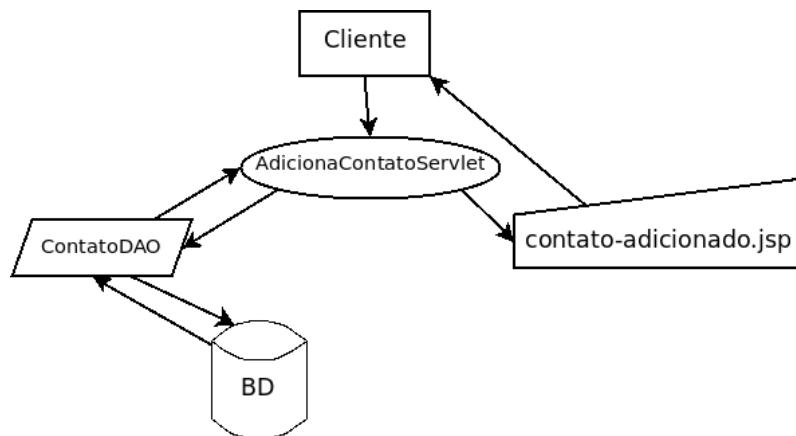
## Resultado

Perceba que já atingimos um resultado que não era possível anteriormente.

Muitos projetos antigos que foram escritos em Java, utilizavam somente jsp ou servlets e o resultado era assustador. Com o conteúdo mostrado até esse momento, é possível escrever um código com muito mais qualidade do que esses projetos.

### 12.4 - Melhorando o processo

Aqui temos várias servlets acessando o banco de dados, trabalhando com os DAOs e pedindo para que o JSP apresente esses dados, o diagrama a seguir mostra a representação do AdicionaContatoServlet após a modificação do exercício anterior.



Agora temos o problema de ter muitas servlets. Para cada lógica de negócios, teríamos uma servlet diferente, que significa oito linhas de código no web.xml... algo abominável em projeto de verdade. Imagine dez classes de modelo, cinco lógicas diferentes, isso totaliza quatrocentas linhas de configuração.

Sabemos da existência de ferramentas para gerar tal código automaticamente, mas isso não resolve o problema da complexidade de administrar tantas servlets.

Utilizaremos uma idéia que diminuirá a configuração para apenas oito linhas: colocar tudo numa Servlet só, e de acordo com que argumentos o cliente nos passa, decidimos o que executar. Teríamos aí uma Servlet monstro.

```

public class TesteServlet extends HttpServlet{

    protected void service(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        String businessLogic = request.getParameter("business");
        ContatoDAO dao;

        try {
            dao = new ContatoDAO();
            if(businessLogic.equals("AdicionaContato")) {
                Contato contato = new Contato();
                contato.setNome(request.getParameter("nome"));
                contato.setSobrenome(request.getParameter("sobrenome"));
                contato.setEmail(request.getParameter("email"));
                contato.setTelefone(request.getParameter("telefone"));
                contato.setEndereco(request.getParameter("endereco"));
                contato.setCidade(request.getParameter("cidade"));
                contato.setEstado(request.getParameter("estado"));
                contato.setBairro(request.getParameter("bairro"));
                contato.setNumero(request.getParameter("numero"));
                contato.setComplemento(request.getParameter("complemento"));
                contato.setCep(request.getParameter("cep"));

                dao.adicionar(contato);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
  
```

```
        contato.setEndereco(request.getParameter("endereco"));
        contato.setEmail(request.getParameter("email"));
        dao.adiciona(contato);
    } else if (businessLogic.equals("ListaContato")) {
        // algo como dao.lista();
    }
} catch (SQLException e) {
    throw new ServletException(e);
}
}
```

Para cada ação teríamos um if / else if, ficaria gigante, não?

Podemos melhorar fazendo refactoring de extrair métodos. Mas continuaria gigante.

Seria melhor colocar cada regra de negócio (como inserir aluno, remover aluno, fazer relatório de um aluno, etc...) em uma classe separada. Cada ação (regra de negócio) em nossa aplicação estaria em uma classe.

Então vamos extrair classes:

```
if (businessLogic.equals("AdicionaContato")) {  
    new AdicionaContato().execute(request,response);  
} else if (businessLogic.equals( "ListaContato")) {  
    new ListaContatos().execute(request,response);  
}
```

Porém, a cada lógica nova, lógica removida, alteração etc, temos que alterar essa servlet. Isso é trabalhoso e muito propenso a erros humanos.

Rpare que o código acima é um switch! E switch em Java é tão ruim que substituímos por polimorfismo, como veremos a seguir.

Vamos tentar generalizar então, queremos executar o seguinte código:

```
String business = request.getParameter("business");
new business().execute(request,response);
```

Entretanto não podemos, pois `business` é o nome de uma variável. O nosso problema é que só sabemos o que vamos instanciar em tempo de execução e não em tempo de compilação.

Temos como fazer isso? Sim.

```
String businessLogicClassName = "br.com.caelum.mvc." + request.getParameter("business");
Class businessLogicClass = Class.forName(businessLogicClassName);
```

Mas e agora como instanciar essa classe?

```
Object obj = businessLogicClass.newInstance();
```

E como chamar o método `execute?` Repare que usamos o mesmo método em todas as lógicas de negócio: isso é um padrão que definimos. Quando isso aparece, é normal extrair uma interface comum a todas essas classes: `BusinessLogic`.

```
BusinessLogic businessLogicObject = (BusinessLogic)businessLogicClass.newInstance();
businessLogicObject.execute(request, response);
```

Alguém precisa controlar então que ação será executada para cada requisição, e que JSP será utilizado. Podemos usar uma servlet para isso, e então ela passa a ser a servlet controladora da nossa aplicação, chamando a ação correta e fazendo o dispatch para o JSP desejado.

Repare na figura a seguir que, apesar dos JSPs não estarem acessando a parte da sua modelagem, isto é, as classes que direta ou indiretamente mexem no banco de dados, ele tem uma referência a um usuário, para poder colocar suas informações na página resultante. Chamamos isso de “push” das informações.

## 12.5 - Retomando o *design pattern Factory*

Note que o método `forName` da classe `Class` retorna um objeto do tipo `Class`, mas esse objeto é novo? Foi reciclado através de um cache desses objetos?

Repare que não sabemos o que acontece exatamente dentro do método `forName`, mas ao invocá-lo e a execução ocorrer com sucesso, sabemos que a classe que foi passada em forma de `String` foi lida e inicializada dentro da virtual machine.

Na primeira chamada a `Class.forName` para determinada classe, ela é inicializada. Já em uma chamada posterior, `Class.forName` devolve a classe que já foi lida e está na memória, tudo isso sem que afete o nosso código.

Esse exemplo do `Class.forName` é ótimo para mostrar que **qualquer** coisa que isola a instanciação através de algum recurso diferente do construtor é uma **factory**.

# Construindo um Framework MVC

*“Há duas tragédias na vida. Uma é a de não obter tudo o que se deseja ardenteamente; a outra, a de obtê-lo.”*  
— Bernard Shaw

Não existe mistério por trás de um framework MVC. Vamos criar aqui o **SYP MVC Framework** – Simple Yet Powerful MVC Framework.

## 13.1 - Nossa interface de execução

Para o nosso *pattern* de comando, que todas as lógicas irão seguir, definiremos a seguinte interface de execução:

```
public interface BusinessLogic {
    void execute(HttpServletRequest req, HttpServletResponse res) throws Exception;
}
```

Parece uma servlet certo? A primeira vista sim, mas perceba que não tem nada com uma servlet. Estende Servlet? Possui método chamado service? Não.

Vamos criar uma lógica de negócio de exemplo para testá-la em breve:

```
public class TestaMVC implements BusinessLogic {
    public void execute(HttpServletRequest req, HttpServletResponse res)
        throws Exception {

        System.out.println("Executando a logica e redirecionando...");
        RequestDispatcher rd = req.getRequestDispatcher("/testa-mvc.jsp");
        rd.forward(req, res);
    }
}
```

## 13.2 - Exercícios

- 1) Crie a sua interface no pacote br.com.caelum.mvc:

```
package br.com.caelum.mvc;

public interface BusinessLogic {
    void execute(HttpServletRequest req, HttpServletResponse res) throws Exception;
}
```

2) Crie uma implementação da interface BusinessLogic, nossa classe TestaMVC, no mesmo pacote:

```
package br.com.caelum.mvc;

public class TestaMVC implements BusinessLogic {
    public void execute(HttpServletRequest req, HttpServletResponse res) throws Exception {
        System.out.println("Executando a logica e redirecionando...");

        RequestDispatcher rd = req.getRequestDispatcher("/mvc-ok.jsp");
        rd.forward(req, res);
    }
}
```

### 13.3 - Criando um controlador e um pouco mais de reflection

Nosso objetivo é que o nome da classe que implementa a interface BusinessLogic seja passada como parâmetro por HTTP. Esse argumento será o business. Nossa servlet controladora é responsável por instanciar essa classe e chamar o seu método execute.

Vamos começar com a declaração da servlet e pegar o nome da classe como parâmetro:

```
public class ControllerServlet extends HttpServlet {

    protected void service(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        String business = request.getParameter("business");
        String className = "br.com.caelum.mvc." + business;
```

Nesse momento, temos o nome da classe que precisamos instanciar.

Quando vimos **JDBC** aprendemos que a classe Class possui um método estático chamado forName que carrega uma determinada classe. Além de carregar a classe, o forName devolve uma referência a um objeto do tipo Class que representa aquela classe. É isso que faremos:

```
Class clazz;
try {
    clazz = Class.forName(className);
} catch (ClassNotFoundException e) {
    throw new ServletException("Não encontro a classe " + className);
}
```

Essa String className deve ser precedida do nome do pacote ao qual essa classe pertence, como “**br.com.caelum.mvc.AdicionaContato**” por exemplo. Caso a classe não for encontrada, uma exceção ClassNotFoundException é disparada.

Um objeto do tipo Class possui um método newInstance, que tenta instanciar um objeto daquele tipo usando o construtor público que não recebe nenhum argumento da classe. Caso o construtor não exista, não seja público, ou lance uma Exception, uma exceção será lançada:

```
BusinessLogic businessLogic;
try {
    businessLogic = (BusinessLogic) clazz.newInstance();
```

```
} catch (InstantiationException e) {
    throw new ServletException(e);
} catch (IllegalAccessException e) {
    throw new ServletException(e);
}
```

Repare que **estamos instanciando um objeto para cada nova requisição**, o que deixa o usuário do nosso framework livre da preocupação de estar sendo acessado por mais de uma *Thread* ao mesmo tempo. Isso possibilita o uso de atributos de instância sem maiores problemas, além de tirar o sabor estático/procedural das servlets.

Tendo em mãos o objeto que representa a nossa lógica de negócios a ser executada, basta chamarmos o método de execução da nossa interface, e nos prepararmos para qualquer problema:

```
try {
    businessLogic.execute(request, response);
} catch (Exception e) {
    throw new ServletException("A lógica de negócios causou uma exceção", e);
}
```

E, assim, termina o nosso controlador que, agora, delegou a responsabilidade de execução para a nossa lógica de negócios.

O código completo com todo o tratamento de erro fica bem grande, por isso usaremos algo simplificado no exercício.

```
public class ControllerServlet extends HttpServlet {

    protected void service (HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        String business = request.getParameter("business");
        String className = "br.com.caelum.mvc." + business;

        try {
            Class clazz = Class.forName(className);

            BusinessLogic businessLogic = (BusinessLogic) clazz.newInstance();
            businessLogic.execute(request, response);

        } catch (Exception e) {
            throw new ServletException("A lógica de negócios causou uma exceção", e);
        }
    }
}
```

**Mais verificações**

Antes de instanciar a classe, talvez seja interessante verificar se essa classe realmente implementa a nossa interface `BusinessLogic`. Caso não implemente, disparamos uma exceção:

```
if (!BusinessLogic.class.isAssignableFrom(clazz)) {  
    throw new ServletException("classe não implementa a interface: " + className);  
}
```

## 13.4 - Configurando o web.xml

Vamos configurar para que o **Controller** responda a URL/MVC:

```
<web-app>  
    <servlet>  
        <servlet-name>syp</servlet-name>  
        <servlet-class>br.com.caelum.mvc.ControllerServlet</servlet-class>  
    </servlet>  
  
    <servlet-mapping>  
        <servlet-name>syp</servlet-name>  
        <url-pattern>/mvc</url-pattern>  
    </servlet-mapping>  
</web-app>
```

## 13.5 - Exercícios

- 1) Crie sua servlet chamada `ControllerServlet` no pacote `br.com.caelum.mvc`:

```
package br.com.caelum.mvc;  
  
// imports...  
  
public class ControllerServlet extends HttpServlet {  
  
    protected void service(HttpServletRequest request, HttpServletResponse response)  
        throws ServletException, IOException {  
  
        String business = request.getParameter("business");  
        String className = "br.com.caelum.mvc." + business;  
  
        try {  
  
            Class clazz = Class.forName(className);  
  
            BusinessLogic businessLogic = (BusinessLogic) clazz.newInstance();  
            businessLogic.execute(request, response);  
  
        } catch (Exception e) {  
            throw new ServletException("A lógica de negócios causou uma exceção", e);  
        }  
    }  
}
```

```
}
```

```
}
```

- 2) Mapeie a URL /mvc para essa servlet no arquivo web.xml.

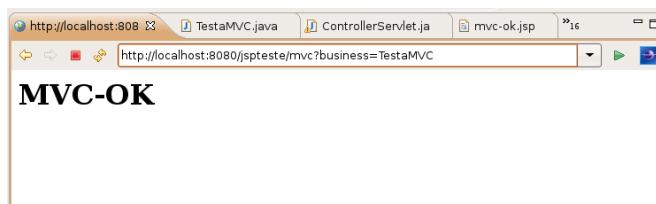
```
<servlet>
    <servlet-name>syp</servlet-name>
    <servlet-class>br.com.caelum.mvc.ControllerServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>syp</servlet-name>
    <url-pattern>/mvc</url-pattern>
</servlet-mapping>
```

- 3) Faça um arquivo jsp chamado mvc-ok.jsp:

```
<html>
    <h1> MVC-OK </h1>
</html>
```

- 4) Teste a url <http://localhost:8080/jspteste/mvc?business=TestaMVC>



## 13.6 - Erros comuns

O erro mais comum é esquecer de concatenar o nome do pacote para adquirir o nome da classe. Isso causa uma `ClassNotFoundException`.

Outro erro similar bastante comum consiste em esquecer o último ponto entre o pacote e o nome da classe:

```
String className = "br.com.caelum.mvc." + business;
```

## 13.7 - Exercícios

- 1) Crie uma nova classe chamada `AlteraContatoLogic` no mesmo pacote `br.com.caelum.mvc`. Devemos implementar a interface `BusinessLogic` e durante sua execução adicione um contato no banco.

```
//import aqui CTRL + SHIFT + O

public class AlteraContatoLogic implements BusinessLogic {

    public void execute(HttpServletRequest request, HttpServletResponse response)
        throws Exception {
```

```
System.out.println("Executando a logica e redirecionando...");

Contato contato = new Contato();
long id = Long.parseLong(request.getParameter("id"));
contato.setId(id);
contato.setNome(request.getParameter("nome"));
contato.setEndereco(request.getParameter("endereco"));
contato.setEmail(request.getParameter("email"));

ContatoDAO dao = new ContatoDAO();
dao.altera(contato);

RequestDispatcher rd = request.getRequestDispatcher("/lista-elegante.jsp");
rd.forward(request, response);
System.out.println("Alterando contato ..." + contato.getNome());
}

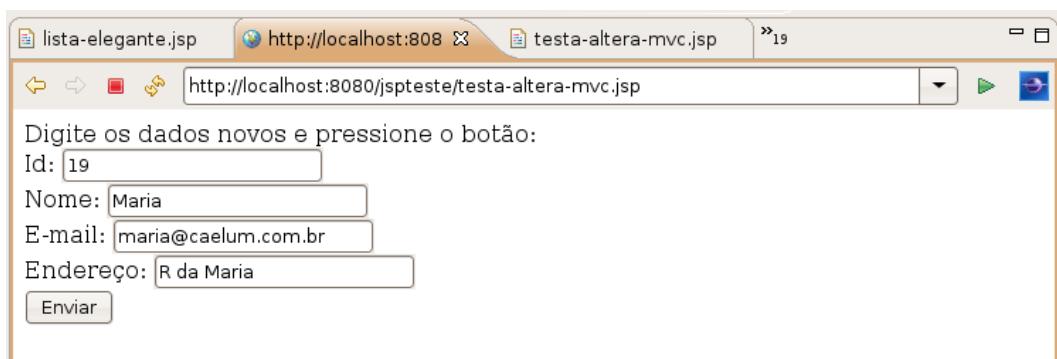
}
```

- 2) Crie um formulário chamado /testa-altera-mvc.jsp através do método POST que chama a lógica criada no exercício anterior.

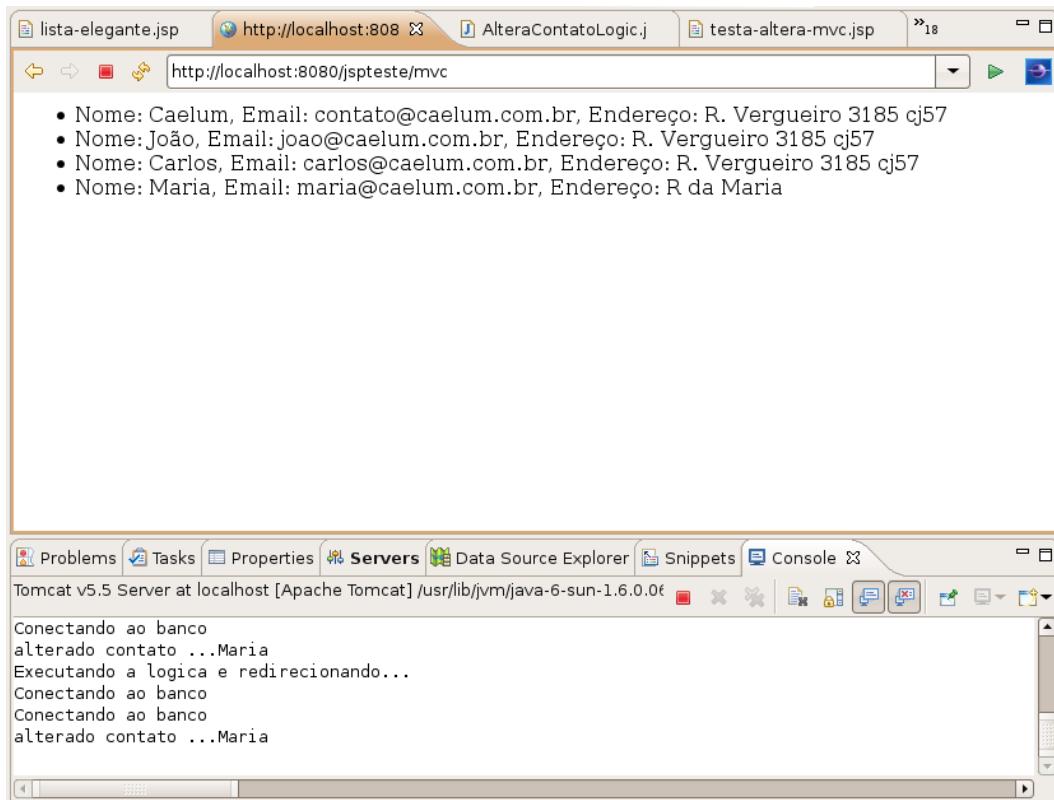
```
<html>
    <body>
        Digite os dados novos e pressione o botão:<br/>

        <form action="mvc" method="POST">
            Id:      <input type="text" name="id"/><br/>
            Nome:    <input type="text" name="nome"/><br/>
            E-mail:  <input type="text" name="email"/><br/>
            Endereço: <input type="text" name="endereco"/><br/>
                        <input type="hidden" name="business" value="AlteraContatoLogic"/>
                        <input type="submit" value="Enviar"/>
        </form>
    </body>
</html>
```

- 3) Teste a url <http://localhost:8080/jsp teste/testa-altera-mvc.jsp>



E olhe seu console:



## 13.8 - Exercícios opcionais

- 1) Crie uma lógica chamada `RemoveContatoLogic` e teste a mesma através de um link na listagem que você criou na aula de servlets.
- 2) Coloque um link na sua `lista-elegante.jsp` que abre a página `testa-altera-mvc.jsp` passando o Id do contato que você quer alterar. Deixe o campo Id visível no form mas não alterável. Não esqueça de passar o campo Id pela requisição. Faça com que os campos do form estejam populados com os dados do contato a ser editado.
- 3) Crie a lógica de adicionar contatos (`AdicionaContatoLogic`). Repare que ela é bem parecida com a `AlterarContatoLogic`. Crie um formulário de adição de novo contato. Coloque um link para adicionar novos contatos dentro do `lista-elegante.jsp`.
- 4) **Desafio:** As lógicas de adição e de alteração ficaram muito parecidas. Tente criar uma versão de uma dessas lógicas que faça essas duas coisas. Dica: A única diferença entre as duas é a presença ou não do parâmetro `Id`.

## 13.9 - Model View Controller

Generalizando o modelo acima, podemos dar nomes a cada uma das partes dessa nossa arquitetura. Quem é responsável por apresentar os resultados na página web é chamado de Apresentação (**View**).

A servlet (e auxiliares) que faz os dispatchs para quem deve executar determinada tarefa é chamada de Controladora (**Controller**).

As classes que representam suas entidades, e as que te ajudam a armazenar e buscar os dados, são chamadas de **Modelo (Model)**.

Esses três formam um padrão arquitetural chamado de **MVC**, ou **Model View Controller**. Ele pode sofrer variações de diversas maneiras. O que o MVC garante é a separação de tarefas, facilitando assim a reescrita de alguma parte, e a manutenção do código.

O famoso **Struts** ajuda você a implementar o **MVC**, pois tem uma controladora já pronta, com uma série de ferramentas para te auxiliar. O **Hibernate** pode ser usado como **Model**, por exemplo. E como **View** você não precisa usar só **JSP**, pode usar a ferramenta **Velocity**, por exemplo.

### 13.10 - Lista de tecnologias: camada de controle

A seguir você encontra uma lista com diversas opções para a camada de controle e um pouco sobre cada uma delas:

- 1) **Struts Action** - o controlador mais famoso do mercado Java, é utilizado principalmente por ser o mais divulgado e com tutoriais mais acessíveis. Possui vantagens características do MVC e desvantagens que na época ainda não eram percebidas. É o controlador pedido na maior parte das vagas em Java hoje em dia. É um projeto que não terá grandes atualizações pois a equipe dele se juntou com o Webwork para fazer o Struts 2, nova versão do Struts incompatível com a primeira e totalmente baseada no Webwork.
- 2) **Webwork** - para aprender a usar o Webwork a linha de aprendizado é mais leve e curta, sendo o framework recomendado para aprendizado pela Caelum quando o aluno já estiver bem adaptado as idéias do MVC. É um projeto que não terá grandes atualizações pois a equipe dele se juntou com o Struts para fazer o Struts 2, continuação do WebWork.
- 3) **Spring framework** - uma coleção de boas idéias colocadas de maneira padronizada em um único controlador. Possui configurações complexas (e longas), mas é extremamente poderoso.
- 4) **Stripes** - um dos frameworks criados em 2005, abusa das anotações para facilitar a configuração.
- 5) **Jboss Seam** - segue as idéias do Stripes na área de anotações e é desenvolvido pelo pessoal que trabalhou também no Hibernate. Trabalha muito bem com o Java Server Faces e EJB 3.0.
- 6) **Struts Shale** - assim como o Jboss Seam, o Struts Shale faz parte da nova geração de controladores, integrando JSF com idéias dos criadores do Struts Action.
- 7) **Wicket** - controlador baseado na idéia de que todas as suas telas deveriam ser criadas através de código Java. Essa linha de pensamento é famosa e existem diversos projetos similares, sendo que é comum ver código onde instanciamos um formulário, adicionamos botões, etc como se o trabalho estivesse sendo feito em uma aplicação Swing, mas na verdade é uma página html.
- 8) **Vraptor2** - desenvolvido por profissionais da Caelum e baseado em diversas idéias dos controladores mencionados acima, o Vraptor2 usa o conceito de favorecer Convenções em vez de Configurações para minimizar o uso de XML e anotações em sua aplicação web.

### 13.11 - Lista de tecnologias: camada de visualização

A seguir você encontra uma lista com diversas opções para a camada de visualização e um pouco sobre cada uma delas:

- **JSP** - como já vimos o JavaServer Pages, temos uma boa idéia do que ele é, suas vantagens e desvantagens. O uso de *taglibs* (a JSTL por exemplo) e expression language é muito importante se você escolher JSP para o seu projeto. É a escolha do mercado hoje em dia.
- **Velocity** - um projeto antigo, no qual a EL do JSP se baseou, capaz de fazer tudo o que você precisa para a sua página de uma maneira extremamente compacta. Indicado pela Caelum para conhecer um pouco mais sobre outras opções para camada de visualização.
- **Freemarker** - similar ao Velocity e com idéias do JSP - como suporte a taglibs - o freemarker vem sendo cada vez mais utilizado, ele possui diversas ferramentas na hora de formatar seu texto que facilitam muito o trabalho do designer.
- **Sitemesh** - não é uma alternativa para as ferramentas anteriores mas sim uma maneira de criar templates para seu site, com uma idéia muito parecida com o *struts tiles*, porém genérica: funciona inclusive com outras linguagens como PHP etc.

Em pequenas equipes é importante uma conversa para mostrar exemplos de cada uma das tecnologias acima para o designer, afinal quem irá trabalhar com as páginas é ele. A que ele preferir, você usa, afinal todas elas fazem a mesma coisa de maneiras diferentes. Como em um projeto é comum ter poucos designers e muitos programadores, talvez seja proveitoso facilitar um pouco o trabalho para aqueles.

### 13.12 - MVC 2

Pronto para aprender o que é o MVC-2? Prepare-se...

O MVC 2 nada mais é que uma genial confusão generalizada. O Model-1 é o antigo padrão que os *blue-prints* (boas práticas) do JEE da sun indicava para o uso de JSP's e javabeans. Depois surgiu o MVC (idéia "emprestada" do Smalltalk), que foi logo chamado de modelo 2... MVC, Modelo 2, Modelo 2, MVC, algumas pessoas acabam falando MVC 2.

Não se engane, MVC 2, MVC e Modelo 2 é tudo a mesma coisa: Model, View, Controller.

# Struts Framework

*"A qualidade é a quantidade de amanhã"*  
— Henri Bergson

Ao término desse capítulo, você será capaz de:

- Utilizar o struts para controlar sua lógica de negócios;
- Criar atalhos para sua camada de visualização;
- Criar e configurar mapeamentos de ações e templates;
- Utilizar form beans para facilitar a leitura de formulários;
- Validar seu formulário de forma simples;
- Controlar os erros de validação do mesmo.

## 14.1 - Struts

**Struts** é um **framework** do grupo Apache que serve como o **controller** de uma arquitetura **MVC**. Apesar de ter suporte para qualquer tipo de Servlets, é focado no uso de `HttpServlets`.

Já na versão 1.2, o Struts suportava JSPs e a integração com frameworks através de plugins, como o Velocity Tools. A última versão é a série 1.3.x.

Sua documentação e .jar podem ser encontrados em:

<http://struts.apache.org>

Para acompanhar esse capítulo, é importante que você **já** tenha lido o capítulo sobre **Model View Controller**.

### Struts 1.x ou 2.x?

Estamos usando aqui no curso a última versão do Struts da série 1.x. É o framework mais famoso usado no desenvolvimento Web em Java.

Hoje, já existe a versão 2.x do Struts, baseada no antigo WebWork e com uma série de melhorias e vantagens em relação à série 1.x. Mas porque não usamos a versão 2 no curso?

Simples: o mercado - sobretudo no Brasil - ainda é muito ligado ao Struts 1.x. Poucas empresas estão usando a versão 2 em relação aos que usam a versão 1. Quando vir uma vaga de emprego pedindo apenas Struts, é da versão 1.x que estão falando.

Mas mesmo que você precise estudar a versão 2, as bases do MVC e recursos fundamentais são as mesmas em qualquer framework.

## 14.2 - Configurando o Struts

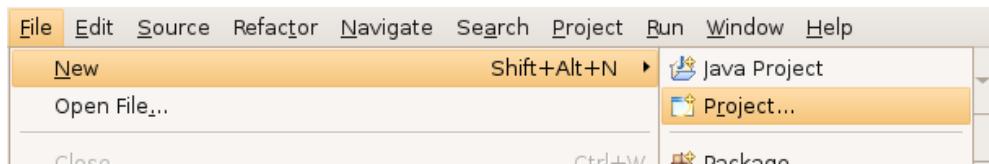
Depois de baixar o Struts do site da Apache, vamos precisar de seus jars e de suas dependências, que são muitos dos commons do Jakarta/Apache (<http://commons.apache.org>).

Descompacte o arquivo baixado em algum diretório e crie a estrutura de diretórios como se segue, o que ilustra como ficam os diretórios em uma típica aplicação que utiliza o Struts.

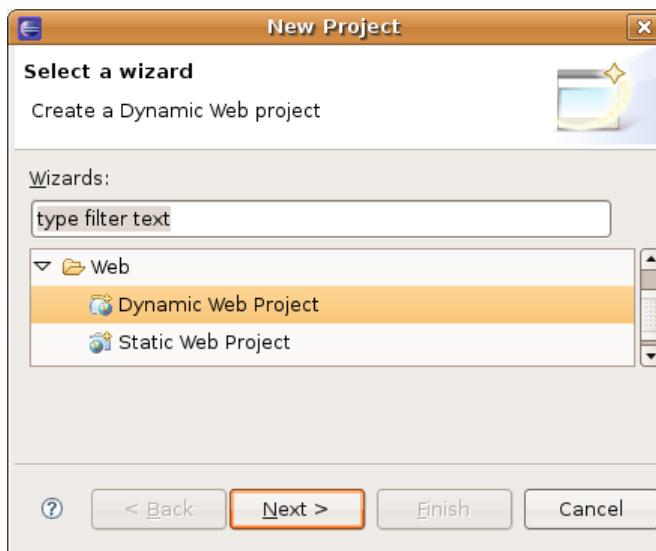
## 14.3 - Exercícios

1) Crie um projeto chamado `struts` no Eclipse, seguindo os passos abaixo:

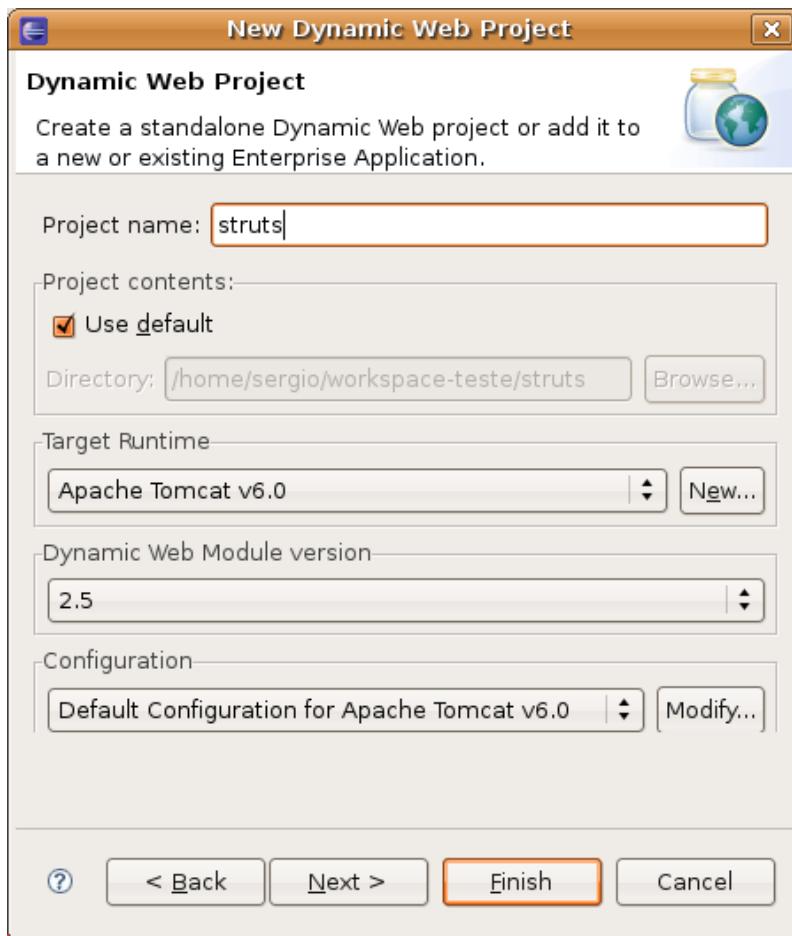
a) No Eclipse, vá em *File > New > Project*



b) Selecione *Dynamic Web project* e clique *next*



c) Preencha *Project name* com `struts` e clique em *Finish*. Deixe o tomcat 6 marcado como opção



Isso cria um novo projeto web no WTP preparado com o contexto **/struts** (isso quer dizer que acessaremos via <http://localhost:8080/struts/>)

- 2) Para fazer o Struts funcionar, precisamos colocar os JARs que baixamos direto do site deles dentro da pasta WEB-INF/lib. Na Caelum, já existe um ZIP que faz isso para nós, copia os arquivos anteriores do projeto JDBC (DAO etc), driver do MySQL, JSTL etc. Enfim, tudo que fizemos até agora mais as configurações do Struts.
  - a) No Eclipse, clique com o botão direito em cima do nome do projeto **struts** e vá em **Import**;
  - b) Selecione **General/Archive File** e clique em **Next**;
  - c) Selecione o arquivo **projeto-struts.zip** na pasta **Desktop/caelum/21**;
  - d) Clique em **Finish** e confirme qualquer overwrite de arquivo.

Dê uma navegada nas pastas e veja as configurações que foram feitas. Em especial, veja os XMLs na pasta WEB-INF.

O **web.xml** já foi configurado para nós com a servlet controladora do struts. Repare que isso é muito parecido com nosso framework do capítulo anterior. Como antes, não vamos precisar ficar configurando mais nada nesse XML.

```
<web-app>
  <servlet>
    <servlet-name>testeDeStruts</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
```

```
<load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>testeDeStruts</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
</web-app>
```

Perceba também que há outro XML, o **struts-config.xml**. Esse arquivo é um arquivo específico do Struts e é lá que colocaremos as configurações do framework. Por enquanto, ele está apenas com o esqueleto:

```
<struts-config>
</struts-config>
```

- 3) Clique da direita no nome do seu projeto, escolha *Run As > Run on Server*.  
Escolha o servidor que você já tem configurado no seu Eclipse e clique em *Finish*
- 4) O Web Browser do Eclipse irá aparecer. Teste a URL `http://localhost:8080/struts/` e você verá uma mensagem de teste.



### Para saber mais: Configurando o projeto no Tomcat sem o plugin do Eclipse

Se você não usa o plugin do WTP, não se esqueça de configurar o Projeto no Tomcat da mesma forma que vimos no capítulo de JSP! Para isso, crie um arquivo:

```
<diretorio_do_tomcat>/conf/Catalina/localhost/struts.xml
```

Coloque nele o seguinte conteúdo:

```
<Context path="/struts" docBase="/home/usuario/workspace/struts/web/" reloadable="true">
</Context>
```

Você obteve a tela seguinte?



Isso pode ter ocorrido porque algum de seus arquivos XML foi digitado errado ou porque no seu sistema não está configurado para que liste os arquivos dos diretórios. Se no seu caso foi porque não está configurado a listagem é um bom sinal, a listagem de arquivos dos diretórios pode trazer riscos ao seu sistemas em caso de usuários maliciosos.

Mas onde configurar para listar ou não meus arquivos? No arquivo `web.xml` do Tomcat que está na pasta `$CATALINA_HOME/conf/`. Mude o seguinte trecho do arquivo:

```
<init-param>
    <param-name>listings</param-name>
    <param-value>false</param-value>
</init-param>
```

Deixe `<param-value>false</param-value>` caso não queira a listagem e `<param-value>true</param-value>`, caso contrário.

## 14.4 - Uma ação Struts

No nosso exemplo anterior de MVC utilizávamos uma interface comum a todas as nossas lógicas de negócio. Com o Struts temos uma classe chamada `Action` que iremos estender para implementar nossas lógicas.

Muitos programadores recomendam como boa prática não colocar a lógica de negócio na `Action`, e sim em uma nova classe que é chamada por ela.

Você deve reescrever o método `execute`, como no exemplo abaixo:

```
package br.com.caelum.struts.action;
// imports....
```

```
public class TesteSimpleAction extends Action {
    @Override
    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
            throws Exception {
        // ...
    }
}
```

Por enquanto, temos quatro parâmetros, dois que já conhecemos, e um retorno. Esses três itens serão explicados em breve, passo a passo para evitar complicações uma vez que é nesse ponto que os programadores sentem uma certa dificuldade inicial com o Struts.

Primeiro, precisamos devolver um objeto do tipo ActionForward, que indica para onde o usuário deve ser redirecionado ao término da execução daquela ação. Pode ser para outra ação, ou, normalmente, para um .jsp.

Poderíamos retornar algo do tipo:

```
return new ActionForward("/exemplo.jsp");
```

Mas isso não é recomendado. Por quê? Uma vez que colocamos o nome do arquivo jsp na nossa camada de lógica de negócios, estamos criando uma ligação muito forte entre as mesmas. Qualquer alteração no nome do arquivo resulta em um grande esforço para encontrar todos os pontos que se referenciam a tal lugar.

Portanto, essa não será a solução que utilizaremos. Partimos para algo que o Struts facilita desde o início de seu projeto: desejamos retornar "ok"! Lembre-se disso.

É muito importante ressaltar que o Struts pode instanciar apenas uma Action de cada tipo, fazendo com que você ainda tenha de se preocupar com problemas de sincronismo, já que existe a possibilidade de existir mais de uma Thread acessando o mesmo objeto Action ao mesmo tempo.

O nosso JSP final será algo bem simples para esse exemplo:

```
<html>
    Minha primeira página usando o Struts!
</html>
```

## 14.5 - Configurando a ação no struts-config.xml

Voltamos ao arquivo struts-config.xml: esse arquivo irá mapear as URLs que os usuários acessarem (chamados de *path*) e as classes (chamadas de *type*). Sendo assim, nada mais natural e elegante do que mapearmos o path /teste para a classe TesteSimples.

**Atenção: o nome do path não precisa ser igual ao nome da classe! Isto é um mapeamento!**

```
<struts-config>

    <action-mappings>
        <action path="/teste" type="br.com.caelum.struts.action.TesteSimples">
            <forward name="ok" path="/exemplo.jsp"/>
        </action>
    </action-mappings>

</struts-config>
```

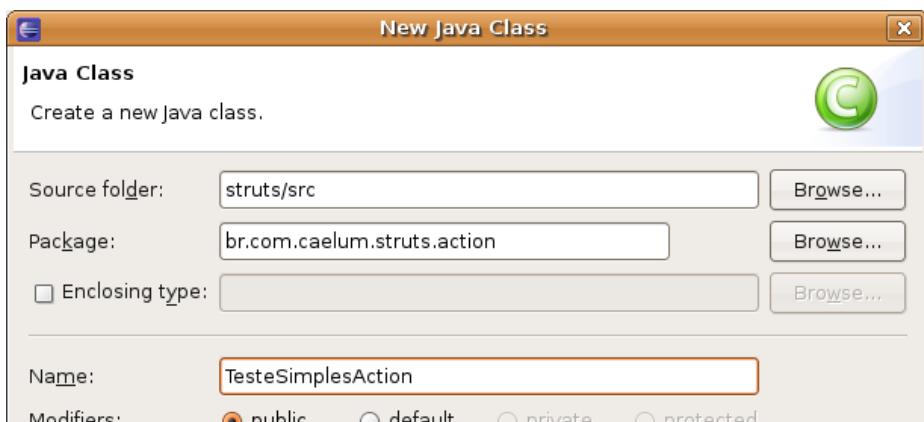
Dentro da tag *action*, colocamos uma tag *forward*. Essa tag define um redirecionamento com um apelido (atributo *name*) e o caminho (*path*). No código, quando fazemos `mapping.findForward("ok")`, o Struts procura um forward com apelido "ok" e devolve o objeto ActionForward correspondente (no caso, redirecionando para exemplo.jsp). Desta forma, podemos trabalhar com nossas lógicas sem nos atrelarmos muito à camada de visualização.

O parâmetro *path* de *action* indica qual URL vai acionar essa ação, no caso será o /teste.do, pois para acessar o struts precisamos da terminação .do no nosso caso.

## 14.6 - Exercícios: TesteSimplesAction

1) Crie sua primeira ação do Struts.

a) Crie uma classe chamada `TesteSimplesAction` no pacote `br.com.caelum.struts.action`.



b) Faça sua classe estender Action (do Struts!).

c) Utilize CTRL+SHIFT+O para importar a classe.

d) Escreva o método `execute` e implemente o conteúdo do mesmo, retornando o resultado exemplo

Você pode fazer o Eclipse gerar para você a assinatura do método. Basta escrever `execute` dentro da classe e apertar **Ctrl+espaço**. **Cuidado para gerar o método que recebe exatamente os parâmetros como mostrados abaixo.**

Agora implemente o conteúdo do método como abaixo:

```
public class TesteSimplesAction extends Action {
    @Override
    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception {
        System.out.println("Executando o código da lógica de negócios...");
        return mapping.findForward("ok");
    }
}
```

2) Crie seu arquivo `exemplo.jsp` dentro do diretório `WebContent`.

```
<html>
    Minha primeira página usando o Struts!
</html>
```

3) Abra o seu arquivo `struts-config.xml` e configure sua ação dentro da tag `action-mappings`, que vem antes do `message-resources`.

```
<struts-config>
    <action-mappings>
        <action path="/teste" type="br.com.caelum.struts.action.TesteSimplesAction">
            <forward name="ok" path="/exemplo.jsp"/>
        </action>
    </action-mappings>
</struts-config>
```

```
</action-mappings>
</struts-config>
```

- 4) Reinicie o Tomcat.
- 5) Teste a URL <http://localhost:8080/struts/teste.do>



#### Reload automático do struts-config.xml

O Struts **não** faz o reload automático do arquivo `struts-config.xml`.

Um truque para fazer isso funcionar (que só é útil durante o desenvolvimento da sua aplicação) é colocar esse arquivo no seu diretório `src`, portanto será jogado no diretório `classes`, o `classpath`. Já no seu arquivo `web.xml` configure o struts com um parâmetro de inicialização de servlet para ler o arquivo dentro do diretório `classes` (`/WEB-INF/classes/struts-config.xml`):

```
<init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/classes/struts-config.xml</param-value>
</init-param>
```

Agora, toda vez que o arquivo for alterado, o Tomcat percebe uma mudança no `classpath` do projeto e reinicia a sua aplicação web.

Cuidado pois essa funcionalidade de reinicialização de contextos pode nem sempre funcionar como você espera. Um caso simples é iniciar threads separadas e deixá-las rodando no background, o que acontece?

## 14.7 - Erros comuns

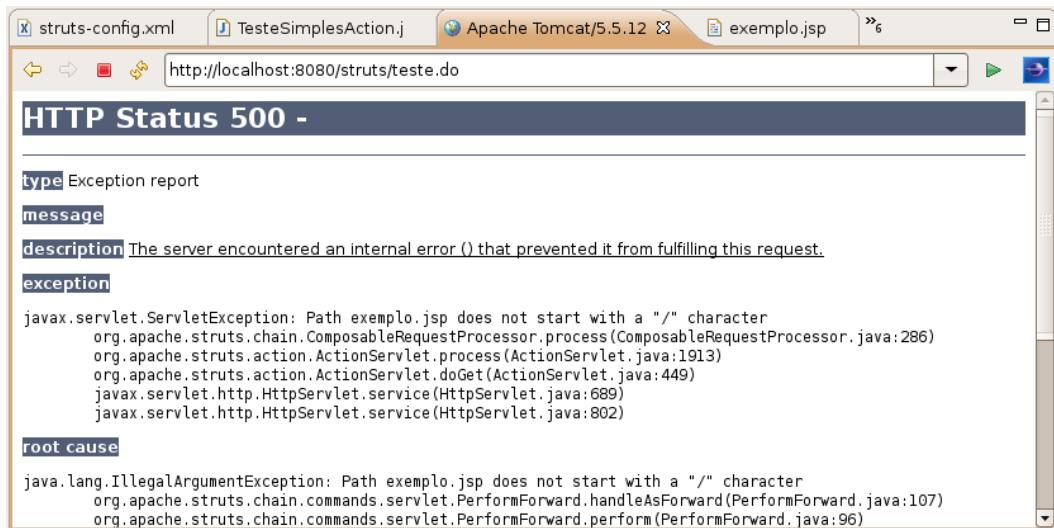
- 1) O erro mais famoso nos primeiros exemplos de uma Action do Struts é colocar o nome do forward de maneira inválida, por exemplo, em minúsculo no `struts-config.xml` e em maiúsculo na sua classe. Lembre-se, o Java é case-sensitive e assim será a maior parte de suas bibliotecas!

Como o Struts não encontra um redirecionamento com tal chave, o método `findForward` retorna `null`, resultado: uma tela em branco.

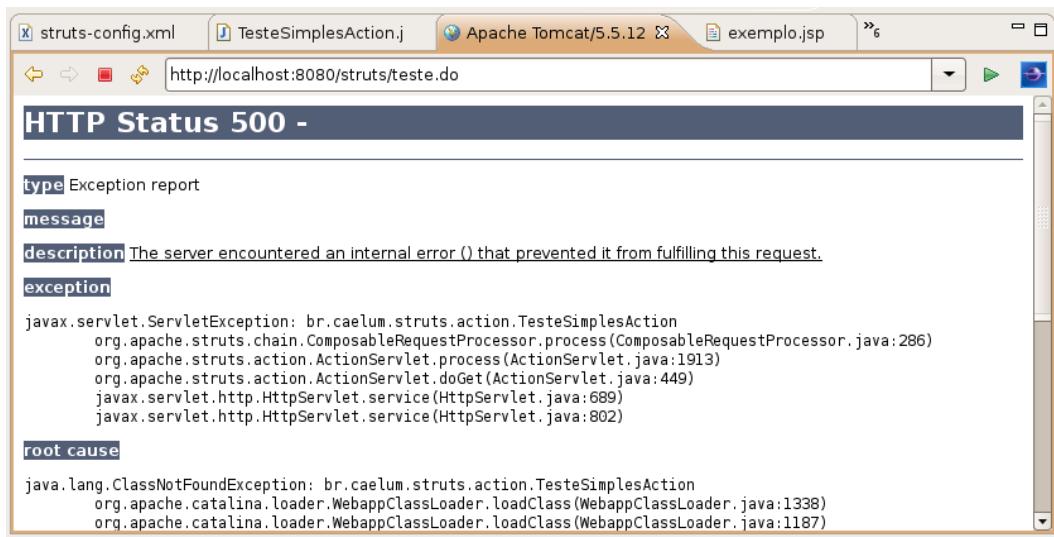


- 2) Outro erro comum é esquecer de colocar a barra antes do nome do redirecionamento. Todo path de forward deve começar com uma barra. Se você colocar somente `exemplo.jsp` o erro diz claramente que faltou uma barra:

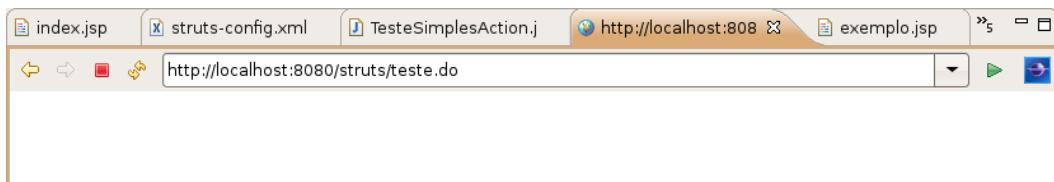
Gostou desta apostila? Conheça nossos treinamentos



- 3) É comum errar o nome da classe de sua action, como por exemplo esquecer o .com e digitar br.caelum.struts.action.TesteSimpleAction. Nesse caso o Struts não consegue instanciar sua action:



- 4) A classe Action possui dois métodos execute: um com HttpServletRequest% e %Response% e um com ServletRequest e Responseprimeiro método. Caso usemos o segundo, recebemos uma página em branco.



- 5) Por último, o erro dos esquecidos. Se você não criar o arquivo JSP ou colocar um nome inválido o erro é o já conhecido 404:



## 14.8 - Pesquisando um banco de dados

Continuando com nossa aplicação criada no capítulo anterior, iremos montar agora um esquema de simulação de acesso a um banco de dados para listar todos os contatos através do MVC e usando Struts, JSP e JDBC.

Repare que já estamos usando três camadas e três bibliotecas diferentes!

## 14.9 - Criando a ação

Para criar a ação de listagem basta utilizarmos a idéia de criar um novo objeto do tipo DAO e chamar o método de lista:

```
// pesquisa no banco de dados a lista completa
List<Contato> lista = new ContatoDAO().getLista();
```

Mas, espere um pouco, esse é o exemplo que vimos no começo da apostila? Até aqui, sem novidades. A questão que fica é como enviar o conteúdo referenciado pela variável lista para a página JSP que será acessada em breve.

Precisamos de um escopo de variável que sobreviva ao método `execute` e continue durante o forward da requisição até o arquivo JSP. Repare que a frase anterior entrega a solução: o escopo da requisição.

Atrelaremos o valor referenciado pela variável lista para um nome qualquer ligada a requisição do cliente. Esse valor só ficará lá até o término da requisição, tempo suficiente para mostrá-lo no arquivo jsp.

Podemos adicioná-la como atributo no request, para que nossa página JSP possa receber tal objeto. Suponha que desejamos chamar nossa lista de "contatos":

```
request.setAttribute("contatos", lista);
```

E o redirecionamento é simples:

```
return mapping.findForward("lista");
```

Portanto, o código final de nossa ação é:

```
package br.com.caelum.struts.action;
// imports aqui

public class ListaContatosAction extends Action {
```

```
public ActionForward execute(ActionMapping mapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse response)
    throws Exception {
    // pesquisa no banco de dados a lista completa
    List<Contato> lista = new ContatoDAO().getLista();
    request.setAttribute("contatos", lista);

    // ok.... para onde ir agora?
    return mapping.findForward("lista");
}
```

## 14.10 - O arquivo WebContent/lista.jsp

Para criarmos o JSP de listagem, temos três opções. A primeira seria escrever o código através de scriptlets, que já vimos no capítulo de JSP: não é uma boa solução.

A segunda opção é utilizar a biblioteca de tags de lógica do Struts, a struts-logic, que funciona e é uma alternativa.

A terceira, é utilizar JSTL. Qual a diferença entre a struts-logic e a JSTL core? Acontece que a biblioteca do Struts veio antes da JSTL: a JSTL é a tentativa de padronizar essas taglibs que apareceram pelo mundo inteiro. Sendo assim, todos, inclusive o grupo Apache, estão migrando para a JSTL. Por essa razão, seguiremos no curso apenas com a JSTL, que é a biblioteca padrão.

Como fizemos antes, primeiro devemos declarar a variável, que está sendo lida do `request`. Logo depois iteramos por todos os itens:

```
<!-- for -->
<c:forEach var="contato" items="${contatos}">
    ${contato.id} - ${contato.nome} <br/>
</c:forEach>
```

Portanto, o arquivo final, com cabeçalho e tudo o que faltava, fica sendo:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>

<!-- for -->
<c:forEach var="contato" items="${contatos}">
    ${contato.id} - ${contato.nome} <br/>
</c:forEach>

</html>
```

Nesse momento, você se pergunta: mas o JSP não declarou a variável `contatos`?! Exato, ele não declarou. A expression language buscará o valor de tal chave no `request` (e em outros lugares, que veremos adiante no curso) e utilizá-la para a iteração, ou seja ele não tem ligação direta com o DAO, ele sabe que vem uma variável `contatos`, mas não sabe de onde.

## 14.11 - ListaContatos no struts-config.xml

Por fim, vamos alterar o struts-config.xml para configurar nossa ação:

```
<action path="/listaContatos" type="br.com.caelum.struts.action.ListaContatosAction">
    <forward name="lista" path="/lista.jsp"/>
</action>
```

Portanto, para testarmos nossa aplicação, devemos reiniciar o Tomcat e utilizar o link /listaContatos.do.

Repare que agora não faz mais sentido acessar o JSP de listagem diretamente pois a variável não existe!

## 14.12 - Exercício: ListaContatosAction

Vamos criar sua listagem de contatos:

1) Crie sua classe de lógica ListaContatosAction no mesmo pacote br.com.caelum.struts.action:

- a) Lembre-se de estender a classe Action
- b) Implemente o método execute:

```
@Override
public ActionForward execute(ActionMapping mapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse response)
    throws Exception {
    // pesquisa no banco de dados a lista completa
    List<Contato> lista = new ContatoDAO().getLista();
    request.setAttribute("contatos", lista);

    // ok.... para onde ir agora?
    return mapping.findForward("lista");
}
```

2) Configure o struts-config.xml

```
<action path="/listaContatos" type="br.com.caelum.struts.action.ListaContatosAction">
    <forward name="lista" path="/lista.jsp"/>
</action>
```

3) Crie seu JSP de resultado WebContent/lista.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>

    <!-- for -->
    <c:forEach var="contato" items="${contatos}">
        ${contato.id} - ${contato.nome} <br/>
    </c:forEach>

</html>
```

4) Reinicie o Tomcat.

5) Teste a URL <http://localhost:8080/struts/listaContatos.do>



6) O que acontece se acessarmos diretamente o JSP? O que estamos fazendo de errado? Teste a URL <http://localhost:8080/struts/lista.jsp>

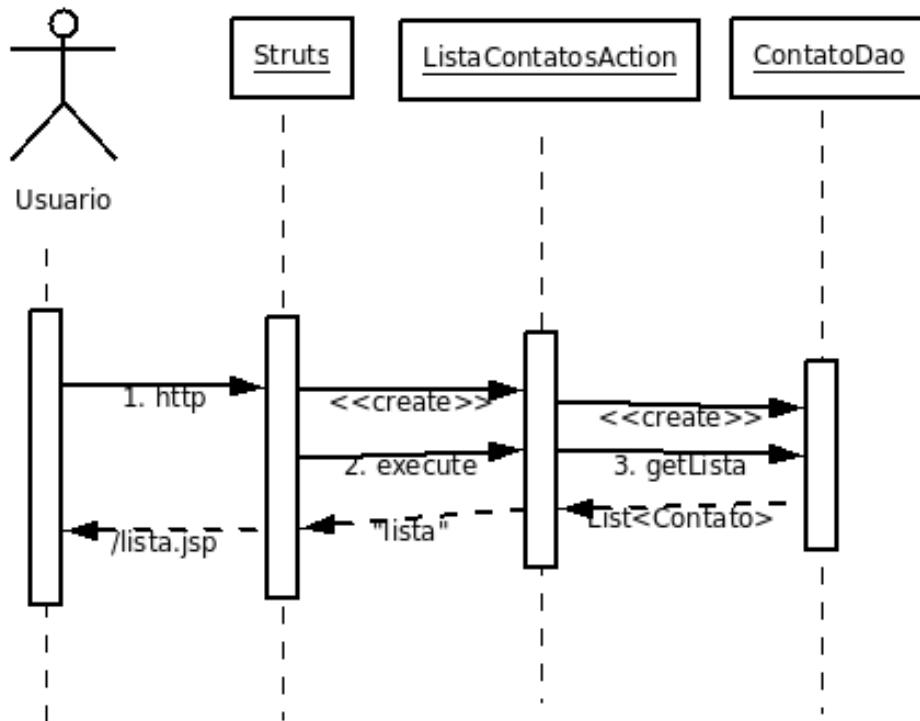


Neste momento, seu arquivo struts-config.xml possui duas actions:

```
<struts-config>
  <action-mappings>
    <action path="/teste" type="br.com.caelum.struts.action.TesteSimplesAction">
      <forward name="ok" path="/exemplo.jsp"/>
    </action>

    <action path="/listaContatos"
           type="br.com.caelum.struts.action.ListaContatosAction">
      <forward name="lista" path="/lista.jsp"/>
    </action>
  </action-mappings>
</struts-config>
```

O seguinte diagrama descreve o que acontece com o nosso sistema ao requisitar a listagem de contatos:



### 14.13 - Resultado condicional com o Struts

Como fazer para mostrar a mensagem “Nenhum contato fora encontrado”?

A primeira idéia é a de colocar um if dentro do seu JSP e resolver o problema, certo? Mas isso só trará problemas para o designer, que não sabe tanto de lógica quanto você e pode ser que o editor que ele usa não suporte tais tipos de lógicas...

Então, a melhor saída é verificar, ainda dentro de sua ação, se o banco de dados retornou uma coleção de tamanho zero. E, nesse caso, redirecionar para outra página.

### 14.14 - Exercícios: listagem vazia

- 1) Crie um JSP novo chamado `lista-vazia.jsp` na pasta `WebContent`.

```

<html>
    Você não possui nenhum contato.
</html>

```

- 2) Alterando somente sua lógica e adicionando um novo forward, quando a lista estiver vazia, a página `lista-vazia.jsp` seja mostrada:

```

// pesquisa no banco de dados a lista completa
List<Contato> lista = new ContatoDAO().getList();
request.setAttribute("contatos", lista);

// ok.... para onde ir agora?
if(lista.isEmpty()) {

```

```

        return mapping.findForward("vazia");
    } else {
        return mapping.findForward("lista");
    }
}

```

- 3) Altere seu struts-config.xml para adicionar um novo forward para a lista-vazia:

```

<action path="/listaContatos" type="br.com.caelum.struts.action.ListaContatosAction">
    <forward name="lista" path="/lista.jsp"/>

    <!-- adicionar a linha a seguir -->
    <forward name="vazia" path="/lista-vazia.jsp"/>
</action>

```

- 4) Teste a sua listagem com o banco de dados vazio. Para simular a lista vazia, você pode, por exemplo, chamar o método lista.clear() ou remover todos os seus contatos do banco.



## 14.15 - Resultado do struts-config.xml

Neste momento, seu arquivo struts-config.xml possui duas actions:

```

<struts-config>
    <action-mappings>
        <action path="/teste" type="br.com.caelum.struts.action.TesteSimplesAction">
            <forward name="ok" path="/exemplo.jsp"/>
        </action>

        <action path="/listaContatos"
               type="br.com.caelum.struts.action.ListaContatosAction">
            <forward name="lista" path="/lista.jsp"/>
            <forward name="vazia" path="/lista-vazia.jsp"/>
        </action>
    </action-mappings>
</struts-config>

```

## 14.16 - Novos contatos

Agora, já estamos prontos para criar a lógica de negócios e a camada de visualização para permitir adicionar novos clientes e, consequentemente, listá-los.

Como de costume, seguiremos os passos:

- 1) Criar a lógica de negócios;

- 2) Criar o JSP de visualização;
- 3) Criar o mapeamento da lógica para a visualização;  
E depois os passos opcionais:
- 4) Criar a validação do formulário na lógica de negócios
- 5) Criar o controle de erro na visualização

## 14.17 - Formulário

Nunca é elegante trabalhar com o método `getParameter` do `request`, já que é muito melhor trabalhar com classes que nós mesmos escrevemos. Assim, vamos imaginar um cenário simples: desejamos adicionar o nome, email e descrição do cliente.

O Struts possui uma classe chamada `ActionForm` que ao ser estendida permite ler os parâmetros do `request` sem nos preocuparmos com o mesmo!

Sendo assim, no Struts, para cada formulário HTML que existe no nosso site, criamos uma classe em Java para representar os campos do mesmo.

No nosso caso, precisamos dos campos `nome`, `email` e `endereco`, mas opa, isso é um `Contato`! Resultado:

```
package br.com.caelum.struts.form;

import org.apache.struts.action.*;

public class ContatoForm extends ActionForm {
    private Contato contato = new Contato();
    public Contato getContato() {
        return this.contato;
    }
}
```

Atenção: o formulário HTML deverá ter os campos com o mesmo nome que as variáveis membro do seu formulário!

Existe uma opção avançada de fazer o formulário através de xml, não deixa de ser bastante código e ainda com a desvantagem de não mostrar erros de compilação.

## 14.18 - Mapeando o formulário no arquivo struts-config.xml

Assim como a `action`, devemos configurar nosso `form` no arquivo `struts-config.xml`. Para isso, usamos a tag chamada `form-bean`.

Atributos de uma tag `form-bean`:

**name:** um nome qualquer que queremos dar a um formulário; **type:** a classe que representa esse formulário.

**Atenção:** tal tag vem antes das definições dos `action-mappings`! Todos os formulários devem ser definidos dentro de uma única tag `form-beans`.

```
<form-beans>
    <form-bean name="ContatoForm" type="br.com.caelum.struts.form.ContatoForm"/>
</form-beans>
```

## 14.19 - Exercício: ContatoForm

1) O primeiro passo é criar o formulário como classe:

- a) Crie a classe ContatoForm no pacote br.com.caelum.struts.form.
- b) Faça com que seu formulário estenda a classe `ActionForm` do Struts.

```
public class ContatoForm extends ActionForm{
}
```

- c) Coloque uma variável do tipo `Contato` no formulário, chame-a de `contato` e instancie um `Contato`:

```
public class ContatoForm extends ActionForm{
    private Contato contato = new Contato();
}
```

- d) Vá no menu Source, *Generate Getters and Setters* e escolha o método `getContato`. (ou digite Ctrl+3 e escreva `ggas`).

```
public class ContatoForm extends ActionForm{
    private Contato contato = new Contato();

    public Contato getContato() {
        return contato;
    }
}
```

2) Agora, vamos mapear esse formulário no `struts-config.xml`. Lembre-se que a tag `form-beans` deve vir **antes** da tag `action-mappings`.

```
<form-beans>
    <form-bean name="ContatoForm" type="br.com.caelum.struts.form.ContatoForm"/>
</form-beans>
```

## 14.20 - Erro comum

O erro mais comum com o struts está no arquivo `struts-config.xml`. Ao configurar seu primeiro `form-bean`, o aluno deve prestar muita atenção que a tag `form-beans` deve vir antes da tag `action-mappings`.

## 14.21 - Lógica de Negócios

Podemos escrever um código bem simples que adiciona um novo contato (recebido através de um formulário) para o banco de dados:

Criamos um contato, recuperamos os valores do formulário e adicionamos este cliente ao banco de dados:

```
package br.com.caelum.struts.action;

// série de imports aqui

public class AdicionaContatoAction extends Action {
    @Override
    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception {
        // log
        System.out.println("Tentando criar um novo contato...");

        // formulário de cliente
        ContatoForm formulario = (ContatoForm) form;
        // acessa o bean
        Contato contato = formulario.getContato();

        // adiciona ao banco de dados
        ContatoDAO dao = new ContatoDAO();
        dao.adiciona(contato);

        // ok.... visualização
        return mapping.findForward("ok");
    }
}
```

## 14.22 - Exercício: AdicionaContatoAction

1) Vamos criar a classe AdicionaContatoAction:

- Crie a classe AdicionaContatoAction em br.com.caelum.struts.action
- Faça com que sua classe estenda Action do Struts.
- Digite execute, CTRL+ESPAÇO e dê enter: implemente o método execute lembrando de verificar os nomes de seus argumentos:

```
public ActionForward execute(ActionMapping mapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse response)
    throws Exception {
    // log
    System.out.println("Tentando criar um novo contato...");

    // formulário de cliente
    ContatoForm formulario = (ContatoForm) form;
    // acessa o bean
    Contato contato = formulario.getContato();

    // adiciona ao banco de dados
    ContatoDAO dao = new ContatoDAO();
    dao.adiciona(contato);
```

```
// ok.... visualização  
return mapping.findForward("ok");  
}
```

## 2) Vamos configurar sua ação:

- a) Define sua ação /novoContato no arquivo struts-config.xml apontando para a classe AdicionaContatoAction.

```
<action path="/novoContato" name="ContatoForm"  
       type="br.com.caelum.struts.action.AdicionaContatoAction">  
</action>
```

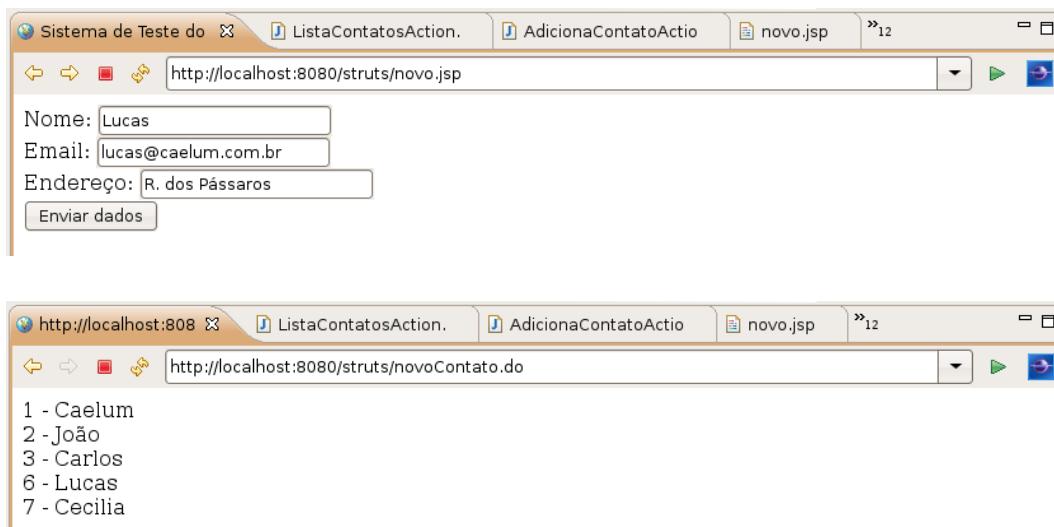
- b) Em caso de sucesso (ok), redirecione para o /listaContatos.do (isto mesmo, estamos encadeando duas ações).

```
<action path="/novoContato" name="ContatoForm"  
       type="br.com.caelum.struts.action.AdicionaContatoAction">  
    <forward name="ok" path="/listaContatos.do"/>  
</action>
```

## 3) Crie seu arquivo novo.jsp na pasta WebContent:

```
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>  
  
<html:html>  
  <head>  
    <title>Sistema de Teste do Struts</title>  
  </head>  
  
  <html:errors/>  
  
<html:form action="/novoContato" focus="contato.nome">  
  Nome:  
    <html:text property="contato.nome"/>  
    <br/>  
  
  Email:  
    <html:text property="contato.email"/>  
    <br/>  
  
  Endereço:  
    <html:text property="contato.endereco"/>  
    <br/>  
    <html:submit>Enviar dados</html:submit>  
    <br/>  
  </html:form>  
</html:html>
```

## 4) Teste a url http://localhost:8080/struts/novo.jsp



5) Agora, tente criar um contato com nome vazio, funciona?

## 14.23 - Erros comuns

A seguir veja os erros mais comuns no exercício anterior:

- 1) Esquecer de fazer seu `ContatoForm` estender `ActionForm`: como `ContatoForm` e `ActionForm` não possuem conexão, o compilador (no caso, o Eclipse), reclama do casting que está sendo feito dentro da sua classe `Action`, afinal nenhum `ActionForm` é um `ContatoForm`. Solução: estenda `ActionForm`.
- 2) Esquecer de colocar o atributo `name="ContatoForm"` na sua tag `action` dentro do `struts-config.xml`. Como você não notificou o Struts que sua `Action` precisa de um `form`, ele passa `null` para seu método e, quando chega na linha que tenta chamar o método `getContato`, acontece um `NullPointerException`. Percebeu que o `NullPointerException` foi um erro do programador? Eles normalmente são descuidos do programador, portanto sempre preste atenção naquilo que você faz e configura! Solução: vá no seu arquivo `struts-config.xml` e coloque o atributo `name="ContatoForm"` na sua tag `action`.

## 14.24 - Arquivo de mensagens

O struts possui um sistema bem simples de internacionalização.

Esse é o processo onde centralizamos todas as mensagens do sistema em um (ou mais) arquivos de configuração que são baseados na antiga idéia de chave-valor, um dicionário de mensagens. Por exemplo:

```
menu.nome = Menu principal
menu.arquivo = Arquivo
menu.editar = Editar
menu.sair = Sair
site.titulo = Sistema de teste do Struts
```

Para configurar o struts e usar um tipo de arquivo como esse, começamos indicando qual o arquivo de configuração que usaremos. O nome mais comum é `MessageResources.properties`. Esse arquivo deve ser criado no nosso diretório `src`.

Para o Struts ler tal arquivo basta configurá-lo no struts-config.xml, localizado no diretório WEB-INF:

```
<struts-config>
    <form-beans>      <!-- beans aqui -->      </form-beans>
    <action-mappings>
        <!-- actions -->
    </action-mappings>

    <!-- Arquivo de Mensagens -->
    <message-resources parameter="MessageResources" />
</struts-config>
```

Para utilizar tal arquivo é bem simples, basta no nosso JSP usar uma taglib do struts chamada bean:

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
```

Dada uma chave (`menu.nome` por exemplo), podemos chamar a tag `message` que é capaz de mostrar a mensagem "Menu principal":

```
<bean:message key="site.titulo" />
```

Portanto, o exemplo a seguir mostra um menu completo usando essa taglib:

```
<html>
    <head>
        <title><bean:message key="site.titulo" /></title>
    </head>

    <body>
        <bean:message key="menu.nome" /><br/>
        <bean:message key="menu.arquivo" /><br/>
        <bean:message key="menu.editar" /><br/>
        <bean:message key="menu.sair" /><br/>
    </body>
</html>
```

## 14.25 - Exercícios: Mensagens

1) Crie um arquivo chamado testa-mensagens.jsp.

a) Inclua a taglib bean:

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
```

b) Inclua as mensagens:

```
<html>
    <head>
        <title><bean:message key="site.titulo" /></title>
    </head>

    <body>
```

```

<bean:message key="menu.nome" /><br/>
<bean:message key="menu.arquivo" /><br/>
<bean:message key="menu.editar" /><br/>
<bean:message key="menu.sair" /><br/>
</body>
</html>

```

- 2) Abra o arquivo chamado `MessageResources.properties` no seu diretório `src` e adicione:

```

# comentario de um arquivo .properties
menu.nome = Nome do menu
menu.arquivo = Escolher Arquivo
menu.editar = Editar Arquivo
menu.sair = Sair da aplicação
site.titulo = Sistema de teste do Struts

```

- 3) Adicione a configuração a seguir no final arquivo `struts-config.xml` para utilizar tal arquivo de mensagens:

```

<struts-config>

    <form-beans>
        <!-- beans aqui -->
    </form-beans>

    <action-mappings>
        <!-- actions aqui-->
    </action-mappings>

    <!-- Arquivo de Mensagens -->
    <message-resources parameter="MessageResources" />

</struts-config>

```

- 4) Reinicie o Tomcat. Será sempre necessário fazer isso ao alterar seu arquivo `struts-config.xml`.  
5) Teste a URL `http://localhost:8080/struts/testa-mensagens.jsp`



## 14.26 - Erros comuns

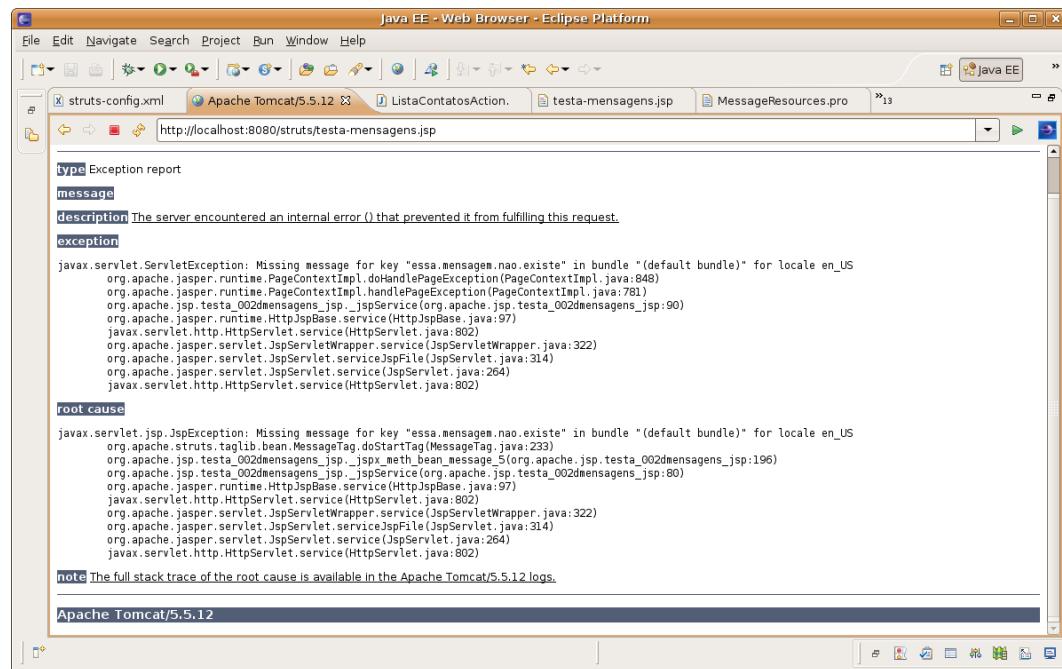
Infelizmente a maior parte dos erros possíveis no exercício anterior trazem a mesma tela de erro: código 500, incapaz de achar o valor de uma mensagem: *Cannot find message resources under key org.apache.struts.action.MESSAGE*.

- 1) Na sua página HTML, digitar o valor de uma mensagem de maneira errada. O Struts encontra seu arquivo mas não encontra a mensagem. Verifique o exercício 1.

Gostou desta apostila? Conheça nossos treinamentos

- 2) Esquecer de alterar o arquivo `MessageResources.properties` e colocar a mensagem nele. O Struts encontra seu arquivo, mas não encontra a mensagem. Verifique o exercício 2.

Para os dois erros acima a mensagem é a mesma:



```

type Exception report
message
description The server encountered an internal error () that prevented it from fulfilling this request.
exception
javax.servlet.ServletException: Missing message for key "essa.mensagem.nao.existe" in bundle "(default bundle)" for locale en_US
	org.apache.jasper.runtime.PageContextImpl.doHandlePageException(PageContextImpl.java:848)
	org.apache.jasper.runtime.PageContextImpl.handlePageException(PageContextImpl.java:781)
	org.apache.jsp.testa_002dmensagens.jsp._jspService(org.apache.jsp.testa_002dmensagens.jsp:90)
	org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:97)
	javax.servlet.http.HttpServlet.service(HttpServlet.java:802)
	org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:322)
	org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:314)
	org.apache.jasper.servlet.JspServlet.service(JspServlet.java:264)
	javax.servlet.http.HttpServlet.service(HttpServlet.java:802)

root cause
javax.servlet.jsp.JspException: Missing message for key "essa.mensagem.nao.existe" in bundle "(default bundle)" for locale en_US
	org.apache.struts.taglib.bean.MessageTag.doStartTag(MessageTag.java:233)
	org.apache.jsp.testa_002dmensagens.jsp._jspx_meth.bean.message_5(org.apache.jsp.testa_002dmensagens.jsp:196)
	org.apache.jsp.testa_002dmensagens.jsp._jspService(org.apache.jsp.testa_002dmensagens.jsp:80)
	org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:97)
	javax.servlet.http.HttpServlet.service(HttpServlet.java:802)
	org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:322)
	org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:314)
	org.apache.jasper.servlet.JspServlet.service(JspServlet.java:264)
	javax.servlet.http.HttpServlet.service(HttpServlet.java:802)

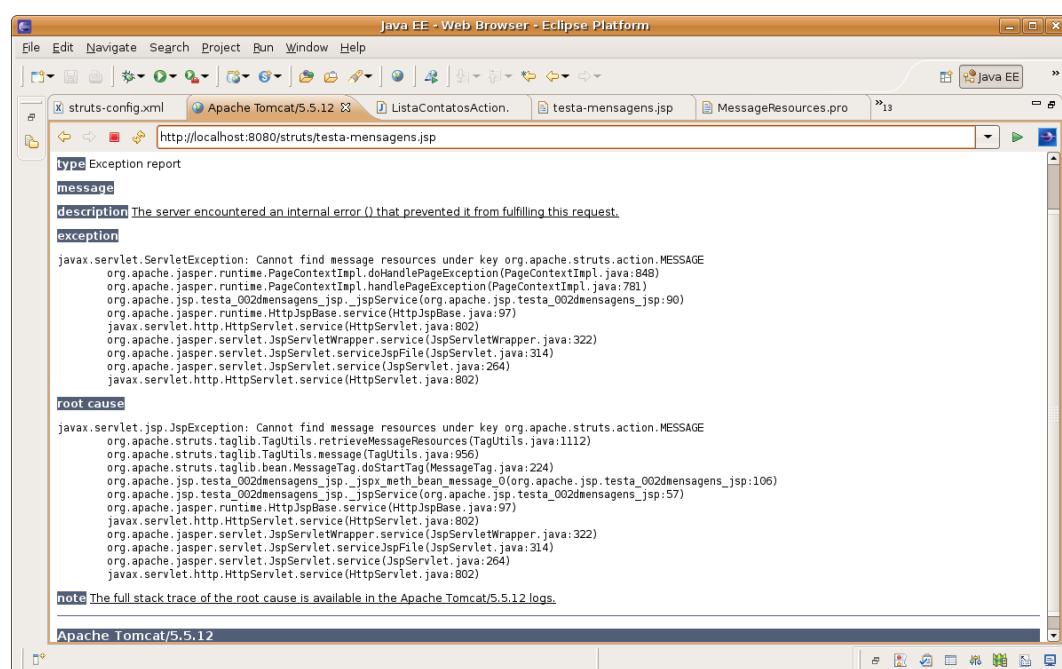
note The full stack trace of the root cause is available in the Apache Tomcat/5.5.12 logs.
Apache Tomcat/5.5.12

```

- 3) Esquecer de alterar o arquivo `struts-config.xml` para configurar o arquivo `MessageResources.properties`. Verifique o exercício 3.

- 4) Esquecer de reiniciar o servidor. Verifique o exercício 4.

Tela dos erros 3 e 4:



```

type Exception report
message
description The server encountered an internal error () that prevented it from fulfilling this request.
exception
javax.servlet.ServletException: Cannot find message resources under key org.apache.struts.action.MESSAGE
	org.apache.jasper.runtime.PageContextImpl.doHandlePageException(PageContextImpl.java:849)
	org.apache.jasper.runtime.PageContextImpl.handlePageException(PageContextImpl.java:781)
	org.apache.jsp.testa_002dmensagens.jsp._jspService(org.apache.jsp.testa_002dmensagens.jsp:90)
	org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:97)
	javax.servlet.http.HttpServlet.service(HttpServlet.java:802)
	org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:322)
	org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:314)
	org.apache.jasper.servlet.JspServlet.service(JspServlet.java:264)
	javax.servlet.http.HttpServlet.service(HttpServlet.java:802)

root cause
javax.servlet.jsp.JspException: Cannot find message resources under key org.apache.struts.action.MESSAGE
	org.apache.struts.taglib.TagUtils.retrieveMessageResources(TagUtils.java:1112)
	org.apache.struts.taglib.TagUtils.message(TagUtils.java:956)
	org.apache.struts.taglib.bean.MessageTag.doStartTag(MessageTag.java:224)
	org.apache.jsp.testa_002dmensagens.jsp._jspx_meth.bean.message_6(org.apache.jsp.testa_002dmensagens.jsp:106)
	org.apache.jsp.testa_002dmensagens.jsp._jspService(org.apache.jsp.testa_002dmensagens.jsp:57)
	org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:97)
	javax.servlet.http.HttpServlet.service(HttpServlet.java:802)
	org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:322)
	org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:314)
	org.apache.jasper.servlet.JspServlet.service(JspServlet.java:264)
	javax.servlet.http.HttpServlet.service(HttpServlet.java:802)

note The full stack trace of the root cause is available in the Apache Tomcat/5.5.12 logs.
Apache Tomcat/5.5.12

```

## 14.27 - Validando os campos

Para facilitar nosso trabalho, podemos agora implementar o método de validação que vem junto com o Struts.

Escreveremos, através do formulário, um método que retorna uma lista de erros encontrados. Para tanto, vamos verificar se as strings de nome, email ou endereço foram preenchidas ou não.

O método de validação do formulário é o método `validate`. Caso haja algum erro de validação, devemos adicionar os erros ao objeto `ActionErrors`. Por exemplo:

```
public ActionErrors validate(ActionMapping mapping, HttpServletRequest request) {  
  
    ActionErrors erros = new ActionErrors();  
  
    if (contato.getNome() == null || contato.getNome().equals("")) {  
        erros.add("nome", new ActionMessage("erro.campoNome"));  
    }  
  
    return erros;  
}
```

Nesse caso, usaremos a palavra `erro.campoNome` como chave para a mensagem de erro! Isso mesmo, fica muito mais fácil controlar o que vai ser apresentado ao seu usuário como mensagem de erro pois vamos configurá-lo no arquivo `MessageResources.properties`.

Acrescentando as verificações dos outros campos, temos o código final do método `validate`:

```
public ActionErrors validate(ActionMapping mapping, HttpServletRequest request) {  
  
    ActionErrors erros = new ActionErrors();  
  
    // verifica o nome  
    if (contato.getNome() == null || contato.getNome().equals("")) {  
        erros.add("nome", new ActionMessage("erro.campoNome"));  
    }  
  
    // verifica o email  
    if (contato.getEmail() == null || contato.getEmail().equals("")) {  
        erros.add("email", new ActionMessage("erro.campoEmail"));  
    }  
  
    // verifica o endereço  
    if (contato.getEndereco() == null || contato.getEndereco().equals("")) {  
        erros.add("endereco", new ActionMessage("erro.campoEndereco"));  
    }  
  
    return erros;  
}
```

Agora, basta alterar nossa configuração do `struts-config.xml` e adicionar o atributo chamado `input`.

```
<action path="/novoContato" name="ContatoForm" input="/novo.jsp"  
       type="br.com.caelum.struts.action.AdicionaContatoAction">
```

```
<forward name="ok" path="/listaContatos.do"/>
</action>
```

## 14.28 - Exercício: validação

1) Abra a sua classe ContatoForm.

a) Crie o método validate:

```
public ActionErrors validate(ActionMapping mapping, HttpServletRequest request) {

    ActionErrors erros = new ActionErrors();

    // verifica o nome
    if (contato.getNome() == null || contato.getNome().equals("")) {
        erros.add("nome", new ActionMessage("erro.campoNome"));
    }

    // verifica o email
    if (contato.getEmail() == null || contato.getEmail().equals("")) {
        erros.add("email", new ActionMessage("erro.campoEmail"));
    }

    // verifica o endereço
    if (contato.getEndereco() == null || contato.getEndereco().equals("")) {
        erros.add("endereco", new ActionMessage("erro.campoEndereco"));
    }

    return erros;
}
```

2) Altere o mapeamento da action de novo contato seu xml, **não adicione!**

```
<action path="/novoContato" name="ContatoForm" input="/novo.jsp"
       type="br.com.caelum.struts.action.AdicionaContatoAction">
    <forward name="ok" path="/listaContatos.do"/>
</action>
```

3) Coloque as mensagens de erro no seu arquivo de resources.

```
erro.campoNome = Preencha o campo nome corretamente.
erro.campoEmail = Preencha o campo email corretamente.
erro.campoEndereco = Preencha o campo endereço corretamente.
```

4) Tente criar um novo contato com nome vazio: <http://localhost:8080/struts/novo.jsp>





## 14.29 - Erros comuns

- 1) Esquecer de colocar o atributo `input="/novo.jsp"`

. O Struts fica sem saber para onde ir no caso da validação dar errado, então ele reclama que você não colocou o atributo `input`.

**Solução:** coloque o atributo `input` na sua tag `action`.

- 2) Em vez de alterar o mapeamento do seu xml, copiar novamente o mapeamento inteiro, isto é, possuir duas actions com o path `/novoContato`. O Struts fica perdido e não funciona.

## 14.30 - Exercícios opcionais

- 1) Abra o seu arquivo `MessageResources.properties`.

- a) Adicione as seguintes linhas:

```
errors.header=<UL>
errors.footer=</UL>
errors.prefix=<LI>
errors.suffix=</LI>
```

- b) Efetue a inserção de um contato que gere diversas mensagens de erro.

O item `header` aparece antes da lista de erros enquanto o item `footer` aparece após a lista. Já os itens `prefix` e `suffix` são prefixos e sufixos a serem adicionados a toda mensagem de erro.

- 2) Utilize a tag `html:errors` com o atributo `properties` para mostrar somente as mensagens de erro de determinado campo. Para mostrar as mensagens relativas ao campo `nome` utilize, por exemplo:

```
<html:errors property="nome"/>
```

## 14.31 - Limpando o formulário

Adicione um contato no banco de dados, acesse um outro site qualquer e volte ao seu `novo.jsp`. O que acontece? O formulário continua preenchido?

Isso ocorre por que o Struts recicla objetos do tipo `ActionForm` entre diferentes `requests`, fazendo com que seu objeto permaneça sujo. Por padrão, os nossos action forms estão no chamado escopo de sessão (`session`), facilitando, por exemplo, a criação de longos wizards.

Mas, na maioria dos casos, queremos que o formulário sirva apenas para o request atual. Para isso, usamos um atributo opcional na tag `action` quando utilizando um formulário: `scope`. Esse atributo é utilizado para deixar os dados do formulário no request ou na sessão.

Utilizamos o escopo de sessão (padrão) quando desejamos que os dados se mantenham atrelados aquele cliente por mais de um request, enquanto que o escopo de `request` atrela os dados somente até o término da requisição.

Portanto, você pode colocar no seu `action`:

`scope="session"`

ou:

`scope="request"`

É comum utilizar o escopo de sessão para manter os dados de um formulário através de diversas requisições.

## 14.32 - Exercícios: scope

- 1) Altere seu `struts-config.xml` para utilizar o escopo do formulário como `request`.

Basta **acrescentar** o atributo `scope="request"`

na `action` do novo contato:

```
<action path="/novoContato" name="ContatoForm" input="/novo.jsp"
       type="br.com.caelum.struts.action.AdicionaContatoAction" scope="request">
    <forward name="ok" path="/listaContatos.do"/>
</action>
```

## 14.33 - Exercícios opcionais

- 1) Crie um formulário chamado `RemoveContatoForm` e mapeie-o no `struts-config.xml`.
- 2) Crie uma ação chamada `RemoveContatoAction`
  - a) Ela recebe um formulário do tipo `RemoveContatoForm`;
  - b) Ela remove do banco (usando o `ContatoDAO`) o contato com `id` igual ao do contato do formulário. Algo como:

```
Contato contato = ((RemoveContatoForm) form).getContato();
new ContatoDAO().remove(contato);
```
  - c) Mapeie uma ação no `struts-config.xml` chamada `removeContato` para sua classe `RemoveContatoAction`;
  - d) Redirecione para `/listaContatos.do` após remover um contato;
  - e) Na sua lista, altere o código para incluir um link para remoção:

```
<c:forEach var="contato" items="#{contatos}">
    ${contato.id} - ${contato.nome}
    (<a href="removeContato.do?contato.id=${contato.id}">remover</a>)<br/>
</c:forEach>
```

- f) Teste a sua listagem e depois remova algum contato;

## 14.34 - O mesmo formulário para duas ações

Nesse momento do aprendizado é bem comum que um aluno pense em reutilizar formulários para duas ações diferentes. Será que vale a pena? Será que as duas ações tem exatamente a mesma validação? Será que elas sempre vão ter a mesma validação?

Como ou: (1) duas ações não costumam ter a mesma validação, ou: (2) não podemos prever o futuro e saber se alterações no negócio trarão validações diferentes para essas duas ações é considerado boa prática criar formulários diferentes para ações diferentes, utilizando sempre o bom senso.

## 14.35 - Exercícios opcionais

- 1) Vamos internacionalizar nosso sistema:

- Crie uma ação chamada `MudaIdiomaAction`;
- Ela chama o método `setLocale` que altera o locale do cliente e retorna o forward de `ok`:

```
String idioma = request.getParameter("idioma");
Locale locale = new Locale(idioma);

System.out.println("Mudando o locale para " + locale);
setLocale(request, locale);

return mapping.findForward("ok");
```

- Mapeie essa action para o path `/mudaIdioma`;

```
<action path="/mudaIdioma" type="br.com.caelum.struts.action.MudaIdiomaAction">
    <forward name="ok" path="/testa-mensagens.jsp"/>
</action>
```

- Altere seu arquivo `testa-mensagens.jsp` para adicionar dois links para a ação de alterar a língua:

```
<a href="mudaIdioma.do?idioma=en">EN</a>
<a href="mudaIdioma.do?idioma=pt">PT</a><br/>
```

- Crie o arquivo `MessageResources_en.properties` no seu diretório `src`;

```
site.titulo = Struts Test
pergunta.usuario = What is your username?
pergunta.senha = What is your password?
pergunta.enviar = Send data

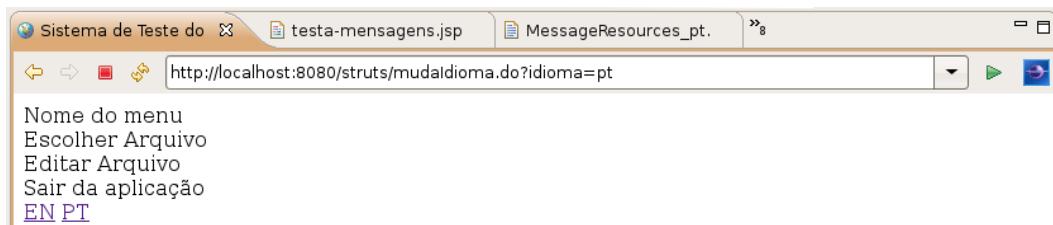
# comentário de um arquivo .properties
menu.nome = Menu name
menu.arquivo = Choose file
menu.editar = Edit file
menu.sair = Quit
```

- Reinic peace o Tomcat;

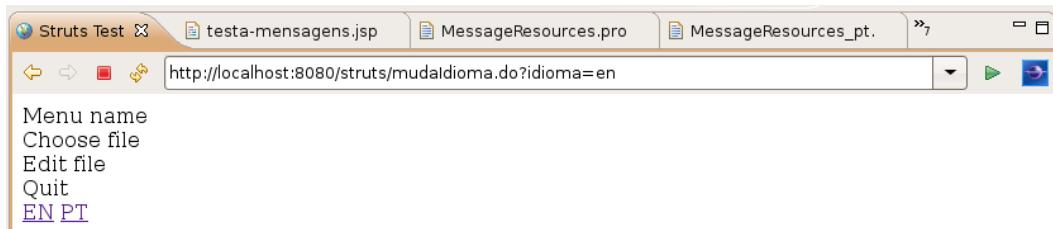
- g) Teste a URL `http://localhost:8080/struts/testa-mensagens.jsp`:

O site aparece por padrão na língua que o seu browser pediu, que é a língua configurada no seu sistema operacional.

- h) Escolha o link para português:



- i) Escolha o link para inglês:



- 2) Vamos mostrar agora os detalhes de um contato.

- Crie a classe `MostraContatoForm` similar ao `ContatoForm`;
- Crie uma ação chamada `MostraContatoAction`;
- Ela chama o método `procura`:

```
MostraContatoForm formulario = (MostraContatoForm) form;
Contato contato = formulario.getContato();
Contato encontrado = new ContatoDAO().procura(contato.getId());
request.setAttribute("contato", encontrado);
```

- Mapeie uma ação no `struts-config.xml` chamada `mostraContato` para sua classe `MostraContatoAction`
- Redirecione para `/mostraContato.jsp` após mostrar um contato. O código dele mostra os dados do contato (sem formulário).
- Na sua `lista.jsp`, altere o código para incluir um link para `mostraContato.do`:

```
<c:forEach var="contato" items="#{contatos}">
    ${contato.id} - ${contato.nome}
    (<a href="removeContato.do?contato.id=${contato.id}">remover</a>
     <a href="mostraContato.do?contato.id=${contato.id}">mostrar</a>)<br/>
</c:forEach>
```

- 3) Vamos terminar a parte de alterar o contato.

- Altere a pagina `mostraContato.jsp` para mostrar um formulario acessando `/alteraContato.do`. Nas tags `html:text` utilize o campo `value="#{...}"` para colocar o valor inicial nos mesmos;
- Crie um form chamado `AlteraContatoForm`;
- Crie uma ação chamada `AlteraContatoAction`;

- d) Ela chama o método altera:

```
AlteraContatoForm formulario = (AlteraContatoForm) form;
Contato contato = formulario.getContato();

new ContatoDAO().altera(contato);
```

- e) Mapeie uma ação no struts.config.xml chamada alteraContato para sua classe AlteraContatoAction;  
f) Redirecione para /listaContatos.do após alterar um contato;

## 14.36 - Struts-logic taglib: um exemplo antigo de for

A seguir a versão do arquivo lista-elegante.jsp usando a taglib de lógica do Struts. Repare como fica mais complexo:

```
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>

<html>
    <head><title>Sistema de Teste do Struts</title></head>

    <body>

        <logic:iterate id="contato" name="contatos" type="br.com.caelum.struts.model.Contato">
            <bean:write name="contato" property="id" />,
            <bean:write name="contato" property="nome" /><br/>
        </logic:iterate>

    </body>
</html>
```

Vale lembrar que o próprio grupo que desenvolve o struts já recomenda o uso das taglibs da JSTL em vez da proprietária do struts.

## 14.37 - Para saber mais

- 1) O **Struts Tiles** ajuda você a componentizar os “pedaços” das suas páginas. Dando nome para os diversos componentes comuns as páginas, você pode incluí-los dinamicamente em qualquer JSP.
- 2) O **Struts Validator** pode ser configurado para que os form beans sejam verificados antes de suas ações serem executadas. Ele é, de longe, o plugin mais famoso e poderoso do Struts.
- 3) O projeto **Velocity Tools** faz a ponte entre o Velocity e o Struts, entre outras coisas.
- 4) Você pode utilizar a tag <html:error property="nome"/>, por exemplo, para mostrar somente os erros relacionados ao campo nome.

### AlwaysLinkToActions

Um dos patterns mais simples e famosos que o Struts construiu é o *Always Link To Actions*. Você sempre deve se referenciar as ações do Struts e nunca as suas páginas JSP diretamente. Se você já esconde suas páginas JSP no diretório WEB-INF, está se obrigando a utilizar tal procedimento. Qual a vantagem?

Se, em algum dia, sua página JSP precisa executar uma lógica antes de ser chamada ou se ela deve ser renomeada, basta alterar o arquivo struts-config.xml, caso contrário você deveria procurar todos os links em sua aplicação!

### Forwards de redirecionamento no cliente

Podemos efetuar o redirecionamento no cliente em vez de fazê-lo no servidor. Utilizando tal recurso, o cliente fica sabendo do redirecionamento e, ao clicar em Refresh (Atualizar) ou pressionar F5 no seu navegador, ele efetuará a requisição do redirecionamento e não da página original.

```
<forward name="ok" redirect="true" path="/listaContatos.do" />
```

No exemplo acima, o redirecionamento após a adição de um contato ao banco será feito para a listagem, portanto ao pressionar F5 o cliente pede a listagem novamente e não a adição.

### A ação padrão

Para marcar uma ação como a padrão, isto é, aquela que deve ser executada caso nenhuma das outras for a correta, basta adicionar um atributo chamado unknown. Somente uma ação pode ter tal atributo com valor true.

```
<action path="/seu path aqui" type="sua classe aqui" unknown="true" />
```

### Ações só de forward

Às vezes, é interessante criar um apelido para uma página JSP. Para isso, uma das alternativas é criar uma ação que em vez de possuir um type, possui um atributo chamado forward:

```
<action path="/apelido" forward="/minha_pagina.jsp" />
```

No exemplo acima, comum no mercado, a URL que termina com /apelido.do será redirecionada para a página JSP dentro do diretório WEB-INF/jsp.

### Global Forwards

O Struts permite configurar no struts-config.xml uma lista de forwards globais que podem ser utilizados por todas as ações. Para isso, basta adicionar a tag global-forwards antes dos action-mappings.

```
<global-forwards>
    <forward name="exception" path="/error.jsp"/>
</global-forwards>
```

Se você quiser controlar os erros através desse forward, basta usar algo similar ao código a seguir:

```
catch (Exception e) {
    return mapping.findForward("exception");
}
```

# Autenticação, cookies e sessão

*“Verifica se o que prometes é justo e possível, pois promessa é dívida.”*  
– Confúcio

Ao término desse capítulo, você será capaz de:

- Utilizar sessões em servlets;
- Um sistema de login.

## 15.1 - Preparando um sistema de login

Vamos nos preparar para um exemplo bem prático utilizando o Struts, começando pelo login de um usuário no nosso site.

Sempre que queremos trabalhar com formulários seguiremos basicamente os passos a seguir:

- 1) Modelar o formulário como classe e configurá-lo no `struts-config.xml`;
- 2) Criar a página HTML com o formulário;
- 3) Criar a ação que será executada e mapeá-la no `struts-config.xml`;
- 4) Criar a página final, após a execução do formulário.

## 15.2 - Nossas classes: Funcionario e FuncionarioDAO

Trabalharemos com usuários que podem logar no sistema, portanto eles devem ter um nome, usuário e senha, como na classe de modelo a seguir:

```
public class Funcionario {  
  
    private String nome;  
    private String usuario, senha;  
  
    // getters e setters  
}
```

E seu respectivo DAO:

```
public class FuncionarioDAO {
```

```

// a conexao com o banco de dados
private Connection connection;

// construtor que recebe a conexao
public FuncionarioDAO(Connection con) {
    this.connection = con;
}

public FuncionarioDAO() throws SQLException {
    this.connection = ConnectionFactory.getConnection();
}

public void adiciona(Funcionario f) throws SQLException {

    // prepared statement para insercao
    PreparedStatement stmt = this.connection
        .prepareStatement("insert into funcionarios (nome,usuario,senha) values (?, ?, ?)");

    // seta os valores
    stmt.setString(1, f.getNome());
    stmt.setString(2, f.getUsuario());
    stmt.setString(3, f.getSenha());

    // executa
    stmt.execute();
    stmt.close();
}

public boolean existeUnico(Funcionario funcionario)
    throws SQLException {

    PreparedStatement stmt = connection
        .prepareStatement("select * from funcionarios where usuario=? and senha=?");
    stmt.setString(1, funcionario.getUsuario());
    stmt.setString(2, funcionario.getSenha());
    ResultSet rs = stmt.executeQuery();

    try {
        // se nao existir nenhum funcionario, da erro
        if (!rs.next()) {
            return false;
        }
        // false se existe mais de um funcionario com esse usuario e senha
        return !rs.next();
    } finally {
        rs.close();
        stmt.close();
    }
}
}

```

Ambas as classes já estão no seu projeto.

Note que o FuncionarioDAO possui um único método que desconhecemos. O método `existeUnico` é capaz de receber um Funcionario e devolve se existe um true somente se existir um único funcionario no banco com

tal usuário e senha. Note que não existe nenhuma novidade quanto a código Java no exemplo acima, é tudo uma questão de código SQL.

### 15.3 - Passo 1: Formbean

O código do nosso formulário é razoavelmente simples:

```
package br.com.caelum.struts.form;

import org.apache.struts.action.*;

public class LoginForm extends ActionForm {

    private Funcionario funcionario = new Funcionario();

    public Funcionario getFuncionario() {
        return this.funcionario;
    }
}
```

E agora vamos a tag de formulário no struts-config.xml:

```
<form-beans>
    <form-bean name="ContatoForm" type="br.com.caelum.struts.form.ContatoForm"/>
    <form-bean name="LoginForm" type="br.com.caelum.struts.form.LoginForm"/>
</form-beans>
```

### 15.4 - Passo 2: A página de login: formulario-login.jsp

Agora, partiremos para o nosso arquivo JSP que define o formulário. Já utilizaremos alguns recursos novos, as taglibs do Struts, que facilitam o trabalho de reutilização de código, simplificam a camada de visualização e permitem uma internacionalização mais fácil da mesma.

A seguir está o arquivo formulario-login.jsp, que usa as taglibs mostradas anteriormente para definir o código html das mesmas. A única tag nova nesse exemplo é a chamada `html:password` que mostra um campo do tipo `password`:

```
<html:password property="funcionario.senha"/>
```

Por razões de segurança, esse campo não volta preenchido no caso de erro de validação. O exemplo a seguir mostra o formulário completo:

```
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>

<html:html>
    <title>Sistema de Testes do Struts</title>

    <body>
        <html:form action="/efetuaLogin" focus="funcionario.usuario">
            Qual é seu usuário?
            <html:text property="funcionario.usuario"/> <br/>
```

```
Qual é sua senha?  
<html:password property="funcionario.senha"/> <br/>  
  
<html:submit>Enviar Dados</html:submit>  
</html:form>  
  
</body>  
</html:html>
```

## 15.5 - Exercícios: formulário de login

- 1) Crie o seu formulário de login: uma classe chamada `LoginForm` no pacote `br.com.caelum.struts.form`. Não esqueça de estender a classe `ActionForm`.

```
package br.com.caelum.struts.form;  
  
//faça os imports aqui CTRL+SHIFT+O  
  
public class LoginForm extends ActionForm {  
  
    private Funcionario funcionario = new Funcionario();  
  
    public Funcionario getFuncionario() {  
        return this.funcionario;  
    }  
}
```

- 2) Mapeie seu formulário no `struts-config.xml`.

```
<form-bean name="LoginForm" type="br.com.caelum.struts.form.LoginForm"/>
```

- 3) Crie o seu arquivo `formulario-login.jsp`.

```
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>  
  
<html:html>  
<title>Sistema de Testes do Struts</title>  
<body>  
  
<html:form action="/efetuaLogin" focus="funcionario.usuario">  
    Qual é seu usuário?  
    <html:text property="funcionario.usuario"/> <br/>  
  
    Qual é sua senha?  
    <html:password property="funcionario.senha"/> <br/>  
  
    <html:submit>Enviar Dados</html:submit>  
</html:form>
```

```
</body>
</html:html>
```

## 15.6 - A ação

Vamos pensar um pouco na nossa ação de login. Que tal utilizarmos o método `getParameter` para ler o usuário e a senha? Já vimos que isso é muito chato e existe uma grande chance de erro.

Podemos então usar nosso `FormBean` e chamar apenas getters elegantemente, usando o conceito que aprendemos no último capítulo.

```
package br.com.caelum.struts.action;

//imports aqui CRTL+SHIFT+O

public class LoginAction extends Action {
    @Override
    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception {
        System.out.println("Algum usuário está tentando se logar...");

        // 1. onde estão as variáveis? Olha que elegância
        LoginForm formulario = (LoginForm) form;
        Funcionario funcionario = formulario.getFuncionario();

        // 2. testa se são válidas
        if (!new FuncionarioDAO().existeUnico(funcionario)) {
            // não são válidas (oops)
            return mapping.findForward("erro");
        }

        // ok.... para onde ir agora?
        return mapping.findForward("ok");
    }
}
```

## 15.7 - A ação no struts-config.xml

Agora vamos configurar nossa ação no `struts-config.xml`.

Adicionaremos uma nova action, chamada `/efetuaLogin` e ela será ligada a classe `br.com.caelum.struts.action.LoginAction`. Introduzimos um novo atributo, chamado `name`, que tem o valor do nosso formulário, aquele que irá receber os dados do request e facilitar sua leitura!

```
<struts-config>

<form-beans>
    <!-- outros form-beans aqui -->
    <form-bean name="LoginForm" type="br.com.caelum.struts.form.LoginForm"/>
</form-beans>
```

```
<action-mappings>
    <!-- outras actions aqui -->
    <action path="/efetuaLogin" name="LoginForm" scope="request"
        type="br.com.caelum.struts.action.LoginAction">
        <!-- Forwards que chamamos na nossa action Login! -->
        <forward name="erro" path="/erro.jsp"/>
        <forward name="ok" path="/ok.jsp"/>
    </action>
</action-mappings>

</struts-config>
```

**O web.xml e o struts-config.xml...**

Você reparou que não alteramos mais o web.xml?

De agora em diante, sempre que você utilizar o Struts, não precisará alterar os dados do web.xml, você somente adicionará novas ações ao seu struts-config.xml.

Se você acha trabalhoso editar o struts-config.xml, você pode utilizar ferramentas como o Struts Console para alterar para uma edição “gráfica” do mesmo.

## 15.8 - ok.jsp e erro.jsp

Agora, vamos escrever as duas páginas mais simples. Ambas ficam no diretório WebContent e não possuem nada de especial:

**ok.jsp**

```
<html>
    Você se logou com sucesso!
</html>
```

**erro.jsp**

```
<html>
    Ocorreu algum erro ao tentar se logar!
</html>
```

## 15.9 - Exercícios: LoginAction

1) Crie sua ação de Login:

```
package br.com.caelum.struts.action;

//imports aqui CTRL+SHIFT+O

public class LoginAction extends Action {

    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception {
```

```

System.out.println("Algum usuário está tentando se logar...");

// 1. onde estão as variáveis? Olha que elegância
LoginForm formulario = (LoginForm) form;
Funcionario funcionario = formulario.getFuncionario();

// 2. testa se são válidas
if (!new FuncionarioDao().existeUnico(funcionario)) {
    // não são válidas (oops)
    return mapping.findForward("erro");
}

// ok.... para onde ir agora?
return mapping.findForward("ok");
}
}

```

- 2) Mapeie sua ação no struts-config.xml:

```

<action path="/efetuaLogin" name="LoginForm" scope="request"
       type="br.com.caelum.struts.action.LoginAction">
    <!-- Forwards que chamamos na nossa action Login! -->
    <forward name="erro" path="/erro.jsp"/>
    <forward name="ok" path="/ok.jsp"/>
</action>

```

- 3) Crie seu arquivo ok.jsp no diretório WebContent:

```

<html>
    Você se logou com sucesso!
</html>

```

- 4) Crie seu arquivo erro.jsp no diretório WebContent:

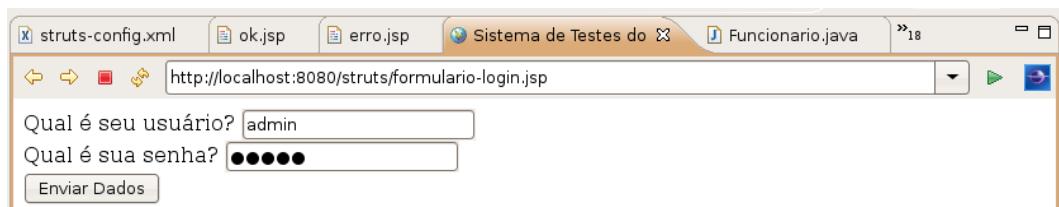
```

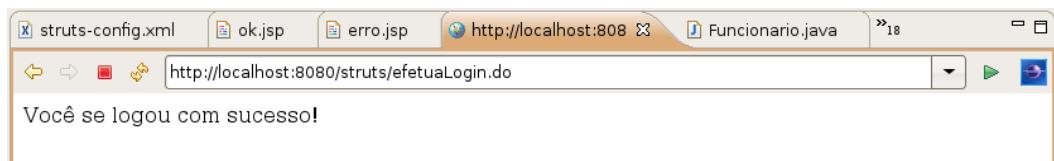
<html>
    Ocorreu algum erro ao tentar se logar!
</html>

```

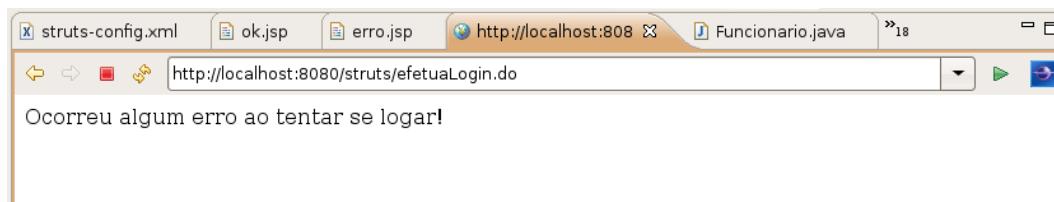
- 5) Agora para testar a sua página de login, basta reiniciar o Tomcat e acessar <http://localhost:8080/struts/formulario-login.jsp>

- a) Teste nosso formulário colocando valores e enviando os dados; deverá aparecer a mensagem de sucesso.  
Agora clique em enviar deixando os campos em branco; deverá aparecer a mensagem de erro.





Ou...



## 15.10 - Erro comum

Esquecer de fazer seu `LoginForm` estender `ActionForm`: como `LoginForm` e `ActionForm` não possuem conexão, o compilador (no caso, o Eclipse), reclama do casting que está sendo feito dentro da sua classe `Action`, afinal nenhum `ActionForm` é um `LoginForm`.

**Solução:** estenda `ActionForm`.

## 15.11 - Exercícios opcionais

- 1) Altere os seus arquivos `formulario-login.jsp`, `ok.jsp` e `erro.jsp` para utilizarem a tag `bean:message`, lembre-se de alterar o arquivo `MessageResources.properties`.
- 2) Faça o restart do tomcat.
- 3) Teste novamente seu sistema de autenticação.

## 15.12 - Cookies

O protocolo HTTP utilizado até agora para o acesso à páginas é limitado por não manter detalhes como quem é quem entre uma conexão e outra. Para resolver isso, foi inventado um sistema para facilitar a vida dos programadores.

Um **cookie** é normalmente um par de strings guardado no cliente, assim como um mapa de strings. Esse par de strings possui diversas limitações que variam de acordo com o cliente utilizado, o que torna a técnica de utilizá-los algo do qual não se deva confiar muito. Já que as informações do cookie são armazenadas no cliente, o mesmo pode alterá-la de alguma maneira... sendo inviável, por exemplo, guardar o nome do usuário logado...

Quando um cookie é salvo no cliente, ele é enviado de volta ao servidor toda vez que o cliente efetuar uma nova requisição. Desta forma, o servidor consegue identificar aquele cliente sempre com os dados que o cookie enviar.

Um exemplo de bom uso de cookies é na tarefa de lembrar o nome de usuário na próxima vez que ele quiser se logar, para que não tenha que redigitar o mesmo.

Cada cookie só é armazenado para um website. Cada website possui seus próprios cookies e estes não são vistos em outra página.

**Cookies: facilidade e segurança**

É arriscado trabalhar com cookies em sessões que necessitam de segurança. O mais indicado é sempre usar sessões, que serão discutidas logo em seguida.

Além disso, é muito penoso ter que iterar por todos os cookies para achar um cookie específico. A estrutura de mapas de uma `Session` facilita em muito o trabalho de valores atrelados a um usuário.

**Adicionando cookies: comportamento estranho**

Na primeira vez que adicionamos um cookie na resposta, ele não está disponível para a leitura através da requisição. Ahn?

Para resolver esse problema costumamos usar um atributo do método `request: request.setAttribute` e `request.getAttribute`. Já podemos perceber que trabalhar com cookies pode ser trabalhoso.

## 15.13 - Sessão

Usar Cookies parece facilitar muito a vida.... exceto que através de um cookie não é possível marcar um cliente com um objeto, somente com `Strings`. Imagine gravar os dados do usuário logado através de cookies. Seria necessário um cookie para cada atributo: `usuario`, `senha`, `id`, `data de inscrição`, etc. Sem contar a falta de segurança.

O Cookie também pode estar desabilitado no cliente, sendo que não será possível lembrar nada que o usuário fez...

Uma sessão facilita a vida de todos por permitir atrelar objetos de qualquer tipo a um cliente, não sendo limitada somente à strings e é independente de cliente.

A abstração da API facilita o trabalho do programador pois ele não precisa saber como é que a sessão foi implementada no servlet container, ele simplesmente sabe que a funcionalidade existe e está lá para o uso. Se os cookies estiverem desabilitados, a seção não funcionará e devemos recorrer para uma técnica (trabalhosa) chamada `url-rewriting`.

A sessão nada mais é que um tempo que o usuário permanece ativo no sistema. A cada página visitada, o tempo de sessão é zerado. Quando o tempo ultrapassa um limite demarcado no arquivo `web.xml`, o cliente perde sua sessão.

## 15.14 - Configurando o tempo limite

Para configurar 3 minutos como o padrão de tempo para o usuário perder a sessão basta incluir o seguinte no arquivo `web.xml`:

```
<session-config>
    <session-timeout>3</session-timeout>
</session-config>
```

## 15.15 - Registrando o usuário logado na sessão

Para utilizar todos os recursos de uma sessão, é necessário primeiro criar uma, assim conseguimos marcar um usuário para a próxima vez que ele visitar uma página na mesma aplicação web.

Existem dois métodos `getSession` na classe `HttpServletRequest`, sendo que o mais simples dele não recebe argumentos e retorna uma nova sessão caso não exista nenhuma, ou a sessão que já existia:

```
HttpSession session = request.getSession();
```

Com uma sessão em mãos podemos utilizar seus métodos mais básicos:

Tabela 15.1:

Object session.getAttribute(String)  
Enumeration session.getAttributeNames()  
session.setAttribute(String, Object)  
session.removeAttribute(String)  
session.invalidate()  
boolean session.isNew()

Retorna o objeto (Object) marcado com uma chave (String)
Retorna uma enumeração com todos os nomes dos atributos
Coloca um objeto (Object) em uma chave (String) do mapa
Remove um atributo da sessão
Inativa a sessão
Verifica se a sessão é nova e foi criada nessa requisição

Como exemplo, podemos trabalhar da seguinte maneira:

```
HttpSession session = request.getSession();
session.setAttribute("nome", "valor");
```

E vale lembrar que o nome deve ser uma `String` e o valor pode ser um objeto qualquer!

#### Vantagens e desvantagens

##### Vantagens:

- Mais difícil para o cliente forjar ser alguém que ele não é;
- Os objetos são armazenados no servidor e não precisam ser reenviados em toda requisição (apenas o link de sessão precisa ser enviado);
- Qualquer objeto Java pode ser utilizado como valor salvo na seção, não se limitando apenas a Strings.

##### Desvantagens:

- O servidor perde o dado se o navegador for fechado;
- Uma seção não é obrigatoriamente a mesma em diferentes instâncias do mesmo navegador no mesmo cliente acessando a mesma aplicação web, o servidor não controla o cliente! Você não sabe como ele vai funcionar!

## 15.16 - Exercícios: autorização

- 1) Ao logar um usuário com sucesso, adicione esse usuário à sessão:

- Trabalhe dentro da ação de login antes de ir para a página de ok
- Acesse a sessão e adicione o funcionário a mesma:

```
HttpSession session = request.getSession();
session.setAttribute("funcionario", funcionario);
```

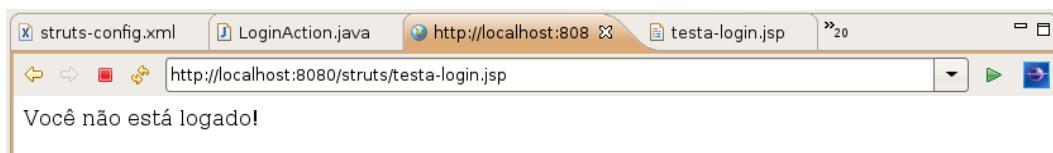
- Crie uma página `testa-login.jsp`

Se o bean `usuario` for `null`, isso significa que o usuário não estava logado; se for diferente que `null` podemos imprimir seu nome!

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<c:choose>
    <c:when test="${empty funcionario}">
        Você não está logado!
    </c:when>
    <c:otherwise>
        Você está logado como ${funcionario.usuario} e senha ${funcionario.senha}
    </c:otherwise>
</c:choose>
```

Note como é fácil aprender a utilizar as tags da JSTL. Os instrutores da Caelum recomendam a todos os alunos que comecem a ler especificações, para aprender cada vez mais sobre determinadas tecnologias, e a especificação da JSTL, assim como seu TLDDoc, é de fácil leitura, podendo ser uma maneira de introduzir o aluno a esse mundo de documentações.

- Teste acessar a página de teste **sem** se logar.



- Teste acessar a página de teste **após** se logar.



#### Logando novamente

Rpare que ao acessar a página de login após ter se logado, seu usuário permanece na sessão. Sendo assim, se você errar o usuário/senha, você continua logado! Para fazer o logout você pode usar o método `removeAttribute` da classe `HttpServletRequest`.

### 15.17 - Exercícios opcionais

- Crie uma lógica para deslogar o usuário do sistema.

- 
- a) Utilize o método `removeAttribute` para remover um atributo da session:

```
HttpSession session = request.getSession();
session.removeAttribute("funcionario");
```

- 2) Altere sua lógica de adicionar contato ao banco para verificar se o usuário está logado no sistema, se ele não estiver, envie ele para a página de formulário de login.
- 3) (desafio) Faça um filtro (Filter de Servlet como vimos no capítulo 11) para fazer o processo de autenticação automaticamente em todas as URLs exceto a que faz o login.

# Hibernate

*“É uma experiência eterna de que todos os homens com poder são tentados a abusar.”*  
– Baron de Montesquieu

Neste capítulo, você aprenderá a:

- Usar a ferramenta de ORM Hibernate;
- Gerar as tabelas em um banco de dados qualquer a partir de suas classes de modelo;
- Automatizar o sistema de adicionar, listar, remover e procurar objetos no banco;
- Utilizar anotações para facilitar o mapeamento de classes para tabelas;
- Criar classes de DAO bem simples utilizando o hibernate.

## 16.1 - Vantagens

A utilização de código SQL dentro de uma aplicação agrava o problema da independência de plataforma de banco de dados e complica, em muito, o trabalho de mapeamento entre classes e banco de dados relacional.

O Hibernate abstrai o código SQL da nossa aplicação e permite escolher o tipo de banco de dados enquanto o programa está rodando, permitindo mudar sua base sem alterar nada no seu código Java.

Além disso, ele permite criar suas tabelas do banco de dados de um jeito bem simples, não se fazendo necessário todo um design de tabelas antes de desenvolver seu projeto que pode ser muito bem utilizado em projetos pequenos.

Já projetos grandes onde o plano de ação padrão tomado pelo Hibernate não satisfaz as necessidades da empresa (como o uso de select \*, joins etc), ele possui dezenas de otimizações que podem ser feitas para atingir tal objetivo.

## 16.2 - Criando seu projeto

Para criar seu projeto, é necessário baixar os arquivos .jar necessários para rodar o Hibernate e colocá-los no *classpath* do mesmo.

O site oficial do hibernate é o [www.hibernate.org](http://www.hibernate.org) e lá você pode baixar a última versão estável do mesmo na seção Download. Após descompactar esse arquivo, basta copiar todos os jars para o nosso projeto.

Ainda falta baixar as classes correspondentes ao *HibernateAnnotations*, que utilizaremos para gerar o mapeamento entre as classes Java e o banco de dados. Eles são encontrados também no site do hibernate e contêm outros jars que devemos colocar no nosso projeto.

Antigamente (até a versão 2 do hibernate) o mapeamento era feito somente através de arquivos xml, que era bem chato, e utilizávamos de uma ferramenta chamada Xdoclet que criava tais xmls. Hoje em dia o Xdoclet foi substituído pelas Annotations.

### Banco de dados

O Hibernate traduz suas necessidades em código SQL para qualquer banco de dados. Continuaremos utilizando o MySQL em nossos exemplos, portanto não esqueça de copiar o arquivo .jar correspondente ao driver para o diretório lib de sua aplicação.

## 16.3 - Modelo

Utilizaremos uma classe que modela um produto para este capítulo:

```
package br.com.caelum.hibernate;

public class Produto {

    private Long id;
    private String nome;
    private String descricao;
    private Double preco;

    // adicione seus getters e setters aqui!
}
```

## 16.4 - Configurando a classe/tabela Produto

Para configurar a nossa classe `Produto`, basta adicionar alguns comentários especiais na definição da classe e nas nossas variáveis membro. O que faremos não são comentários de verdade, mas sim o que chamamos de anotações.

A grande diferença entre os dois – anotações e comentários – é que as anotações são bem estruturadas, seguem um padrão e são mantidas em tempo de execução, enquanto os comentários são perdidos em tempo de compilação, que impossibilita descobrir em tempo de execução o que havia sido comentado.

O código a seguir coloca nossa classe na tabela “`Produto`” e seta algumas propriedades e o id.

**Atenção:** toda classe que vai trabalhar com o Hibernate precisa de um (ou mais) campo(s) que será a chave primária (composta ou não).

Fora isso, existem diversas opções que podemos colocar como configurar para não aceitar campos `null` ou mudar o nome da coluna por exemplo. Para ler:

```
package br.com.caelum.hibernate;

@Entity
public class Produto {

    @Id @GeneratedValue
    private Long id;
```

```

@Column(name = "descricao", nullable = true, length = 50)
private String descricao;

private Double preco;
private String nome;

//metodos...
}

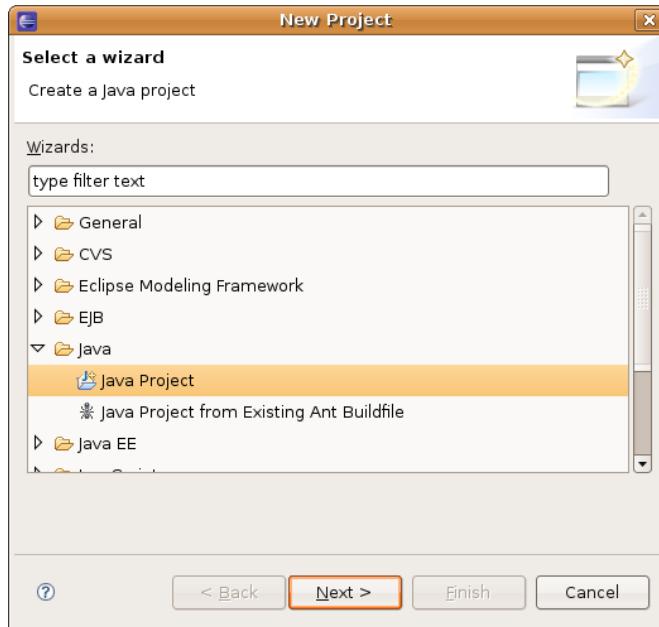
```

A especificação do EJB3 define tais anotações e possui diversas opções que podemos utilizar em nosso projeto. Sempre que possuir alguma dúvida em relação as anotações lembre-se de ler a tal especificação.

## 16.5 - Exercícios

1) Crie um novo projeto:

- Vá em *File -> New -> Project*.
- Escolha a opção *Java Project*.

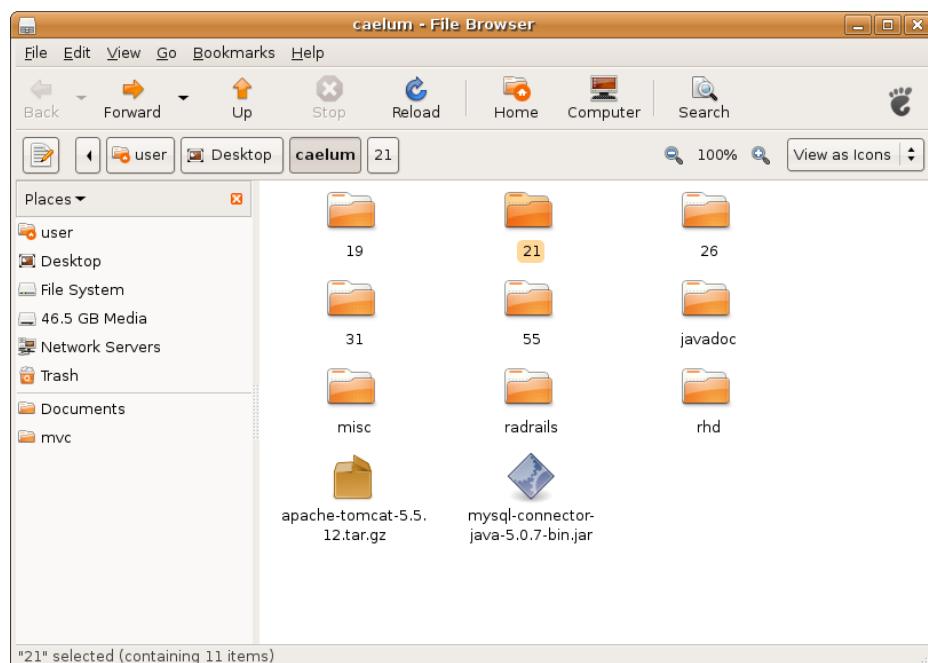


- Escolha **hibernate** como nome do projeto e clique em *Finish*. Confirme a mudança de perspectiva.

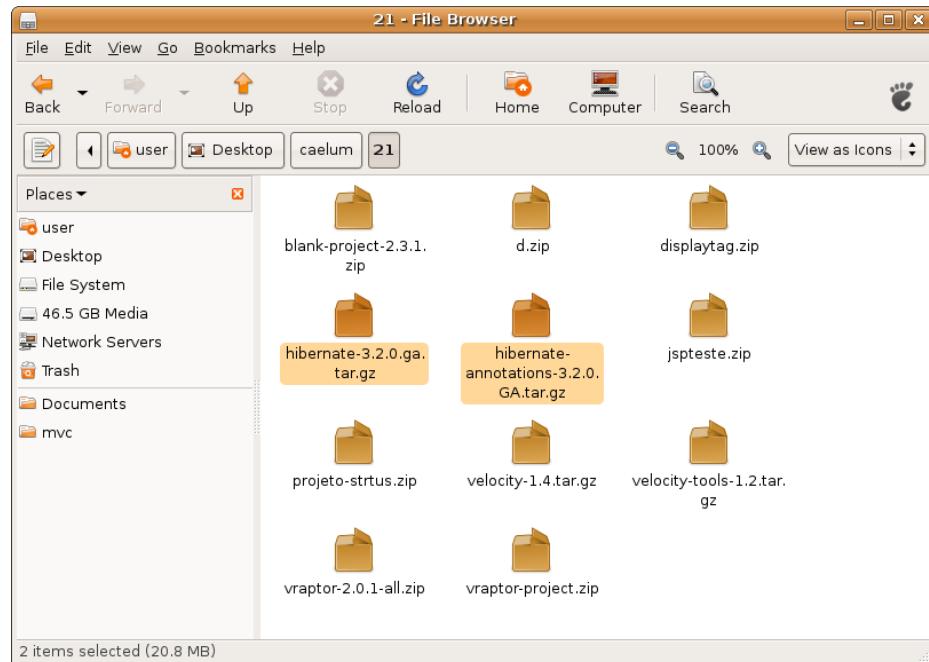


2) Descompacte o hibernate e o hibernate-annotations:

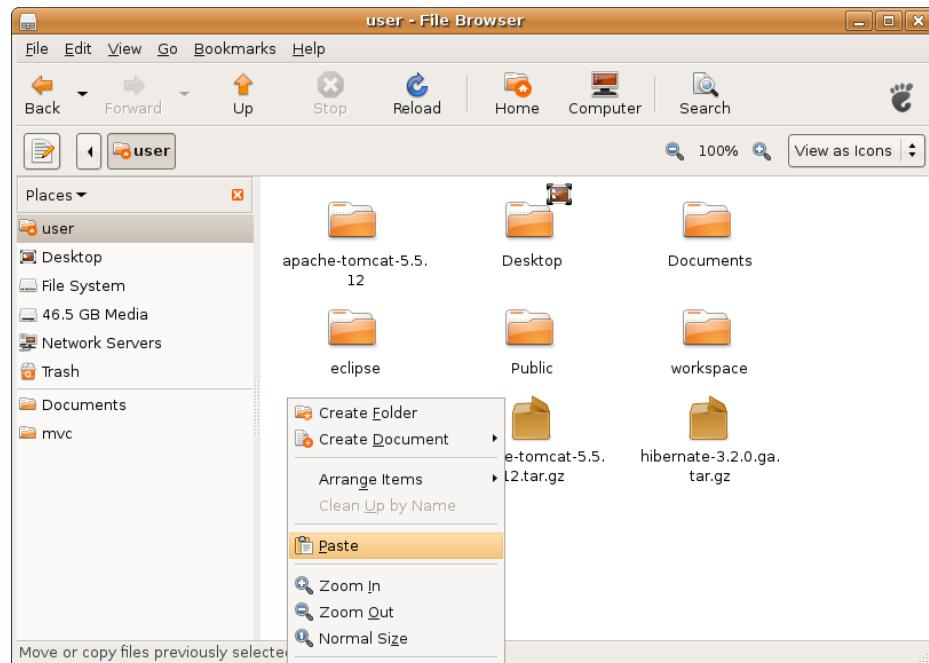
- Entre no diretório caelum, clicando no ícone da caelum no seu Desktop;
- Vá para a pasta 21:



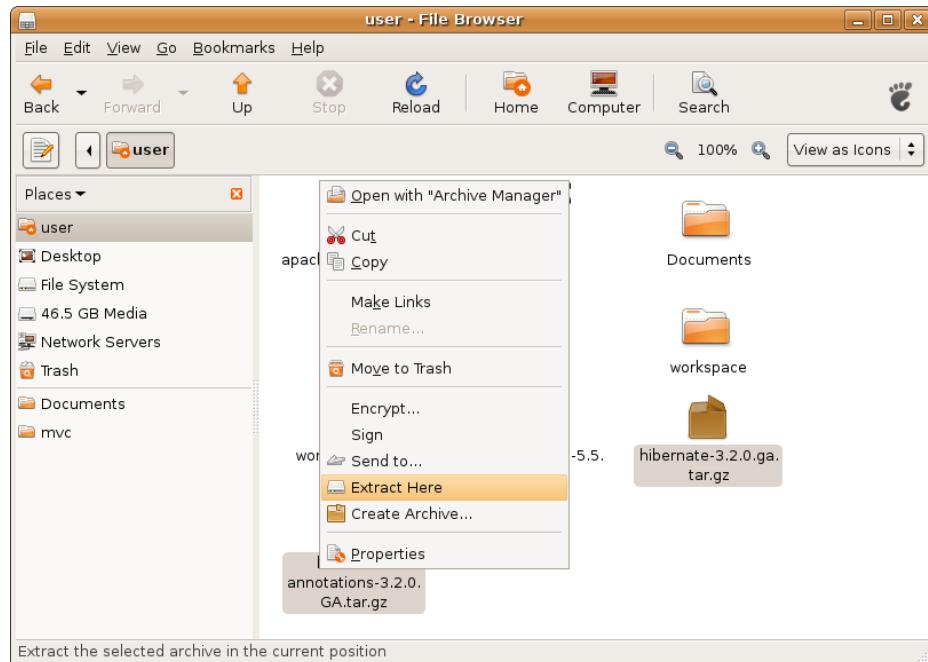
- Marque os dois arquivos do hibernate, clique com o botão direito e escolha *Copy*;



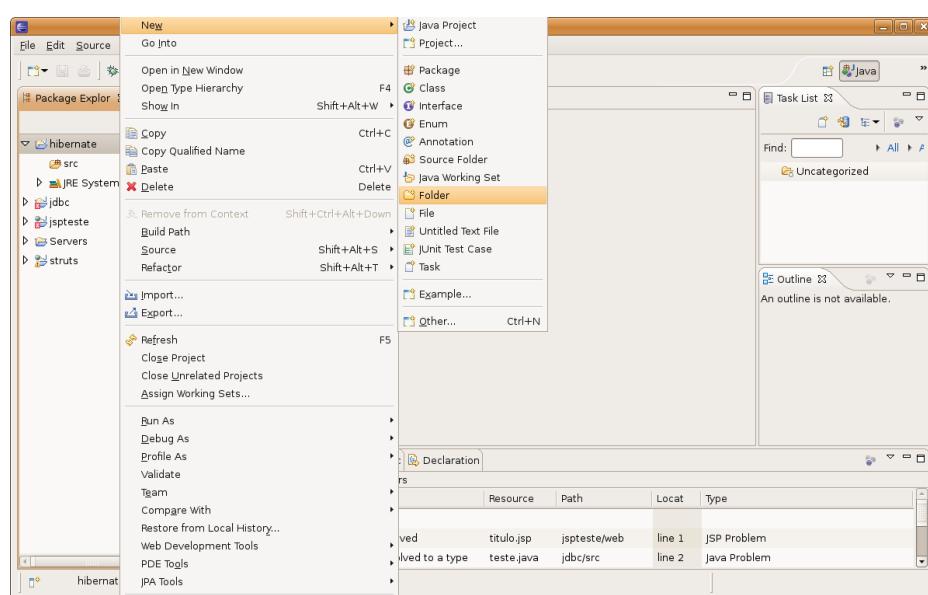
d) Vá para sua pasta padrão: webXXX, clique com o botão direito e escolha *Paste*;



e) Marque os dois arquivos, clique com o botão direito e escolha *Extract here*;

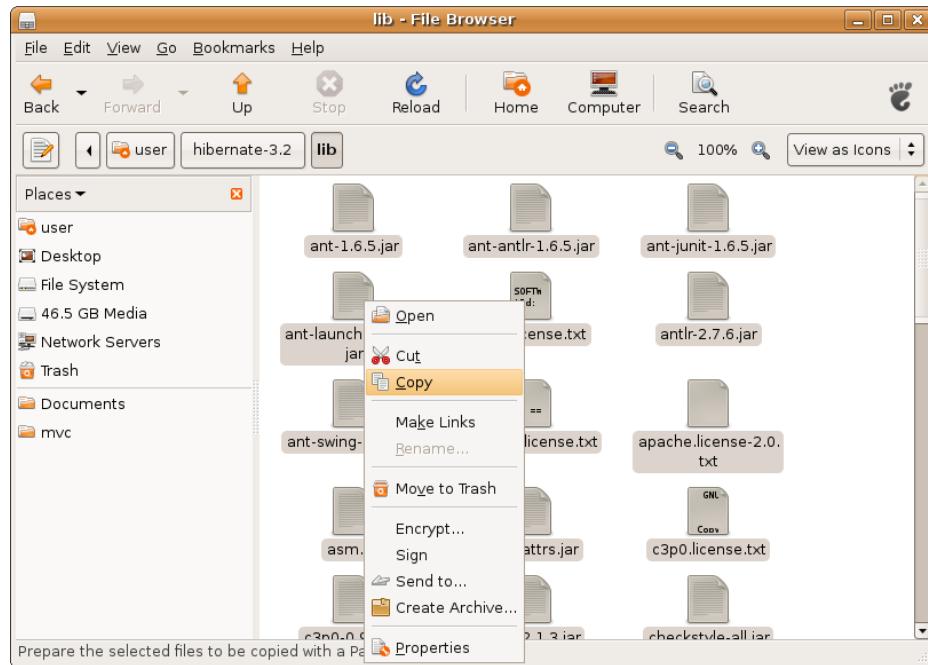


- f) Na pasta do seu usuário foram criadas duas pastas do hibernate, nelas estão os jar's que você utilizará no seu projeto.
- 3) Copie os jars do hibernate para a pasta **lib** do seu projeto.
- a) Clique com o botão direito no nome do projeto do hibernate e escolha *New -> Folder*. Escolha **lib** como nome dessa pasta.

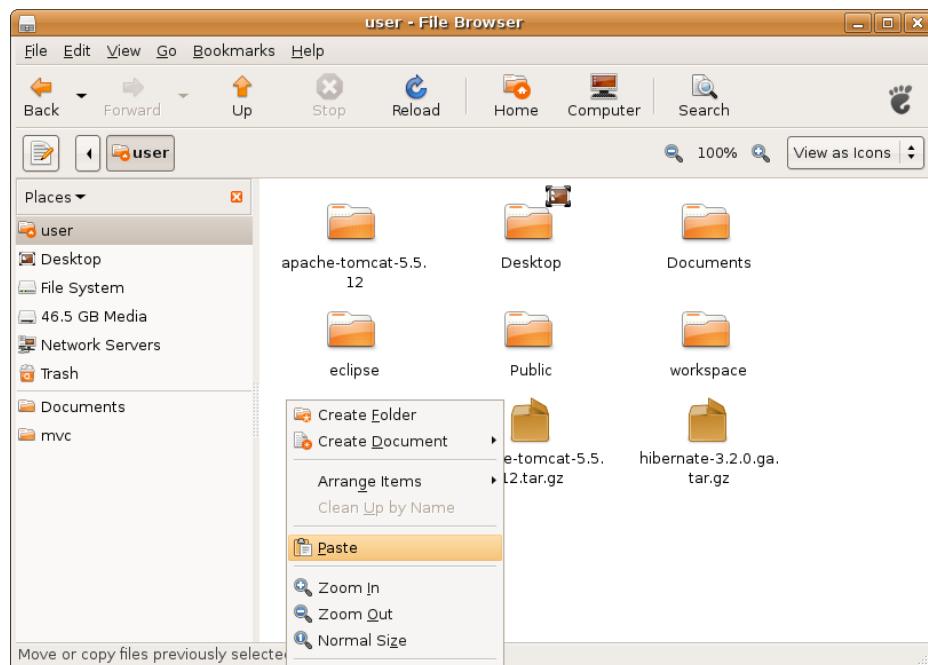


- b) Vá para a pasta do seu usuário e copie os jar's *hibernate-3.2/hibernate3.jar* e *hibernate-annotations-3.2.0-GA/hibernate-annotations.jar* para pasta *workspace/hibernate/lib*
- c) Volte para a pasta do seu usuário. Abra a pasta *hibernate-3.2/lib*. Selecione todos os arquivos (Ctrl+A), clique da direita em um deles e escolha *Copy*.

Conheça nossos treinamentos e aprenda Java com a Caelum

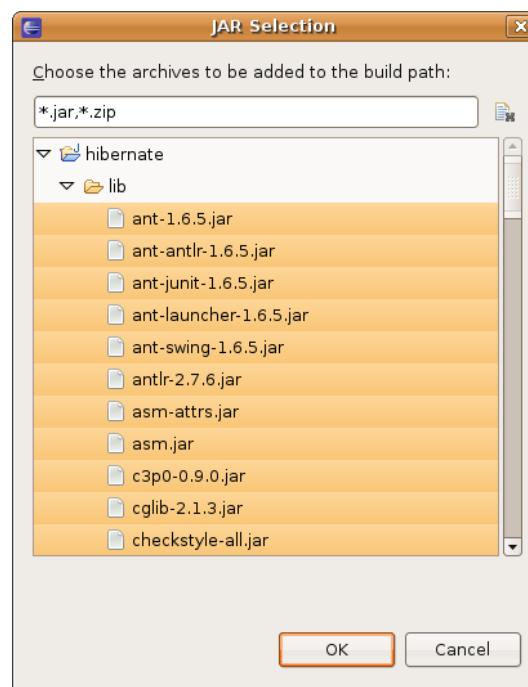
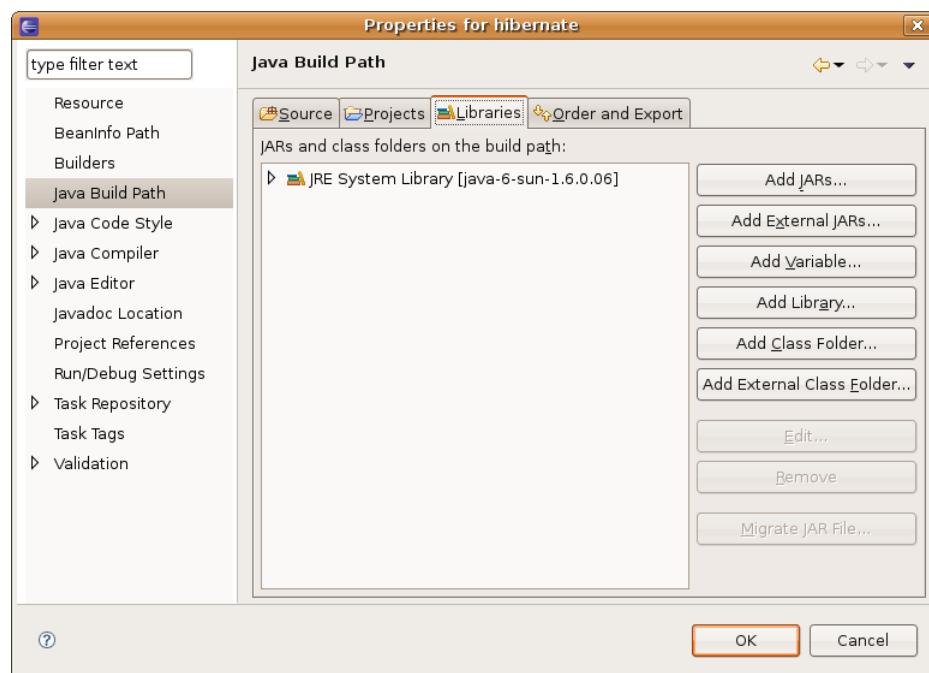


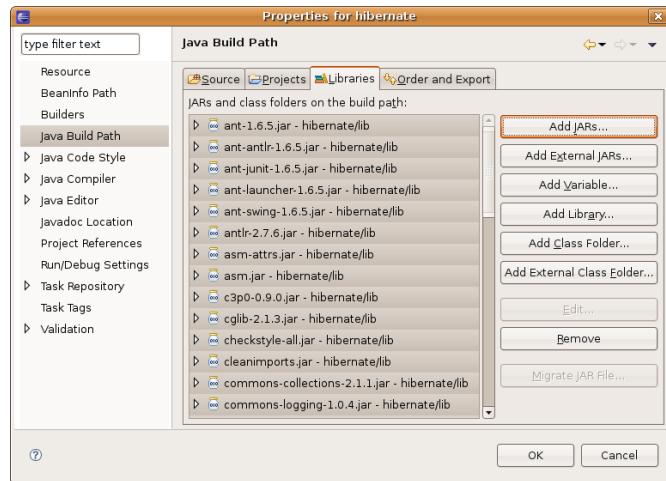
- d) Vá para a pasta do seu usuário e então em *workspace/hibernate/lib*. Clique da direita e escolha *Paste* para colar os jars nessa pasta.



- e) Agora repita esses dois últimos passos para a pasta *hibernate-annotations-3.2.0-GA/lib*.
- 4) Copie o jar do driver do mysql para a pasta lib do projeto também. Você pode achá-lo na pasta *workspace/jdbc*.
- 5) Vamos adicionar os jars no classpath do eclipse:
- Clique da direita no nome do seu projeto, escolha Refresh.
  - Clique novamente da direita, escolha o menu *Properties*;

- c) Escolha a opção *Java Build Path*;
- d) Escolha a aba *Libraries*;
- e) Escolha a opção *Add JARs* e selecione todos os jars do diretório lib;





6) Crie uma classe chamada `Produto` no pacote `br.com.caelum.hibernate`.

7) Adicione as seguintes variáveis membro:

```
private Long id;
private String nome;
private String descricao;
private Double preco;
```

8) Gere os getters e setters usando o eclipse. (*Source -> Generate Getters and Setters* ou *Ctrl + 3 -> ggas*)

9) Anote a sua classe como uma entidade de banco de dados. Lembre-se de importar as anotações do pacote `javax.persistence`.

```
@Entity
public class Produto {
```

}

10) Anote seu field `id` como chave primária e como campo auto-gerado:

```
@Id
@GeneratedValue
private Long id;
```

## 16.6 - Propriedades do banco

Precisamos criar nosso arquivo de configuração, o `hibernate.properties`.

Os dados que vão nesse arquivo são específicos do hibernate e podem ser bem avançados, sobre controle de cache, transações, connection pool etc, tópicos que são abordados no curso FJ-26.

Para nosso sistema, precisamos de quatro linhas básicas, que configuram o banco, o driver, o usuário e senha, que já conhecemos, e uma linha adicional, que diz para o hibernate qual dialeto de SQL ele deve “falar”: o dialeto do MySQL.

Uma das maneiras mais práticas é copiar o arquivo de mesmo nome que está no diretório etc, do hibernate descompactado que você baixou, no diretório src de sua aplicação.

Por padrão a configuração está de tal maneira que o hibernate irá usar um banco de dados do tipo Hyper-sonicSQL. Comente as linhas do mesmo (colocando # no começo). Se você copiar tal arquivo, descomente a parte que utiliza o mysql e configure corretamente a mesma, por exemplo:

```
hibernate.dialect = org.hibernate.dialect.MySQLDialect
hibernate.connection.driver_class = com.mysql.jdbc.Driver
hibernate.connection.url = jdbc:mysql://localhost/teste
hibernate.connection.username = root
hibernate.connection.password =
```

## 16.7 - Exercícios

- 1) Crie o arquivo **hibernate.properties** no seu diretório **src**.

```
hibernate.dialect = org.hibernate.dialect.MySQLDialect
hibernate.connection.driver_class = com.mysql.jdbc.Driver
hibernate.connection.url = jdbc:mysql://localhost/teste
hibernate.connection.username = root
hibernate.connection.password =
```

## 16.8 - Configurando

Nosso primeiro passo é configurar o hibernate, portanto iniciamos instanciando uma `org.hibernate.cfg.AnnotationConfiguration`.

```
// Cria uma configuração para a classe Produto
AnnotationConfiguration cfg = new AnnotationConfiguration();
```

A partir daí podemos adicionar quantas classes desejarmos a nossa configuração.

```
// Adiciona a classe Produto
cfg.addAnnotatedClass(Produto.class);
```

No nosso caso, iremos adicionar somente a nossa classe `Produto`, gerando o seguinte resultado para configurar o Hibernate:

```
// Cria uma configuração para a classe Produto
AnnotationConfiguration cfg = new AnnotationConfiguration();
// Adiciona a classe Produto
cfg.addAnnotatedClass(Produto.class);
```

Só isso não é suficiente para que o Hibernate esteja configurado com a classe `Produto`. O Hibernate requer que descrevamos como a classe se relaciona com as tabelas no banco de dados e fizemos isso através das anotações do Hibernate.

Essa configuração poderia ser feita através de um arquivo XML chamado `hibernate.cfg.xml`, e em vez de utilizarmos as chamadas ao método `addAnnotatedClass` executaríamos o método a seguir:

```
// lê o arquivo hibernate.cfg.xml
cfg.configure();
```

Tal arquivo não será utilizado nesse curso mas sim no curso de Laboratório de MVC e Hibernate com JSF avançado (FJ-26), onde serão mostrados todos os detalhes do hibernate.

## 16.9 - Criando as tabelas

Vamos criar um programa que gera as tabelas do banco. Dada uma configuração, a classe SchemaExport é capaz de gerar o código DDL de criação de tabelas em determinado banco (no nosso caso, o MySQL).

Para exportar tais tabelas, fazemos uso do método create que recebe dois argumentos booleanos. O primeiro diz se desejamos ver o código DDL e o segundo se desejamos executá-lo.

```
new SchemaExport(cfg).create(true, true);
```

## 16.10 - Exercícios

- 1) Crie a classe GeraTabelas.

```
package br.com.caelum.hibernate;

public class GeraTabelas {

    public static void main(String[] args) {
        // Cria uma configuração para a classe Produto
        AnnotationConfiguration cfg = new AnnotationConfiguration();
        cfg.addAnnotatedClass(Produto.class);

        new SchemaExport(cfg).create(true, false);
    }
}
```

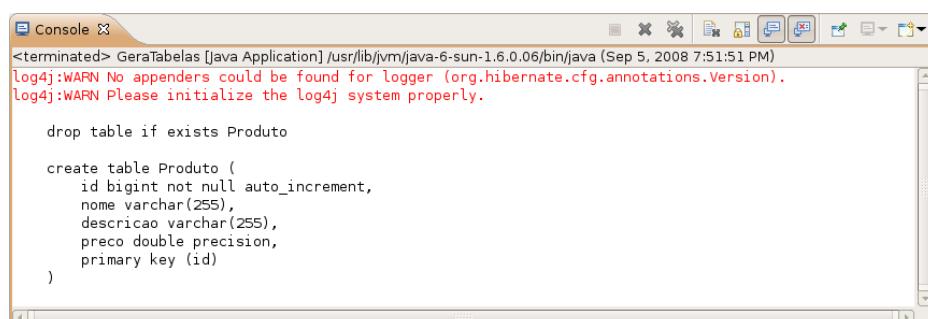
- 2) Adicione as seguintes linhas no seu arquivo `hibernate.properties`.

```
hibernate.show_sql = true
hibernate.format_sql = true
```

Essas linhas fazem com que todo sql gerado pelo hibernate apareça no console.

- 3) Crie suas tabelas executando o código anterior. Clique da direita no meio do código e vá em *Run As -> Java Application*.

O Hibernate deve reclamar que não configuramos nenhum arquivo de log para ele (dois warnings) e mostrar o código SQL que ele executou no banco.



```
Console >
<terminated> GeraTabelas [Java Application] /usr/lib/jvm/java-6-sun-1.6.0.06/bin/java (Sep 5, 2008 7:51:51 PM)
log4j:WARN No appenders could be found for logger (org.hibernate.cfg.annotations.Version).
log4j:WARN Please initialize the log4j system properly.

drop table if exists Produto

create table Produto (
    id bigint not null auto_increment,
    nome varchar(255),
    descricao varchar(255),
    preco double precision,
    primary key (id)
)
```

Se o seu resultado não foi esse, você pode seguir a dica a seguir para configurar o log do hibernate.

### 16.11 - Dica: log do Hibernate

Você pode configurar o log do hibernate para verificar exatamente o que ele está fazendo, sendo tal atitude recomendada para todos os seus projetos.

Uma vez que essa parte de configuração do log não é o foco do curso, deixamos a dica aqui para você copiar tal configuração do próprio zip do hibernate para seu projeto.

- 1) Vá para o Desktop, escolha o File Browser;
- 2) Entre no diretório que você descompactou o hibernate;
- 3) Entre no diretório etc;
- 4) Copie o arquivo log4j.properties;
- 5) Volte para sua Home;
- 6) Entre no diretório workspace/hibernate/src;
- 7) Cole o arquivo log4j.properties nesse diretório;
- 8) Execute um refresh do seu projeto no eclipse (clique da direita no nome do projeto, Refresh).

Agora já podemos pegar uma fábrica de sessões do tipo SessionFactory, para isso basta chamar o método buildSessionFactory do objeto cfg.

```
package br.com.caelum.hibernate;

public class TesteDeConfiguracao {

    public static void main(String[] args) {

        AnnotationConfiguration cfg = new AnnotationConfiguration();
        cfg.addAnnotatedClass(Produto.class);
        SessionFactory factory = cfg.buildSessionFactory();
        factory.close();
    }
}
```

O Hibernate gera sessões através dessa factory. Essas sessões são responsáveis por se conectar ao banco de dados e persistir e buscar objetos no mesmo.

A maneira mais simples de buscar uma nova sessão e fechar a mesma é:

```
package br.com.caelum.hibernate;

public class TesteDeConfiguracao {

    public static void main(String[] args) {

        AnnotationConfiguration cfg = new AnnotationConfiguration();
```

```
cfg.addAnnotatedClass(Produto.class);

SessionFactory factory = cfg.buildSessionFactory();

// cria a sessão
Session session = factory.openSession();

// fecha a sessão
session.close();

factory.close();
}
```

## 16.12 - HibernateUtil

Vamos criar agora uma classe `HibernateUtil` que cuidará de:

- Instanciar a `SessionFactory` do `Hibernate`;
- Nos dar `Sessions` do `hibernate` quando solicitada.

```
public class HibernateUtil {

    private static SessionFactory factory;

    static {
        AnnotationConfiguration cfg = new AnnotationConfiguration();
        cfg.addAnnotatedClass(Produto.class);
        factory = cfg.buildSessionFactory();
    }

    public Session getSession() {
        return factory.openSession();
    }
}
```

O bloco estático das linhas 4 a 8 cuidará de configurar o `Hibernate` e pegar uma `SessionFactory`. Lembre-se que o bloco estático é executado automaticamente quando a classe é carregada pelo Class Loader e só neste momento; ele não será executado outras vezes, como quando você der `new HibernateUtil()`.

O método `getSession` devolverá uma `Session`, conseguida através do `SessionFactory` do `Hibernate`.

## 16.13 - Exercícios

- 1) Crie a sua classe `HibernateUtil` no pacote `br.com.caelum.hibernate`. No momento de importar `Session` lembre-se que não é a `classic`!

```
package br.com.caelum.hibernate;
```

```
public class HibernateUtil {  
  
    private static SessionFactory factory;  
  
    static {  
        AnnotationConfiguration cfg = new AnnotationConfiguration();  
        cfg.addAnnotatedClass(Produto.class);  
        factory = cfg.buildSessionFactory();  
    }  
  
    public Session getSession() {  
        return factory.openSession();  
    }  
}
```

## 16.14 - Erros comuns

O erro mais comum ao criar a classe `HibernateUtil` está em importar `org.hibernate.classic.Session` ao invés de `org.hibernate.Session`. Uma vez que o método `openSession` devolve uma `Session` que não é do tipo `classic`, o Eclipse pede para fazer um casting. Não faça o casting! Remova o seu import e adiciona o import correto.

## 16.15 - Salvando novos objetos

Através de um objeto do tipo `Session` é possível gravar novos objetos do tipo `Produto` no banco. Para tanto basta criar o objeto e depois utilizar o método `save`.

```
Produto p = new Produto();  
p.setNome("Nome aqui");  
p.setDescricao("Descrição aqui");  
p.setPreco(100.50);  
  
Session session = new HibernateUtil().getSession();  
session.save(p);  
System.out.println("ID do produto: " + p.getId());  
session.close();
```

## 16.16 - Exercícios

- 1) Crie uma classe chamada `AdicionaProduto` no pacote `br.com.caelum.hibernate`.

```
package br.com.caelum.hibernate;  
  
//Imports aqui CTRL+SHIFT+O  
  
public class AdicionaProduto {  
  
    public static void main(String[] args) {  
  
        Produto p = new Produto();
```

```

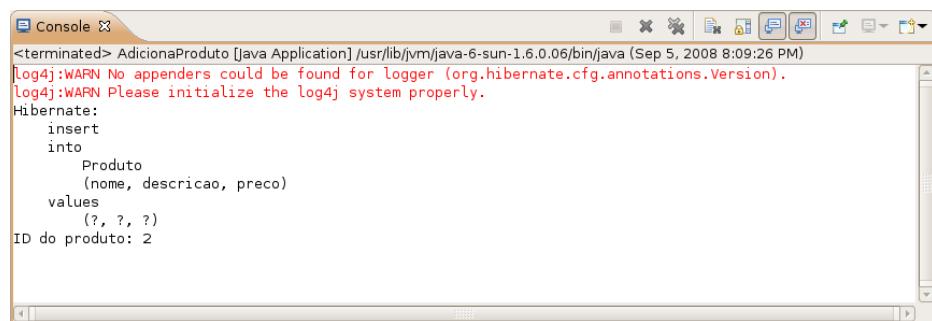
        p.setNome("Nome aqui");
        p.setDescricao("Descrição aqui");
        p.setPreco(100.50);

        Session session = new HibernateUtil().getSession();
        session.save(p);
        System.out.println("ID do produto: " + p.getId());

        session.close();
    }
}

```

- 2) Rode a classe e adicione cerca de 5 produtos no banco. Saída possível:



The screenshot shows a Java console window titled 'Console'. The output is as follows:

```

<terminated> AdicionaProduto [java Application] /usr/lib/jvm/java-6-sun-1.6.0.06/bin/java (Sep 5, 2008 8:09:26 PM)
[Log4j:WARN No appenders could be found for logger (org.hibernate.cfg.annotations.Version).
Log4j:WARN Please initialize the log4j system properly.
Hibernate:
    insert
    into
        Produto
        (nome, descricao, preco)
    values
        (?, ?, ?)
ID do produto: 2

```

## 16.17 - Buscando pelo id

Para buscar um objeto pela chave primária, no caso o seu id, utilizamos o método `load`, conforme o exemplo a seguir:

```

Session session = new HibernateUtil().getSession();
Produto encontrado = (Produto) session.load(Produto.class, 1L);
System.out.println(encontrado.getNome());

```

## 16.18 - Criando o ProdutoDAO

Com os métodos que já conhecemos, podemos criar uma classe DAO para o nosso Produto:

```

public class ProdutoDAO {

    private Session session;

    public ProdutoDAO (Session session) {
        this.session = session;
    }

    public void salva (Produto p) {
        this.session.save(p);
    }

    public void remove (Produto p) {
        this.session.delete(p);
    }
}

```

```
}

public Produto procura (Long id) {
    return (Produto) this.session.load(Produto.class, id);
}

public void atualiza (Produto p) {
    this.session.update(p);
}
}
```

Através desse DAO, podemos salvar, remover, atualizar e procurar Produtos.

## 16.19 - Exercícios

- 1) Faça uma classe chamada ProdutoDAO dentro do pacote br.com.caelum.hibernate.dao

```
package br.com.caelum.hibernate.dao;

//imports aqui CTRL+SHIFT+O

public class ProdutoDAO {

    private Session session;

    public ProdutoDAO (Session session) {
        this.session = session;
    }

    public void salva (Produto p) {
        this.session.save(p);
    }

    public void remove (Produto p) {
        this.session.delete(p);
    }

    public Produto procura (Long id) {
        return (Produto) this.session.load(Produto.class, id);
    }

    public void atualiza (Produto p) {
        this.session.update(p);
    }
}
```

- 2) Adicione diversos objetos diferentes no banco de dados usando sua classe ProdutoDAO.

```
Session session = new HibernateUtil().getSession();
ProdutoDAO dao = new ProdutoDAO(session);
Produto produto = new Produto();
//...
```

```
dao.salva(produto);  
session.close();
```

- 3) Busque um id inválido, que não existe no banco de dados e descubra qual o erro que o Hibernate gera. Ele retorna null? Ele joga uma Exception? Qual?

#### session.flush() e transações

Note que os métodos `remove` e `update` precisam de `session.flush()`(ou transações). Isso na verdade é um detalhe do driver do MySQL com o JDBC. Sendo assim, sempre utilizamos o método `session.flush()` ou transações após a remoção ou atualização de algum objeto, independentemente de qual banco estamos utilizando. Na prática não existe obrigação do driver enviar o statement enquanto você não forcá-lo com o flush.

O uso clássico de uma transação é bem simples:

```
Transaction tx = session.beginTransaction();  
// executa tarefas  
tx.commit();
```

## 16.20 - Buscando com uma cláusula where

O Hibernate possui uma linguagem própria de queries para facilitar a busca de objetos. Por exemplo, o código a seguir mostra uma pesquisa que retorna todos os produtos com id maior que 2:

```
Session session = new HibernateUtil().getSession();  
List<Produto> lista = null;  
lista = session.createQuery("from br.com.caelum.hibernate.Produto where id>2").list();  
for (Produto atual : lista) {  
    System.out.println(atual.getNome());  
}
```

## 16.21 - ProdutoDAO: Listar tudo e fazer paginação

Vamos incluir mais três métodos na nossa classe `ProdutoDAO`.

Primeiro, vamos listar todos os produtos existentes no banco de dados. Para isso, vamos usar o método `createCriteria` de `Session` que cria um `Criteria`. Através de um `Criteria`, temos acesso a diversas operações no banco de dados; uma delas é listar tudo com o método `list()`.

Nosso método `listaTudo()` fica assim:

```
public List<Produto> listaTudo() {  
    return this.session.createCriteria(Produto.class).list();  
}
```

Mas o método acima devolve a lista com todos os produtos no banco de dados. Em um sistema com listagens longas, normalmente apresentamos a lista por páginas. Para implementar paginação, precisamos determinar que a listagem deve começar em um determinado ponto e ser de um determinado tamanho.

Usando o Criteria, como no listaTudo anteriormente, isso é bastante simples. Nosso método página fica assim:

```
public List<Produto> pagina (int inicio, int quantia) {  
    return this.session.createCriteria(Produto.class)  
        .setMaxResults(quantia).setFirstResult(inicio).list();  
}
```

O método `setMaxResults` determina o tamanho da lista (resultados por página) e o método `setFirstResult` determina em que ponto a listagem deve ter início. Por fim, basta chamar o método `list()` e a listagem devolvida será apenas daquela página!

E vamos adicionar também o método `listaAPartirDoTerceiro`:

```
public List<Produto> listaAPartirDoTerceiro (){  
    return this.session.createQuery("from Produto where id > 2").list();  
}
```

Poderíamos passar alguns parâmetros para essa busca, mas isso só será visto no cursos de Web Avançado.

## 16.22 - Exercícios

1) Modifique a sua classe `ProdutoDAO` e acrescente os métodos `listaTudo()` e `pagina()`

```
public List<Produto> listaTudo() {  
    return this.session.createCriteria(Produto.class).list();  
}  
  
public List<Produto> pagina (int inicio, int quantia) {  
    return this.session.createCriteria(Produto.class)  
        .setMaxResults(quantia).setFirstResult(inicio).list();  
}  
  
public List<Produto> listaAParitrDoTerceiro (){  
    return this.session.createQuery("from Produto where id > 2").list();  
}
```

2) Crie uma classe chamada `TestaBuscas`:

```
public class TestaBuscas {  
  
    public static void main(String [] args){  
        Session session = new HibernateUtil().getSession();  
        ProdutoDAO produtoDao = new ProdutoDAO(session);  
  
        System.out.println("*****Listando Tudo*****");  
        for(Produto p : produtoDao.listaTudo()) {  
            System.out.println(p.getNome());  
        }  
  
        System.out.println("*****Listando Paginado*****");  
        for(Produto p : produtoDao.pagina(2,3)) {
```

```
        System.out.println(p.getNome());
    }

    System.out.println("*****Listando a partir do terceiro*****");
    for(Produto p : produtoDao.listaAPartirDoTerceiro()) {
        System.out.println(p.getNome());
    }
}
```

## 16.23 - Exercícios para o preguiçoso

- 1) Mude a propriedade `hibernate.show_sql` para `true` no arquivo `hibernate.properties` e rode a classe acima.
- 2) Teste um programa que faz somente o seguinte: busca um produto por id. O código deve somente buscar o produto e não imprimir nada! Qual o resultado?

```
Session session = new HibernateUtil().getSession();
Produto encontrado = (Produto) session.load(Produto.class, 1L);
```

- 3) Tente imprimir o nome do produto do teste anterior, o que acontece?

```
Session session = new HibernateUtil().getSession();
Produto encontrado = (Produto) session.load(Produto.class, 1L);
System.out.println(encontrado.getNome());
```

- 4) Antes de imprimir o nome do produto, tente imprimir uma mensagem qualquer, do tipo: “O select já foi feito”. E agora? Como isso é possível?

```
Session session = new HibernateUtil().getSession();
Produto encontrado = (Produto) session.load(Produto.class, 1L);
System.out.println("O select já foi feito");
System.out.println(encontrado.getNome());
```

Então, onde está o código do select? Ele deve estar no método `getNome()`, certo?

- 5) Imprima o nome da classe do objeto referenciado pela variável `encontrado`:

```
Session session = new HibernateUtil().getSession();
Produto encontrado = (Produto) session.load(Produto.class, 1L);
System.out.println("O select já foi feito");
System.out.println(encontrado.getNome());
System.out.println(encontrado.getClass().getName());
```

O Hibernate retorna um objeto cujo tipo estende `Produto`: ele não deixa de ser um `Produto` mas não é somente um `Produto`.

O método `getNome` foi sobrescrito nessa classe para fazer a busca na primeira vez que é chamado, economizando tempo de processamento.

É claro que para fazer o *fine-tuning* do Hibernate é interessante conhecer muito mais a fundo o que o Hibernate faz e como ele faz isso.

## 16.24 - Exercício opcional

- 1) Crie um sistema para cadastro e listagem de produtos usando o struts. Siga o padrão que utilizamos para cadastro e listagem de contatos.

# VRaptor

*“Aquele que castiga quando está irritado, não corrige, vinga-se”*  
– Michel de Montaigne

Neste capítulo, você aprenderá:

- O que é Inversão de Controle, Injeção de Dependências e *Convention over Configuration*;
- Como utilizar um framework MVC baseado em tais idéias;
- Como abstrair a camada HTTP da sua lógica de negócios;
- Como *não* utilizar arquivos XML para configuração da sua aplicação.

## 17.1 - Eu não quero o que eu não conheço

No mundo Java para a web, vamos lembrar como fica um código utilizando um controlador MVC simples para acessar os parâmetros enviados pelo cliente.

É fácil notar como as classes, interfaces e apetrechos daquele controlador infectam o nosso código e começamos a programar voltado a tal framework. O código a seguir mostra uma ação que utiliza um DAO para incluir um contato no banco de dados.

```
public class AdicionaContato implements Action {

    public String executa(HttpServletRequest req, HttpServletResponse res) throws Exception{
        Contato contato = new Contato();
        contato.setNome(req.getParameter("nome"));
        contato.setEndereco(req.getParameter("endereco"));
        contato.setEmail(req.getParameter("email"));

        ContatoDAO dao = new ContatoDAO();
        dao.adiciona(contato);

        return "/ok.jsp";
    }
}
```

Baseado no código acima, percebemos que estamos fortemente atrelados a `HttpServletRequest` e seu método `getParameter`. Fora isso, usamos diversas classes estranhas ao nosso projeto: `Action`, `HttpServletRequest` e `HttpServletResponse`. Se estivéssemos controlando melhor a conexão, seria necessário importar `Connection` também! Nenhuma dessas classes e interfaces citadas faz parte do nosso projeto! Não é o código que modela minha lógica!

É muito chato, e nada prático, repetir isso em toda a sua aplicação. Sendo assim, visando facilitar esse tipo de trabalho, vimos que o *Struts Action*, por exemplo, utiliza alguns recursos que facilitam o nosso trabalho:

```
public class AdicionaContato extends Action {  
  
    public ActionForward execute(ActionMapping map, ActionForm form,  
        HttpServletRequest req, HttpServletResponse res)  
        throws Exception {  
  
        Contato contato = ((ContatoForm) form).getContato();  
  
        ContatoDAO dao = new ContatoDAO();  
        dao.adiciona(contato);  
  
        return map.findForward("ok");  
    }  
}
```

Mas, mesmo assim, imagine os limites de tal código:

- Você **não** pode receber mais de um action form;
- Sua classe **deve** estender `ActionForm`. Se você queria estender outra, azar;
- Você **deve** receber todos esses argumentos que não foi você quem criou;
- Você **deve** retornar esse tipo que não foi você quem criou;
- Você **deve** trabalhar com Strings ou tipos muito pobres. O sistema de conversão é complexo para iniciantes;
- O código fica muito alienado: ele é escrito de tal forma que o programador precisa agradar o framework e não o framework para agradar o programador;
- Você acaba criando classes repetidas: deve criar dois beans repetidos ou parecidos, ou ainda escrever muito código xml para substituir um deles.

De tais problemas surgiram diversos outros frameworks, inclusive diversos patterns novos, entre eles, Injeção de Dependências (*Dependency Injection*), Inversão de Controle (*Inversion of Control – IoC*) e **prefira** convenções em vez de configuração (*Convention over Configuration – CoC*).

Imagine deixar de estender `Action`:

```
public class AdicionaContato {  
    public ActionForward execute(ActionMapping map, ActionForm form,  
        HttpServletRequest req, HttpServletResponse res)  
        throws Exception {  
        Contato contato = ((ContatoForm) form).getContato();  
  
        ContatoDAO dao = new ContatoDAO();  
        dao.adiciona(contato);  
  
        return map.findForward("ok");  
    }  
}
```

}

Nesse momento estamos livres para mudar nosso método execute. Desejamos que ele se chame adiciona, e não execute, sem contar que não precisamos daqueles quatro argumentos. Afinal, não utilizamos o request e response:

```
public class AdicionaContato {  
    public ActionForward adiciona(ActionMapping map, ActionForm form) throws Exception {  
  
        Contato contato = ((ContatoForm) form).getContato();  
  
        ContatoDAO dao = new ContatoDAO();  
        dao.adiciona(contato);  
  
        return map.findForward("ok");  
    }  
}
```

Aos poucos, o código vai ficando mais simples. Vamos agora devolver ok, em vez de usar esse ActionMapping, que aliena o nosso código até esse momento:

```
public class AdicionaContato {  
    public String adiciona(ActionForm form) throws Exception {  
  
        Contato contato = ((ContatoForm) form).getContato();  
  
        ContatoDAO dao = new ContatoDAO();  
        dao.adiciona(contato);  
  
        return "ok";  
    }  
}
```

Por fim, em vez de criar uma classe que estende ActionForm, ou configurar toneladas de XML, desejamos receber um Contato como parâmetro do método.

Portanto nada mais natural que o parâmetro seja Contato, e não ContatoForm.

```
public class AdicionaContato {  
  
    public String adiciona(Contato contato) throws Exception {  
  
        ContatoDAO dao = new ContatoDAO();  
        dao.adiciona(contato);  
  
        return "ok";  
    }  
}
```

O resultado é um código bem mais legível, e um controlador menos intrusivo no seu código.

## 17.2 - Vantagens

Você desconecta o seu programa da camada web, criando ações ou comandos que não trabalham com `request` e `response`.

Note que, enquanto utilizávamos a lógica daquela maneira, o mesmo servia de adaptador para a web. Agora não precisamos mais desse adaptador, nossa própria lógica que é uma classe Java comum, pode servir para a web ou a qualquer outro propósito.

Além disso, você recebe todos os objetos que precisa para trabalhar, não se preocupando em buscá-los. Isso é chamado de **injeção de dependências**. Por exemplo, se você precisa do usuário logado no sistema e, supondo que ele seja do tipo `Funcionario`, você pode criar um construtor que requer tal objeto:

```
public class AdicionaContato {  
  
    public AdicionaContato(Funcionario funcionario) {  
        // o parametro é o funcionario logado no sistema  
    }  
  
    public String adiciona(Contato contato) throws Exception {  
  
        ContatoDAO dao = new ContatoDAO();  
        dao.adiciona(contato);  
  
        return "ok";  
    }  
}
```

## 17.3 - Vraptor 2

Tudo o que faremos neste capítulo está baseado no framework **Vraptor 2**. Sua documentação pode ser encontrada em inglês em <http://www.vraptor.org> e em português em <http://www.vraptor.org/pt>. Além disso existe, uma página em português: <http://www.vraptor.com.br> e um curso na Caelum (FJ-28) que também possui a apostila disponível para download.

Iniciativa brasileira, o VRaptor foi criado inicialmente para o desenvolvimento do GUJ 2 ([www.guj.com.br](http://www.guj.com.br)) em 2004, que o utiliza até hoje. O VRaptor também é utilizado, atualmente, para a construção do JForum 3, o fórum escrito em Java mais famoso do mundo.

## 17.4 - Exercícios

Vamos instalar o Vraptor utilizando um projeto novo para o Eclipse.

- 1) Crie um novo **Dynamic Web Project** chamado **vraptor**.
- 2) Clique da direita no nome do projeto **vraptor** e vá em **Import > Archive File**. Importe o arquivo **Desktop/caelum/21/projeto-vraptor.zip**.
- 3) Clique da direita e vá em **Run As -> Run On Server** e teste a URL: <http://localhost:8080/vraptor>

## 17.5 - Internacionalização

Repare que o projeto em branco que foi utilizado já vem com o sistema de internacionalização do JSTL configurado. Abra o arquivo `index.jsp` e verifique a tag

```
<fmt:message key="..." />
```

A chave utilizada nessa tag será buscada no arquivo `messages.properties`, que será mostrado para o usuário que acessa tal página via web.

## 17.6 - A classe de modelo

O projeto já vem com uma classe de modelo pronta, chamada `Produto`, que utilizaremos em nossos exemplos.

```
@Entity  
public class Produto {  
  
    @Id  
    @GeneratedValue  
    private Long id;  
  
    private String nome;  
    private String descricao;  
  
    private Double preco;  
  
    // getters e setters  
  
}
```

A classe `ProdutoDAO` também já existe e utiliza o hibernate para acessar um banco de dado (mysql).

```
public class ProdutoDAO {  
  
    private Session session;  
  
    public ProdutoDao() {  
        this.session = new HibernateUtil().getSession();  
    }  
  
    public void adiciona(Produto p) {  
        Transaction tx = session.beginTransaction();  
        session.save(p);  
        tx.commit();  
    }  
  
    public void atualiza(Produto p) {  
        Transaction tx = session.beginTransaction();  
        session.update(p);  
        tx.commit();  
    }  
  
    @SuppressWarnings("unchecked")
```

```
public List<Produto> lista() {  
    return session.createCriteria(Produto.class).list();  
}  
}
```

O foco desse capítulo não está em como configurar o hibernate ou em boas práticas da camada de persistência portanto o código do DAO pode não ser o ideal por utilizar uma transação para cada chamada de método, mas é o ideal para nosso exemplo.

## 17.7 - Minha primeira lógica de negócios

Vamos agora escrever uma classe que adiciona um `Produto` a um banco de dados através do DAO e do uso do Vraptor 2:

```
@Component("produto")  
public class ProdutoLogic {  
  
    public void adiciona(Produto produto) {  
        new ProdutoDao().adiciona(produto);  
    }  
}
```

Pronto! É assim que fica uma ação de adicionar produto utilizando esse controlador (e podemos melhorar mais ainda!).

E se surgir a necessidade de criar um método `atualiza`? Poderíamos reutilizar a mesma classe para os dois métodos:

```
@Component  
public class ProdutoLogic {  
  
    // a ação adiciona  
    public void adiciona(Produto produto) {  
        new ProdutoDao().adiciona(produto);  
    }  
  
    // a ação atualiza  
    public void atualiza(Produto produto) {  
        new ProdutoDao().atualiza(produto);  
    }  
}
```

O próprio controlador se encarrega de preencher o produto para chamar nosso método. O que estamos fazendo através da anotação `@Component` é dizer para o Vraptor disponibilizar essa classe para ser instanciada e exposta para a web através do nome `produto`.

## 17.8 - Como configurar a minha lógica?

Não precisa. Não existe nenhum XML do VRaptor que seja de preenchimento obrigatório.

## 17.9 - E o JSP com o formulário?

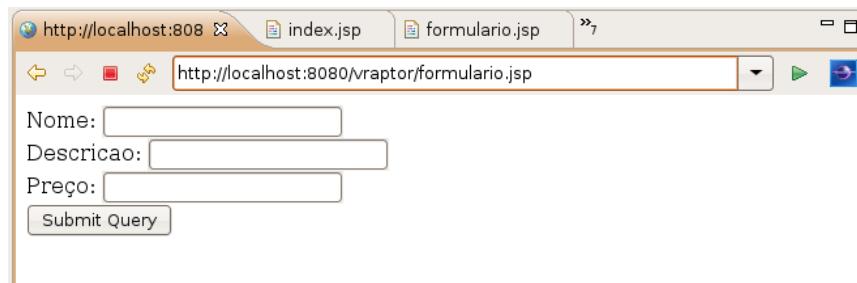
Para fazer com que a lógica seja invocada, basta configurarmos corretamente os campos do formulário no nosso código html. Não há segredo algum.

Criaremos o arquivo WebContent/formulario.jsp que será responsável por invocar a nossa lógica de novo Produto. Mas, qual o link para ela?

Note que foi dado o nome de produto ao nosso componente e o nome do método que desejamos invocar se chama adiciona. Como a extensão padrão do vraptor para a web se chama logic, a URL que devemos acessar é produto.adiciona.logic.

Já os parâmetros devem ser enviados com o nome do tipo da variável que desejamos preencher, no nosso caso, produto:

```
<html>
    <form action="produto.adiciona.logic">
        Nome: <input name="produto.nome"/><br/>
        Descricao: <input name="produto.descricao"/><br/>
        Preço: <input name="produto.preco"/><br/>
        <input type="submit"/>
    </form>
</html>
```



## 17.10 - E a página final?

Precisamos criar uma página que mostre uma mensagem de sucesso, aproveitamos e confirmamos a inclusão mostrando os dados que foram incluídos:

```
<html>
    Seu produto foi adicionado com sucesso!<br/>
</html>
```

Mas qual o nome desse arquivo? Uma vez que o nome do componente é produto, o nome da lógica é adiciona e o resultado é ok, o nome de seu arquivo de saída deve ser: WebContent/produto/adiciona.ok.jsp.

Você pode alterar o resultado de sua lógica devolvendo uma String qualquer e pode ainda alterar completamente o resultado de sua lógica utilizando um arquivo de configuração.

Quanto mais você segue o padrão, menos configuração você possui e mais simples fica para outra pessoa entender o que acontece.

Um exemplo de possível saída diferente seria:

```
// a ação adiciona
public String adiciona(Produto produto) {
    new ProdutoDao().adiciona(produto);
    return "valido";
}
```

E a criação do arquivo `produto/adiciona.valido.jsp`. Uma mesma lógica pode retornar uma entre diversas strings diferentes dependendo do que acontece, e então o usuário será redirecionado para o JSP correspondente.

### Como configurar esse redirecionamento?

Não precisa!

## 17.11 - Exercícios

1) Crie um sistema de inclusão de produtos

a) Crie a classe `ProdutoLogic` no pacote `br.com.caelum.vraptor.logic`

```
@Component("produto")
public class ProdutoLogic {

    // a ação adiciona
    public void adiciona(Produto produto) {
        new ProdutoDao().adiciona(produto);
    }

    // a ação atualiza
    public void atualiza(Produto produto) {
        new ProdutoDao().atualiza(produto);
    }
}
```

b) Crie o arquivo `WebContent/formulario.jsp`

```
<html>
    <form action="produto.adiciona.logic">
        Nome: <input name="produto.nome"/><br/>
        Descrição: <input name="produto.descricao"/><br/>
        Preço: <input name="produto.preco"/><br/>
        <input type="submit"/>
    </form>
</html>
```

c) Crie o diretório `WebContent/produto`

d) Crie o arquivo `adiciona.ok.jsp` no diretório `WebContent/produto`

```
<html>
```

```
Seu produto foi adicionado com sucesso!<br/>
</html>
```

- e) Você importou alguma classe do pacote javax.servlet?
- f) Teste o seu formulário: http://localhost:8080/vraptor/formulario.jsp . Adicione cinco produtos diferentes.

## 17.12 - A lista de produtos

O segundo passo será criar a listagem de todos os produtos do sistema. Nossa lógica de negócios, mais uma vez, possui somente aquilo que precisa:

```
@Component("produto")
public class ListaProdutoLogic {

    private List<Produto> produtos;

    public void lista() {
        this.produtos = new ProdutoDao().lista();
    }
}
```

Mas dessa vez precisamos disponibilizar o conteúdo da variável produtos para a nossa camada de visualização. A maneira mais simples encontrada até hoje de encapsular uma variável é colocá-la como private, como fizemos, e para disponibilizá-la, geramos o seu getter:

```
public List<Produto> getProdutos() {
    return produtos;
}
```

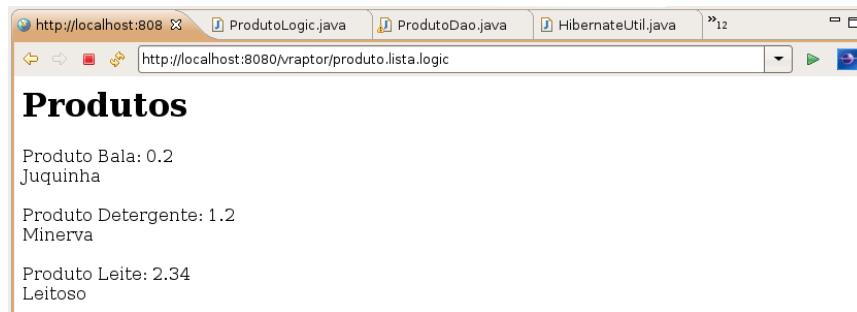
Repare que nossa classe pode ser testada facilmente, basta instanciar o bean `ListaProdutoLogic`, chamar o método `lista` e depois o método `getProdutos`. Todas essas convenções evitando o uso de configurações ajudam bastante no momento de criar testes unitários para o seu sistema.

Por fim, vamos criar o arquivo `lista.ok.jsp` no diretório `WebContent/produto` utilizando a taglib core da JSTL:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<h1>Produtos</h1>
<c:forEach var="produto" items="#{produtos}">

    Produto ${produto.nome}: ${produto.preco}<br/>
    ${produto.descricao}<br/><br/>

</c:forEach>
</html>
```



## 17.13 - Exercícios opcionais

1) Crie um sistema de listagem de produtos

a) Crie a classe `ListaProdutoLogic` no pacote `br.com.caelum.vraptor.logic`

```
@Component("produto")
public class ListaProdutoLogic {

    private List<Produto> produtos;

    public void lista() {
        this.produtos = new ProdutoDao().lista();
    }

    public List<Produto> getProdutos() {
        return produtos;
    }
}
```

b) Crie o arquivo `lista.ok.jsp` no diretório `WebContent/produto`

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<h1>Produtos</h1>
<c:forEach var="produto" items="${produtos}">

    Produto ${produto.nome}: ${produto.preco}<br/>
    ${produto.descricao}<br/><br/>

</c:forEach>
```

c) Teste a sua lógica: <http://localhost:8080/vraptor/produto.lista.logic>.

## 17.14 - Velocity, Freemarker e Sitemesh

O Vraptor funciona sem problemas com outras camadas de visualização como **Velocity** e **Freemarker**, além de ser facilmente acoplado a controles de visualização como o **Sitemesh**.

## 17.15 - Configurações

Todos os padrões mostrados aqui podem ser configurados através de anotações, XML ou alguns arquivos do tipo properties. Para mais informações, consulte a documentação do projeto.

## 17.16 - Um pouco mais...

Através de interceptadores é possível controlar o acesso a um sistema para implementar seu componente de login ou fornecer objetos do tipo DAO para toda a sua aplicação.

## 17.17 - Plugin para o Eclipse

Existem plugin abertos e pagos para o Eclipse se conectar ao VRaptor, que vão desde a visualização de todos os seus componentes passando pela geração automática de listas, páginas do tipo *master-detail*, sistemas de controle de envio de email através de filas de espera e até mesmo a geração de um projeto através de *wizards* no Eclipse.

## 17.18 - Pequenos exemplos de simplicidade

Com o intuito de mostrar o que os controladores tentam fazer hoje em dia, dê uma olhada no código a seguir, que pode ser utilizado com o VRaptor. Repare na injeção de dependências através do construtor e dos parâmetros. Outro ponto importante é a anotação @Role que verifica se o usuário possui ou não tal perfil.

```
@Component
public class ContatoLogic {

    private EmailQueue fila;

    public ContatoLogic(EmailQueue fila) throws SQLException {
        this.fila = fila;
    }

    @Role("gerente")
    public void adiciona(Contato contato) throws SQLException {
        new ContatoDAO().adiciona(contato);
        fila.envia(new Email("de", "para", "conteudo"));
    }

    @Role("admin")
    public void atualiza(Contato contato) throws SQLException {
        new ContatoDAO().altera(contato);
    }
}
```

E uma classe de teste unitário, bem natural utilizando **JUnit**, só código Java, sem configuração extra:

```
public class ContatoLogicTest class TestCase {

    public void testAdicionaNovoContato() throws SQLException {
        Contato c = new Contato();
        c.setNome("Meu nome");
```

```
    new ContatoLogic(new EmailQueue()).adiciona(c);
}
}
```

# E agora?

*“A nuca é um mistério para a vista.”*

– Paul Valéry

Onde continuar ao terminar o curso ‘Java para o desenvolvimento web’.

## 18.1 - Certificação

Entrar em detalhes nos assuntos contidos até agora iriam no mínimo tornar cada capítulo quatro vezes maior do que já é.

Os tópicos abordados (com a adição e remoção de alguns) constituem boa parte do que é cobrado na certificação oficial para desenvolvedores Web da Sun.

Para maiores informações sobre certificações consulte a própria Sun, o <http://www.javaranch.com> ou o <http://www.guj.com.br>, que possui diversas informações sobre o assunto.

## 18.2 - Frameworks

Diversos frameworks foram desenvolvidos para facilitar o trabalho de equipes de desenvolvimento.

Aqueles que pretendem trabalhar com Java devem a qualquer custo analisar as vantagens e desvantagens da maior parte desses frameworks, que diminuem o número de linha de código necessárias e facilitam o controle e organização de uma aplicação.

Por exemplo, o VRaptor é um exemplo de controlador simples e bom para iniciantes. O Hibernate é um ótimo passo para persistência de objetos.

Do mesmo jeito que esses arcabouços surgem e crescem repentinamente, eles podem perder força para outros que tenham novidades criativas. E o ciclo se repete.

## 18.3 - Revistas

Diversas revistas, no Brasil e no exterior, estudam o mundo Java como ninguém e podem ajudar o iniciante a conhecer muito do que está acontecendo lá fora nas aplicações comerciais.

## 18.4 - Grupo de Usuários

Diversos programadores com o mínimo ou máximo de conhecimento se reúnem online para a troca de dúvidas, informações e idéias sobre projetos, bibliotecas e muito mais. Um dos mais importantes e famosos no Brasil é o GUJ – <http://www.guj.com.br>

## 18.5 - Falando em Java - Próximos módulos

O ‘Falando em Java’ não pára por aqui. A Caelum oferece uma grande variedade de cursos que você pode seguir:

**FJ-19:** Preparatório para Certificação Java

**FJ-21:** Java para desenvolvimento Web

**FJ-26:** Laboratório de MVC com JSF e Hibernate para Web

**FJ-28:** Desenvolvimento ágil para Web 2.0 com VRaptor, Hibernate e AJAX

**FJ-31:** Enterprise JavaBeans (EJB)

**FJ-55:** Java para pequenos dispositivos (Java ME)

**FJ-91:** Arquitetura e Design de Projetos Java

**PM-51:** Programação Extrema (XP) com Java

**PM-81:** Gerenciamento de Projetos de Software com Scrum

**RR-11:** Desenvolvimento ágil para Web 2.0 com Ruby on Rails

Consulte mais informações no nosso site e entre em contato conosco.

# Apêndice A - Servlet e JSP API

*“Do rio que tudo arrasta, diz-se que é violento. Mas ninguém chama violentas as margens que o comprimem.”*  
— Bertolt Brecht

Nesse capítulo, você vai aprender a:

- Configurar parâmetros para uma servlet;
- Acessar um escopo global de variáveis em sua aplicação;
- Executar tarefas no início e término de sua aplicação.

## 19.1 - Início e término da sua aplicação

O que acontece se desejamos executar algum código no início e término da nossa aplicação? Isso é, quando o nosso contexto é inicializado e destruído?

Um exemplo clássico é a inicialização das conexões com o banco de dados, dos arquivos de log ou de recursos externos ao sistema.

A API de servlets disponibiliza uma interface chamada `ServletContextListener` capaz de receber notificações desses dois eventos.

Ao implementar tal interface, precisamos escrever dois métodos, um para inicialização e outro para a destruição de nossa aplicação (que também será chamada *servlet context*, contexto de servlets ou ainda, escopo de aplicação).

```
public void contextDestroyed(ServletContextEvent event) {
    // ...
}

public void contextInitialized(ServletContextEvent event) {
    // ...
}
```

Podemos, por exemplo, manter em uma variável o momento de inicialização da nossa aplicação:

```
Date inicializacao = new Date();
```

Mas como ver esse resultado através de uma servlet? Precisamos colocar esse objeto em algum escopo que seja o mesmo para todos os usuários e todas as servlets, isso existe? Sim: o escopo de aplicação.

```
ServletContext context = event.getServletContext();
```

Utilizamos tal escopo como um mapa (a palavra mapa é dos físicos, quer dizer que para cada chave tem um valor associado, nós usamos normalmente a palavra dicionário, mas como em java usa a palavra map traduzimos para mapa) portanto definimos uma chave qualquer para nosso objeto, por exemplo: "inicializacao" e utilizamos o método setAttribute para atribuir o objeto referenciado pela variável inicialização para tal chave:

```
Date inicializacao = new Date();
ServletContext context = event.getServletContext();
context.setAttribute("inicializacao", inicializacao);
```

Falta ainda configurar nossa aplicação para procurar por tal Listener e registrá-lo devidamente. Como isso é na realidade uma configuração, cadastraremos um listener no nosso web.xml:

```
<listener>
    <listener-class>br.com.caelum.servlet.ControleDeAplicacao</listener-class>
</listener>
```

## 19.2 - Exercícios

1) Crie a classe ControleDeAplicacao no pacote br.com.caelum.servlet. Escolha o menu File > New > Class.

a) Implemente ServletContextListener.

```
public class ControleDeAplicacao implements ServletContextListener {
```

```
}
```

b) Utilize o CTRL+SHIFT+O

c) Resolva os problemas que o compilador reclama utilizando o CTRL+1 na linha do erro: implemente os dois métodos:

```
public void contextDestroyed(ServletContextEvent event) {
```

```
}
```

```
public void contextInitialized(ServletContextEvent event) {
```

```
    Date inicializacao = new Date();
```

```
    ServletContext context = event.getServletContext();
```

```
    context.setAttribute("inicializacao", inicializacao);
```

```
}
```

2) Abra o arquivo web.xml e mapeie o seu listener.

```
<listener>
    <listener-class>br.com.caelum.servlet.ControleDeAplicacao</listener-class>
</listener>
```

## 19.3 - getServletContext()

Toda servlet possui um método chamado `getServletContext` que retorna uma referência para o nosso contexto de aplicação:

```
ServletContext aplicacao = getServletContext();
```

A partir daí basta buscar o valor do atributo inicialização:

```
Date inicializacao = (Date) aplicacao.getAttribute("inicializacao");
```

E imprimir os resultados:

```
Date agora = new Date();

long diferenca = agora.getTime() - inicializacao.getTime();
double minutos = diferenca / (60 * 1000.0);

PrintWriter writer = response.getWriter();
writer.println("<html>");
writer.println("Momento inicial: " + inicializacao + "<br/>");
writer.println("Momento atual: " + agora + "<br/>");
writer.println("Minutos: " + minutos + "<br/>");
writer.println("</html>");
```

## 19.4 - Exercícios opcionais

1) Crie a classe `AcessaAplicacao` no pacote `br.com.caelum.servlet`.

- Estenda `HttpServlet`. Utilize CTRL+SHIFT+O.
- Escreva o método `service`:

```
@Override
protected void service(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    ServletContext aplicacao = getServletContext();

    Date inicializacao = (Date) aplicacao.getAttribute("inicializacao");
    Date agora = new Date();

    long diferenca = agora.getTime() - inicializacao.getTime();
    double minutos = diferenca / (60 * 1000.0);

    PrintWriter writer = response.getWriter();
    writer.println("<html>");
    writer.println("Momento inicial: " + inicializacao + "<br/>");
    writer.println("Momento atual: " + agora + "<br/>");
    writer.println("Minutos: " + minutos + "<br/>");
    writer.println("</html>");}
```

2) Mapeie a sua servlet para a URL /testa-aplicacao:

```
<servlet>
    <servlet-name>acessaAplicacao</servlet-name>
    <servlet-class>br.com.caelum.servlet.AcessaAplicacao</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>acessaAplicacao</servlet-name>
    <url-pattern>/testa-aplicacao</url-pattern>
</servlet-mapping>
```

3) Teste a URL <http://localhost:8080/jsp teste/testa-aplicacao>.



## 19.5 - Acessando a aplicação no JSP

E como fazemos para acessar o escopo de aplicação no nosso JSP?

Simples, uma das variáveis que já existe em um JSP se chama `application`, algo como:

```
ServletContext application = getServletContext();
```

Portanto podemos utilizá-la através de scriptlet:

```
<%= application.getAttribute("inicializacao") %><br/>
```

Como já vimos anteriormente, o código do tipo scriptlet pode ser maléfico para nossa aplicação, sendo assim vamos utilizar Expression Language para acessar um atributo do escopo aplicação:

Acessando com EL: \${inicializacao}<br/>

Repare que a Expression Language procurará tal atributo não só no escopo do `application`, como veremos mais a frente. Para deixar claro que você procura uma variável do escopo de aplicação, usamos a variável implícita chamada `applicationScope`:

Acessando escopo `application`: \${applicationScope['inicializacao']}<br/>

## 19.6 - Exercícios

1) Crie um arquivo chamado `testa-aplicacao.jsp`.

```
<html>
    Acessando com EL: ${inicializacao}<br/>
```

```
Acessando escopo application: ${applicationScope['inicializacao']}
```

```
Acessando application com scriptlet: <%= application.getAttribute("inicializacao") %>
```

```
</html>
```

2) Teste a URL <http://localhost:8080/jsp teste/testa-aplicacao.jsp>



## 19.7 - Configuração de uma servlet

Já vimos o arquivo `web.xml` sendo utilizado para configurar uma ponte entre as URL's acessadas através de um navegador e as servlets a serem utilizadas.

Agora enfrentaremos um problema comum que utilizará esse arquivo como parte de sua solução. Imagine que precisamos de uma conexão a um banco de dados dentro de nossa servlet. Podemos escrever a URL do banco, o usuário e senha dentro da nossa servlet, mas isso é uma péssima solução, uma vez que, imagine a situação onde se faz necessário alterar o usuário e a senha. Precisamos alterar a nossa servlet e recompilá-la!

O que falta aqui é um arquivo de configuração para a nossa servlet e o arquivo `web.xml` pode ser utilizado para isso!

O código a seguir define um parâmetro no arquivo `web.xml` para representar a URL de conexão do banco de dados.

```
<!-- Definicao de uma servlet -->
<servlet>

    <servlet-name>minhaServlet</servlet-name>
    <servlet-class>br.com.caelum.servlet.TestaBancoDeDados</servlet-class>

    <!-- Parametro de inicializacao -->
    <init-param>
        <param-name>bancodedados.url</param-name>
        <param-value>jdbc:mysql://localhost/banco</param-value>
    </init-param>

</servlet>
```

O parâmetro deve ser lido através do método `getInitParameter` da classe `ServletConfig`. Para obter acesso ao `servlet config` basta chamar o método `getServletConfig`. Por exemplo:

```
String url = getServletConfig().getInitParameter("bancodedados.url");
```

### Arquivos de configuração

O mais comum é marcar no `web.xml` um arquivo de configuração para cada parte da aplicação. Por exemplo, um parâmetro indica qual arquivo possui a configuração de logs, outro indica a configuração do banco de dados. Assim fica ainda mais claro qual configuração fica aonde e ainda mais simples para o responsável pelo processo de deploy de configurar a aplicação pois ele não precisa alterar o `web.xml`, mas sim os arquivos de configuração.

## 19.8 - Exercícios

- 1) Crie uma servlet que utilize o parâmetro de inicialização para conectar-se ao banco de dados:
  - a) Crie a servlet `TestaBancoDeDados`;
  - b) Mapeie a URL `/testaparametros` para a servlet `TestaBancoDeDados`;
  - c) Implemente o método `service`, que lê o argumento `bancodedados.url`, `bancodedados.usuario`, `bancodedados.senha` e imprima para o cliente esses dados;
  - d) Teste sua servlet através da URL mapeada.

## 19.9 - Descobrindo todos os parâmetros do request

Para ler todos os parâmetros do `request` basta acessar o método `getParameterMap` do `request`.

```
Map<String, Object> parametros = request.getParameterMap();
for(String parametro:parametros.keySet()) {
    // faça algo com o parametro
}
```

# Apêndice B - Design Patterns

*“Nunca chegamos aos pensamentos. São eles que vêm.”*  
— Martin Heidegger

Neste capítulo, você aprenderá como utilizar o Factory para criar um cache de objetos.

## 20.1 - Factory – exemplo de cache de objetos

Tomemos como exemplo a classe a seguir que seria responsável por pegar o conteúdo de uma PaginaWeb da internet:

```
public class PaginaWeb {

    private String url;

    public PaginaWeb(String url) {
        this.url = url;
    }
}
```

Poderíamos colocar um aviso na nossa aplicação, avisando todos os programadores de que eles devem verificar o cache antes de dar um `new`. Não funciona. Com um construtor acessível, qualquer um poderia criar um novo objeto sem verificar o cache antes, algo que não podemos permitir.

E se deixarmos o construtor privado? Continuando com o exemplo e faremos o cache desse tipo de objeto:

```
public class PaginaWeb {

    private String url;

    /**
     * Construtor privado
     */
    private PaginaWeb(String url) {
        this.url = url;
    }
}
```

Não parece fazer muito sentido deixar o construtor de uma classe privado. Quem pode acessar algo privado de uma classe? Somente ela mesma. Então, o seguinte código não funcionaria fora dela:

```
class Pesquisa {
    PaginaWeb pesquise(String url) {
```

```

        return new PaginaWeb(url);
    }
}

```

Podemos dar acesso a construção dos objetos do tipo PaginaWeb a partir de um método dentro da classe. Esse método ditaria as regras da fabricação de novos objetos, ou em vez de fabricar um novo, devolver um já existente.

Imagine agora manter um objeto do tipo `HashMap` chamado `paginasJaCriadas` que seria o nosso cache. O código a seguir mostra como usá-lo para evitar a criação de objetos cacheados:

```

PaginaWeb pagina;
if (this.paginasJaCriadas.get(url) != null) {
    pagina = this.paginasJaCriadas.get(url);
} else {
    pagina = new PaginaWeb(url);
    this.paginasJaCriadas.put(url, pagina);
}
return pagina;

```

#### Cache de quê?

Não é aconselhado sair por aí fazendo cache de qualquer tipo de objetos. Objetos pesados são indicados para serem cacheados ou então para serem tirados de um pool.

Objetos pesados são aqueles que nitidamente consomem muitos recursos do sistema operacional ou da plataforma, como `Threads`, conexões com banco de dados e outros.

Isto é, se a `PaginaWeb` para esta URL já foi criada algum dia, não há a necessidade de criar um novo objeto: podemos pegá-lo de um cache. Se ele não existe, vamos criar, guardar e retornar, para que possa ser reutilizado.

Em que método colocamos essa implementação? Se for em um método para o objeto, só poderemos acessar esse método já tendo um objeto antes, o que não faz sentido. Precisamos de um método estático para poder ser acessado pela classe. Além disso, não podemos usar um atributo de objeto dentro de um método estático, então o nosso mapa não pode ser acessado pelo `this`, pois trataria-se de um atributo de objeto. Nossa mapa deve ser um atributo estático.

```

[/java] public class PaginaWeb {

    private static Map<String,PaginaWeb> paginasJaCriadas = new HashMap<String,PaginaWeb>(); private
    String url;

    private PaginaWeb(String url) { this.url = url; }

    public static PaginaWeb getPaginaWeb(String url) {

        // se não existir no cache if (PaginaWeb.paginasJaCriadas.get(url) == null) { // coloca no cache PaginaWeb.paginasJaCriadas.put(url, new PaginaWeb(url)); }

        // retorna do cache return PaginaWeb.paginasJaCriadas.get(url); } } [/java]

```

Acabamos de encapsular como nossos objetos são fabricados. Quando alguém quiser criar uma `PaginaWeb`:

```
class Pesquisa {  
    PaginaWeb pesquise(String url) {  
        return PaginaWeb.getPaginaWeb(url);  
    }  
}
```

O interessante é perceber aqui que, para o cliente, é indiferente o que esse método fez, se ele pegou do cache ou não. Aliás, você pode mudar as regras do seu cache da noite pro dia, sem interferir no bom funcionamento da aplicação.

Esse *Design Pattern* tem o nome de *Factory* (Fábrica) e existem diversas variações do mesmo.

#### Memória cheia?

Fazer o cache de todos os objetos pode consumir muita memória. Você pode estar usando um tipo de mapa que guarda apenas os últimos 50 objetos instanciados ou então os mais acessados, por exemplo.

## 20.2 - Singleton

Em alguns casos, desejamos que determinada classe só possa ser instanciada uma vez. Isto é, só queremos um objeto daquela classe e que a referência para ao mesmo seja compartilhada por diversos outros objetos.

Isto é muito comum quando a classe representa uma entidade única no sistema, como o presidente de uma empresa, o gerenciador de logs da aplicação ou o controlador de usuários.

Como faremos? Com certeza aproveitaremos o truque anterior, deixando o construtor privado e criando um método estático para retornar a referência. Mas o que fazer para retornar sempre uma referência para a mesma instância?

```
public Presidente getPresidente() {  
    return referenciaUnica;  
}
```

Quem é essa `referenciaUnica`? Um atributo estático, como era o nosso mapa de páginas no cache.

```
public class Presidente {  
    private static Presidente referenciaUnica = new Presidente("Nome do Presidente");  
    private String nome;  
  
    private Presidente(String nome) {  
        this.nome = nome;  
    }  
  
    public static Presidente getPresidente() {  
        return Presidente.referenciaUnica;  
    }  
}
```

E, quando alguém quiser uma referência para o presidente:

```
Presidente presidente = Presidente.getPresidente();
```

E poderia até ser que o método `getPresidente` não retornasse sempre uma referência para mesma instância, depende do que você quer que sua aplicação faça. Se um dia quisermos deixar que mais de um `Presidente` exista na nossa aplicação, podemos deixar o construtor público ou, melhor ainda, criar um método de Factory que receba o nome do `Presidente` como argumento.

#### Classes com tudo estático

Uma outra alternativa seria criar uma classe cheia de métodos e atributos estáticos, tais como a `System` e a `Math`. Apesar de funcionar, essa solução não é orientada a objetos. Se você precisar passar um `System` como argumento, como poderia fazer isso?

Uma classe só com métodos estáticos tem a característica de uma pequena biblioteca de funções.

### 20.3 - Exercícios

- 1) Dado o sistema a seguir, aplique o *pattern Singleton*. A classe `Logger` usa a variável membro ativo para realmente imprimir informações, enquanto que a classe `Aplicacao` é o nosso programa e utiliza dois objetos de tipo `Logger`.

```
public class Logger {  
  
    /* por default não imprime o log */  
    private boolean ativo = false;  
  
    public Logger() {}  
  
    public boolean isAtivo() {  
        return this.ativo;  
    }  
  
    public void setAtivo(boolean b) {  
        this.ativo = b;  
    }  
  
    public void log(String s) {  
        if(this.ativo) {  
            System.out.println("LOG :: " + s);  
        }  
    }  
}
```

Agora a classe `Aplicacao`, em outro arquivo:

```
public class Aplicacao {  
  
    public static void main(String[] args) {  
  
        Logger log1 = new Logger();  
        log1.setAtivo(true);  
    }  
}
```

```
    log1.log("PRIMEIRA MENSAGEM DE LOG");

    Logger log2 = new Logger();
    log2.log("SEGUNDA MENSAGEM DE LOG");

}
```

Resultado da aplicação antes de aplicar o *pattern* :

PRIMEIRA MENSAGEM DE LOG

Ao aplicar o pattern, a classe aplicação deverá utilizar o mesmo objeto do tipo `Logger` nas duas chamadas ao método `log`, portanto o resultado da aplicação será:

PRIMEIRA MENSAGEM DE LOG

SEGUNDA MENSAGEM DE LOG

- **Passo 1:** Torne o construtor de `Logger` privado;
- **Passo 2:** Crie uma variável estática `logger` para conter uma referência única ao objeto de `Logger`; instancie a variável na declaração;
- **Passo 3:** Crie um método estático `getLogger` que devolve a referência para o `logger`;
- **Passo 4:** Na classe `Aplicacao`, substitua o “`new Logger()`” pelo uso do método estático `getLogger` criado no passo 3.

## 20.4 - Um pouco mais...

- 1) Pesquise sobre outros *Patterns*. *Strategy*, *Facade*, *Proxy* e *Decorator* são alguns interessantes;
- 2) Como um Design Pattern deve ser aplicado? Em que momento? Antes, durante ou depois da implementação?

# Respostas dos Exercícios

## 2.7 - Exercícios

```
3. package br.com.caelum.jdbc;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConnectionFactory {

    public static Connection getConnection() throws SQLException {
        try {
            Class.forName("com.mysql.jdbc.Driver");
            System.out.println("Conectando ao banco");
            return DriverManager.getConnection("jdbc:mysql://localhost/teste", "root", "");
        } catch (ClassNotFoundException e) {
            throw new SQLException(e.getMessage());
        }
    }
}
```

```
4. package br.com.caelum.jdbc.teste;

import java.sql.Connection;
import java.sql.SQLException;

import br.com.caelum.jdbc.ConnectionFactory;

public class TestaConexao {

    public static void main(String[] args) throws SQLException {
        Connection connection = ConnectionFactory.getConnection();
        System.out.println("Conectado ao banco!");
        connection.close();
    }
}
```

5. Ao rodar, ocorre:

```
Exception in thread "main" java.sql.SQLException: com.mysql.jdbc.Driver
at br.com.caelum.jdbc.ConnectionFactory.getConnection(ConnectionFactory.java:15)
at br.com.caelum.jdbc.teste.TestaConexao.main(TestaConexao.java:11)
```

6. No seu console:

Conectado ao banco!

## 2.10 - Exercícios

```
1. package br.com.caelum.jdbc.modelo;

public class Contato {

    private Long id;
    private String nome;
    private String email;
    private String endereco;

    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }

    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }

    public String getEndereco() {
        return endereco;
    }
    public void setEndereco(String endereco) {
        this.endereco = endereco;
    }
}
```

## 2.13 - Exercícios

```
2. package br.com.caelum.jdbc.teste;

import java.sql.SQLException;

import br.com.caelum.jdbc.dao.ContatoDAO;
import br.com.caelum.jdbc.modelo.Contato;
```

```
public class TestaInsere {  
  
    public static void main(String[] args) throws SQLException {  
        Contato contato = new Contato();  
        contato.setNome("Caelum");  
        contato.setEmail("contato@caelum.com.br");  
        contato.setEndereco("R. Vergueiro 3185 cj57");  
  
        ContatoDAO dao = new ContatoDAO();  
  
        dao.adiciona(contato);  
  
        System.out.println("Gravado!");  
    }  
}
```

3. Saída no seu console.

```
Conectando ao banco  
Gravado!
```

4. Seu console do MySQL deve mostrar algo assim:

```
+-----+-----+-----+-----+  
| id | nome | email | endereco |  
+-----+-----+-----+-----+  
| 1 | Caelum | contato@caelum.com.br | R. Vergueiro 3185 cj57 |  
+-----+-----+-----+-----+  
1 row in set (0.00 sec)
```

## 2.14 - Exercícios

1. Saída no console do eclipse:

```
Nome: Contato Caelum  
E-mail: contato@caelum.com.br  
Endereço: R. Vergueiro 3185 cj51  
Conectando ao banco  
Gravado!
```

## 2.16 - Exercícios

3. package br.com.caelum.jdbc.teste;  
  
import java.sql.SQLException;  
import java.util.List;  
  
import br.com.caelum.jdbc.dao.ContatoDAO;

```

import br.com.caelum.jdbc.modelo.Contato;

public class TestaListaDAO {

    public static void main(String[] args) throws SQLException {
        ContatoDAO dao = new ContatoDAO();

        List<Contato> lista = dao.getLista();
        for (Contato contato : lista) {
            System.out.println("Nome: " + contato.getNome());
            System.out.println("E-mail: " + contato.getEmail());
            System.out.println("Endereço: " + contato.getEndereco() + " \n");
        }
    }
}

```

#### 4. Saída no console:

```

Conectando ao banco
Nome: Caelum
E-mail: contato@caelum.com.br
Endereço: R. Vergueiro 3185 cj57

```

### 2.18 - Exercícios

1. PreparedStatement stmt = this.connection.prepareStatement("select \* from contatos where nome like 'C%'"); ResultSet rs = stmt.executeQuery(); ...
  
2.

```

1 public Contato pesquisa(int id) throws SQLException {
2
3     PreparedStatement stmt = this.connection.prepareStatement(
4             "select * from contatos where id = ?");
5     stmt.setInt(1, id);
6     ResultSet rs = stmt.executeQuery();
7
8     //se o contato não existir retorna null
9     Contato contato = null;
10
11    if (rs.next()) {
12        // criando o objeto Contato
13        contato = new Contato();
14        contato.setNome(rs.getString("nome"));
15        contato.setEmail(rs.getString("email"));
16        contato.setEndereco(rs.getString("endereco"));
17    }
18    rs.close();
19    stmt.close();
20
21    return contato;
22 }
```

## 2.20 - Exercícios

1.

2.

3.

4. package br.com.caelum.jdbc.modelo;

```
public class Funcionario {  
    private Long id;  
    private String nome;  
    private String usuario;  
    private String senha;  
  
    public Long getId() {  
        return id;  
    }  
  
    public void setId(Long id) {  
        this.id = id;  
    }  
  
    public String getNome() {  
        return nome;  
    }  
  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
  
    public String getUsuario() {  
        return usuario;  
    }  
  
    public void setUsuario(String email) {  
        this.usuario = email;  
    }  
  
    public String getSenha() {  
        return senha;  
    }  
  
    public void setSenha(String endereco) {  
        this.senha = endereco;  
    }  
}
```

5. \$ mysql -u root  
mysql> use teste

```
mysql> create table funcionarios
-> (id bigint,
-> nome varchar,
-> usuario varchar,
-> senha varchar);

6. package br.com.caelum.jdbc.dao;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

import br.com.caelum.jdbc.ConnectionFactory;
import br.com.caelum.modelo.Funcionario;

public class FuncionarioDAO {
    // a conexão com o banco de dados
    private Connection connection;

    public FuncionarioDAO() throws SQLException {
        this.connection = ConnectionFactory.getConnection();
    }

    public void adiciona(Funcionario funcionario) throws SQLException {

        // prepared statement para inserção
        PreparedStatement stmt = this.connection
            .prepareStatement("insert into "
                + "funcionarios (nome,usuario,senha) values (?, ?, ?)");

        // seta os valores
        stmt.setString(1, funcionario.getNome());
        stmt.setString(2, funcionario.getUsuario());
        stmt.setString(3, funcionario.getSenha());

        // executa
        stmt.execute();
        stmt.close();
    }

    public List<Funcionario> getLista() throws SQLException {

        PreparedStatement stmt = this.connection
            .prepareStatement("select * from funcionarios");
        ResultSet rs = stmt.executeQuery();

        List<Funcionario> funcionarios = new ArrayList<Funcionario>();

        while (rs.next()) {
            // criando o objeto Funcionario
            Funcionario funcionario = new Funcionario();
            funcionario.setId(rs.getLong("id"));

            // setando os valores
            funcionario.setNome(rs.getString("nome"));
            funcionario.setUsuario(rs.getString("usuario"));
            funcionario.setSenha(rs.getString("senha"));

            // adicionando ao vetor
            funcionarios.add(funcionario);
        }
    }
}
```

```

funcionario.setNome(rs.getString("nome"));
funcionario.setUsuario(rs.getString("usuario"));
funcionario.setSenha(rs.getString("senha"));

// adicionando o objeto à lista
funcionarios.add(funcionario);
}

rs.close();
stmt.close();

return funcionarios;
}

public Funcionario pesquisar(int id) throws SQLException {

PreparedStatement stmt = this.connection
.prepareStatement("select * from funcionarios where id = ?");
stmt.setInt(1, id);
ResultSet rs = stmt.executeQuery();

// se o funcionario não existir retorna null
Funcionario funcionario = null;

if (rs.next()) {
    // criando o objeto Funcionario
    funcionario = new Funcionario();
    funcionario.setNome(rs.getString("nome"));
    funcionario.setUsuario(rs.getString("usuario"));
    funcionario.setSenha(rs.getString("senha"));
}
rs.close();
stmt.close();

return funcionario;
}

public void remover(Funcionario funcionario) throws SQLException {

PreparedStatement stmt = this.connection.prepareStatement(
    "delete from funcionarios where id = ?");
stmt.setLong(1, funcionario.getId());
stmt.execute();

stmt.close();
}

public void atualizar(Funcionario funcionario) throws SQLException {

PreparedStatement stmt = this.connection.prepareStatement(
    "update funcionarios set nome = ?, senha = ?, usuario = ? where id = ?");
stmt.setString(1, funcionario.getNome());
stmt.setString(2, funcionario.getSenha());
stmt.setString(3, funcionario.getUsuario());
stmt.setLong(4, funcionario.getId());
}

```

```
        stmt.execute();

        stmt.close();
    }
}
```

7.

### 8.11 - Exercícios

1. <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>  
<c:import url="cabecalho.jsp"/>  
Uma mensagem de texto qualquer  
<c:import url="rodape.jsp"/>

### 10.11 - Exercícios

1. @Override  

```
protected void service(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
    // recebe o writer
    PrintWriter out = response.getWriter();
    out.println("<html>");
    Enumeration parameterNames = request.getParameterNames();
    while (parameterNames.hasMoreElements()) {
        String parameter = (String) parameterNames.nextElement();
        out.println(parameter + " = " + request.getParameter(parameter) + "<br/>");
    }
    out.println("</html>");
}
```

### 10.14 - Exercícios

7. try {  
 SimpleDateFormat format = new SimpleDateFormat("dd/MM/yyyy");
 Date data = format.parse(request.getParameter("data"));
} catch (ParseException e) {
 throw new ServletException(e);
}

### 10.32 - Exercícios

- 1) package br.com.caelum.servlet;  
  
import java.io.IOException;

```

import java.io.PrintWriter;
import java.sql.SQLException;
import java.util.List;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import br.com.caelum.jdbc.dao.ContatoDAO;
import br.com.caelum.jdbc.modelo.Contato;

public class MostraContatosServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    protected void service(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        PrintWriter writer = response.getWriter();
        writer.println("<html>");
        writer.println("<ul>");
        try{
            ContatoDAO dao = new ContatoDAO();
            List<Contato> lista = dao.getLista();
            for (Contato contato : lista) {
                writer.println("<li>" + contato.getNome() + ", " +
                    contato.getEmail() + ", " + contato.getEndereco()
                    + ". <a href='remove-contato?id='"
                    + contato.getId() + "'>Remove</a>" +
                    "</li>");
            }
        } catch (SQLException e) {
            throw new ServletException(e);
        }
        writer.println("</ul>");

        writer.println("Contato Adicionado");
        writer.println("</html>");

    }

}

2) <servlet>
    <servlet-name>MostraContatosServlet</servlet-name>
    <servlet-class>br.com.caelum.servlet.MostraContatosServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>MostraContatosServlet</servlet-name>
    <url-pattern>/lista-contatos</url-pattern>
</servlet-mapping>
3) package br.com.caelum.servlet;

import java.io.IOException;
import java.io.PrintWriter;

```

```

import java.sql.SQLException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import br.com.caelum.jdbc.dao.ContatoDAO;
import br.com.caelum.jdbc.modelo.Contato;

public class RemoveContatoServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    protected void service(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        Contato contato = new Contato();
        String id = request.getParameter("id");

        contato.setId(Long.parseLong(id));
        try{
            ContatoDAO dao = new ContatoDAO();
            dao.remove(contato);
        } catch (SQLException e) {
            throw new ServletException(e);
        }

        PrintWriter writer = response.getWriter();
        writer.println("<html>");
        writer.println("Contato " + id + " removido");
        writer.println("</html>");
    }
}

4) <servlet>
    <servlet-name>RemoveContatoServlet</servlet-name>
    <servlet-class>br.com.caelum.servlet.RemoveContatoServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>RemoveContatoServlet</servlet-name>
    <url-pattern>/remove-contato</url-pattern>
</servlet-mapping>

```

## 11.7 - Exercícios

1. public void doFilter(ServletRequest request, ServletResponse response,
 FilterChain chain) throws IOException, ServletException {
 long antes = System.currentTimeMillis();
 chain.doFilter(request, response);
 long depois = System.currentTimeMillis() - antes;
 System.out.println("Tempo gasto: " + depois + "ms");
}

### 13.8 - Exercícios

```

1. package br.com.caelum.mvc;

import javax.servlet.RequestDispatcher;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import br.com.caelum.jdbc.dao.ContatoDAO;
import br.com.caelum.jdbc.modelo.Contato;

public class RemoveContatoLogic implements BusinessLogic {
    public void execute(HttpServletRequest request, HttpServletResponse response)
        throws Exception {
        System.out.println("Executando a logica e redirecionando...");

        Contato contato = new Contato();
        long id = Long.parseLong(request.getParameter("id"));
        contato.setId(id);
        ContatoDAO dao = new ContatoDAO();
        dao.remove(contato);

        RequestDispatcher rd = request
            .getRequestDispatcher("/lista-elegante.jsp");
        rd.forward(request, response);
        System.out.println("removendo contato ..." + contato.getId());
    }
}
-----  

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
    <!-- cria a lista -->
    <jsp:useBean id="dao" class="br.com.caelum.jdbc.dao.ContatoDAO"/>
    <ul>
        <c:forEach var="contato" items="${dao.lista}" varStatus="id">
            <li>Nome: ${contato.nome}, Email: ${contato.email}, Endereço: ${contato.endereco}
                <a href="mvc?business=RemoveContatoLogic&id=${contato.id }">Remover</a>
            </li>
        </c:forEach>
    </ul>
</html>

```

### 2. lista-elegante.jsp

```

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
    <!-- cria a lista -->
    <jsp:useBean id="dao" class="br.com.caelum.jdbc.dao.ContatoDAO"/>
    <ul>
        <c:forEach var="contato" items="${dao.lista}" varStatus="id">

```

```

<li>Nome: ${contato.nome}, Email: ${contato.email}, Endereço: ${contato.endereco},
    <a href="testa-altera-mvc.jsp?id=${contato.id}&nome=${contato.nome}
        &email=${contato.email}&endereco=${contato.endereco}">Editar</a>
    <a href="mvc?business=RemoveContatoLogic&id=${contato.id}">Remover</a>
</li>
</c:forEach>
</ul>
</html>

testa-altera-mvc.jsp

<html>
<body>
    Digite os dados novos e pressione o botão:<br/>

    <form action="mvc" method="POST">
        Id:      ${param.id}<input type="hidden" name="id" value="${param.id}" /><br/>
        Nome:    <input type="text" name="nome" value="${param.nome}" /><br/>
        E-mail:   <input type="text" name="email" value="${param.email}" /><br/>
        Endereço: <input type="text" name="endereco" value="${param.endereco}" /><br/>
                    <input type="hidden" name="business" value="AlteraContatoLogic"/>
                    <input type="submit" value="Enviar" />
    </form>
</body>
</html>

```

3. Cópia do AlteraContatoLogic e do testa-altera-mvc.jsp, sem os campos de Id.

#### 14.33 - Exercícios

2. RemoveContatoForm.java

```

public class RemoveContatoForm extends ActionForm {
    private Contato contato = new Contato();

    public Contato getContato() {
        return contato;
    }
    @Override
    public void reset(ActionMapping mapping, HttpServletRequest request) {
        this.contato = new Contato();
    }
}

```

RemoveContatoAction.java

```

public class RemoveContatoAction extends Action {
    @Override
    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception {

```

```

// log
System.out.println("Tentando remover um novo contato...");

// formulário de cliente
RemoveContatoForm formulario = ((RemoveContatoForm) form);
// acessa o bean
Contato contato = formulario.getContato();

// remove ao banco de dados
ContatoDAO dao = new ContatoDAO();
dao.remove(contato);

// ok.... visualização
return mapping.findForward("ok");
}
}
}

```

struts-config.xml

```

<struts-config>
...
<form-beans>
...
<form-bean name="RemoveContatoForm" type="br.com.caelum.struts.form.RemoveContatoForm" />
</form-beans>
<action-mappings>
...
<action path="/removeContato" name="RemoveContatoForm"
       type="br.com.caelum.struts.action.RemoveContatoAction">
    <forward name="ok" path="/listaContatos.do" />
</action>
</action-mappings>
...
</struts-config>

```

#### 14.35 - Exercícios

2. <html>
 <head>
 <title>Sistema de Teste do Struts</title>
 </head>
 <body>
 Id: \${contato.id} <br/>

 Nome: \${contato.nome} <br/>

 Email: \${contato.email} <br/>

 Endereço: \${contato.endereco} <br/>
 </body>

```
</html>
```

### 15.17 - Exercícios

```
1. public class LogoutAction extends Action {  
    @Override  
    public ActionForward execute(ActionMapping mapping, ActionForm form,  
        HttpServletRequest request, HttpServletResponse response)  
        throws Exception {  
        System.out.println("Algum usuário está tentando se deslogar...");  
  
        HttpSession session = request.getSession();  
        session.removeAttribute("funcionario");  
  
        // ok.... para onde ir agora?  
        return mapping.findForward("ok");  
    }  
}  
  
<action path="/efetuaLogout" type="br.com.caelum.struts.action.LogoutAction">  
    <forward name="ok" path="/formulario-login.jsp" />  
</action>  
  
2. if (request.getSession().getAttribute("funcionario") == null)  
    return mapping.findForward("login");  
  
<global-forwards>  
    <forward name="login" path="/formulario-login.jsp"></forward>  
</global-forwards>
```

---

# Índice Remissivo

ActionForm, 132  
always link to actions, 147  
AnnotationConfiguration, 169  
  
Banco de dados, 4  
bean:message, 137  
  
c:forEach, 56  
c:import, 59  
c:out, 55  
c:set, 58  
c:url, 62  
CGI, 70  
Conteúdo dinâmico, 31  
Content directory, 42  
Context, 41  
Controller, 114  
Convention over configuration, 181  
Cookie, 155  
Criteria, 176  
  
DAO, 21  
Design Patterns, 8  
destroy, 76  
doGet, 80  
doPost, 80  
DriverManager, 5  
  
Eclipse, 3  
Error page, 91  
Escopo de aplicação, 194  
Expression Language, 50, 56  
  
filters, 99  
FirstResults, 177  
  
hibernate.cfg.xml, 169  
hibernate.properties, 168  
HQL, 176  
  
Http, 70  
HttpServletRequest, 71  
  
init, 75  
Injeção dee Dependências, 181  
Inversão de controle, 181  
  
Java EE, 29  
Javabeans, 16  
JSP, 45  
JSTL, 54  
  
Listener, 195  
load, 174  
  
MaxResults, 177  
Message Resources, 136  
Model, 115  
MVC, 115  
MYSQL, 4  
  
Persistência, 4  
Prepared Statement, 19  
  
Regras de negócio, 102  
Request, 70  
Request Dispatcher, 103  
Response, 70  
ResultSet, 24  
  
Scriptlet, 45  
service, 71  
Servlet, 71  
Servlet Context, 195  
Servlet Context Listener, 194  
SingleThreadModel, 85  
struts, 117  
struts MVC, 117

---

Validação, 140

View, 114

war, 86

WEB-INF, 42

WEB-INF/classes, 43

WEB-INF/lib, 42

web.xml, 42