

Sendo um JSP

Ele não conhece uma diretiva do scriptlet, mas ELE conseguiu a sala do canto, a poltrona reclinável e uma massagem duas vezes por semana? Eu já tive isso.

Relaxe... quando ele for reprovado no exame, NÓS sabemos o que acontecerá. Só espero que eles não sujem a poltrona de sangue...



Um JSP torna-se um servlet. Um servlet que você não cria.

O Container olha o seu JSP, o traduz em código-fonte Java e o compila em uma classe servlet de Java completa. Porém, você tem que saber o que acontece quando o código que você escreveu em JSP se transforma em código Java. Você *pode* escrever códigos Java em JSP, mas será que você deveria? E se você não escrever o código Java, o que você escreverá? Como ele faz a tradução para o código Java? Neste capítulo, veremos seis diferentes tipos de elementos JSP – cada um com seu próprio propósito e, sim, *sintaxe única*. Você aprenderá como, por que e o que escrever no seu JSP. Talvez o mais importante, você aprenderá o que *não* escrever no seu JSP.

Objetivos



O Modelo de Tecnologia do JSP

- 6.1 Identificar, descrever ou escrever o código JSP para os seguintes elementos: (a) template text, (b) elementos de scripting (comentários, diretivas, declarações, scriptlets e expressões), (c) ações-padrão e customizadas, e (d) elementos da expression language.
- 6.2 Escrever o código JSP que usa as diretivas: (a) *page* (com os atributos *import*, *session*, *contentType* e *isELIgnored*), (b) *include*, e (c) *taglib*.
- 6.3 Escrever um Documento JSP (*documento baseado em XML*) que usa a sintaxe correta.
- 6.4 Descrever o propósito e a seqüência de eventos do ciclo de vida de uma página JSP: (1) tradução da página JSP, (2) compilação da página JSP, (3) carregar a classe, (4) criar a instância, (5) chamar o método *jspInit*, (6) chamar o método *jspService*, e (7) chamar o método *jspDestroy*.
- 6.5 Dado um objetivo de design, escrever o código JSP, usando os objetos implícitos apropriados: (a) *request*, (b) *response*, (c) *out*, (d) *session*, (e) *config*, (f) *application*, (g) *page*, (h) *pageContext*, e (i) *exception*.
- 6.6 Configurar o deployment descriptor para declarar uma ou mais bibliotecas de tags, desativar a linguagem de avaliação e desativar a linguagem de scripting.
- 6.7 Dado um objetivo de projeto específico para incluir um segmento JSP em outra página, escrever o código JSP que usa o mecanismo de inclusão mais apropriado (a diretiva *include* ou a ação-padrão *jsp:include*).

Notas sobre a Abrangência:

A maioria é tratada neste capítulo, mas os detalhes que envolvem as (c) ações-padrão e customizadas e (d) os elementos da expression language serão abordados nos próximos capítulos.

*A diretiva de página será abordada neste capítulo, mas o *include* e a *taglib* serão abordados nos próximos capítulos.*

Não será abordado aqui; verifique o capítulo sobre Distribuição.

Tudo abordado neste capítulo. (Dica: estas serão algumas das perguntas mais tranqüilas do exame, uma vez que você aprendeu o básico neste capítulo.)

Tudo abordado neste capítulo, embora se espera que você já saiba o que a maioria deles significa, baseado nos dois capítulos anteriores.

Nós falamos sobre tudo aqui, exceto declarar as bibliotecas de tags, que será visto no capítulo Usando a JSTL.

Não é abrangido aqui; verifique no próximo capítulo (JSP sem scripts).

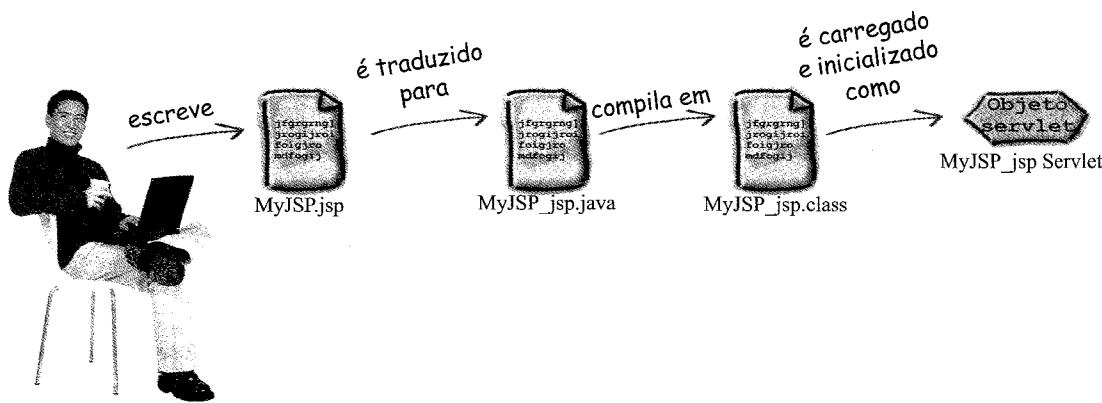
No fim das contas, o JSP é só um servlet

O seu JSP torna-se um servlet completo rodando em sua aplicação. É muito parecido com qualquer outro servlet, exceto pelo fato de que a classe do servlet é escrita *para* você – pelo Container.

O Container utiliza o que você escreveu no seu JSP, o *traduz* para um arquivo-fonte da classe servlet (.java) e o *compila* em uma classe servlet Java. Depois disso, ele será só um servlet e rodará exatamente do mesmo jeito, como se você mesmo tivesse escrito e compilado o código. Ou seja, o Container carrega a classe servlet, instancia-a e inicializa-a, cria uma thread para cada solicitação e chama o método service() do servlet.

A questão mais importante para este capítulo é a seguinte: qual a função que seu código JSP executa na classe do servlet final?

Em outras palavras, onde vão parar os elementos do JSP no código-fonte do servlet gerado?



Algumas das perguntas que responderemos neste capítulo são:

- ▶ Para onde vai cada parte do seu arquivo JSP no código-fonte do servlet?
- ▶ Você tem acesso às características “sem servlet” da sua página JSP? Por exemplo, o JSP tem noção do ServletConfig ou do ServletContext?
- ▶ Quais são os tipos de elementos que você pode inserir em um JSP?
- ▶ Qual é a sintaxe para os diferentes elementos de um JSP?
- ▶ Qual é o ciclo de vida de um JSP? Você pode interferir nele?
- ▶ Como os diferentes elementos de um JSP interagem no servlet final?

Criando um JSP que exiba quantas vezes ele foi acessado

Pauline quer usar JSPs em suas aplicações – ela está *realmente* cansada de escrever HTML no seu método println() do PrintWriter do seu servlet.

Ela decide aprender JSPs, criando uma página dinâmica simples que exibe o número de vezes que ela foi solicitada. Ela sabe que você pode colocar um código Java normal em um JSP usando um *scriptlet* – que significa simplesmente o código Java dentro de uma tag <% ... %>.



Já que eu posso colocar código Java no JSP, vou criar um método estático em uma classe Counter, para manter a variável estática de contagem de acessos e chamar este método a partir do JSP...

BasicCounter.jsp

```
<html>
<body>
The page count is:
<%
    out.println(Counter.getCount());
%>
</body>
</html>
```

O objeto "out" está implícito lá.
Tudo entre <% e %> é um scriptlet,
que é apenas Java simples.

Counter.java

```
package foo;

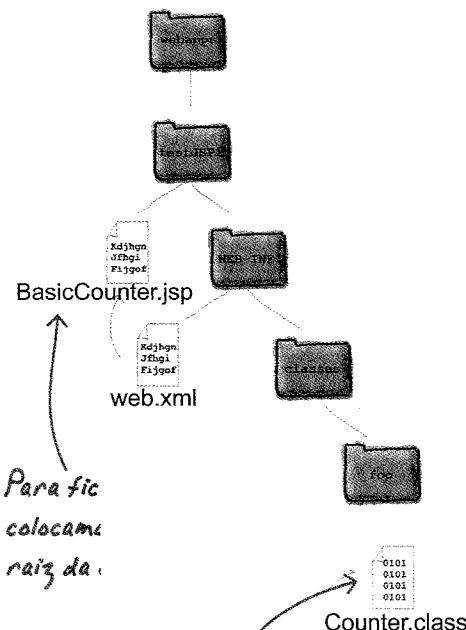
public class Counter {
    private static int count;
    public static synchronized int getCount()
    {
        count++;
        return count;
    }
}
```

Classe assistente
Java simples.

Ela distribui e testa o JSP

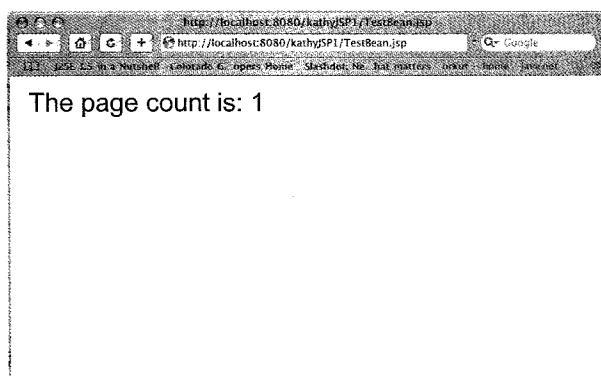
Distribuir e testar é trivial. A única parte confusa é garantir que a classe Counter esteja disponível para o JSP. E isso é fácil: basta colocá-la no diretório WEB-INF/classes da aplicação. Ela acessa o JSP direto pelo browser:

<http://localhost:8080/testJSP1/BasicCounter.jsp>



Coleque o diretório do pacote e o arquivo .class no diretório WEB-INF/classes, e qualquer parte desta aplicação será capaz de vê-lo.

O que ela esperava:



O que ela conseguiu:



O JSP não reconhece a classe Counter

A classe Counter está no pacote *foo*, mas não há nada no JSP capaz de reconhecê-la. É idêntico ao que acontece com você em qualquer outro código Java. E você conhece a regra: importe o pacote ou use o nome da classe totalmente qualificado no seu código.



Eu acho que você tem que usar o nome da classe totalmente qualificado dentro dos JSPs. Faz sentido, pois todos os JSPs foram transformados em código servlet Java simples pelo Container. Mas é claro que eu gostaria que você pudesse colocar imports no seu código JSP...

Counter.java

```
package foo;  
  
public class Counter {  
    private static int count;  
    public static int getCount() {  
        count++;  
        return count;  
    }  
}
```

O código JSP era:

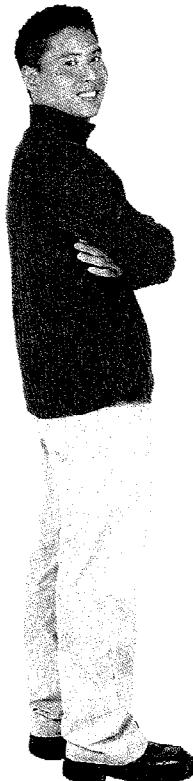
```
<% out.println(Counter.getCount()); %>
```

O código JSP deveria ser:

```
<% out.println(foo.Counter.getCount()); %>
```

Agora vai funcionar!

Mas você PODE
incluir declarações
import num JSP...
basta ter uma
diretiva.



Use a diretiva de página para importar pacotes

A **diretiva** é um recurso que você tem para dar instruções especiais ao Container no momento da tradução da página. As diretivas vêm em três sabores: **page**, **include** e **taglib**. Nós veremos as diretivas include e taglib nos próximos capítulos. Por enquanto, nossa preocupação será com a diretiva **page**, porque ela é a única que nos permite *importar*.

Para importar um único pacote:

```
<%@ page import="foo.*" %> ←
<html>
<body>
The page count is:
<%
    out.println(Counter.getCount());
%>
</body>
</html>
```

Esta é uma diretiva **page**
com um atributo **import**.

(Observe que não há
nenhum ponto-e-vírgula no
final da diretiva).

Os **scriptlets** são Java
normais; logo, todas as
declarações em um scriptlet
devem terminar com um
ponto-e-vírgula!

Para importar múltiplos pacotes:

```
<%@ page import="foo.* , java.util.*" %>
```

↑
Use uma vírgula para separar os pacotes. A
lista inteira de pacotes vem entre aspas!

Reparou a diferença entre o código Java que exibe o contador e a diretiva de página?

O código Java vem entre os símbolos **<%** e **%>**. Mas a diretiva acrescenta um caractere a mais no começo do elemento – o símbolo **@** (arroba)!

Se você vir um código JSP que começa com <%@, você já sabe que se trata de uma diretiva. (Nós entraremos em maiores detalhes sobre a diretiva de página mais adiante.)

Mas em seguida Kim menciona as “expressões”

Justo quando você pensou que fosse seguro, o Kim percebe o scriptlet com uma declaração `out.println()`. Isto é JSP, pessoal. Grande parte da razão do JSP é *evitar* o `println()`! É por isso que existe um elemento *expressão* no JSP – ele exibe automaticamente aquilo que você colocou entre as tags.

Você não precisa dizer `out.println()` em um JSP! Basta usar uma **expressão**...

O código scriptlet:

```
<%@ page import="foo.*" %>
<html>
<body>
The page count is:
<% out.println(Counter.getCount()); %>
</body>
</html>
```

O código expressão:

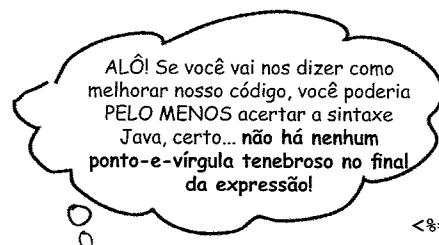
```
<%@ page import="foo.*" %>
<html>
<body>
The page count is now:
<%= Counter.getCount() %>
</body>
</html>
```

A expressão é mais curta – nós não precisamos escrever o print...

Notou a diferença entre a tag para o código scriptlet e a tag da expressão? O código *scriptlet* vem entre os sinais de porcentagem `<%` e `%>`. Já as *expressões* recebem um caractere adicional no início do elemento: um sinal de *igual* (`=`).

Até agora nós já vimos três tipos diferentes de elementos do JSP:

Scriptlet:	<code><%</code>	<code>%></code>
Diretiva:	<code><%@</code>	<code>%></code>
Expressão:	<code><%=</code>	<code>%></code>



Onde está o ponto-e-vírgula?

`<%= Counter.getCount() %>`



As expressões se tornam o argumento para um `out.print()`

Em outras palavras, o Container carrega *tudo* o que você digita entre `<%=` e `%>` e acrescenta como argumento para uma declaração que exibe para a resposta implícita *out* do PrintWriter.

Quando o Container encontra isso:

`<%= Counter.getCount() %>`

Ele o transforma nisso:

`out.print(Counter.getCount());`

Se você colocou mesmo um ponto-e-vírgula na sua expressão:

`<%= Counter.getCount(); %>`

Isso é mau. Significaria isso:

`out.print(Counter.getCount());`

X!!! Isto nunca compilará.

NUNCA termine uma expressão com um ponto-e-vírgula!

`<%= nuncaColoqueUmPontoEVírgulaAqui %>`

`<%= porquelstoÉUmArgumentoParaPrint() %>`

Não existem Perguntas Idiotas

P: Bem, se você tem que usar as expressões EM VEZ DE colocar o `out.println()` em um scriptlet, então por que existe um "out" implícito?

R: Você provavelmente não usará a variável out implícita de dentro da sua página JSP, mas você pode passá-la *adiante*... alguns outros objetos que são parte da sua aplicação não têm acesso direto ao stream de saída de dados da resposta.

P: Em uma expressão, o que acontece se o método não retornar coisa alguma?

R: Você receberá um erro!! Você não pode, e NÃO DEVE, usar um método com um tipo de retorno void como uma expressão. O Container é esperto o suficiente para descobrir que **não haverá nada a exibir se o método tem um tipo de retorno void!**

P: Por que a diretiva import começa com a palavra "page"? Por que é `<%@ page import...%>` em vez de simplesmente `<%@ import... %>`?

R: Boa pergunta! Em vez de possuir uma enorme pilha de diretivas, a especificação JSP tem apenas três diretivas, que podem ter atributos. O que você chamava de "a diretiva import" é na verdade "o atributo import da diretiva de página".

P: Quais são os outros atributos para a diretiva de página?

R: Lembre-se, a diretiva da página é para dar ao Container a informação de que ele precisa quando traduz seu JSP para servlet. Os atributos que nos importam (além do import) são: session, content-Type e isELIgnored (falaremos deles mais adiante neste capítulo).



Aponte seu lápis

Diga quais das seguintes expressões são válidas, ou não, e por quê. Nós não abordamos todos os exemplos aqui; portanto, use a sua imaginação, baseado no que você já sabe sobre o funcionamento das expressões.

(As respostas estão mais adiante neste capítulo; então, faça o exercício AGORA).

Válido? (Verifique se é válido ou não, justificando quando não for.)

`<%= 27 %>`

`<%= ((Math.random() + 5)*2); %>`

`<%= "27" %>`

`<%= Math.random() %>`

`<%= String s = "foo" %>`

`<%= new String[3]) %>`

`<% = 42*20; %>`

`<%= 5 > 3 %>`

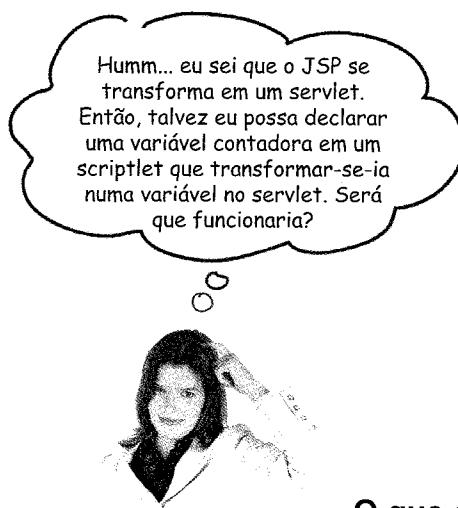
`<%= false %>`

`<%= new Counter() %>`

Kim solta a bomba final...



Você não PRECISA nem da classe Counter... você pode fazer tudo no JSP.



Humm... eu sei que o JSP se transforma em um servlet. Então, talvez eu possa declarar uma variável contadora em um scriptlet que transformar-se-ia numa variável no servlet. Será que funcionaria?

O que ela tentou:

```
<html>
<body>
<% int count=0; %>
The page count is now:
<%= ++count %>
</body>
</html>
```

Isto vai **compilar?**

Vai **funcionar?**

Declarando uma variável em um scriptlet

A declaração da variável é *válida*, mas não funcionará conforme Pauline esperava.

O que ela tentou:

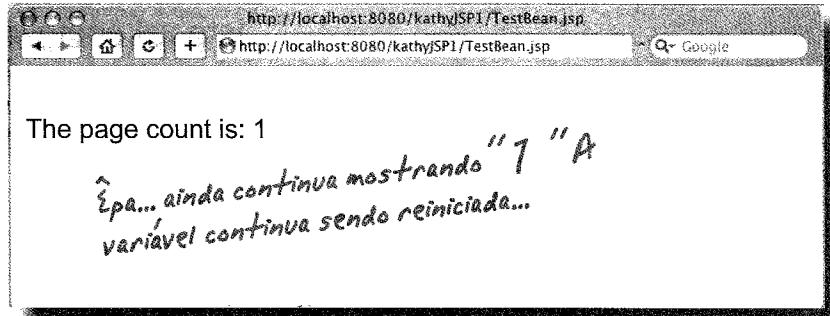
Nós não precisamos importar nada, então nós pulamos a diretiva da página.

```
<html> ←  
<body> ←  
script → <% int count=0; %> ← The page count is now:  
expressão → <%= ++count %> Declara a variável contadora.  
</body> ← Incrementa a variável contadora  
</html> e exibe o valor.
```

O que ela viu na primeira vez que abriu a página:



O que ela viu na segunda, na terceira e em todas as outras vezes em que ela abriu a página:



O que **REALMENTE** acontece com o seu código JSP?

Você *escreve* um JSP, mas ele *vira* um servlet. A única maneira de saber realmente o que está acontecendo é ver o que o Container faz com o seu código JSP. Ou seja, como o Container *traduz* o seu JSP em servlet?

Uma vez que você saiba onde os diferentes elementos do JSP se encontram no arquivo de classe do servlet, você achará muito mais fácil de saber como criar a estrutura do seu JSP.

O código do servlet nesta página *não* é o código verdadeiro gerado pelo Container – nós o simplificamos e aproveitamos o essencial. O arquivo do servlet gerado pelo Container é, digamos, *mais feio*. O verdadeiro código-fonte do servlet gerado é um pouco mais difícil de ser lido, mas nós o veremos nas próximas páginas. Contudo, por enquanto, o que nos interessa é *onde* o nosso código JSP realmente termina na classe servlet.

Este JSP:

Torna-se este servlet:

```
public class basicCounter_jsp extends
SomeSpecialHttpServlet {

    public void _jspService(HttpServletRequest request,
                           HttpServletResponse response) throws java.
                           io.IOException,
                           ServletException {

        PrintWriter out = response.getWriter();
        response.setContentType("text/html");

<html><body>-----> out.write("<html><body>");
<% int count=0; %>-----> int count=0;
The page count is now:-----> out.write("The page count is now:");
<%= ++count %>-----> out.print( ++count );
</body></html>-----> out.write("</body></html>");

    }
}
```

O Container coloca todo o código em um método service genérico. Pense nele como um pacotão doGet/doPost que pega tudo.

TODOS os códigos do scriptlet e das expressões ficam em um método service.

Isto significa que as variáveis declaradas em um scriptlet são sempre variáveis LOCAIS!

Nota: Se você quiser checar o código do servlet gerado a partir do Tomcat, veja em HomeDirDoSeuTomcat/work/Catalina/ NomeDoSeuServidor/NomeDaSuaAplicação/org/apache/jsp. (Os nomes sublinhados mudarão, dependendo do seu sistema e da sua aplicação.)



Não me diga - deve existir outro tipo de elemento JSP para a declaração de variáveis de instância, em vez de variáveis locais...

Precisamos de outro elemento JSP...

Declarar a variável contadora no scriptlet nos mostrou que a variável foi reinicializada cada vez que o método de serviço rodou. Quer dizer que *ela foi reiniciada para 0 a cada solicitação*. De alguma forma, precisamos tornar count uma variável de *instância*.

Até agora, nós vimos as diretivas, os scriptlets e as expressões. As *diretivas* são para instruções especiais para o Container, os *scriptlets* são Java simples localizados dentro de um método service do servlet gerado, e o resultado de uma *expressão* sempre se torna o argumento para um método print().

Mas existe um outro elemento JSP chamado *declaração*.

```
<%! int count=0; %>
```

Coloque um ponto de exclamação (!) depois do sinal de porcentagem (%).

Isto não é uma expressão - você PRECISA do ponto-e-vírgula aqui!

As declarações JSP servem para declarar membros da classe do servlet gerado. *Isto significa as variáveis e os métodos!* Portanto, tudo que estiver entre a tag <%! e %> é adicionado à classe *fora* do método service. Logo, você pode declarar os métodos e as variáveis estáticas e de instância.

As declarações JSP

Uma declaração JSP é sempre definida *dentro* da classe, mas *fora* do método service (ou qualquer outro). É simples assim: as declarações servem para as variáveis estáticas e de instância e para os métodos. (Na teoria, sim, você poderia definir outros membros, incluindo as classes internas, mas em 99,9999% das vezes você utilizará as declarações para os métodos e as variáveis.) O código abaixo resolve o problema de Pauline; agora, o contador é incrementado cada vez que um cliente solicitar a página.

A Declaração da Variável

Este JSP:

```
<html><body>
<%! int count=0; %>
The page count is now:
<%= ++count %>
</body></html>
```

Torna-se este servlet:

```
public class basicCounter_jsp extends HttpServlet {
    int count=0;

    public void _jspService(HttpServletRequest request,
                           HttpServletResponse response) throws java.io.IOException {
        PrintWriter out = response.getWriter();
        response.setContentType("text/html");
        out.write("<html><body>");
        out.write("The page count is now:");
        out.print( ++count );
        out.write("</body></html>");
    }
}
```

Desta vez, estamos incrementando uma variável de instância em vez de uma variável local.

A Declaração do Método

Este JSP:

```
<html>
<body>
<%! int doubleCount() {
    count = count*2;
    return count;
}
%>
<%! int count=1; %>
The page count is now:
<%= doubleCount() %>
</body>
</html>
```

Torna-se este servlet:

```
public class basicCounter_jsp extends HttpServlet {
    int doubleCount() {
        count = count*2;
        return count;
    }
    int count=1;
    public void _jspService(HttpServletRequest request,
                           HttpServletResponse response) throws java.io.IOException {
        PrintWriter out = response.getWriter();
        response.setContentType("text/html");
        out.write("<html><body>");
        out.write("The page count is now:");
        out.print( doubleCount() );
        out.write("</body></html>");
    }
}
```

O método permanece idêntico ao que você digitou no seu JSP.

Isto é Java; então, sem problemas quanto às referências de forward (declaração da variável DEPOIS que você a utilizou em um método).

Hora de ver o VERDADEIRO servlet gerado

Estávamos vendo uma versão bem simplificada do servlet que o Container realmente cria a partir do seu JSP. Não há necessidade de se ver o código gerado pelo Container durante o desenvolvimento, mas você pode usá-lo para ajudá-lo a *aprender*. Uma vez que você veja o que o Container faz com os diferentes elementos de um JSP, você não precisará mais ver os arquivos-fonte .java gerados pelo Container. Alguns fabricantes não *deixarão* que você veja a fonte Java gerada, mantendo apenas os arquivos .class compilados.

Não se intimide quando você notar partes da API que você não conheça. Os tipos de classe e interface são, na maioria, implementações específicas do fabricante que você não deve se preocupar.

O que o Container faz com o seu JSP

- Olha as *diretivas*, em busca de informações de que ele possa necessitar durante a tradução.
- Cria uma subclasse HttpServlet. Para o Tomcat 5, o servlet gerado estende:
`org.apache.jasper.runtime.HttpJspBase`
- Se houver uma *diretiva de página* com um atributo *import*, ele escreve as instruções do import no topo do arquivo classe, logo abaixo da declaração do pacote. Para o Tomcat 5, a declaração do pacote (*com a qual você não deve se preocupar*) é:
`package org.apache.jsp;`
- Se houver *declarações*, ele escreve-as no arquivo classe, geralmente logo abaixo da declaração da classe e antes do método service. O Tomcat 5 por si só declara uma variável estática e um método de instância.
- Cria o método **service**. O nome real do método é `_jspService()`. Ele é chamado pelo método ativado `service()` da superclasse do servlet e recebe o `HttpServletRequest` e o `HttpServletResponse`. Como parte da construção desse método, o Container declara e inicializa todos os *objetos implícitos*. (Você verá mais objetos implícitos quando mudar de página.)
- Mistura o HTML simples (chamado template text) com os *scriptlets* e as *expressões* no método service, formatando tudo e escrevendo no PrintWriter da resposta.



A prova fala pouco sobre a classe gerada.

Nós estamos mostrando o código gerado de forma que você possa entender como o JSP é traduzido para o código servlet. Mas você não precisa saber em detalhes como um determinado fabricante faz isso e nem a aparência verdadeira do código gerado. Tudo o que você precisa saber é como se comporta cada tipo de elemento (scriptlet, diretiva, declaração, etc.) e como aquele elemento funciona dentro do servlet gerado. Você precisa saber, por exemplo, que seu scriptlet pode usar objetos implícitos, e você precisa conhecer os tipos de objetos implícitos da API Servlet. Mas você NÃO precisa conhecer o código usado para disponibilizar tais objetos.

A única coisa que você precisa saber sobre o código gerado são os três métodos do ciclo de vida do JSP: `jspInit()`, `jspDestroy` e `_jspService()`. (Falaremos sobre eles ainda neste capítulo.)

A classe gerada pelo Tomcat 5

```

package org.apache.jsp;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

Se você tiver imports da
diretiva de página, eles
aparecerão aqui (nós não temos
nenhum import neste JSP).

```

<html><body>
<%! int count=0; %>
The page count is now:
<%= ++count %>
</body></html>

```

public final class BasicCounter_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent
{

    int count=0;
    private static java.util.Vector _jspx_dependants;

    public java.util.List getDependants() {
        return _jspx_dependants;
    }

    public void _jspService(HttpServletRequest request, HttpServletResponse
response)
        throws java.io.IOException,
ServletException {
    JspFactory _jspxFactory = null;
    PageContext pageContext = null;
    HttpSession session = null;
    ServletContext application = null;
    ServletConfig config = null;
    JspWriter out = null;
    Object page = this;
    JspWriter _jspx_out = null;
    PageContext _jspx_page_context = null;

    try {
        _jspxFactory = JspFactory.getDefaultFactory();
        response.setContentType("text/html");
        pageContext = _jspxFactory.getPageContext(this, request, response, null,
true, 8192, true);
        _jspx_page_context = pageContext;
        application = pageContext.getServletContext();
        config = pageContext.getServletConfig();
        session = pageContext.getSession();
        out = pageContext.getOut();
        _jspx_out = out;
        out.write("\r<html>\r<body>\r");
        out.write("\rThe page count is now: \r");
        out.print( ++count );
        out.write("\r</body>\r</html>\r");
    } catch (Throwable t) {
        if (!(t instanceof SkipPageException)){
            out = _jspx_out;
            if (out != null && out.getBufferSize() != 0)
                out.clearBuffer();
            if (_jspx_page_context != null) _jspx_page_context.
handlePageException(t);
        }
    } finally {
        if (_jspxFactory != null) _jspxFactory.releasePageContext(_jspx_page_
context);
    }
}

```

O Container coloca as suas declarações (aqueelas dentro das tags <%! %>) e algumas delas próprias abaixo da declaração da classe.

O Container declara um mante de variáveis locais dele próprio, incluindo aquelas que representam os objetos implícitos, que seu código pode precisar, como out e request.

Aqui ele tenta inicializar os objetos implícitos.

Ele tenta rodar e enviar seu HTML JSP, seu scriptlet e seu código de expressão.

É claro que algo pode dar errado...

A variável out não é o único objeto implícito...

Quando o Container transforma o JSP em um servlet, o começo do método service é uma pilha de declarações de **objetos implícitos** e atribuições.

Com os objetos implícitos, você pode escrever um JSP, sabendo que o seu código fará parte de um servlet. Em outras palavras, você pode aproveitar as suas características de servlet, mesmo que você não esteja escrevendo diretamente uma classe servlet.

Lembre-se dos capítulos 4, 5 e 6. Quais foram alguns dos objetos importantes que você usou? Como o seu servlet conseguiu seus parâmetros init? Como o seu servlet obteve a sessão? Como ele conseguiu os parâmetros enviados pelo cliente em um formulário?

Estas são algumas das razões que seu JSP pode precisar para usar alguns dos recursos disponíveis para o servlet. Todos os objetos implícitos apontam para algum item da API Servlet/JSP. O objeto implícito *request*, por exemplo, é uma referência ao objeto *HttpServletRequest*, passado para o método service pelo Container.

API	Objeto Implícito	Observações
JspWriter	out	Quais destes representam os escopos do atributo para a solicitação, a sessão e a aplicação? (Tudo bem, muito óbvio). Mas agora temos um quarto NOVO escopo, page-level, e os atributos do escopo da página são armazenados no pageContext.
HttpServletRequest	request	
HttpServletResponse	response	
HttpSession	session	
ServletContext	application	Este objeto implícito é disponível apenas para páginas de erro definidas. (Você verá isso mais tarde.)
ServletConfig	config	
JspException	exception	
PageContext	pageContext	Um PageContext encapsula outros objetos implícitos. Assim, se você passar uma referência PageContext a algum objeto helper, ele poderá usar essa referência para obter referências para OUTROS objetos implícitos e atributos de todos os escopos.
Object	page	

P: Qual é a diferença entre um JspWriter e um PrintWriter que eu recebo de um HttpServletResponse?

R: Não muita. O JspWriter É UM PrintWriter com algumas funções de buffer. O único momento em que você deve realmente dar maior atenção a ele é quando você estiver criando tags customizadas. Portanto, falaremos um pouco a respeito das habilidades especiais do JspWriter no capítulo que aborda o desenvolvimento das tags customizadas.



SEJA o Container

Exercícios



Cada um dos trechos vem de um JSP. Seu trabalho será descobrir o que acontecerá quando o Container tentar transformar o J S P num servlet. O Container será capaz de traduzir o seu JSP em um código servlet legítimo e compilável? Se não, por que? Em caso afirmativo, o que acontece quando um cliente acessar o JSP?

1

```
<html><body>
Test scriptlets...
<% int y=5+x; %>
<% int x=2; %>
</body></html>
```



2

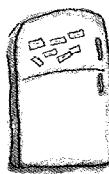
```
<%@ page import="java.util.*" %>
<html><body>
Test scriptlets...
<% ArrayList list = new ArrayList();
   list.add(new String("foo"));
%>
<%= list.get(0)  %>
</body></html>
```



3

```
<html><body>
Test scriptlets...
<%! int x = 42; %>
<% int x = 22; %>
<%= x %>
</body></html>
```





Ímã de Geladeira (Teste Preparatório)

Estude o cenário (e o restante nesta página) e coloque os ímãs no JSP para criar um arquivo legítimo que produziria o resultado correto. Você não deve usar um ímã mais de uma vez e nem usará todos eles. Este exercício supõe que existe um servlet (que você não precisa ver) que recebe a solicitação inicial, associa um atributo no escopo da solicitação e encaminha para o JSP que você está criando.

(Nota: nós chamamos este exercício de “Ímã de Geladeira (Teste Preparatório)” em vez de “Ímã de Geladeira”, pois a prova está repleta de perguntas “arrastar e soltar” como esta).

Objetivo do Projeto

Criar um JSP que produzirá isto:

The friends who share your hobby of extreme knitting are:
Fred
Pradeep
Philippe

Os três nomes vêm de uma **ArrayList** de **atributos** da solicitação, chamados “names”. Você terá que *receber* o atributo do objeto solicitação. Considere que o servlet recebeu esta solicitação e configurou um atributo no escopo da mesma.

Isto vai para o servlet que configura o atributo solicitação e depois encaminha a solicitação para o JSP que você está escrevendo.

O formulário HTML

```
<html><body>
<form method="POST"
      action="HobbyPage.do">
  Choose a hobby:<p>

  <select name="hobby" size="1">
    <option>horse skiing
    <option>extreme knitting
    <option>alpine scuba
    <option>speed dating
  </select>
  <br><br>
  <center>
    <input type="SUBMIT">
  </center>
</form>
</body></html>
```

O texto “extreme knitting” vem de um *parâmetro* do formulário de solicitação. Você precisa receber esse parâmetro do seu JSP. O servlet obterá a solicitação primeiro (e então a encaminhará ao JSP), mas isto não muda a forma como você receberá o parâmetro no seu JSP.

Conselhos e dicas importantes

- O atributo da solicitação é do tipo `java.util.ArrayList`.
- A variável implícita para o objeto `HttpServletRequest` é chamada de `request` e você pode usá-la em scriptlets ou expressões, mas *não* em diretivas ou declarações. Independentemente do que você fizer com um objeto solicitação em um servlet, você o fará dentro do seu JSP.
- O método `servlet` do JSP pode processar os parâmetros `request`, pois lembre-se, seu código vai estar dentro do método `service` do servlet. Você não precisa se preocupar com o método HTTP (GET ou POST) que foi usado na solicitação.

Nós colocamos algumas linhas para você. O código que você colocar neste JSP DEVE funcionar com o código que já está aqui. Quando você tiver terminado, ele deve ser compilável e produzir o resultado na página anterior (você deve CONSIDERAR que já existe um servlet funcionando, que primeiro recebe a solicitação, configura o atributo "names" da solicitação e encaminha a solicitação para esse JSP).

PARE!

Este não é um exercício
opcional.
É parte da lição sobre a
sintaxe do JSP!

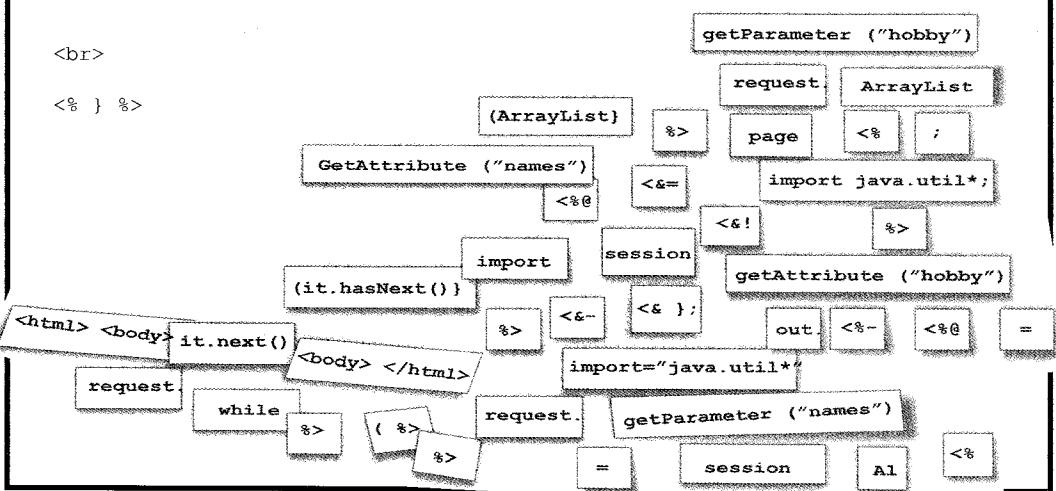
The friends who share your hobby of

are:

<% Iterator it = al.iterator();

Você não usará todos eles!

<% } %>





SEJA o Container

Respostas

O nº 2 é fácil de entender e funciona. O nº 1 é uma questão básica da linguagem Java (usar uma variável local antes que ela seja declarada) e o nº 3 também demonstra uma questão básica da linguagem Java – aquilo que acontece quando você tem uma variável local e de instância com o mesmo nome. Então veja... se você traduzir o código JSP no código Java do servlet, você não terá problemas em descobrir o resultado. Uma vez que o conteúdo do seu JSP estiver dentro do servlet, isso é Java.

Este não irá compilar! Exatamente como escrever um método com:

1

```
<html><body>
Test scriptlets...
<% int y=5+x; %>
<% int x=2; %>
</body></html>
```

```
void foo() {
    int y = 5 + x; ←
    int x = 2;
}
```

Você está tentando usar a variável x ANTES que ela seja definida. A linguagem Java não permite isso e o Container não se incomoda em rearranjar a ordem do seu código scriptlet.

2

```
<%@ page import="java.util.*" %>
<html><body>
Test scriptlets...
<% ArrayList list = new ArrayList();
   list.add(new String("foo"));
%>
<%= list.get(0) %>
</body></html>
```

Test scriptlets... foo

Sem problemas, exibe o primeiro (e único) objeto da ArrayList.

3

```
<html><body>
Test scriptlets...
<%! int x = 42; %>
<% int x = 22; %>
<%= x %> ←
</body></html>
```

O scriptlet decifra uma variável local x (que esconde a variável de instância x), portanto, se você quer exibir a variável de instância x (42), em vez da variável local x (22), mude a expressão para: <%= \${x} %>

Test scriptlets... 22



Ímã de Geladeira Respostas

Se a sua resposta parecer um pouco diferente e você ainda acreditar que ela deve funcionar – tente! Você terá que fazer o servlet que carrega o formulário de solicitação configurar um atributo e encaminhar (despachar) a solicitação para o JSP.

```
<%@ page import="java.util.*" %>
<html><body>
```

Nós precisamos da diretiva de página import, por causa da ArrayList e do Iterator.

The friends who share your hobby of

```
<%= request.getParameter("hobby") %>
```

are:


```
<% ArrayList Al = (ArrayList) request.getAttribute("names"); %>
```

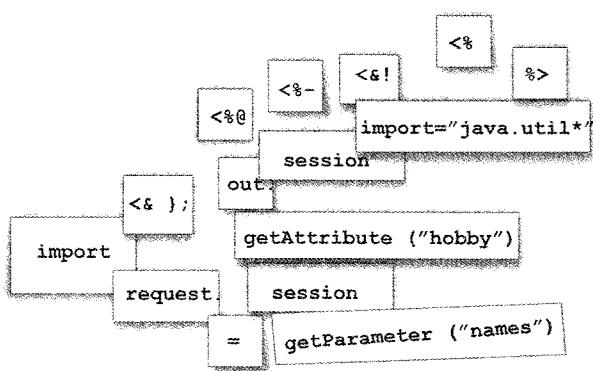
```
<% Iterator it = al.iterator(); %>
while (it.hasNext()) { %>
<%= it.next() %>
```

*Inicie o scriptlet aqui...
e o finalize aqui.
Use uma expressão.*


```
<% } %>
```

Encerre o bloco de loop while! (Se você esquecer isto, ele não compilará.)

```
</body></html>
```



Um comentário...

Sim, você pode colocar comentários no seu JSP. Se você é um programador Java com alguma experiência em HTML, você pegar-se-á digitando:

// isso é um comentário

sem pensar duas vezes. Mas se você o fizer, a não ser que esteja dentro de um scriptlet ou em uma tag de declaração, você acabará EXIBINDO isso para o cliente como parte da resposta. Portanto, para o Container aquelas duas barras são simplesmente mais texto, como "Olá" ou "E-mail é".

Você pode colocar dois tipos diferentes de comentários em um JSP:

► <!-- comentário HTML -->

O Container passa isso diretamente para o cliente e o browser o interpreta como um comentário.

► <%-- comentário JSP --%>

Estes são para os desenvolvedores de páginas e, tal qual com os comentários Java em um arquivo-fonte Java, eles são tirados da página traduzida. Se você estiver digitando um JSP e quiser adicionar comentários da maneira como você os usaria em um arquivo-fonte Java, use um comentário JSP. Se você quiser que os comentários façam parte da resposta HTML que vai para o cliente (embora o browser não os mostre ao cliente), use um comentário HTML.



Aponte seu lápis

Respostas

As Expressões Válidas e Inválidas

Válidas?

<%= 27 %>

Todas as literais primitivas valem.

<%= ((Math.random() + 5)*2); %>

NÃO! O ponto-e-vírgula não pode estar aqui.

<%= "27" %>

A String literal é válida.

<%= Math.random() %>

Sim, o método retorna uma cópia.

<%= String s = "foo" %>

NÃO! Você não pode ter uma declaração de variável aqui.

<%= new String[3] %>

Sim, porque o array new String é um objeto e QUALQUER objeto pode ser enviado para uma declaração println().

<%= 42*20; %>

NÃO! A aritmética está boa, mas há um espaço entre o % e o =. Não pode ser <%=, tem que ser <%=>.

<%= 5 > 3 %>

Claro, por ser um booleano, exibe true.

<%= false %>

Nós já dissemos que as literais primitivas são válidas.

<%= new Counter() %>

Sem problema. Isto é exatamente como a String [...]... irá exibir o resultado do método toString() do objeto.

A API para o servlet gerado

O Container gera uma classe a partir do seu JSP, que implementa a interface `HttpJspPage`. Esta é a única parte da API do servlet gerado que você precisa saber. Não se incomode se no Tomcat, por exemplo, a sua classe gerada estenda:

```
org.apache.jasper.runtime.HttpJspBase
```

Tudo o que você precisa saber são os três métodos principais:

► `jspInit()`

Este método é chamado pelo método `init()`. Você pode anular este método. (Você consegue imaginar *como?*)

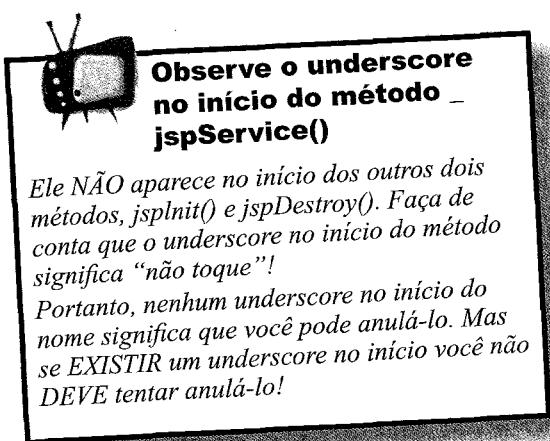
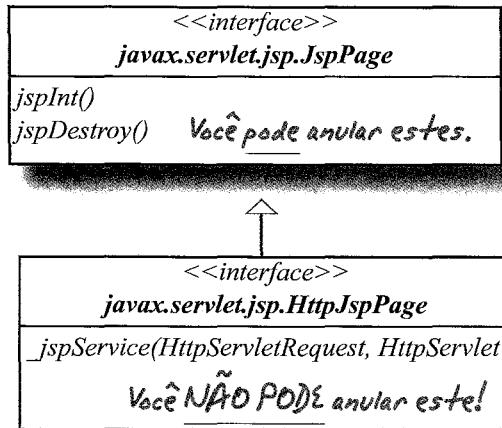
► `jspDestroy()`

Este método é chamado pelo método `destroy()` do servlet. Você também pode anular este método.

► `jspService()`

Este método é chamado pelo método `service()` do servlet, o que significa que ele roda em uma thread separada para cada solicitação. O Container passa para este método os objetos Solicitação e Resposta.

Você pode anular este método! Você não pode fazer NADA por sua própria conta neste método (exceto escrever o código que vai dentro dele), e é responsabilidade do fabricante do Container aceitar seu código JSP e adaptar o método `_jspService()` que irá utilizá-lo.



O ciclo de vida de um JSP

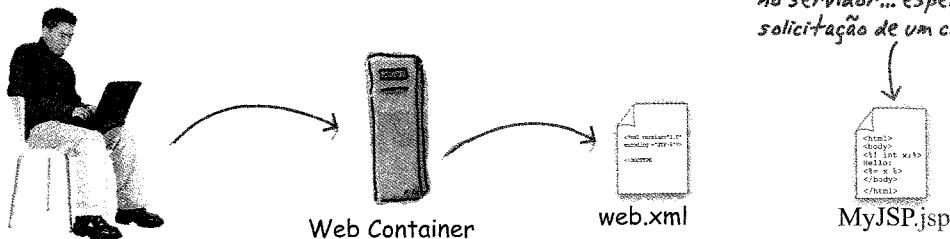
Você escreve o arquivo `.jsp`.

O **Container** escreve o arquivo `.java` para o servlet no qual o seu JSP se transformou.

- 1 Kim escreve um arquivo `.jsp` e o distribui como parte de uma aplicação.

O Container "lê" o `web.xml` (DD) para esta aplicação, mas não faz mais nada com o arquivo `.jsp` (até que ele seja solicitado pela primeira vez).

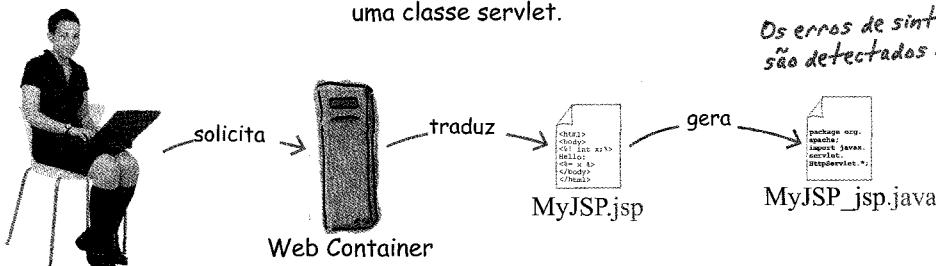
Está justamente localizado no servidor... esperando pela solicitação de um cliente.



- 2 O cliente clica em um link que solicita o `.jsp`.

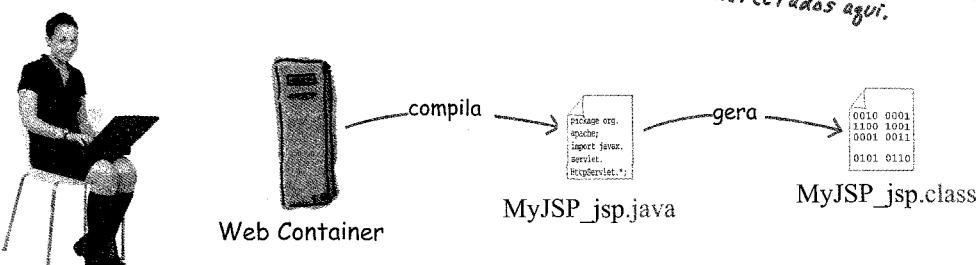
O Container tenta TRADUZIR o `.jsp` em código-fonte `.java` para uma classe servlet.

Os erros de sintaxe JSP
são detectados nesta fase.



- 3 O Container tenta COMPILAR o código-fonte `.java` do servlet em um arquivo `.class`.

Os erros de linguagem/ sintaxe Java
são detectados aqui.

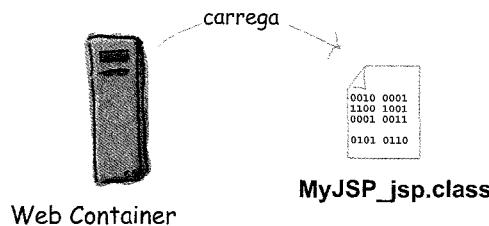


O ciclo de vida do JSP continua...

4



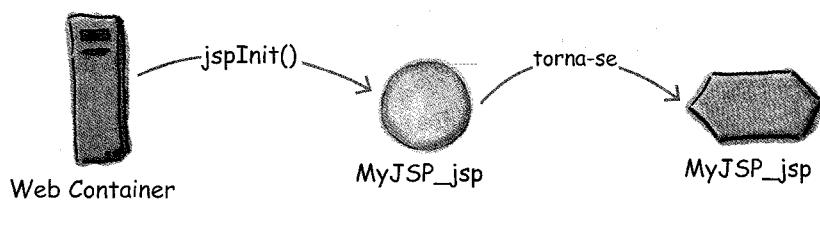
O Container CARREGA a classe do servlet gerada recentemente.



5



O Container insta o servlet e faz com que o método `jspInit()` dele rode.

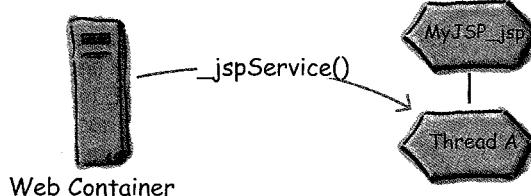


O objeto agora é um servlet completo, pronto para aceitar as solicitações do cliente.

6

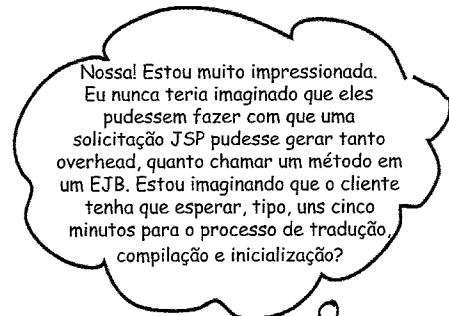


O Container cria uma nova thread para tratar a solicitação deste cliente, e o método `_jspService()` do servlet roda.



Tudo o que acontece depois disso é simplesmente a rotina normal do servlet para tratamento de solicitações.

Por fim, o servlet envia uma resposta de volta para o cliente (ou encaminha a solicitação para outro componente da aplicação).



Nossa! Estou muito impressionada.
Eu nunca teria imaginado que eles
pudessem fazer com que uma
solicitação JSP pudesse gerar tanto
overhead, quanto chamar um método em
um EJB. Estou imaginando que o cliente
tenha que esperar, tipo, uns cinco
minutos para o processo de tradução,
compilação e inicialização?

A tradução e a compilação acontecem UMA única vez

Quando você distribui uma aplicação com um JSP, toda a etapa de tradução e compilação acontece uma única vez na vida do JSP. Uma vez que esteja traduzido e compilado, ele fica exatamente como qualquer outro servlet. É como qualquer outro servlet, uma vez que tenha sido carregado e inicializado, a única coisa que acontece na hora da solicitação é a criação ou a alocação de uma thread para o método service. Portanto, a figura mostrada na página anterior e nesta é apenas para *a primeira solicitação*.

P: Tudo bem, quer dizer que somente o primeiro cliente a solicitar o JSP é que leva a pancada. Mas DEVE haver uma maneira de configurar o servidor para pré-traduzir e compilar... certo?

R: Embora apenas o primeiro cliente tenha que esperar, a maioria dos fabricantes de Container OFERECE uma forma de fazer toda a tarefa de tradução e compilação acontecer antecipadamente, de forma que até mesmo a primeira solicitação aconteça como em qualquer outra solicitação do servlet. Mas cuidado, isso depende do fabricante e não é garantido. EXISTE uma menção na especificação JSP (JSP 11.4.2) que sugere um protocolo para a pré-compilação do JSP. Você faz a solicitação para o JSP anexando uma query string "?jsp_compile" e o Container pode (caso queira) fazer a tradução/compilação ali mesmo, em vez de esperar pela primeira solicitação real do cliente.

Se o JSP se transforma em um servlet, eu gostaria de saber se posso configurar os parâmetros init do servlet... e já que estou aqui, será que posso anular o método init() do servlet...



Aponte seu lápis

Observe estas perguntas. Dê uma olhada nas páginas anteriores (e capítulos), caso ache necessário, mas não continue antes de terminar.

Sim, você PODE obter os parâmetros init do servlet através de um JSP, então, as questões são:

- 1) Como você *restaurou* de seu código? (Grande, imensa, embarçosa dica: muito parecido com a maneira que você o restaura em um servlet "comum". De qual objeto você normalmente obtém os parâmetros init do servlet? Ele está disponível para o seu código JSP?)
- 2) Como e onde você *configura* os parâmetros init do servlet?
- 3) Suponha que você queira *mesmo* anular o método init()... como você o faria? Há alguma outra maneira de se fazer e que traga o mesmo resultado?

Inicializando seu JSP

Você *pode* fazer tarefas relacionadas à inicialização do servlet com seu JSP, mas é *um pouco* diferente do que você faz num servlet comum.

Configurando os parâmetros init do servlet

Você configura os parâmetros init do servlet para o seu JSP, quase da mesma maneira que você os configura para um servlet normal. A única diferença é que você tem que acrescentar um elemento `<jsp-file>` na tag `<servlet>`.

```
<web-app ...>
  <servlet>
    <servlet-name>MyTestInit</servlet-name>
    <jsp-file>/TestInit.jsp</jsp-file> ←
    <init-param>
      <param-name>email</param-name>
      <param-value>ikickedbutt@wickedlysmart.com</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>MyTestInit</servlet-name> ←
    <url-pattern>/TestInit.jsp</url-pattern>
  </servlet-mapping>
</web-app>
```

Esta é a única linha que é diferente de um servlet comum. Dig basicamente: aplique tudo desta tag `<servlet>` no servlet criado por esta página JSP...

Quando você define um servlet para um JSP, você deve igualmente definir um servlet mapeando à página de JSP

Anulando o `jsplInit()`

É simples assim. Se você implementar um método `jsplInit()`, o Container o chama no começo da vida desta página como um servlet. Ele é chamado pelo método `init()` do servlet, portanto, na hora em que este método roda, haverá um `ServletConfig` e um `ServletContext` disponíveis para o servlet. Isto significa que você pode chamar o `getServletConfig()` e o `getServletContext()` de dentro do `jsplInit()`.

Este exemplo usa o `jsplInit()` para recuperar um parâmetro init do servlet (configurado no DD) e usa este valor para configurar um atributo de escopo application.

```
<%! ←
  Anula o método jsplInit() usando
  uma declaração.
  public void jsplInit() {
    ServletConfig sConfig = getServletConfig();
    String emailAddr = sConfig.getInitParameter("email");
    ServletContext ctx = getServletContext();
    ctx.setAttribute("mail", emailAddr);
  }
%>
  Você está num servlet, portanto, pode chamar
  seu getServletConfig() herdado!
  Consegue uma referência para o ServletContext e
  configura um atributo de escopo application.
  Isto é EXATAMENTE o que você
  faz num servlet normal.
```

Os atributos em um JSP

O exemplo da página anterior mostra o JSP configurando um atributo `application`, usando uma declaração que anule o `jsplInit()`. Porém, na maioria das vezes você usará um dos quatro *objetos implícitos* para receber e configurar os atributos correspondentes aos quatro escopos disponíveis no JSP.

Sim, quatro. Lembre-se, além dos escopos-padrão do servlet – `request`, `session` e `application` (contexto), o JSP acrescenta um quarto escopo – o escopo `page` –, que você obtém de um objeto `pageContext`.

Geralmente, você não precisará (e nem se preocupar) do escopo `page`, a menos que você esteja desenvolvendo tags personalizadas. Por isso, não entraremos em maiores detalhes, até o capítulo que trata das tags personalizadas.

Em um servlet

Em um JSP (usando objetos implícitos)

Application	<code>getServletContext().setAttribute("foo", barObj);</code>	<code>application.setAttribute("foo", barObj);</code>
Request	<code>request.setAttribute("foo", barObj);</code>	<code>request.setAttribute("foo", barObj);</code>
Session	<code>request.getSession().setAttribute("foo", barObj);</code>	<code>session.setAttribute("foo", barObj);</code>
Page	Não se aplica!	<code>pageContext.setAttribute("foo", barObj);</code>

Mas não é só isso! Em um JSP, existe uma *outra* maneira de receber e configurar os atributos em *qualquer* escopo, usando apenas o objeto implícito `pageContext`. Continue e descubra como...



Não existe um escopo “context”... mesmo que os atributos no escopo application estejam associados ao objeto ServletContext.

A convenção de nomenclatura pode levá-lo a pensar que os atributos armazenados no `ServletContext` são... do escopo `context`. Não existe isso. Lembre-se, quando você encontrar “Context”, pense em “application”. Porém, existe uma diferença entre os nomes `servlet` e `JSP` usados para obter os atributos do escopo `application`: num `servlet`, você diz:

`getServletContext().getAttribute("foo")`

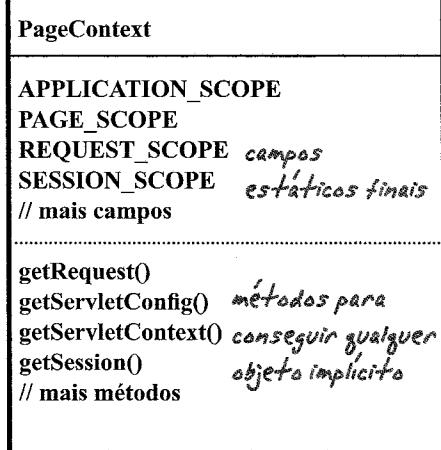
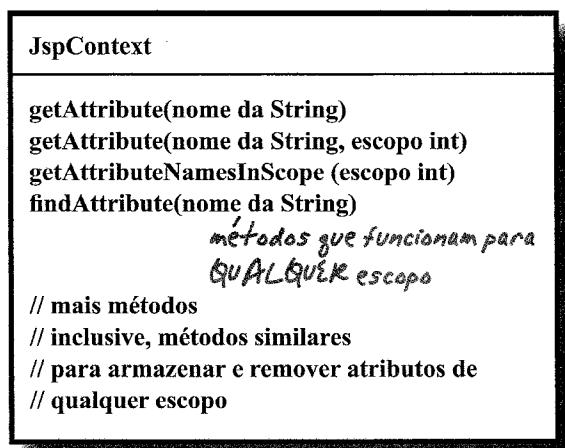
mas em um `JSP` você diz:

`application.getAttribute("foo")`

Usando o PageContext para atributos

Você pode usar uma referência PageContext para obter atributos a partir de qualquer escopo, inclusive o escopo page para atributos associados ao PageContext.

Os métodos que funcionam com os outros escopos usam um argumento int para indicar o escopo. Embora os métodos de acesso derivem do JspContext, você encontrará as constantes para os escopos na classe PageContext.



Exemplos usando o pageContext para obter e configurar atributos

Configurando um atributo do escopo page

```
<% Float one = new Float(42.5); %>
<% pageContext.setAttribute("foo", one); %>
```

Obtendo um atributo do escopo page

```
<%= pageContext.getAttribute("foo") %>
```

Usando o pageContext para configurar um atributo do escopo session

```
<% Float two = new Float(22.4); %>
<% pageContext.setAttribute("foo", two, PageContext.SESSION_SCOPE); %>
```

Usando o pageContext para obter um atributo do escopo session

```
<%= pageContext.getAttribute("foo", PageContext.SESSION_SCOPE) %>
(Que é idêntico a: <%= session.getAttribute("foo") %>)
```

Usando o pageContext para obter um atributo do escopo application

Email is:
`<%= pageContext.getAttribute("mail", PageContext.APPLICATION_SCOPE) %>`

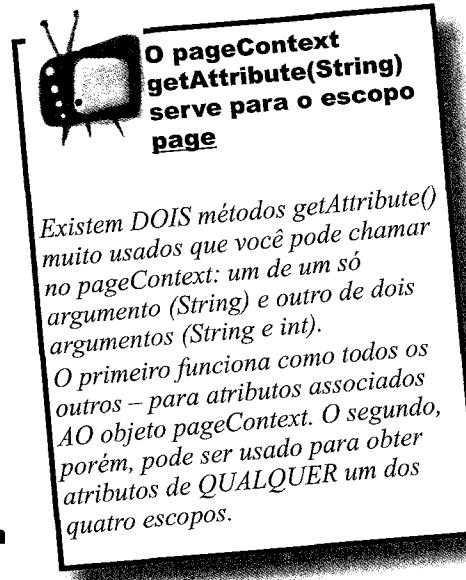
Dentro de um JSP, o código acima é idêntico a:

Email is:
`<%= application.getAttribute("mail") %>`

Usando o pageContext para encontrar um atributo quando você não conhece o escopo

`<%= pageContext.findAttribute("foo") %>` encontrá-lo onde?

Onde o método findAttribute() olha? Ele olha primeiro no page context, e se houver um atributo "foo" com o escopo page context, então, chamar o `findAttribute(nome da String)` num PageContext funcionará exatamente como chamar o `getAttribute(nome da String)` num PageContext. Mas se não existir nenhum atributo "foo", o método começa procurando em outros escopos, do mais restrito ao menos restrito – ou seja, primeiro o escopo request, depois o session e, finalmente, o application. **O primeiro que ele encontrar com este nome ganha.**



Existem **DOIS** métodos `getAttribute()` muito usados que você pode chamar no `pageContext`: um de um só argumento (`String`) e outro de dois argumentos (`String e int`). O primeiro funciona como todos os outros – para atributos associados ao objeto `pageContext`. O segundo, porém, pode ser usado para obter atributos de **QUALQUER** um dos quatro escopos.

Já que estamos falando sobre isso... vamos falar mais sobre as três diretivas

Nós já vimos as diretivas usadas para inserir as declarações import na classe do servlet gerado pelo seu JSP. Foi a diretiva *page* (um dos três tipos de diretivas) com um atributo *import* (um dos 13 atributos da diretiva *page*). Nós agora daremos uma rápida olhada nas outras, embora algumas não sejam abordadas em detalhes até os próximos capítulos. Outras, inclusive, nem serão *muito* faladas aqui, pois são raramente usadas.

1 A diretiva page

```
<%@ page import="foo.*" session="false" %>
```

Define propriedades específicas da página, como códigos de caracteres, o tipo de conteúdo para a resposta e se esta página deveria possuir o objeto implícito *session*. Uma diretiva *page* pode utilizar até 13 atributos diferentes (como o atributo *import*), embora apenas quatro sejam cobrados no exame.

2 A diretiva taglib

```
<%@ taglib tagdir="/WEB-INF/tags/cool" prefix="cool" %>
```

Define as bibliotecas de tags disponíveis para o JSP. Nós ainda não falamos sobre o uso das tags customizadas e ações padronizadas; portanto, talvez não faça sentido neste momento. Fique com isto por enquanto... teremos dois capítulos inteiros sobre bibliotecas de tags vindo em breve.

3 A diretiva include

```
<%@ include file="wickedHeader.html" %>
```

Define os textos e os códigos que são acrescentados na página atual no momento da tradução. Isto lhe permite construir pedaços reutilizáveis (como um título-padrão para a página ou a barra de navegação), que podem ser acrescentados a cada página, sem precisar repetir todo aquele código em cada JSP.

P: Estou confuso... o título deste tópico diz: “Já que estamos falando sobre isso...”, mas eu não vejo o que as diretivas têm a ver com *pageContext* e atributos.

R: Nada a ver, realmente. Nós só dissemos aquilo para disfarçar a transição inexistente e patética entre dois tópicos não relacionados. Esperávamos que ninguém notasse, mas NÃO... você simplesmente não permitiria, não é mesmo?

Os atributos para a diretiva page

Dos 13 atributos da diretiva page da especificação JSP 2.0, apenas *quatro* são cobrados no exame. Você NÃO precisa memorizar a lista inteira, mas veja o que você pode fazer. (Veremos os atributos *isELIgnored* e os dois atributos relacionados a erros mais adiante.)

PODE cair na prova

import	Define as declarações import do Java que serão adicionadas à classe do servlet gerado. Você tem alguns import de graça (por padrão): <i>java.lang</i> (óbvio), <i>javax.servlet</i> , <i>javax.servlet.http</i> e <i>javax.servlet.jsp</i> .
isThreadSafe	Define se o servlet gerado precisa implementar o <i>SingleThreadModel</i> que, como você já sabe, é uma Coisa Terrivelmente Ruim. O valor-padrão é... “true”, o que significa: “Minha aplicação é thread-safe e eu NÃO preciso implementar o <i>SingleThreadModel</i> , pois sei que é ruim por natureza.” A única razão para especificar este atributo seria se você precisasse configurar o valor do atributo para “false”, o que significa que você quer que o servlet gerado use o <i>SingleThreadModel</i> , mas você nunca fará isto.
contentType	Define o tipo MIME (e o código opcional do caractere) para a resposta JSP. <i>Você conhece o padrão.</i>
isELIgnored	Define se as expressões EL serão ignoradas quando a página for traduzida. Nós só falaremos a respeito da EL no próximo capítulo. Por enquanto, saiba apenas que você talvez queira ignorar a sintaxe EL em sua página, e este é um dos dois modos que você pode informar ao Container.
isErrorPage	Define se a página atual representa <i>uma outra</i> página de erro dos JSPs. O valor-padrão é “false”, mas se for true, as páginas têm acesso ao objeto implícito <i>exception</i> (que é uma referência ao inconveniente <i>Throwable</i>). Se for false, o objeto implícito <i>exception</i> fica indisponível para o JSP.
errorPage	Define uma URL para o recurso para onde os <i>uncaught Throwables</i> devem ser enviados. Se você definir um JSP aqui, então <i>este</i> JSP terá um atributo isErrorPage=”true” na sua diretiva page.

NÃO cairá na prova

language	Define a linguagem scripting usada nos scriptlets, expressões e declarações. No momento, o único valor possível é “java”, mas ele está aqui pensando no futuro – quando outras linguagens provavelmente serão usadas.
extends	Define a superclasse da classe que este JSP tornar-se-á. Você não o usará, a menos que REALMENTE saiba o que está fazendo – ele anula a hierarquia da classe fornecida pelo Container.
session	Define se a página terá um objeto implícito <i>session</i> . O valor-padrão é “true”.
buffer	Define como o buffering é tratado pelo objeto implícito <i>out</i> (referente ao <i>JspWriter</i>).
autoFlush	Define se a saída bufferizada está limpa automaticamente. O valor-padrão é “true”.
info	Define uma String que é inserida na página traduzida, exatamente para que você possa obtê-la usando o método <i>getServletInfo()</i> herdado do servlet.
pageEncoding	Define o código de caracteres para o JSP. O padrão é “ISO-8859-1” (a menos que o atributo <i>contentType</i> já o tenha definido ou a página use a sintaxe XML Document).

Que capítulo lindo, com explicações APAIXONANTES de como colocar um código Java em um JSP! Mas olha só este memorando para a empresa inteira que acabei de receber.



**Memorando Intra-escritórios do
Gerente de Tecnologia**

URGENTE

Para efeito imediato, qualquer um que seja pego usando scriptlets, expressões ou declarações no seu código JSP será suspenso, sem direito a pagamento, até que seja determinado se o programador foi totalmente responsável ou estava simplesmente tentando manter o código de algum OUTRO idiota.

Se, aliás, for detectado que o programador é na verdade o responsável, a companhia efetuará a sua dispensa.

--
Rick Forester
Gerente de Tecnologia

--
“Lembre-se: there is no ‘I’ in TEAM.”

“Ao escrever seu código, faça de conta que o próximo cara* a mantê-lo é um maníaco homicida que sabe onde você mora.”

[*Nota para o RH: ao usarmos “cara” não especificamos gênero.]

Scriptlets consideradas prejudiciais?

Isso é verdade? Poderia haver uma desvantagem em se colocar todo este Java no seu JSP? Afinal, não é este todo o PROPÓSITO do JSP? De forma que você escreva seu Java no que é essencialmente uma página HTML, ao contrário de escrever o HTML em uma classe Java?

Algumas pessoas acreditam (tudo bem, tecnicamente um *monte* de gente, inclusive as equipes que criaram as especificações do JSP e do servlet) ser uma *prática ruim* colocarmos todo este Java no seu JSP.

Por que? Imagine que você foi contratado para construir um grande site. A sua equipe inclui uns poucos programadores Java e um enorme grupo de “webdesigners” – artistas gráficos e profissionais que criam páginas utilizando o Dreamweaver e o Photoshop para construírem aquelas páginas fabulosas. Eles não são *programadores* (bem, existem aqueles que continuam achando que HTML é “programação”).



Atores aspirantes trabalhando como webdesigners enquanto esperam por sua grande oportunidade no showbiz.



Duas perguntas: POR QUE você está nos ensinando isso e QUAL é a alternativa? Que outra m**** EXISTE, além do HTML, já que não podemos usar scriptlets, declarações e expressões no JSP?

Não EXISTIA nenhuma alternativa.

Isso significa que já existem *montanhas* de arquivos JSP abarrotados de código Java enfiados em cada pedaço da página, acomodados entre scriptlets, expressões e tags de declarações. Já está lá e não há nada que alguém possa fazer para mudar o passado. Portanto, significa que você tem que saber como *ler* e *entender* estes elementos e como *manter* páginas escritas com eles (a menos que você tenha a chance de recriar todo o JSP da sua aplicação).

Cá entre nós, até achamos que ainda há lugar para coisas desse tipo – não há nada melhor do que um pouco de Java no JSP para testar rapidamente algo no seu servidor. Mas na maioria das vezes, você não vai querer usar isso nas suas páginas verdadeiras e em produção.

A razão para tudo isto estar no exame é que as *alternativas* ainda são novidade. Por isso, a maioria das páginas hoje ainda é “da antiga”. **Por enquanto, você ainda tem que ser capaz de trabalhar assim!** Em algum momento, quando novas técnicas que dispensem Java atingirem um público considerável, os objetivos deste capítulo provavelmente não constarão mais na prova. E respiraremos aliviados pela morte do Java-em-JSPs.

Mas esse dia ainda não é hoje.

(Nota para pais e professores: a palavra de cinco letras implícita no balão acima, que começa com “m” seguida por quatro asteriscos, NÃO é o que vocês estão pensando. É simplesmente uma palavra que achamos engraçada demais para ser incluída aqui, sem que distraísse o leitor. Por isso ela foi truncada. Porque é engraçada. E não, *imprópria*.)

Poxa, se pelo menos existisse uma maneira do JSP usar tags simples que permitissem que os métodos Java rodassem, sem ter que colocar na página o código Java em si.



EL: a resposta para, digamos, tudo.

Ou *quase* tudo. Mas certamente uma resposta para duas grandes reclamações sobre colocar o Java no JSP:

- 1. Os designers não precisariam saber Java.**
- 2. O código Java existente no JSP é difícil de mudar e manter.**

EL significa “Expression Language” e, oficialmente, tornou-se parte da especificação a partir da especificação JSP 2.0. A EL é quase sempre o jeito mais simples de se fazer algo que você normalmente faria com scriptlets e expressões.

É claro que neste momento você está pensando: “Mas se eu quiser que meu JSP use métodos customizados, como poderei declará-los e escrevê-los se não posso usar Java?”

Ahhh... escrever a funcionalidade real (o código do método) *não* é o propósito da EL. O propósito da EL é oferecer um jeito simples de *invocar* o código Java – mas o código em si pertence a *algum outro lugar*. Ou seja, uma classe Java simples normal, que funcione como um JavaBean, uma classe com métodos estáticos, ou aquilo que chamamos de Tag Handler. Em outras palavras, você não escreve o código do método no seu JSP se você estiver seguindo as Melhores Práticas de hoje em dia. Você escreve o método Java em *algum outro lugar* e o *chama* usando a EL.

Uma apresentação rápida da EL

O capítulo seguinte fala só sobre EL, portanto não entraremos em detalhes aqui. A única razão de estarmos falando dela é devido ao fato de ela ser um outro elemento (com sua própria sintaxe) que o JSP aceita. E os objetivos da prova que constam neste capítulo incluem reconhecer tudo o que possa estar em um JSP.

Uma expressão EL SEMPRE se parece com: \${alguma coisa}

Ou seja, ela vem SEMPRE entre chaves e precedida pelo símbolo (\$).

Esta expressão EL:

```
Please contact: ${applicationScope.mail}
```

É o mesmo que esta expressão Java:

```
Please contact: <%= application.getAttribute("mail") %>
```

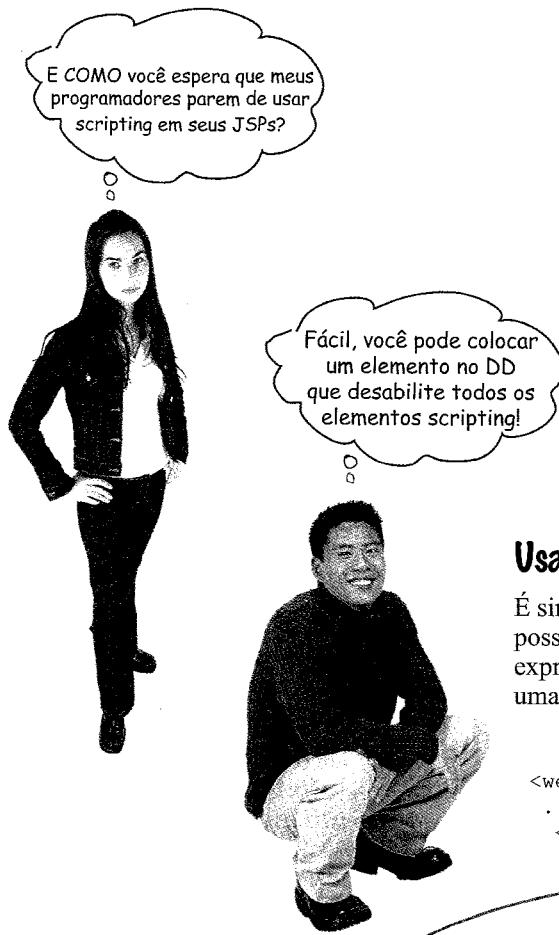
Não existem Perguntas Idiotas

P: Não querendo ser chato, mas creio não ter visto nenhuma grande diferença entre a EL e a expressão Java. Claro que ela é um pouco menor, mas vale a pena mudar toda a linguagem scripting e os códigos no JSP?

R: Você AINDA não viu as vantagens da EL. As diferenças tornar-se-ão óbvias no próximo capítulo, quando mergulharemos no assunto de cabeça. Mas você deve se lembrar que para um programador Java, a EL NÃO oferece, necessariamente, uma gigantesca vantagem no desenvolvimento. Na realidade, para um programador Java ela simplesmente significa "uma coisa a mais (com sua própria sintaxe e tudo) para aprender, quando, psiu, eu já SEI Java..."

Mas não é só com relação a você. A EL é *muito* mais fácil para alguém que não programe em Java aprender rapidamente. E para um programador Java, ainda é muito mais fácil manter uma página sem scripts.

Sim, é mais uma coisa a aprender. Ela não deixa os webdesigners completamente a salvo, mas você verá em breve que é mais intuitivo e natural para eles usar a EL. Por enquanto, aqui neste capítulo, você simplesmente precisará ser capaz de *reconhecer* uma EL quando se deparar com ela. E não se preocupe ainda em diferenciar se a EL é válida – tudo que queremos saber agora é se você consegue identificar uma expressão EL em uma página JSP.



*Isto desabilita os elementos scripting para TODOS os JSPs na aplicação (porque usamos o *jsp como padrão de URL).*

Usando <scripting-invalid>

É simples, você pode impedir que um JSP possua elementos scripting (scriptlets, expressões Java ou declarações), colocando uma tag <scripting-invalid> no DD:

```
<web-app ...>
...
<jsp-config>
  <jsp-property-group>
    <url-pattern>*.jsp</url-pattern>
    <scripting-invalid>
      true
    </scripting-invalid>
  </jsp-property-group>
</jsp-config>
...
</web-app>
```

Cuidado – você pode ter visto outros livros e artigos mostrando uma diretiva page que desabilita o scripting. Na versão *rascunho* da especificação 2.0, havia um atributo da diretiva page:

<%@ page isScriptingEnabled="false" %>

mas ele foi removido da especificação definitiva!!

A **única** forma de invalidarmos o scripting agora é através da tag <scripting-invalid> no DD.

Isto não funciona! O atributo isScriptingEnabled não consta mais na especificação do JSP!

Você pode decidir ignorar a EL

Sim, a EL é uma coisa legal que salvará o mundo, como nós já sabemos. Mas algumas vezes você pode querer desabilitá-la. Por quê?

Lembre-se de quando a palavra-chave *assert* foi adicionada na linguagem Java versão 1.4. De uma hora para outra, o identificador perfeitamente legal e não reservado “assert” passou a *significar* algo para o compilador. Então, se você tivesse, digamos, uma variável chamada *assert*, você estaria “ferrado”. Com exceção da versão J2SE 1.4, que veio com as declarações desabilitadas por padrão. Se você soubesse que estaria escrevendo (ou recompilando) um código que não usasse *assert* como identificador, você poderia escolher habilitar as tais declarações.

Ou seja, é quase o mesmo que desabilitar a EL – se você decidiu ter template text (HTML simples ou texto) em um JSP que incluísse algo parecido com a EL (`${alguma coisa}`), você estaria com um Grande Problema, caso não pudesse informar ao Container para ignorar tudo que se parecesse com a EL, em vez de tratar como qualquer outro texto comum. Exceto pelo fato de haver uma grande diferença entre a EL e as assertions:

A EL é habilitada por padrão!

Se você quiser que os elementos do seu JSP parecidos com a EL sejam ignorados, você tem que dizer explicitamente, ou através de uma diretiva page ou de um elemento no DD.

Inserindo `<el-ignored>` no DD

```
<web-app ...>
  ...
  <jsp-config>
    <jsp-property-group>
      <url-pattern>*.jsp</url-pattern>
      <el-ignored>
        true
      </el-ignored>
    </jsp-property-group>
  </jsp-config>
  ...
</web-app>
```

 A diretiva page tem prioridade sobre a configuração do DD!

Se houver um conflito entre a tag <el-ignored> configurada no DD e o atributo isELIgnored da diretiva page, a diretiva sempre ganha! Isto permite a você determinar o comportamento-padrão no DD, mas, também, anulá-lo em uma página específica, usando uma diretiva page.

 Cuidado com as inconsistências na nomenclatura!

A tag do DD é <el-ignored>, portanto, alguém poderia pensar, com toda razão, que o atributo da diretiva page seria, talvez, elIgnored. Mas não, esse alguém estaria errado se chegasse a tal conclusão. O DD e a diretiva para ignorar a EL são distintos! Não caia na armadilha da <is-el-ignored>.

Usando o atributo `isELIgnored` da diretiva page

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

O atributo da diretiva page começa com “is”, mas a tag no DD não!

Mas espere... existe um outro elemento JSP que nós ainda não vimos: as ações

Até aqui, você viu cinco diferentes tipos de elementos que podem aparecer em um JSP: scriptlets, diretivas, declarações, expressões Java e expressões EL.

Mas nós não vimos as *ações*. Elas vêm em dois sabores: *padrões* e... *não-padrões*.

Ação-Padrão:

```
<jsp:include page="wickedFooter.jsp" />
```

Outra Ação:

```
<c:set var="rate" value="32" />
```

Por enquanto, não se preocupe com o que elas fazem ou como funcionam, apenas reconheça uma ação quando você encontrar a sintaxe em um JSP. Mais tarde, entraremos em detalhes.

Embora possa parecer confuso, existem ações que não são consideradas *ações-padrão*, mas que ainda fazem parte de uma biblioteca agora padronizada. Ou seja, você aprenderá mais tarde que algumas ações não-padrão (os objetivos referem-se a elas como *customizadas*) são... padrões, mas ainda não são consideradas “ações-padrão”. Sim, é isso mesmo: elas são ações customizadas padronizadas não-padrão. Agora não ficou mais claro?

Num capítulo mais adiante, quando chegarmos em “usando tags”, usaremos um vocabulário um pouco mais rico, em que trataremos deste assunto mais detalhadamente. Então, relaxe. **Agora, tudo o que nos interessa é que você reconheça uma ação quando se deparar com ela em um JSP!**



Aponte seu lápis

Observe a sintaxe de uma ação e compare-a com a sintaxe de outros tipos de elementos JSP. E responda:

- 1) Quais são as diferenças entre o elemento de uma ação e um scriptlet?

- 2) Como você reconheceria uma ação?



Exercícios

Matriz de Avaliação

O que acontece quando cada uma destas configurações (ou a combinação delas) ocorre? Você verá as respostas quando virar a página; portanto, faça AGORA.

Faça um X na coluna avaliado, se as combinações levarem as expressões EL a serem avaliadas, OU faça um X na coluna ignorado, se a EL for tratada como outro texto template. Nenhuma linha terá as duas opções marcadas, é claro.

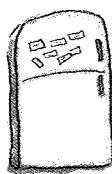
1 Avaliação da EL

Configuração no DD <el-ignored>	diretiva de página isELIgnored	avaliado	ignorado
não-especificado	não-especificado		
falsa	não-especificado		
verdadeira	não-especificado		
falsa	falsa		
falsa	verdadeira		
verdadeira	falsa		

Faça um X na coluna avaliado, se as configurações levarem as expressões de scripting a serem avaliadas, OU faça um X na coluna erro, se o scripting gerar um erro de tradução.

2 Validade do scripting

Configuração no DD <scripting-invalid>	avaliado	ignorado
não-especificado		
verdadeira		
falsa		



Ímã de Geladeira (Elementos JSP)

Correlacione o elemento JSP com o seu trecho de código, associando o código à caixa que o representa. Lembre-se de que você terá questões “arrastar e soltar” no exame, semelhante a este exercício; então, não o pule!

Tipo de elemento JSP

diretiva

Trecho de código JSP

declaração

Arraste e solte na caixa correspondente.

```
<% Float one = new Float (42.5); %>
```

expressão EL

```
<%! int y = 3; %>
```

```
<% @ page import="java.util.*"%>
```

scriptlet

```
<jsp:include file="foo.html"/>
```

```
<%= pageContext.getAttribute("foo")%>
```

```
email: ${applicationScope.mail}
```

expressão

ação



Ímã de Geladeira (Elementos JSP): Continuação

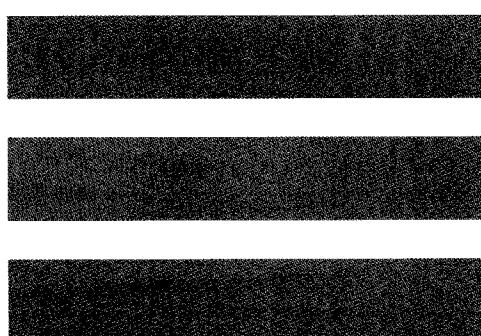
Você sabe o nome deles, mas você se lembra *onde eles ficam no servlet gerado?*
É claro que sim. Mas este é só mais um reforço antes de passarmos para um capítulo e um assunto diferentes.

(Considerando o arquivo da classe servlet, coloque o elemento no quadro onde o código gerado por ele ficará. Note que o ímã, em si, não representa o código REAL que será gerado.)

```
public final class BasicCounter_jsp extends org.apache.jasper.runtime.HttpJspBase  
    implements org.apache.jasper.runtime.JspSourceDependent {
```

```
public void _jspService(HttpServletRequest request, HttpServletResponse response)  
    throws java.io.IOException, ServletException {
```

...



A ordem destes três ímãs
é irrelevante.

```
}
```

```
<%= request.getAttribute("foo")%>  
email: ${applicationScope.mail}  
<% @ Page import="java.util.*"%>  
<% Float one = new Float (42.5); %>  
<%! int y = 3; %>
```



Matriz de Avaliação RESPOSTAS

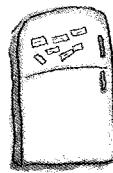
Exercícios

1 Avaliação da EL

Configuração no DD <el-ignored>	diretiva de página isELIgnored	avaliado	ignorado
não-especificado	não-especificado	✓	
falsa	não-especificado	✓	
verdadeira	não-especificado		✓
falsa	falsa	✓	
falsa	verdadeira		✓
verdadeira	falsa	✓	

2 Validez do scripting

Configuração no DD <scripting-invalid>	avaliado	ignorado
não-especificado	✓	
verdadeira		✓
falsa	✓	



Ímã de Geladeira (Elementos JSP)

Respostas

Tipo de elemento JSP

```
<% @ page import="java.util.*"%>
```

diretiva

```
<%! int y = 3; %>
```

declaração

```
email: ${applicationScope.mail}
```

expressão EL

```
<% Float one = new Float (42.5); %>
```

scriptlet

```
<%= pageContext.getAttribute("foo") %>
```

expressão

```
<jsp:include file="foo.html" />
```

ação

Trecho de código JSP

A palavra “expressão” significa “expressão scripting”, e NÃO “expressão EL”.



É claro que a palavra “expressão” é muito usada pelos elementos JSP. Se você encontrar a palavra “expressão”, ou “expressão scripting”, elas querem dizer o mesmo: uma expressão que usa a sintaxe da linguagem Java:

```
<%= foo.getName() %>
```

A única vez que a palavra “expressão” se refere à EL, é quando você especificar “EL” no label ou nas descrições! Portanto, considere sempre que o padrão para a palavra “expressão” é “scripting/expressão Java”, e não EL.



Ímã de Geladeira (Elementos JSP): Continuação RESPOSTAS

```
<% @ page import="java.util.*"%>
```

Uma diretiva de página com um atributo `import` vira uma declaração `import Java`.

```
public final class BasicCounter_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {
```

```
<%! int y = 3; %>
```

As declarações servem para declarar MEMBROS; logo, elas ficam dentro da classe e fora dos métodos.

```
public void _jspService(HttpServletRequest request, HttpServletResponse response)
    throws java.io.IOException, ServletException {
```

```
<%=request.getAttribute("foo")%>
```

As expressões viram declarações `out.print()` no método `service`.

```
<% Float one = new Float (42.5); %>
```

Os scriptlets ficam dentro do método `service`.

```
email: ${applicationScope.mail}
```

As expressões EL ficam dentro do método `service`.

```
}
```

(Nota: a ordem destes três itens é irrelevante.)

NOTA: lembre-se de que o código JSP não FICA, de fato, assim dentro do servlet... é todo traduzido para o código Java. Este exercício serve para mostrar em qual parte da classe gerada estes elementos FICAM, mas não estamos mostrando-lhe o código real gerado para o qual elementos são traduzidos. Por exemplo, a declaração que era `<%! int y = 3;%>` tornar-se simplesmente `int y = 3;`



Teste Preparatório - Capítulo 7

1 Dado o elemento DD:

- 47. <jsp-property-group>
- 48. <url-pattern>*.jsp</url-pattern>
- 49. <el-ignored>true</el-ignored>
- 50. </jsp-property-group>

O que o elemento faz? (Escolha todas as que se aplicam.)

- A. Todos os arquivos com o mapeamento da extensão especificada devem ser tratados pelo container JSP como arquivos que respeitam a sintaxe XML.
- B. Todos os arquivos com o mapeamento da extensão especificada devem ter seus códigos EL avaliados pelo container JSP.
- C. Por padrão, NENHUM arquivo com o mapeamento da extensão especificada deve ter seus códigos Expression Language avaliados pelo container JSP.
- D. Nada. Esta tag NÃO é reconhecida pelo container.
- E. Embora esta tag seja válida, ela é redundante, pois o container já faz isso por padrão.

2 Quais das diretivas abaixo representam uma resposta HTTP do tipo "image/svg"? (Escolha todas as que se aplicam.)

- A. <%@ page type="image/svg" %>
- B. <%@ page mimeType="image/svg" %>
- C. <%@ page language="image/svg" %>
- D. <%@ page contentType="image/svg" %>
- E. <%@ page pageEncoding="image/svg" %>

3 Seja o JSP:

```
1. <%@ page import="java.util.*" %>
2. <html><body> The people who like
3. <%= request.getParameter("hobby") %>
4. are: <br>
5. <% ArrayList al = (ArrayList) request.
getattribute("names"); %>
6. <% Iterator it = al.iterator();
7. while (it.hasNext()) { %>
8. <%= it.next() %>
9. <br>
10. <% } %>
11. </body></html>
```

Que tipos de códigos encontramos nele? (Escolha todas as que se aplicam.)

- A. EL
- B. diretiva
- C. expressão
- D. template text
- E. scriptlet

4 Que declarações sobre o `jspInit()` são verdadeiras? (Escolha todas as que se aplicam.)

- A. Ele tem acesso ao `ServletConfig`.
- B. Ele tem acesso ao `ServletContext`
- C. Só é chamado uma vez.
- D. Pode ser anulado.

5 Que tipos de objetos estão disponíveis para o método `jspInit()`?
(Escolha todas as que se aplicam.)

- A. `ServletConfig`
 - B. `ServletContext`
 - C. `JspServletConfig`
 - D. `JspServletContext`
 - E. `HttpServletRequest`
 - F. `HttpServletResponse`
-

6 Dado:

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

O que acontece? (Escolha todas as que se aplicam.)

- A. Nada. A diretiva `page` NÃO foi definida.
 - B. A diretiva anula a avaliação que o container JSP faz do código da Expression Language em todos os JSPs da aplicação.
 - C. O JSP que possui esta diretiva será tratado pelo container JSP como um arquivo que respeita a sintaxe XML.
 - D. O JSP que possui esta diretiva NÃO deveria possuir nenhum código da Expression Language avaliado pelo container JSP.
 - E. Esta diretiva apenas cancelará a avaliação da EL se o DD declarar um elemento `<el-ignored>true</el-ignored>` com um padrão URL que inclui este JSP.
-

7 Qual declaração referente aos JSPs é verdadeira? (Escolha uma.)

- A. Apenas o `jspInit()` pode ser anulado.
- B. Apenas o `jspDestroy()` pode ser anulado.
- C. Apenas o `_jspService()` pode ser anulado.
- D. O `jspInit()` e o `jspDestroy()` podem ser anulados.
- E. O `jspInit()`, o `jspDestroy()` e o `_jspService()` podem ser anulados.

8 Qual das etapas do ciclo de vida do JSP está fora de ordem?

- A. Traduzir o JSP em servlet.
 - B. Compilar o código-fonte do servlet.
 - C. Call `_jspService()`
 - D. Instar a classe servlet.
 - E. Call `jspInit()`
 - F. Call `jspDestroy()`
-

9 Quais das variáveis implícitas JSP são válidas? (Escolha todas as que se aplicam.)

- A. `stream`
 - B. `context`
 - C. `exception`
 - D. `listener`
 - E. `application`
-

10 Seja uma solicitação com dois parâmetros: um chamado “first”, que representa o primeiro nome do usuário, e o outro chamado “last”, que representa seu último nome.

Qual código scriptlet JSP gera os valores para estes parâmetros?

- A. `<% out.println(request.getParameter("first"));
out.println(request.getParameter("last")); %>`
- B. `<% out.println(application.getInitParameter("first"));
out.println(application.getInitParameter("last")); %>`
- C. `<% println(request.getParameter("first"));
println(request.getParameter("last")); %>`
- D. `<% println(application.getInitParameter("first"));
println(application.getInitParameter("last")); %>`

11 Dado:

```
11. Hello ${user.name}!
12. Your number is <c:out value="${user.phone}" />.
13. Your address is <jsp:getProperty name="user" property="addr" />
14. <% if (user.isValid()) {>You are valid!<% } %>
```

Quais declarações são verdadeiras? (Escolha todas as que se aplicam.)

- A. As linhas 11 e 12 (e nenhuma outra) contêm exemplos de elementos EL.
- B. A linha 14 é um exemplo de código scriptlet.
- C. Nenhuma das linhas deste exemplo contém um template text.
- D. As linhas 12 e 13 incluem exemplos de ações-padrão JSP.
- E. A linha 11 demonstra o uso incorreto da EL.
- F. Todas as quatro linhas deste exemplo seriam válidas em uma página JSP.

12 Qual tag JSP exibirá o parâmetro de inicialização de contexto chamado “javax.sql.DataSource”?

- A. <%= application.getAttribute("javax.sql.DataSource") %>
- B. <%= application.getInitParameter("javax.sql.DataSource") %>
- C. <%= request.getParameter("javax.sql.DataSource") %>
- D. <%= contextParam.get("javax.sql.DataSource") %>

13 Quais declarações sobre desabilitar elementos scripting são verdadeiras? (Escolha todas as que se aplicam.)

- A. Você não pode desabilitar scripting via DD.
- B. Você só pode desabilitar scripting no nível da aplicação.
- C. Você pode desabilitar scripting programaticamente, utilizando o atributo de diretiva de página `isScriptingEnabled`.
- D. Você pode desabilitar scripting via DD usando o elemento `<scripting-invalid>`.

14 Em seqüência, quais são os tipos Java para os seguintes objetos implícitos JSP: `application, out, request, response, session?`

- A. `java.lang.Throwable`
`java.lang.Object`
`java.util.Map`
`java.util.Set`
`java.util.List`
 - B. `javax.servlet.ServletConfig`
`java.lang.Throwable`
`java.lang.Object`
`javax.servlet.jsp.PageContext`
`java.util.Map`
 - C. `javax.servlet.ServletContext`
`javax.servlet.jsp.JspWriter`
`javax.servlet.ServletRequest`
`javax.servlet.ServletResponse`
`javax.servlet.http.HttpSession`
 - D. `javax.servlet.ServletContext`
`java.io.PrintWriter`
`javax.servlet.ServletConfig`
`java.lang.Exception`
`javax.servlet.RequestDispatcher`
-

15 Qual das opções representa um exemplo da sintaxe usada para importar uma classe em um JSP?

- A. `<% page import="java.util.Date" %>`
 - B. `<%@ page import="java.util.Date" @%>`
 - C. `<%@ page import="java.util.Date" %>`
 - D. `<% import java.util.Date; %>`
 - E. `<%@ import file="java.util.Date" %>`
-

16 Dado o JSP:

```

1. <%@ page isELIgnored="true" %>
2. <%@ taglib uri="http://java.sun.com/jsp/jstl/core"
prefix="c" %>
3. <c:set var="awesomeBand" value="LIMOZEEN"/>
4. ${awesomeBand}

```

Qual será a saída?

- A. `${awesomeBand}`
- B. `LIMOZEEN`
- C. Nenhuma saída
- D. Uma exceção será enviada porque todas as diretivas taglib devem preceder qualquer diretiva page.



Pausa para o café

Respostas – Capítulo 7

1 Dado o elemento DD:

(JSPv2.0, pág. 7-87)

47. <jsp-property-group>
48. <url-pattern>*.jsp</url-pattern>
49. <el-ignored>true</el-ignored>
50. </jsp-property-group>

- A opção C cancela o cálculo das expressões EL pelo container JSP 2.0. Por padrão, o container calcula a EL.

O que o elemento faz? (Escolha todas as que se aplicam.)

- A. Todos os arquivos com o mapeamento da extensão especificada devem ser tratados pelo container JSP como arquivos que respeitam a sintaxe XML.
- B. Todos os arquivos com o mapeamento da extensão especificada devem ter seus códigos EL avaliados pelo container JSP.
- C. Por padrão, NENHUM arquivo com o mapeamento da extensão especificada deve ter seus códigos Expression Language avaliados pelo container JSP.
- D. Nada. Esta tag NÃO é reconhecida pelo container.
- E. Embora esta tag seja válida, ela é redundante, pois o container já faz isso por padrão.

2 Quais das diretivas abaixo representam uma resposta HTTP do tipo “image/svg”? (Escolha todas as que se aplicam.)

(JSPv2.0, seção 7.10.7)

- A. <%@ page type="image/svg" %>
- B. <%@ page mimeType="image/svg" %>
- C. <%@ page language="image/svg" %>
- D. <%@ page contentType="image/svg" %> - A opção D apresenta a sintaxe correta
- E. <%@ page pageEncoding="image/svg" %> para esta diretiva.

3 Seja o JSP:

(JSPv2.0, seção 7)

```

1. <%@ page import="java.util.*" %>
2. <html><body> The people who like
3. <%= request.getParameter("hobby") %>
4. are: <br>
5. <% ArrayList al = (ArrayList) request.
getattribute("names"); %>
6. <% Iterator it = al.iterator();
7. while (it.hasNext()) { %>
8.     <%= it.next() %>
9. <br>
10. <% } %>
11. </body></html>
```

Que tipos de códigos encontramos nele? (Escolha todas as que se aplicam.)

- A. EL
- B. diretiva
- C. expressão
- D. template text
- E. scriptlet

— Não existe EL neste JSP. Temos uma diretiva na linha 7, expressões nas linhas 3 e 8, template text por todo lado (como na linha 2) e, lógico, elementos scripting.

4 Que declarações sobre o `jspInit()` são verdadeiras? (Escolha todas as que se aplicam.)

(JSPv2.0, seção 11.2.7)

- A. Ele tem acesso ao `ServletConfig`.
- B. Ele tem acesso ao `ServletContext`
- C. Só é chamado uma vez.
- D. Pode ser anulado.

5 Que tipos de objetos estão disponíveis para o método `jspInit()`?

(Escolha todas as que se aplicam.)

- A. `ServletConfig`
- B. `ServletContext`
- C. `JspServletConfig`
- D. `JspServletContext`
- E. `HttpServletRequest`
- F. `HttpServletResponse`

(JSPv2.0, seção 7.2.7)

– Os JSPs viram servlets simples e por isso têm acesso aos objetos simples `ServletConfig` e `ServletContext`... e é um pouco cedo no ciclo de vida para falarmos sobre solicitações e respostas

6 Dado:

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

(JSPv2.0, pág. 7-49)

O que acontece? (Escolha todas as que se aplicam.)

- A. Nada. A diretiva `page` NÃO foi definida.
- B. A diretiva anula a avaliação que o container JSP faz do código da Expression Language em todos os JSPs da aplicação.
- C. O JSP que possui esta diretiva será tratado pelo container JSP como um arquivo que respeita a sintaxe XML.
- D. O JSP que possui esta diretiva NÃO deveria possuir nenhum código da Expression Language avaliado pelo container JSP.
- E. Esta diretiva apenas cancelará a avaliação da EL se o DD declarar um elemento `<el-ignored>true</el-ignored>` com um padrão URL que inclui este JSP.

– A opção B está incorreta, porque a diretiva afeta apenas o JSP que foi incluído.

7 Qual declaração referente aos JSPs é verdadeira? (Escolha uma.)

- A. Apenas o `jspInit()` pode ser anulado.
- B. Apenas o `jspDestroy()` pode ser anulado.
- C. Apenas o `_jspService()` pode ser anulado.
- D. O `jspInit()` e o `jspDestroy()` podem ser anulados.
- E. O `jspInit()`, o `jspDestroy()` e o `_jspService()` podem ser anulados.

(JSPv2.0, seção 7.7)

– Lembre-se de que o underscore é a dica de que um método não pode ser anulado.

8 Qual das etapas do ciclo de vida do JSP está fora de ordem?

- A. Traduzir o JSP em servlet.
- B. Compilar o código-fonte do servlet.
- C. Call `_jspService()`
- D. Instar a classe servlet.
- E. Call `jspInit()`
- F. Call `jspDestroy()`

(JSPv2.0, seção 7.7)

- O método `jspService` nunca pode ser chamado antes do `jspInit`.

9 Quais das variáveis implícitas JSP são válidas? (Escolha todas as que se aplicam.)

- A. `stream`
- B. `context`
- C. `exception`
- D. `listener`
- E. `application`

(JSPv2.0, seção 7.8.3)

- As opções A, B e D não existem como objetos implícitos criados pelo container para os JSPs.

10 Seja uma solicitação com dois parâmetros: um chamado “first”, que representa o primeiro nome do usuário, e o outro chamado “last”, que representa seu último nome.

Qual código scriptlet JSP gera os valores para estes parâmetros?

(JSPv2.0, pág. 7-47)

- A. `<% out.println(request.getParameter("first"));
out.println(request.getParameter("last")); %>`
- B. `<% out.println(application.getInitParameter("first"));
out.println(application.getInitParameter("last")); %>`
- C. `<% println(request.getParameter("first"));
println(request.getParameter("last")); %>`
- D. `<% println(application.getInitParameter("first"));
println(application.getInitParameter("last")); %>`

, A opção A usa o método `out` com o respectivo método `println()`.

- As opções C e D, não envolvem o objeto implícito `out`.

11 Dado:

JSPv2.0, pág. 7-70

```
11. Hello ${user.name}!
12. Your number is <c:out value="${user.phone}" />.
13. Your address is <jsp:getProperty name="user" property="addr" />
14. <% if (user.isValid()) {>You are valid!<% } %>
```

Quais declarações são verdadeiras? (Escolha todas as que se aplicam.)

- A. As linhas 11 e 12 (e nenhuma outra) contêm exemplos de elementos EL.
- B. A linha 14 é um exemplo de código scriptlet.
- C. Nenhuma das linhas deste exemplo contém um template text.
- D. As linhas 12 e 13 incluem exemplos de ações-padrão JSP.
- E. A linha 11 demonstra o uso incorreto da EL.
- F. Todas as quatro linhas deste exemplo seriam válidas em uma página JSP.

– A opção C está incorreta, porque todas as quatro linhas possuem template text.
– A opção D está incorreta, porque a linha 12 não inclui uma ação-padrão JSP.
– A opção E está incorreta, porque a EL na linha 11 é válida.

12 Qual tag JSP exibirá o parâmetro de inicialização de contexto chamado “javax.sql.DataSource”?

JSPv2.0, pág. 7-47

- A. <%= application.getAttribute("javax.sql.DataSource") %>
- B. <%= application.getInitParameter("javax.sql.DataSource") %>
- C. <%= request.getParameter("javax.sql.DataSource") %>
- D. <%= contextParam.get("javax.sql.DataSource") %>

– A opção B mostra o uso correto do objeto implícito application.

13 Quais declarações sobre desabilitar elementos scripting são verdadeiras? (Escolha todas as que se aplicam.)

JSPv2.0, seção 3.3.3

- A. Você não pode desabilitar scripting via DD.
- B. Você só pode desabilitar scripting no nível da aplicação.
- C. Você pode desabilitar scripting programaticamente, utilizando o atributo de diretiva de página `isScriptingEnabled`.
- D. Você pode desabilitar scripting via DD usando o elemento

`<scripting-invalid>`.

– Você só pode desabilitar elementos scripting pela DD. O elemento `<JSP-property-group>` possibilita que desabilitemos scripting em alguns JSPs, definindo padrões de URLs para serem desabilitadas.

14 Em seqüência, quais são os tipos Java para os seguintes objetos implícitos JSP: `application, out, request, response, session?`

- A. `java.lang.Throwable`
`java.lang.Object`
`java.util.Map`
`java.util.Set`
`java.util.List`
- B. `javax.servlet.ServletConfig`
`java.lang.Throwable`
`java.lang.Object`
`javax.servlet.jsp.PageContext`
`java.util.Map`
- C. `javax.servlet.ServletContext`
`javax.servlet.jsp.JspWriter`
`javax.servlet.ServletRequest`
`javax.servlet.ServletResponse`
`javax.servlet.http.HttpSession`
- D. `javax.servlet.ServletContext`
`java.io.PrintWriter`
`javax.servlet.ServletConfig`
`java.lang.Exception`
`javax.servlet.RequestDispatcher`

(JSPv2.0, pág. 7-47)

- A opção C mostra o tipo de cada objeto implícito.

15 Qual das opções representa um exemplo da sintaxe usada para importar uma classe em um JSP?

(JSPv2.0, pág. 7-44)

- A. `<% page import="java.util.Date" %>`
- B. `<%@ page import="java.util.Date" @%`
- C. `<%@ page import="java.util.Date" %>`
- D. `<% import java.util.Date; %>`
- E. `<%@ import file="java.util.Date" %>`

- As opções A e D são inválidas, porque apenas as declarações Java podem vir entre as tags `<% ... %>`.
- A opção C é o único exemplo que mostra a sintaxe correta.
- A opção E é inválida, porque não existe diretiva import.

16 Dado o JSP:

```
1. <%@ page isELIgnored="true" %>
2. <%@ taglib uri="http://java.sun.com/jsp/jstl/core"
prefix="c" %>
3. <c:set var="awesomeBand" value="LIMOZEEN"/>
4. ${awesomeBand}
```

(JSPv2.0, seção 1.10.1)

Qual será a saída?

- Opção A: a expressão EL é ignorada e passou textualmente.

- A. `${awesomeBand}`
- B. LIMOZEEN
- C. Nenhuma saída
- D. Uma exceção será enviada porque todas as diretivas taglib devem preceder qualquer diretiva page.