Módulo 6

Programação WEB

Lição 4

Páginas JSP Básicas

Versão 1.0 - Nov/2007

Autor
Daniel Villafuerte

Equipe
Rommel Feria
John Paul Petines

Necessidades para os Exercícios

Sistemas Operacionais Suportados

NetBeans IDE 5.5 para os seguintes sistemas operacionais:

- Microsoft Windows XP Profissional SP2 ou superior
- Mac OS X 10.4.5 ou superior
- Red Hat Fedora Core 3
- Solaris™ 10 Operating System (SPARC® e x86/x64 Platform Edition)

NetBeans Enterprise Pack, poderá ser executado nas seguintes plataformas:

- Microsoft Windows 2000 Profissional SP4
- Solaris™ 8 OS (SPARC e x86/x64 Platform Edition) e Solaris 9 OS (SPARC e x86/x64 Platform Edition)
- Várias outras distribuições Linux

Configuração Mínima de Hardware

Nota: IDE NetBeans com resolução de tela em 1024x768 pixel

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	500 MHz Intel Pentium III workstation ou equivalente	512 MB	850 MB
Linux	500 MHz Intel Pentium III workstation ou equivalente	512 MB	450 MB
Solaris OS (SPARC)	UltraSPARC II 450 MHz	512 MB	450 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Série 1.8 GHz	512 MB	450 MB
Mac OS X	PowerPC G4	512 MB	450 MB

Configuração Recomendada de Hardware

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	1 GB
Linux	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	850 MB
Solaris OS (SPARC)	UltraSPARC IIIi 1 GHz	1 GB	850 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Series 1.8 GHz	1 GB	850 MB
Mac OS X	PowerPC G5	1 GB	850 MB

Requerimentos de Software

NetBeans Enterprise Pack 5.5 executando sobre Java 2 Platform Standard Edition Development Kit 5.0 ou superior (JDK 5.0, versão 1.5.0_01 ou superior), contemplando a Java Runtime Environment, ferramentas de desenvolvimento para compilar, depurar, e executar aplicações escritas em linguagem Java. Sun Java System Application Server Platform Edition 9.

- Para **Solaris**, **Windows**, e **Linux**, os arquivos da JDK podem ser obtidos para sua plataforma em http://java.sun.com/j2se/1.5.0/download.html
- Para **Mac OS X**, Java 2 Plataform Standard Edition (J2SE) 5.0 Release 4, pode ser obtida diretamente da Apple's Developer Connection, no endereço: http://developer.apple.com/java (é necessário registrar o download da JDK).

Para mais informações: http://www.netbeans.org/community/releases/55/relnotes.html

Colaboradores que auxiliaram no processo de tradução e revisão

Aécio Júnior
Alexandre Mori
Alexis da Rocha Silva
Allan Souza Nunes
Allan Wojcik da Silva
Angelo de Oliveira
Aurélio Soares Neto
Bruno da Silva Bonfim
Carlos Fernando Gonçalves
Denis Mitsuo Nakasaki
Emanoel Tadeu da Silva Freitas
Felipe Gaúcho
Jacqueline Susann Barbosa
João Vianney Barrozo Costa
Luciana Rocha de Oliveira
Luiz Fernandes de Oliveira Junior
Marco Aurélio Martins Bessa
Maria Carolina Ferreira da Silva
Massimiliano Giroldi
Mauro Cardoso Mortoni
Paulo Oliveira Sampaio Reis
Pedro Henrique Pereira de Andrade
Ronie Dotzlaw
Sergio Terzella
Thiago Magela Rodrigues Dias
Vanessa dos Santos Almeida
Wagner Eliezer Roncoletta

Auxiliadores especiais

Revisão Geral do texto para os seguintes Países:

- **Brasil** – Tiago Flach
- **Guiné Bissau** – Alfredo Cá, Bunene Sisse e Buon Olossato Quebi – ONG Asas de Socorro

Coordenação do DFJUG

- **Daniel deOliveira** – JUGLeader responsável pelos acordos de parcerias
- **Luci Campos -** Idealizadora do DFJUG responsável pelo apoio social
- **Fernando Anselmo -** Coordenador responsável pelo processo de tradução e revisão, disponibilização dos materiais e inserção de novos módulos
- **Rodrigo Nunes -** Coordenador responsável pela parte multimídia
- **Sérgio Gomes Veloso -** Coordenador responsável pelo ambiente JEDI™ (Moodle)

Agradecimento Especial

John Paul Petines – Criador da Iniciativa JEDI™
Rommel Feria – Criador da Iniciativa JEDI™

1. Objetivos

Nas lições anteriores, aprendemos como produzir conteúdo dinâmico para nossos usuários usando a Tecnologia Java por meio do uso de classes *servlets*. Entretanto, enquanto os desenvolvedores Java podem criar sites com conteúdo dinâmico usando *servlet*, existem desvantagens em fazê-lo.

Primeiro, utilizar *servlet* somente para produzir conteúdo HTML não torna o código limpo e de fácil manutenção. Strings a serem usadas para saída em HTML podem se tornar muito grandes – medindo várias linhas – e podem facilmente tornar-se um pesadelo de manutenção.

Segundo, usar *servlet* somente para produzir conteúdo HTML assume que o desenvolvedor está familiarizado com Java e HTML. Especialmente no caso de grandes organizações, projetistas de site são grupos separados de programadores. Ter que treinar os projetistas que possuam um entendimento básico de Java, ou para desenvolvedores Java obterem habilidade em projeto de site em HTML, pode consumir tempo e recursos caros para uma organização.

É aqui que a tecnologia JSP entra.

Ao final desta lição, o estudante será capaz de:

- Ter uma visão geral sobre páginas JSP
- Conhecer o ciclo de vida de páginas JSP
- Compreender a sintaxe e a semântica das páginas JSP

2. Visão Geral

2.1. O que é JSP?

JavaServer Pages (JSP) é uma tecnologia baseada em servlet usada na camada WEB para apresentar conteúdo dinâmico e estático. Ela é baseada em texto e contém, em sua maioria, modelo de texto em HTML misturado com *tags* que especificam conteúdo dinâmico.

2.2. Porque JSP?

- Considerando que JSP são documentos de texto como HTML, desenvolvedores evitam ter que formatar e manipular uma String longa para produzir saída. O conteúdo HTML não estará embutido dentro de código em Java. Isto torna mais fácil sua manutenção.
- JSP são familiares para qualquer um com conhecimento de HTML, porque possuem somente marcação dinâmica, isto é, tags. Isto torna possível para projetistas de site criar o modelo HTML do site restando aos desenvolvedores processando-o posteriormente a inclusão das *tags* para produzir o conteúdo dinâmico. Isto torna fácil o desenvolvimento de páginas WEB.
- JSP têm suporte interno para o uso de componentes de software reusáveis (JavaBeans). Estes não somente permitem que os desenvolvedores evitem "reinventar a roda" para cada aplicação. Ter suporte para componentes de software separados para manipular lógica promove separação de apresentação e lógica de negócios também.
- JSP, como parte da solução Java para desenvolvimento de aplicação WEB, são inerentemente multiplataforma e podem ser executadas em qualquer contêiner compatível, independente do distribuidor ou sistema operacional.
- Devido ao modo como JSP funciona, não necessita de compilação explícita pelo desenvolvedor. Esta compilação é feita pelo contêiner. Modificações na JSP é também detectadas automaticamente e resultam em uma nova compilação. Isto torna as páginas JSP relativamente simples de desenvolver.

2.3. JSP Exemplo

```
<HTML>
 <TITLE>Welcome</TITLE>
 <BODY>
  <H1>Greetings!</H1><br>
  Thank you for accessing our site. <br>
  The time is now <%= new java.util.Date()%>
 </BODY>
</HTML>
```

Este é um simples arquivo em JSP que apresenta uma saudação genérica para os usuários do site, bem como informa a data e hora corrente de acesso. A **figura 1** corresponde à saída do arquivo JSP anterior.

Figura 1: Saída da página welcome.jsp

Podemos ver que o arquivo JSP é em sua maioria de natureza HTML. A única parte representativa é este pedaço de instrução: <%=new java.util.Date()%>, responsável por exibir a data e hora corrente. Realizando este trabalho criando uma nova instância do objeto *Date* e exibindo sua String correspondente.

2.4. Executando o JSP Exemplo

2.4.1. Usando o IDE

Uma JSP pode executar sobre qualquer projeto WEB criado pela IDE. Assumindo a existência de um projeto, poderemos simplesmente adicionar um arquivo do tipo JSP na pasta *Web Pages*.

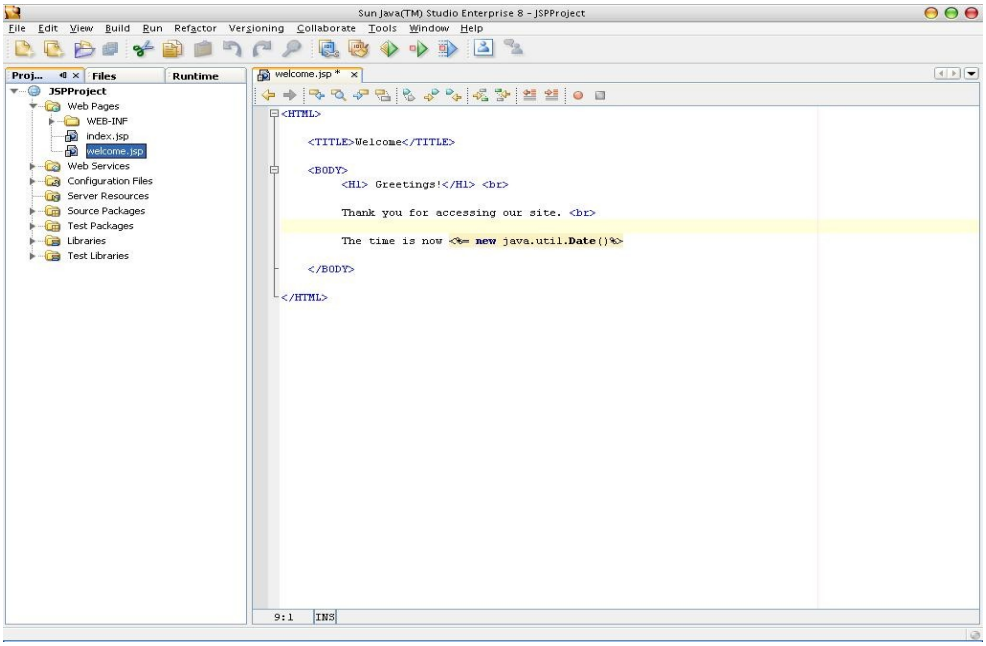


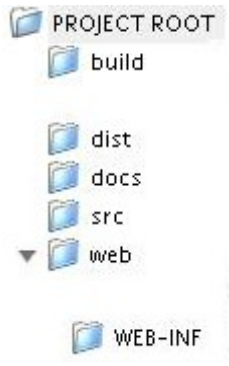
Figura 2: Página JSP para um projeto WEB

Isto especifica que a página JSP pode então ser diretamente executado a partir do IDE pressionado-se Shift+F6. Alternativamente, o projeto WEB pode ser empacotado como um arquivo WAR e transferido para o servidor. A página JSP pode então ser acessada digitando-se a seguinte URL:

```
http://[host]:[port]/[WEB_PROJECT_NAME]/[JSP_NAME]
```

2.4.2. Usando um arquivo de construção Ant

A página JSP pode também ser executada pelo empacotamento em um arquivo WAR usando uma ferramenta de construção (tal como esboçado na lição 2 – Classes *Servlet*), e então colocando o arquivo WAR em um servidor WEB. A estrutura do diretório é replicada abaixo para lembrete:



build	Armazenará a aplicação após executar o script de construção em uma estrutura de diretório reconhecido pelo contêiner.
dist	Depois de executar o script construído, conterá o arquivo WAR.
docs	Usado para armazenar a documentação que é utilizada pelo time de desenvolvimento.
src	Diretório dentro do qual todos os arquivos fonte Java devem ser colocados.
web	Diretório contendo todo conteúdo static da app (HTML, JSP), será a base da raiz do documento de seu projeto.
WEB-INF	Diretório contendo arquivos de configuração como o descritor de desenvolvimento (web.xml), descritores de biblioteca de tags, etc.

Figura 3: *Estrutura de diretório recomendado para desenvolvimento de aplicação web.*

3. Ciclo de vida JSP

O *WEB Server* gerencia as páginas JSP de modo similar como as classes *Servlets*. Isto é feito através de um ciclo de vida bem definido.

JSP têm ciclos de vida de três fases: inicialização, serviço e destruição. Estes eventos de ciclo de vida são similares aos dos *servlets*, embora os métodos invocados pelo *WEB Server* sejam diferentes:

- jspInit() para a fase de inicialização
- _jspService() para a fase de serviço
- jspDestroy() para a fase de destruição

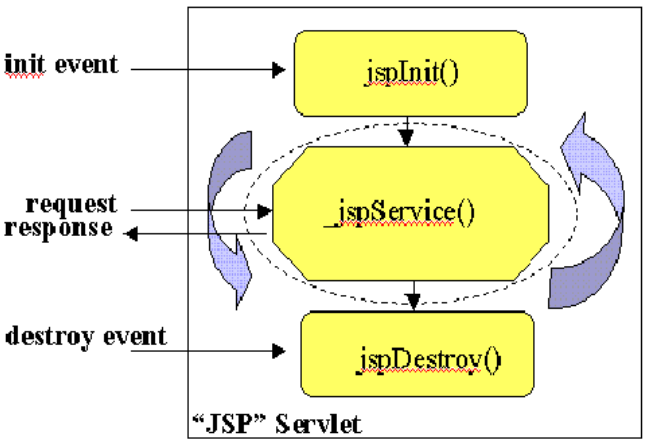


Figura 4: *Ciclo de vida de 3 fases da JSP*

Dado o JSP exemplo acima, parece confuso falar a respeito dos métodos jspInit ou _jspService(). A página JSP exemplo é apenas uma simples página de texto com a maior parte de conteúdo HTML. Não possui quaisquer métodos. Páginas JSP criam uma classe *Servlet* que são compiladas pelo *WEB Server*. Esta classe *Servlet* manipula todas as requisições para a página JSP. Esta tradução em *Servlet* e subseqüente compilação é feita de modo transparente pelo *WEB Server*. Não é necessário o desenvolvedor preocupar-se com o modo de como este procedimento é realizado.

Para ver os arquivos da classe *Servlet* gerados, o *Sun Application Server 8.1* colocam estes arquivos no seguinte diretório:

```
${SERVER_HOME}/domains/domain1/generated/jsp/j2ee-modules/
${WEB_APP_NAME}/org/apache/jsp
```

Abaixo está a classe *Servlet* equivalente ao nosso exemplo.

```
package org.apache.jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

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

  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");
        response.addHeader("X-Powered-By", "JSP/2.0");
        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("<HTML>\n");
        out.write("\n");
        out.write(" <TITLE>Welcome</TITLE>\n");
        out.write("\n");
        out.write(" <BODY>\n");
        out.write(" <H1> Greetings!</H1> <br>\n");
        out.write("\n");
        out.write(" Thank you for accessing our site. <br>\n");
        out.write("\n");
        out.write(" The time is now ");
        out.print( new java.util.Date());
        out.write("\n");
        out.write("\n");
        out.write(" </BODY>\n");
        out.write("\n");
        out.write("</HTML>");
      } 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);
      }
    }
  }
```

Neste momento não é importante entender completamente o funcionamento classe acima. É importante conhecer como as páginas JSP são manipuladas da mesma forma que as classes *Servlets*, mesmo que isto não seja imediatamente óbvio. Outro ponto é que as páginas JSP são transformadas classes *Servlets*. Somente diferem no modo como um desenvolvedor produz o conteúdo. Páginas JSP são orientadas a texto, enquanto que as classes *Servlets* permite ao desenvolvedor aplicar fortes conhecimentos de Java.

4. Sintaxe JSP e Semântica

Embora páginas JSP sejam baseadas em linguagem Java, sendo manipuladas como uma classe Java pela *Servlet.* A sintaxe permite aos desenvolvedores os usar de forma diferente daquela da especificação 2.0 de Java. Ao invés disso, segue regras definidas na especificação JSP. A seção seguinte descreve a sintaxe JSP em mais detalhes.

4.1. Elementos e Modelo de Dados

Componentes de todas *JavaServer Pages* podem ser qualificados em duas categorias gerais: elementos e modelo de dados. Elementos são informação produzida dinamicamente. Modelo de Dados são informação estática que são responsáveis pela apresentação. Na página **hello.jsp**, a expressão JSP <%=new java.util.Date()%> é o único elemento de dados e o resto são dados modelo de dados.

```
<html>
 <head>
  <title>Hello World!</title>
 </head>
 <body>
  <center>
   <h1>Hello World! It's <%= new java.util.Date()%></h1>
  </center>
 </body>
</html>
```

4.1.1. Dois Tipos de Sintaxe

Dois estilos de criação de páginas JSP são suportados pelos *WEB Server*: o estilo JSP e o estilo XML. Ambos estão presentes nesse texto. Escolher um estilo ao invés da outro é apenas uma questão de preferência e padronização. A estilo JSP foi desenvolvido para ser de fácil criação. O estilo XML é simplesmente o estilo JSP modificado para ser compatível com a sintaxe XML. O estilo XML é preferido quando utilizamos uma ferramenta de criação de JSP. No entanto, a maioria prefere o estilo JSP por ser mais fácil de ler e entender. Esse texto irá utilizar o estilo JSP nos exemplos.

4.2. Elementos de Scripting

Como mencionado na lição anterior, páginas JSP devem ser vistas como documentos HTML ou XML com *scripts* JSP embutido. Elementos de *scripting* JSP permitem inserir instruções Java na classe *Servlet* gerada a partir da página JSP.

A forma mais simples de se criar uma JSP dinâmica é inserir diretamente elementos de *scripting* no modelo de dados.

Nessa lição iremos aprender os seguintes elementos de scripting JSP:

1. *Scriptlets*,
2. Expressões, e
3. Declarações.

4.2.1. Scriptlets

Proporcionam uma maneira de inserir diretamente pedaços de instruções Java em diferentes partes da modelo de dados e tem a seguinte forma:

```
<% Instruções Java; %>
```

Definir instruções Java entre <% e %> é o mesmo que estar codificando dentro de um método. *Scriptlets* são úteis para embutir instruções Java como comandos condicionais, de repetição, entre outros. Não existe um limite específico sobre a complexidade das instruções Java que podem ser

inseridas em *scriptlets*. No entanto, devemos ter grande precaução a utilização exagerada de *scriptlets*. Colocar processos computacionais pesados dentro de códigos *scriptlets* gera um problema para manutenção. Além disso, o excesso de utilização de *scriptlets* viola a regra de JSP de ser prioritariamente um componente da camada de apresentação.

Discutiremos mais adiante como poderemos utilizar *JavaBeans* para encapsular o resultado de dados passados por outro componente, reduzindo drasticamente a quantidade de *scriptlets* necessários em uma página. Mais adiante, iremos discutir como utilizar *tags* customizadas para executar tarefas comuns como decisão lógica, repetição, entre outras. Isso, combinado com a utilização dos *JavaBeans*, desta forma iremos diminuir a necessidade de *scriptlet*.

Se quisermos utilizar os caracteres "**%>**" dentro de um *scriptlet*, escrevemos "**%\>**" . Isso evitará que o compilador interprete os caracteres como *tag* de fechamento de *scriptlet*.

Abaixo temos dois exemplos que definem códigos Java dentro de *tags* HTML.

Método println

```
1  <html>
2  <head>
3  <title>Scriptlet Example 1</title>
4  </head>
5  <body>
6  <%
7  String username="jedi";
8  out.println ( username ) ;
9  %>
10 </body>
11 </html>
```

Figura 5: PrintlnScriplet.jsp

O exemplo mostra a implementação de instruções Java embutidas em uma página JSP. Esta página escreve o texto contido no atributo *username* no navegador WEB.

Comando *for* em *scriptlet*

```
1  <html>
2  <head>
3  <title>Scriptlet Example 2</title>
4  </head>
5  <body>
6  <% for (int i=0; i < 10; i++) { %>
7  This line is printed 10 times.
8  <br/>
9  <% } %>
10 </body>
11 </html>
```

Figura 6: LoopScriplet.jsp

Este outro exemplo mostra a implementação do código Java de um laço **for** inserido dentro das *tags* de *scriptlet* (<%...%>). A saída que teremos no navegador após a execução dessa JSP deve ser o texto "This line is printed 10 times." mostrado 10 vezes devido a iteração do comando **for** de 0 até 9 conforme mostrado na figura a seguir:

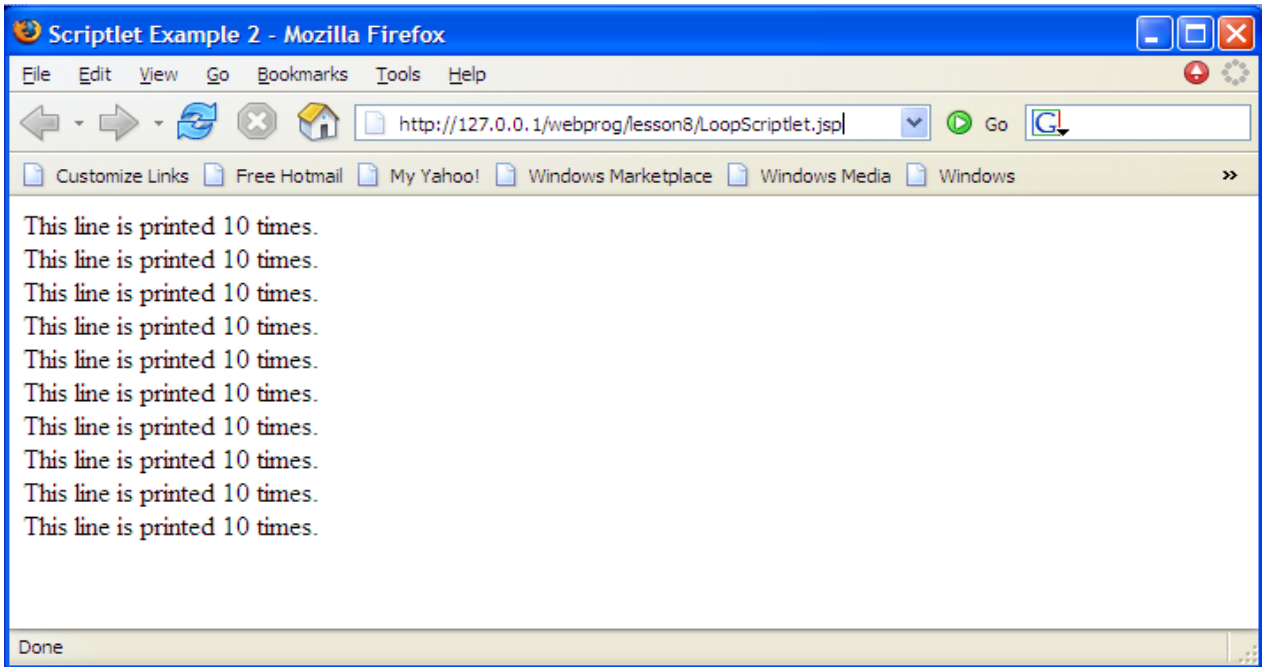


Figura 7: Saída do LoopScriplet.jsp

Note que o *scriptlet* não é enviado para o cliente, apenas sua saída. Tente ver o código da saída do JSP que foi produzida no *browser* para entender esse ponto. Tudo o que vemos são *tags HTML* mais a saída do *scriptlet* mas não vemos o *scriptlet*.

O estilo XML para escrever *scriptlets* é:

```
<jsp:declaration>
 Código Java;
</jsp:declaration>
```

4.2.2. Expressões

Expressões fornecem uma maneira de inserir valores Java diretamente na saída. Elas tem a seguinte forma:

```
<%= Expressão Java %>
```

Expressões nada mais são do que uma abreviação para *out.println()*.

Note que o ponto-e-vírgula (;) **não aparece** no final da instrução dentro da *tag*.

Qualquer expressão Java colocada entre <%= e %> é avaliada em tempo de execução, convertida para String e inserida na página. Uma expressão sempre envia uma *String* de texto para o cliente, mas o objeto produzido como resultado da expressão não necessariamente será uma instância do objeto String. Todas as instâncias de objetos não-String são convertidos para strings através de seus métodos membros herdados *toString().* Se o resultado for primitivo, então uma representação String do valor primitivo será mostrada.

Como mencionado anteriormente, avaliações são efetuadas em tempo de execução (quando a página é requisitada). Isso dá às expressões acesso total às informações sobre a requisição (request).

Diversos objetos predefinidos estão disponíveis para os desenvolvedores de JSP para simplificar expressões. Esses objetos são chamados de **objetos implícitos** e serão discutidos em maiores detalhes posteriormente. Para o propósito das expressões, as mais importantes são:

- requisição (request), o HttpServletRequest;
- resposta (response), o HttpServletResponse;
- sessão (session), o HttpSession associado com à requisição (se houver); e

- saída (out), o PrintWriter (uma versão armazenada do tipo JspWriter) utilizado para enviar dados de saída para o cliente.

Por exemplo, para imprimir o nome da máquina (hostname), só precisamos incluir essa simples expressão JSP abaixo:

Nome da Máquina: <%= request.getRemoteHost() %>

O estilo XML para a tag **<%=** Expressão Java **%>** é:

```
<jsp:expression>
 Expressão Java
</jsp:expression>
```

4.2.3. Declarações

Declaração permitem definir métodos ou variáveis. Elas possuem o seguinte formato:

```
<%! Código Java %>
```

Declarações são usadas para embutir código de modo semelhante aos *scriptlets*. No entanto, declarações são inseridas na parte principal (*main body*) da classe *Servlet*, fora do método _jspService() que processa a requisição. Por essa razão, os códigos embutidos em declarações podem ser usados para declarar métodos ou variáveis globais. Por outro lado, código das declarações NÃO são protegidos, a não ser que seja explicitamente programado pelo desenvolvedor da JSP, logo devemos tomar cuidado ao escrever declarações.

A seguir temos alguns lembretes simples sobre a utilização da utilização da *tag* de declaração:

- Antes da declaração deve ter **<%!** e ao final da declaração **%>**
- As instruções devem seguir a sintaxe Java padrão
- Declarações não geram saída mas usadas com expressões JSP ou *scriptlets*

Como declarações não geram qualquer saída, elas são normalmente usadas em conjunto com expressões JSP ou *scriptlets*. Por exemplo, aqui temos uma JSP que imprime o número de vezes que a página corrente foi requisitada desde que o servidor foi iniciado (ou a classe servlet foi mudada ou recarregada):

Exemplo

```
1  <%! private int accessCount = 0; %>
2
3  <html>
4  <head>
5  <title>Declaration Example</title>
6  </head>
7  <body>
8  <% accessCount++;%>
9  Accesses to page since server reboot:
10 <%= accessCount %>
11 </body>
12 </html>
```

Figura 8: AccessCountDeclaration.jsp

A JSP é capaz de mostrar o número de visitas através da declaração de uma atributo com escopo na classe *accessCount*, utilizando um *scriptlet* para incrementar o valor do número de vezes que a página foi visitada e uma expressão para mostrar o valor.

A ilustração abaixo mostra um exemplo de saída dessa JSP carregada quatro vezes.

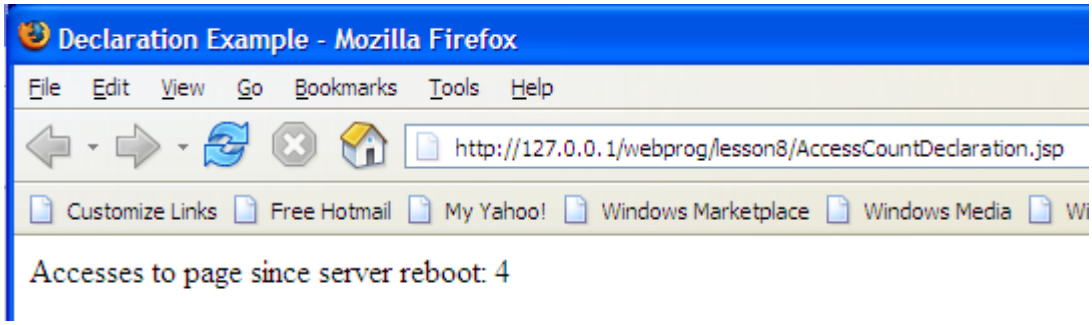


Figura 9: Saída da AccessCountDeclaration.jsp

Para entendermos melhor como uma página JSP é transformada em uma servlet, vamos examinar a saída da *servlet* do contêiner para a página AccessCountDeclaration.jsp. O contêiner gerou um arquivo Java chamado AccessCountDeclaration_jsp.java, a seguir veremos o conteúdo desse arquivo. Observe que a declaração, o *scriptlet* e a expressão foram destacadas para facilitar a referência.

AccessCountDeclaration_jsp.java

```
package org.apache.jsp.JSP;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

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

  private int accessCount = 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("\n");
      out.write("\n");
      out.write("<html>\n");
      out.write("<head>\n");
```

```
        out.write("<title>Declaration Example</title>\n");
        out.write("</head>\n");
        out.write("<body>\n");
        accessCount++;
        out.write("Accesses to page since server reboot: \n");
        out.print( accessCount );
        out.write("\n");
        out.write("</body>\n");
        out.write("</html>\n");
        out.write("\n");
        out.write("\n");
      } 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);
      }
    }
  }
```

Note como a declaração para accessCount é colocada fora do método _jspservice() como um atributo. Isso torna accessCount disponível para qualquer outro método definido na JSP, e não apenas para o método _jspservice(). O código anterior nos mostrou a colocação das declarações, *scriptlets* e expressões no código fonte Java traduzido da página JSP.

O estilo XML para a tag `<%! Instruções Java %>` é

```
<jsp:declaration>
Código Java;
</jsp:declaration>
```

4.2.4. Texto Template

- Utilize <\% para ter <% na saída.
- <%-- Comentário JSP --%>
- <!-- Comentário HTML -->
- Todos os outro textos não específicos a JSP são passados para a saída da página.

4.3. Objetos Predefinidos

Na discussão da *tag* de expressões, abordamos objetos JSP implícitos. Essa sessão descreve esses objetos em detalhe.

Objetos JSP implícitos são automaticamente declarados pelo contêiner e estão sempre disponíveis para utilização pelas expressões e *scriptlets* (mas não em declarações). A seguir temos a lista de objetos implícitos:

request: Instância da classe javax.servlet.http.HttpServletRequest, este objeto é associado com a requisição do cliente.

response: Instância da classe *javax.servlet.http.HttpServletResponse*, este objeto é associado com a resposta para o cliente.

pageContext: Objeto associado com a página corrente.

out: Instância da classe *javax.servlet.jsp.JspWriter*, este objeto pode ser usado para escrever ações e modelo dos dados em páginas JSP, parecido com objetos da classe *PrintWriter* que utilizamos na discussão da *Servlet*. O objeto *out* é inicializado automaticamente utilizando métodos de objetos da classe *PageContext*.

session: Instância da classe javax.servlet.http.HttpSession. É equivalente a chamarmos o método HttpServletRequest.getSession().

application: Instância da classe *javax.servlet.ServletContext*. É equivalente a chamarmo o método *getServletConfig*().*getContext*(). Esse objeto implícito é compartilhado por todos as *Servlets* e páginas JSP no servidor.

config: Instância da classe *javax.servlet.ServletConfig* para essa página. Igual as *Servlets*, JSP tem acesso aos parâmetros inicialmente definidos no *Deployment Descriptor* do servidor de aplicação.

4.4. Diretivas JSP

Diretivas são mensagens para o contêiner. Afetam toda estrutura da classe servlet. Geralmente tem a seguinte forma:

```
<%@ atributo da diretiva="valor" %>
```

Uma lista de configuração de atributos pode também ser enumerada para uma simples diretiva como segue:

```
<%@ atributo1 da diretiva="valor1"
              atributo2="valor2"
              ...
              atributoN="valorN" %>
```

Nota: Espaços em branco após <%@ e depois %> são opcionais.

Diretivas NÃO produzem qualquer saída visível quando a página é requisitada mas elas mudam a forma que o contêiner processa a página. Por exemplo, podemos ocultar dados da sessão para uma página configurando uma diretiva (*session*) para falso.

Uma diretiva JSP dá informações especiais sobre a página para o contêiner. A diretiva pode ser page, include ou taglib e cada uma dessas diretivas tem seus próprios conjuntos de atributos.

4.4.1. Diretiva Page

A diretiva *page* define o processamento de informação da página. Permite importar classes, preparar super-classes *Servlets* e coisas assim.

As diretivas possuem atributos opcionais que provê ao mecanismo JSP uma forma especial de processar a informação como se segue; Nota: as letras maiúsculas e minúsculas são importantes nos nomes dos atributos:

Diretiva	***Descrição***	***Exemplo de uso***
extends	Super-classe usada pelo mecanismo JSP para traduzir o servlet. Analogamente às extensões das palavras-chave da linguagem de programação em Java.	`<%@ page extends = "com.taglib…"%>`
language	Indica qual a linguagem foi usada nos *scriptlets*, expressões e declarações encontradas nas páginas JSP. O único valor definido para este atributo é ***java***.	`<%@ page language="java"%>`
import	Importa pacotes de classes java para a página JSP corrente.	`<%@ page import="java.util.*"%>`
session	Indica se a página faz uso de sessões. Por padrão toda JSP possui dados da *session* disponíveis. Alterando *session* para *false* implica em benefícios de performance.	`O valor padrão é true. (verdadeiro)`
buffer	Controla o uso dos *buffers* de saída da página JSP. Se o valor for "none" então não existirão *buffers* e a saída será escrita diretamente no PrintWriter apropriado. Por padrão o valor é de 8kb.	`<%@ page buffer="none"%>`

Diretiva	*Descrição*	*Exemplo de uso*
autoFlush	Quando atribuído *true* esvazia o buffer de saída logo quando ele ficar cheio.	`<%@ page autoFlush="true"%>`
isThreadSafe	Indica se as transações de *Servlet* podem receber múltiplos pedidos. Se *true*, a nova *thread* será iniciada capturando requisições simultâneas, por padrão o valor é *true*.	
info	Desenvolvedores usam este atributo para adicionar informações e documento para a página. Tipicamente é usado para informar o nome autor do código, versão, *copyright* e datas.	`<%@ page info="jedi.org test page, alpha version "%>`
errorPage	Define qual a pagina que será usada para tratar os erros, deve ser um endereço URL para uma página de erro.	`<%@ page errorPage="/errors.jsp"%>`
IsErrorPage	Se for atribuído *true* transforma esta página JSP em uma página de erro. Esta página tem acesso ao objeto implícito *exception*.	

A sintaxe XML para definir as diretivas é:

```
<jsp:directive.directiveType attribute=value />
```

por exemplo o equivalente XML de:

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

é

```
<jsp:directive.page import="java.util.*" />
```

4.4.1. Diretiva *include*

A diretiva *include* define qual arquivo será incluído na página. Isso permite inserir um aquivo na classe servlet (inclui o conteúdo do arquivo dentro de outro arquivo) durante a tradução. Tipicamente a inclusão de arquivos é usada pelos componentes comuns a várias páginas como navegação, tabelas, cabeçalhos rodapés e etc.

A diretiva Include possui a seguinte sintaxe:

```
<%@ include file="relativeURL"%>
```

Por exemplo, para incluir uma barra de menu encontrada no diretório corrente, a diretiva será escrita da seguinte forma:

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

4.4.1. Diretiva *taglib*

A biblioteca de *tag* é uma coleção de *tags* predefinidas. A diretiva *taglib* define a biblioteca de *tags* usada nesta página. As *taglibs* são escritas da seguinte forma:

```
<%@ taglib uri="tag library URI" prefix="tag Prefix"%>
```

Esta diretiva informa ao contêiner que será considerado um código predefinido como remarcação e como esse código de remarcação irá se apresentar.

Vejamos como exemplo o arquivo index.jsp na listagem seguinte. A primeira linha declara que o arquivo index.jsp usa o código predefinido "*struts/template*", identificando como "*template*" para simplificar a digitação. As linhas seguintes referenciam a *taglib* antepondo a remarcação corretamente.

index.jsp

```
<%@ taglib uri="struts/template" prefix="template"%>
<template:insert template="/WEB-INF/view/template.jsp">
 <template:put name="header" content="/WEB-INF/view/header.jsp"/>
 <template:put name="content" content="/WEB-INF/view/index_content.jsp"/>
 <template:put name="footer" content="/WEB-INF/view/footer.jsp"/>
</template:insert>
```

Tags predefinidas foram introduzias na JSP 1.1 e permitem aos desenvolvedores JSP esconder dos *web designers* aspectos complexos do servidor.

4.5. JavaBeans em JSP

O uso dos *JavaBeans* em JSP não é definido pela especificação JSP. Entretanto, provêem funcionalidades interessantes. O uso de *JavaBeans* tem reduzido bastante a quantidade de elementos encontrados nas páginas JSP.

Primeiramente, *JavaBeans* são simplesmente classes Java que seguem um determinado padrão de codificação:

- por padrão, provê um construtor sem argumentos
- possui exclusivamente métodos *gets* e *sets*

Um exemplo de *JavaBean* é demonstrado a seguir:

```
package jedi.beans;

public class User {
  private String name;
  private String address;
  private String contactNo;

  public User() { }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public String getAddress() {
    return address;
  }
  public void setAddress(String address) {
    this.address = address;
  }
  public String getContactNo() {
    return contactNo;
  }
  public void setContactNo(String contactNo) {
    this.contactNo = contactNo;
  }
}
```

4.5.1. Como são usados os JavaBeans em JSP?

Como um objeto de transferência de dados – *JavaBeans* são amplamente utilizados em páginas JSP como objeto de transferência de dados. Em muitas aplicações os processos são executados em uma *Servlet*, não em um JSP. Só os resultados do transcurso são passados para o JSP na forma de um ou mais JavaBeans.

Como um objeto auxiliar - Em algumas aplicações menores não é eficiente ter um servlet separado para processar os dados. Neste caso, é melhor colocar a funcionalidade dentro de um *JavaBean*.

4.5.2.*JavaBeans* relacionados com ações de JSP

JSP define ações padrão que simplificam o uso de JavaBeans.

<jsp:useBean>

Para usar um componente *JavaBean*, deve-se habilitar seu uso por instanciação, o qual pode feito por intermédio das ações abaixo:

Os atributos das ações <jsp:useBean> são as seguintes:

- **id** – especifica o nome do *bean* e como ele será referenciado na página.
- **scope** – especifica a região na qual se pode armazenar uma instância do *JavaBean*. Pode ser anexado à página (*page*), à sessão (*session*), à requisição (*request*), ou à aplicação (*application*).
- **class** – especifica o nome da classe Java na qual o *JavaBean* é criado. Se foi especificado o nome do *JavaBean* não será preciso especificar a classe.
- **beanName** – Especifica o nome do bean que está armazenado no servidor. Refira-se a este atributo como a uma classe (por exemplo, *com.projectalpha.PowerBean*). Se especificado o atributo ***class***, não é necessário especificar este atributo.
- **type** - especifica o tipo de variável de *scripting* devolvido pelo *bean*. O tipo tem que se relacionar à classe do *bean*.

A seguir, um exemplo de como usar um *JavaBean* em uma pagina JSP:

```
<jsp:useBean id="user" scope="session" class="jedi.bean.User"/>
```

Quando a página encontra uma ação *useBean*, primeiro tentará verificar se já existe uma instância do *JavaBean* deste tipo determinado com a determinada extensão.

Caso não exista, o contêiner automaticamente cria uma nova instância do *JavaBean* usando o construtor padrão sem argumentos. Colocando o *JavaBean* no escopo determinado.

Caso a funcionalidade anterior seja expressada como um *scriptlet*, se pareceria com:

```
<%
jedi.bean.User user = (jedi.bean.User)session.getAttribute("user");
if (user == null) {
  user = new User();
  session.setAttribute("user", user);
}
%>
```

Uma vez que o *JavaBean* tenha sido habilitado usando a ação *jsp:useBean*, pode ser usado dentro da pagina JSP como qualquer instância de objeto, apenas usando o nome especificado no atributo **id**. Vejamos o seguinte exemplo:

```
<jsp:useBean id="user" scope="session" class="jedi.bean.User"/>
<HTML>
 <BODY>
  Hello <%= user.getName() %> !!
 </BODY>
</HTML>
```

<jsp:getProperty>

Esta ação retorna o valor de uma propriedade específica dentro de um *JavaBean* e retorna imediatamente ao fluxo de resposta.

Esta ação possui dois atributos:

- **name** – Nome do *JavaBean* cuja propriedade será retornada. Ela deve possuir o mesmo valor do atributo id usado na ação anterior <jsp:useBean>
- **property** – Nome da propriedade cujo valor será retornado.

Se o desenvolvedor desejar recuperar o valor de uma propriedade de JavaBean sem colocar isto imediatamente no fluxo saída, não se terá outra escolha a não ser fazer uso de *scriptlet* ou expressão (a ser discutido em capítulos posteriores).

Seguindo os exemplos anteriores, para exibir a propriedade *name* através do JavaBean, faz-se uso da ação getProperty da seguinte forma:

```
<jsp:getProperty name="user" property="name"/>
```

<jsp:setProperty>

Esta ação permite aos desenvolvedores atribuir valores a propriedades do *JavaBean* sem que seja necessário escrever uma linha de código no *scriptlet*.

Esta ação possui os mesmos atributos da ação getProperty e dois adicionais:

- **value** – Valor que deve ser atribuído à propriedade. Pode ser um valor estático ou uma expressão avaliada durante a execução.
- **param** – Especifica o parâmetro de requisição pelo qual a propriedade irá retornar.

O desenvolvedor que faz uso desta ação da seguinte forma: deve especificar o valor, ou o atributo de **param**, mas, ao incluir ambos atributos na ação, irá causar uma exceção.

4.6. Capturando erros

Páginas JSP podem fazer uso da diretiva de página para especificar uma página que controlará qualquer exceção que não possa ser capturada. O atributo *errorPage* da diretiva de página pode ser passado por uma URL relativa à página de JSP, identificada como uma página de erro. A página assim designada pode fixar o atributo *isErrorPage* para processar o erro. Deste modo, será realizado acesso a um objeto implícito chamado ***exception*** – que contém detalhes sobre a exceção lançada.

Um exemplo disto é fornecido a seguir:

```
<%@page errorPage="errorPage.jsp"%>
<HTML>
 <%
 String nullString = null;

 // chama um método de um objeto nulo, assim será lançada uma exception
 // e a página de erro será chamada.
 %>
 The length of the string is <%= nullString.length() %>
</HTML>
```

errorPage.jsp:

```
<%@page isErrorPage="true"%>
<HTML>
 <TITLE>Error!</TITLE>
 <BODY>
  <H1>An Error has occurred.</H1><br/>
  Laçamento mas um erro ocorreu com a pagina acessada anteriormente, por favor
  contacte um membro do suporte e informe a mensagem:
  <%=exception.getMessage()%> que foi a causa do erro.
 </BODY>
</HTML>
```

A primeira página usa a diretiva *page* que descreve uma pagina de erro que será usada quando uma exceção não puder ser capturada.

Na página de erro indicada, à diretiva de página *isErrorPage* é atribuída com o valor *true*. A página pode usar um objeto *exception* para mostrar a mensagem de erro gerada pela página chamada.

5. Exercícios

1) Considere a seguinte classe:

```
public class Recipe {
    private String recipeName;
    private String[] ingredients;
    private int cookTime;

    // métodos getter and setter aqui.
 }
```

Temos um cenário onde uma instância desta classe que foi armazenada em um escopo de requisição por intermédio da chave "currentRecipe". Crie uma pagina JSP que faça o seguinte:

a) Retorne uma instância de **Recipe** usando as ações JSP.

b) Mostre detalhes contidos na instância. Isso inclui mostrar cada elemento do array *ingredients*. detalhes devem ser mostrados usando ações JSP.

2) Usando a mesma classe do exercício anterior crie outra página que mostre os mesmos detalhes. Desta vez, faça uso de objetos implícitos para retornar a instância do objeto *Recipe* e use expressões para mostrar os detalhes.

3) Usando a mesma classe de definição do objeto Recipe realize as seguintes tarefas:

a) Crie uma pagina JSP que irá criar instâncias usando ações JSP.

b) Atribua à propriedade *recipeName* o valor "Monte Carlo Sandwich" e a propriedade *cookTime* o valor 0 (zero). atribua estas propriedades usando as melhores ações JSP.

c) Mostre os valores de instância deste objeto.

4) Uma página JSP chamada **otherContent.jsp** está localizada na raiz. É definida da seguinte forma:

```
<TABLE cellspacing="5">
 <tr>
  <td colspan="2"> This is very important content </td>
 </tr><tr>
  <td colspan="2"> Inclusion successful! </td>
 </tr>
</TABLE>
```

Crie uma página JSP que irá incluir o conteúdo definido em **otherContent.jsp**.

Parceiros que tornaram JEDI™ possível

Instituto CTS
Patrocinador do DFJUG.

Sun Microsystems
Fornecimento de servidor de dados para o armazenamento dos vídeo-aulas.

Java Research and Development Center da Universidade das Filipinas
Criador da Iniciativa JEDI™.

DFJUG
Detentor dos direitos do JEDI™ nos países de língua portuguesa.

Banco do Brasil
Disponibilização de seus *telecentros* para abrigar e difundir a Iniciativa JEDI™.

Politec
Suporte e apoio financeiro e logístico a todo o processo.

Borland
Apoio internacional para que possamos alcançar os outros países de língua portuguesa.

Instituto Gaudium/CNBB
Fornecimento da sua infra-estrutura de hardware de seus servidores para que os milhares de alunos possam acessar o material do curso simultaneamente.