UNIDADE 3 – JAVA SERVER PAGES (JSP)
MÓDULO 1 – JSP

01

1- A API JSP

Java Server Pages (JSP) é uma outra tecnologia Java para desenvolver aplicativos web. JSP foi lançada em meados de 1999 quando a tecnologia Servlet tinha atingido popularidade como uma das melhores tecnologias disponíveis no mercado. JSP, porém, não se destina a substituir o Servlet. Na verdade, JSP é uma extensão da tecnologia de Servlet, e como tal, é utilizada de maneira conjunta e complementar à tecnologia que estudamos anteriormente.

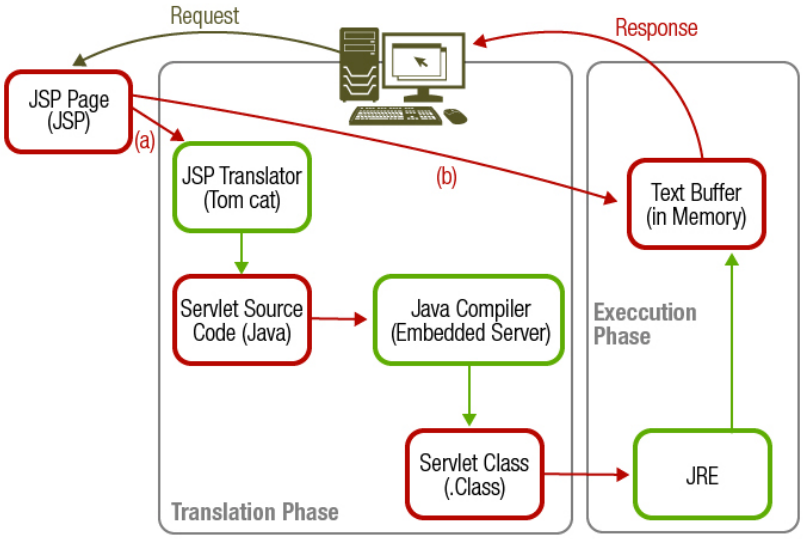
Até agora foi apresentada a geração de conteúdo dinâmico por meio dos Servlets. Porém, construir um sistema web apenas com Servlets dificulta a manutenção e a legibilidade dos códigos que compõem o sistema, pois todo código HTML escrito dinamicamente, até o presente momento, estava misturado ao código Java. Ou seja, a escrita dos Servlets pode ser vista sob uma de suas dimensões como sendo código HTML, em menor parte, embutido dentro de código Java, em maior parte.

Como forma de melhorar essa relação entre HTML e Java, as páginas JSP surgem como um recurso complementar, uma vez que elas invertem essa relação. Isso significa que as páginas JSP são códigos Java, em menor parte, embutido dentro de código HTML, em maior parte. Deste modo, podemos dizer que páginas JSP possuem, em sua maior parte, código HTML e, em menor parte, código java.

02

Um ponto importante que deve ser memorizado é que toda JSP é um Servlet, mas nem todo Servlet é uma JSP. Isso significa que toda págna JSP é traduzida para um Servlet de modo automático pelo Container web no primeiro acesso. Ou seja, a tradução significa que o Container irá ler o código da página JSP e reescrevê-lo como um Servlet para, somente depois, compilá-lo e executá-lo. Nos acessos seguintes, a tradução não é mais necessária, salvo se a página JSP tiver seu conjunto de instruções alterado.

A figura abaixo mostra o ciclo de requisição de uma página JSP.



03

A tecnologia JSP é baseada em uma API que consiste em dois pacotes: "*javax.servlet.jsp*" e "*javax.servlet.jsp.tagext*". O primeiro será estudado neste momento, enquanto que o segundo trata da definição, criação e utilização de tags personalizadas, criadas pelo próprio programador, que não será objeto de estudo desta disciplina.

O pacote "javax.servlet.jsp" define, atualmente, três interfaces e, basicamente, seis classes essenciais.

As **interfaces** são:

- JspPage
- HttpJspPage
- JspApplicationContext (JSP 2.1)

As seis **classes essenciais** são:

- JspEngineInfo
- JsPFactory
- PageContext
- JspWriter
- JspContext (JSP 2.0)
- ErrorData (JSP 2.0)

Observe que em algumas classes e interfaces existem os números de versões (*jsp 2.0 ou jsp 2.1*) em que tais recursos passaram a existir na especificação. Aquelas que não possuem tal número existem deste o princípio/nascimento da referida especificação.

04

1.1 A Interface JspPage

JspPage é uma interface que precisa ser implementada por todas as classes de servlet da JSP. A interface *JspPage* extende a interface "*javax.servlet.Servlet*" que vimos anteriormente.

A referida interface especifica duas operações: *jspInit* e *jspDestroy*. Ambas as operações se comportam de modo análogo às suas operações semelhantes da interface "*javax.servlet.Servlet*".

Isso significa que a *jspInit* é chamada uma única vez no **início** do ciclo de vida de uma página JSP e a *jspDestroy* é chamada uma única vez no **final** do ciclo de vida de uma página JSP.

A assinatura das operações está descrita logo abaixo:

```
public void jspInit();

public void jspDestroy();
```

1.2 A Interface HttpJspPage

HttpJspPage possui apenas uma única operação denominada de *_jspService*. A referida interface estende a interface *JspPage* para introduzir uma operaçãoo análoga à operação *service* da interface "*javax.servlet.Servlet*". A assinatura da referida operação está descrita logo abaixo:

```
public void _jspService
           (HttpServletRequest request, HttpServletResponse response)
                                 throws ServletException, IOException;
```

Um ponto importante que deve ser observado é que o conteúdo deste método *_jspService* será determinado automaticamente pelo Container em função do conjunto de instruções descritos na página JSP pelo programador. Ou seja, o conjunto de instruções da página JSP representa o conteúdo da operação *_jspService*.

05

1.3 A interface JspApplicationContext

O Container de JSP é responsável por criar uma instância de *JspApplicationContext* para cada instância de *ServletContext* existente na aplicação. Essa interface especifica um objeto que será importante para a manipulação de Expression Language (EL). Desta forma, no momento oportuno, tal interface será abordada.

1.4 A classe JspFactory

A classe JspFactory é uma classe abstrata que oferece métodos para obter outros objetos necessários ao processamento da página JSP. Ou seja, a partir de um objeto desta classe obtém-se os objetos do tipo *JspApplicationContext*, *JspEngineInfo* e *PageContext,* estes dois últimos descritos logo adiante.

1.5 A classe JspEngineInfo

A classe *JspEngineInfo* é uma classe abstrata que oferece informações sobre o Container de JSP. A referida classe define apenas um método denominado de *getSpecificationVersion* o qual retorna o número da versão do container JSP.

1.6 A classe PageContext

A classe PageContext oferece métodos que são independentes de implementação. Os referidos métodos são utilizados para se criar outros objetos dos seguintes tipos: ServletRequest, ServletResponse, ServletConfig, ServletContext, HttpSession, dentre outros.

06

1.7 A classe JspWriter

A classe *JspWriter* é uma classe abstrata que estende a classe "*java.io.Writer*". Um objeto da referida classe deve ser utilizado para escrever dados que serão transmitidos para o navegador web do cliente por meio dos métodos *print* ou *println*. Quando estudamos o uso dos Servlets, um objeto da classe *PrintWriter* foi bastante utilizado para escrever dados que seriam transmitidos aos browser clientes. Um ponto importante que se deve levar em consideração é que a classe *JspWriter* emula várias funcionalidades da classe *PrintWriter*, porém ela cria distinções no que diz respeito ao lançamento da exceção *IOException* para os métodos *print's.* Já a classe *Printwriter* não realiza qualquer distinção, no que diz respeito ao lançamento da exceção IOException, entre os métodos de escrita (*print's*).

1.8 A classe JspContext

A classe *JspContext* passou a existir na especificação JEE somente a partir da versão JSP 2.0. A introdução da JSP 2.0 na especificação JEE somente aconteceu em 11 de novembro de 2003, o que coincide com a versão 1.4 da especificação JEE. A partir de então, tal classe passou a ser a base para a classe *PageContext* descrita anteriormente.

A classe *JspContext* tem por objetivo abstrair todas as informações que não são especificas dos servlets como por exemplo as *Tags* personalizadas e a Expression Language (EL).

1.9 A classe ErrorData

A classe *ErrorData* possui o mesmo momento histórico da classe *JspContext*. Porém, sua finalidade é informar sobre os erros, como o próprio nome já diz. A informação pertencente a este objeto somente tem significado se utilizada no contexto de uma "*error page*" (páginas JSP criadas especificamente para esse fim.)

07

2- O SERVLET GERADO AUTOMATICAMENTE

Como forma de visualizar tecnicamente aquilo que foi descrito até agora, será escrito uma página JSP de exemplo, extremamente simples, cujo objetivo é apenas subsidiar uma análise superficial, pelo menos nesse momento, do servlet gerado a partir da mesma. A sintaxe específica para escrita de uma página JSP será apresentada oportunamente.

Deste modo, abaixo está o código fonte de exemplo da primeira página JSP:

```
<%@ page language="java"
         contentType="text/html; charset=UTF-8"
         pageEncoding="UTF-8"
%>
<!DOCTYPE html>
<html>
      <head>
            <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
            <title>Primeira Página JSP - Exemplo</title>
      </head>
      <body>
            <%
                  out.print("Hello World!");
            %>
      </body>
</html>
```

08

Uma vez que a referida página JSP esteja implantada no servidor de aplicação (Tomcat, por exemplo), o referido Servlet será criado automaticamente pelo Container após o primeiro acesso a referida página. Abaixo o código fonte do Servlet:

```
/*
 * Generated by the Jasper component of Apache Tomcat
 * Version: Apache Tomcat/7.0.63
 * Generated at: 2015-07-12 18:55:15 UTC
 * Note: The last modified time of this file was set to
 *       the last modified time of the source file after
 *       generation to assist with modification tracking.
 */
```

```
package org.apache.jsp;

import javax.servlet.ServletException;

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

     private static final javax.servlet.jsp.JspFactory _jspxFactory =
javax.servlet.jsp.JspFactory
                .getDefaultFactory();

     private static java.util.Map<java.lang.String, java.lang.Long>
_jspx_dependants;

     private javax.el.ExpressionFactory _el_expressionfactory;
     private org.apache.tomcat.InstanceManager _jsp_instancemanager;

     public java.util.Map<java.lang.String, java.lang.Long>
getDependants() {
           return _jspx_dependants;
     }

     public void _jspInit() {
           _el_expressionfactory =
_jspxFactory.getJspApplicationContext(

     getServletConfig().getServletContext()).getExpressionFactory();
           _jsp_instancemanager =
org.apache.jasper.runtime.InstanceManagerFactory
                     .getInstanceManager(getServletConfig());
     }

     public void _jspDestroy() {
     }

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

           final javax.servlet.jsp.PageContext pageContext;
```

```
            javax.servlet.http.HttpSession session = null;
            final javax.servlet.ServletContext application;
            final javax.servlet.ServletConfig config;
            javax.servlet.jsp.JspWriter out = null;
            final java.lang.Object page = this;
            javax.servlet.jsp.JspWriter _jspx_out = null;
            javax.servlet.jsp.PageContext _jspx_page_context = null;

            try {
                  response.setContentType("text/html; charset=UTF-8");
                  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("<!DOCTYPE html>\n");
                  out.write("<html>\n");
                  out.write("<head>\n");
                  out.write("<meta http-equiv=\"Content-Type\"
content=\"text/html; charset=UTF-8\">\n");
                  out.write("<title>Primeira Página JSP -
Exemplo</title>\n");
                  out.write("</head>\n");
                  out.write("<body>\n");
                  out.write("\t");

                  out.print("Hello World!");

                  out.write("\n");
                  out.write("</body>\n");
                  out.write("</html>");
            } catch (java.lang.Throwable t) {
                  if (!(t instanceof
javax.servlet.jsp.SkipPageException)) {
                        out = _jspx_out;
                        if (out != null && out.getBufferSize() != 0)
                              try {
```

```
                                    if (response.isCommitted()) {
                                          out.flush();
                                    } else {
                                          out.clearBuffer();
                                    }
                              } catch (java.io.IOException e) {
                              }
                        if (_jspx_page_context != null)
                              _jspx_page_context.handlePageException(t);
                        else
                              throw new ServletException(t);
                  }
            } finally {
                  _jspxFactory.releasePageContext(_jspx_page_context);
            }
      }
}
```

Observe que o código fonte acima é bastante complexo e com inúmeros detalhes. O objetivo aqui é apenas identificar os recursos que foram descritos ao longo do item 1.1 bem como a instrução "*out.print("Hello Wolrd!")*" que foi introduzida na página JSP de exemplo. Todo o restante terá mais significado à medida que os demais conteúdos forem sendo adequadamente compreendidos.

09

3 - OBJETOS IMPLÍCITOS

Examinando o código fonte do servlet gerado automaticamente pelo Container web a partir da página JSP de exemplo, podemos observar que o método *_jspService* possui dois parâmetros formais (são análogos as variáveis locais) e algumas variáveis locais propriamente ditas.

Apesar de neste exemplo específico não estarem presentes todas as possíveis variáveis locais, a especificação JSP mais recente considera como sendo objetos implícitos as variáveis descritas na tabela abaixo:

Objeto	Tipo
request	javax.servlet.http.HttpServletRequest
response	javax.servlet.http.HttpServletResponse
out	javax.servlet.jsp.JspWriter
session	javax.servlet.http.HttpSession

application	javax.servlet.ServletContext
config	javax.servlet.ServletConfig
pageContext	javax.servlet.jsp.PageContext
page	javax.servlet.jsp.HttpJspPage
exception	javax.servlet.jsp.JspException

É importante salientar que tais objetos implícitos são disponibilizados automaticamente ao programador de um JSP. O objetivo do uso de objetos implícitos é permitir que todas as capacidades do Servlet continuem disponíveis ao programador de modo que o desenvolvimento da página, utilizando-se a tecnologia JSP, não careça de limitações.

10

RESUMO

Neste módulo definimos o modelo de funcionamento de uma página JSP bem como o detalhamento de sua API. Além disso, identificamos e caracterizamos os objetos implícitos definidos na especificação JSP2.0, os quais são disponibilizados automaticamente ao programador de um JSP.

Vimos que o objetivo do uso de objetos implícitos é permitir que todas as capacidades do Servlet continuem disponíveis ao programador de modo que o desenvolvimento da página, utilizando-se a tecnologia JSP, não careça de limitações.

UNIDADE 3 – JAVA SERVER PAGES (JSP)

MÓDULO 2 – SINTAXE JSP

01

1 - SINTAXE JSP

O objetivo deste módulo é descrever a sintaxe necessária à escrita de páginas JSP bem como uma forma de uniformizar tal escrita. A escrita de páginas JSP se assemelha muito com a escrita de páginas dinâmicas para sistemas web utilizando-se outras linguagens de programação tais como ASP e PHP. Deste modo, as semelhanças com outras linguagens de programação encontradas ao longo deste módulo não podem ser vistas como mera coincidência.

Uma página JSP pode ter código Java e código HTML misturado em um mesmo arquivo fonte. De um modo formal, pode-se dizer que uma página JSP pode ter elementos e *template data*. Os elementos,

que também são chamados de ***tags JSP***, formam a sintaxe e a semântica de uma página JSP sendo os responsáveis, no lado do servidor, pela parte dinâmica do conteúdo.

Já os *template data* são todo o restante, ou seja, aquelas partes que o Container JSP não compreende, como por exemplo, as tags HTML, CSS, dentre outros.

A especificação JSP define três elementos, a saber:

- Elementos de diretiva;
- Elementos de script;
- Elementos de ação.

02

1.1 Elementos de Diretiva

Diretivas são mensagens enviadas ao Container de JSP contendo informações necessárias que são responsáveis por orientar o procedimento de tradução da página JSP em comento para o seu respectivo Servlet.

A sintaxe básica de uma diretiva é:

```
<%@ directive attribute0="value0" attribute1="value1" …
attributeN="valueN" %>
```

As diretivas podem ser de três tipos:

a) Diretivas de Página
b) Diretivas de inclusão
c) Diretivas de tag de biblioteca

As duas primeiras serão estudadas neste módulo enquanto que a terceira será estudada no módulo 4 (JSTL) desta unidade.

03

a) Diretivas de Página

A diretiva de página, como o próprio nome diz, se aplica a página JSP em que a mesma se encontra presente. A sintaxe básica é a seguinte:

```
<%@ page attribute0="value0" attribute1="value1" … attributeN="valueN"
%>
```

Essa diretiva suporta 11 tipos de atributos, descritos na tabela abaixo:

Atributo	Tipo de Valor	Valor Padrão
language	Nome de linguagem de script	"Java"
info	String	Depende do conteiner JSP
contenType	Tipo MIME	Depende do conteiner JSP. Em geral é "tet/html;charset=UTF-8"
extends	Nome da super classe	Nenhum
import	Nome de classe fqdn	Nenhum
buffer	Integer	8192
autoFlush	boolean	"true"
session	boolean	"true"
isThreadSafe	boolean	"true"
errorPage	URL	Nenhum
isErrorPage	boolean	"false"
pageEncoding	String	Depende do conteiner JSP. EM geral é "UTF-8"
isElIgnored	boolean	"false"

A sintaxe JSP não lhe permite repetir o mesmo atributo dentro uma diretiva *page* ou em múltiplas diretivas *page* na mesma página. A única exceção a esta regra é para o atributo *import* que pode aparecer múltiplas vezes na mesma página.

04

b) Diretivas de Inclusão

Essa diretiva permite aos programadores de página JSP incluírem o conteúdo de outros arquivos na página JSP atual. Essa diretiva é útil nos casos em que se possui uma página comum que será utilizada por mais de uma página JSP. Muitas interfaces web fazem uso deste tipo de recurso para dividir a página em seções denominadas de cabeçalho (header), rodapé (footer) e conteúdo (content).

A sintaxe básica é a seguinte:

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

1.2 Elementos de Script

Os elementos de script permitem que seja possível inserir código JAVA nas páginas JSP.

Existem três tipos de elementos de script:

a) Scriptlets
b) Declarations
c) Expressions

05

a) Scriptlets

Scriptlets são blocos de código de uma página JSP cuja sintaxe básica é a seguinte:

```
<% ...Código Java... %>
```

É importante frisar que todo o conteúdo dos scriptlets irão compor o corpo/escopo do método *_jspService*. Ou seja, o código java permitido em um scriptlet é o mesmo permitido em um método qualquer tais como:

- declaração e inicialização de variáveis/referencias locais;

- instruções de decisão e repetição;

- operadores aritméticos, lógicos, relacionais e bitwise;

- chamadas de métodos.

Ao escrever scriptlets, é muito comum e, por vezes, conveniente, alternar entre *tags HTML* e *tags JSP* (*scriptlets*). Esse tipo de técnica será demonstrado nos exemplos práticos.

06

b) Declarations

As declarações permitem que se declarem métodos e atributos em uma página JSP e que poderão ser utilizados em qualquer parte da mesma. As declarações também oferecem uma maneira de se criar o código relativo aos métodos *jspInit* e *jspDestro.* A sintaxe básica de uma declaração é a seguinte:

```
<%! ...Código Java... %>
```

É importante frisar que todo o conteúdo das declarações irão compor o corpo/escopo do Servlet (não se esqueça que é uma classe java) gerado automaticamente pelo Container.

c) Expressions

As expressões são avaliadas quando a página JSP é solicitada e seus resultados são convertidos para uma String que alimenta o parâmetro do método *print* do objeto implícito *out* do tipo "*javax.servlet.jsp.JspWriter"*. Caso não seja possível converter o resultado para o tipo String, um erro em tempo de execução será lançado. A sintaxe básica para uma expressão é a seguinte:

```
<%= ... Código Java cujo resultado será convertido para String... %>
```

07

1.3 Elementos de Ação

Os elementos de ação são tags que podem ser embutidas em uma página JSP para executar diversos tipos de tarefas predefinidas. Abaixo, alguns dos elementos de ação mais utilizados em páginas JSP:

- jsp:include

- jsp:forward

- jsp:param

- jsp:useBean

- jsp:setProperty

- jsp:getProperty

1.3.1 jsp:include

O elemento *jsp:include* é utilizado para incorporar recursos estáticos ou dinâmicos à página JSP atual. Esse elemento de ação é semelhante a diretiva *include*, mas *jsp:include* oferece maior flexibilidade pois permite a passagem de parâmetros por meio da outra ação padrão *jsp:param*. A sintaxe básica possui duas formas: uma sem a passagem de parâmetros e outra com a passagem de parâmetro:

Sem parâmetro

```
<jsp:include page="relativeURL" flush="true" />
```

Com parâmetro

```
<jsp:include page=”relativeURL” flush=”true” >
     <jsp:param … />
</jsp:inlcude>
```

08

1.3.2 jsp:forward

O elemento jsp:forward é usado para fazer um redirecionamento interno, ou seja, encerrar a execução da página JSP atual e trocar o controle para um outro recurso (estático ou dinâmico). A sintaxe básica possui duas formas: uma sem a passagem de parâmetros e outra com a passagem de parâmetro:

Sem parâmetro

```
<jsp:forward page=”relativeURL” />
```

Com parâmetro

```
<jsp:include page=”relativeURL” >
     <jsp:param … />
</jsp:inlcude>
```

Esse redirecionamento interno é equivalente àquele produzido pelo método *forward* da interface *javax.servlet.http.RequestDispacther*

09

1.3.3 jsp:param

O elemento *jsp:param* é utilizado conjuntamente com os elementos *jsp:include* ou *jsp:forward* conforme visto nos itens anteriores. Seu objetivo, como o próprio nome diz, é passar parâmetros adicionais e necessários ao processamento das outras páginas JSP. A sintaxe básica é a seguinte:

```
<jsp:param name=”param1” value=”value1” />
<jsp:param name=”param2” value=”value2” />
.
.
.
<jsp:param name=”paramN” value=”valueN” />
```

10

1.3.4 JavaBeans

São classes que encapsulam muitos objetos em um objeto simples (the bean).

As classes que são consideradas JavaBeans devem atender às seguintes **regras**:

- Devem ser serializáveis;
- Devem ter um construtor sem parâmetros (com zero parâmetros);
- Devem permitir o acesso às propriedades (atributos) por meio dos métodos gettter and setter.

Essa padronização na escrita de classes java tem como objetivo melhorar a reusabilidade destes componentes de *software* que são denominados de JavaBeans.

11

1.3.4.1 jsp:useBean

Antes de poder utilizar um *javaBean* em uma página JSP, é preciso torná-lo disponível utilizando o elemento padrão *jsp:useBean*. Esse elemento possui mecanismos que podem ser utilizados para controlar o *javaBean*. A sintaxe para o elemento *jsp:useBean* possui duas formas:

- a primeira para utilizar um *javaBean* existente, cujas propriedades já possuem valores e
- o segundo para criar um *javaBean* e inicializar suas propriedades com valores predefinidos.

Primeira forma

```
<jsp:useBean attribute0=”value0” attribute1=”value1” …
attributeN=”valueN” />
```

Segunda forma

```
<jsp:useBean attribute0=”value0” attribute1=”value1” …
attributeN=”valueN” >
     … initialization code …
</jsp:useBean>
```

Esse elemento de ação suporta os tipos de atributos descritos abaixo:

a) id

b) class

c) type

d) scope

12

a) Id

O atributo *id* define um identificador único para o javaBean. Esse identificador pode ser usado ao longo da página e pode ser percebido com uma referência para o javaBean.

b) class

O atributo *class* especifica o nome completamente qualificado (fqdn) da classe *javaBean.* Porém, um nome de simples de classe é permitido caso o javaBean seja importado por meio da diretiva *page.*

c) type

O atributo type é opcional e frequentemente utilizado nos casos em que se pretende utilizar comportamentos polimórficos. Ou seja, se o type estiver presente, ele especificará: ou o próprio tipo da classe javaBean, ou o tipo de sua super classe, ou o tipo de uma interface que o javaBean implementa.

d) scope

O atributo *scope* define a visibilidade e o tempo de vida do javaBean. Esse atributo pode assumir um dos seguintes valores, conforme tabela abaixo:

Valor	Significado
page	O javaBean somente existirá na página atual.
request	O javaBean existirá para a página incluida ou referenciada pelos elementos *jsp:include* ou *jsp:forward*
session	O javaBean existirá por toda a sessão desta requisição específica
application	O javaBean existirá durante a vida do aplicativo web.

13

1.3.4.2 jsp:setProperty

O elemento de ação *jsp:setProperty* é utilizado para atribuir o valor à uma propriedade do objeto *javaBean*. Para isso, os metodos setters do objeto javaBean são utilizados.

A sintaxe básica é a seguinte:

```
<jsp:setProperty name=”beanName” property=”propertyName” value=”value”
/>
```

Além dessa sintaxe básica, algumas outras formas também são válidas:

a) Essa sintaxe permite atribuir ao valor da propriedade do bean, o valor do parâmetro da requisição:

```
<jsp:setProperty name=”beanName” property=”propertyName”
param=”parameterName” />
```

b) Nos casos em que os parâmetros da requisição tenham, exatamente, os mesmos nomes que as propriedades do bean, a sintaxe abaixo poderá ser utilizada:

```
<jsp:setProperty name=”beanName” property=”*”  />
```

14

1.3.4.3 jsp:getProperty

O elemento de ação *jsp:getProperty* é utilizado para ler o valor de uma propriedade do objeto *javaBean*. Para isso, os metodos getters do objeto javaBean são utilizados.

A sintaxe básica é a seguinte:

```
<jsp:getProperty name=”beanName” property=”propertyName” />
```

15

2 - Sintaxe JSP (Conversão para XML)

A sintaxe XML é uma outra forma de se escrever uma página JSP. Com exceção dos elementos de ação que foram mostrados já em uma sintaxe XML, os demais foram demonstrados ao longo deste módulo com a sintaxe tradicional.

Abaixo, cada um dos elementos bem como suas duas possíveis formas essenciais: sintaxe XML e sintaxe tradicional.

Elementos	Sintaxe XML	Sintaxe Tradicional
Diretiva	<jsp:directive: ….. />	<%@ ….. %>
Declaração	<jsp:declaration> ….. </jsp:declaration>	<%! ….. %>
Scriptlet	<jsp:scriptlet>	<%

	.....	.....
	</jsp:scriptlet>	%>
Expressão	<jsp:expression>	<%=
	.....	.....
	</jsp:expression>	%>

Para escrever uma página JSP eficaz, é preciso compreender como esses quatro elementos podem trabalhar de maneira coordenada. Lembrando que a sintaxe XML e a sintaxe tradicional são, apenas, duas formas diferentes de se escrever o mesmo código cada uma delas com suas vantagens e desvantagens.

Importante salientar que uma única página JSP deve possuir apenas uma destas duas formas, ou seja, não é possível programar uma única página JSP que utilize as duas formas simultaneamente. Nesta disciplina, em sua maior parte, adotaremos a sintaxe tradicional.

16

2.3 - Comentários

Os comentários têm por finalidade explicar o funcionamento das partes que compõem o código fonte de um programa de computador. Deste modo, os comentários em uma página JSP podem ser de três tipos:

- **tipo HTML** cuja sintaxe é: `<!-- ...comentários... -->;`
- **tipo JSP** cuja sintaxe é: `<%-- ...comentários... -%>;`
- **tipo Java** cuja sintaxe é: `<%--`

```
//comentários de uma linha

// OU
/* comentários
    de múltiplas
    linhas
*/
-%>;
```

Observe que os comentários do tipo Java somente são permitidos dentro de comentários do tipo JSP.

17

2.4 Exemplos de páginas JSP

Como forma de demonstrar o uso dos elementos descritos neste módulo, abaixo seguem os códigos de exemplo. Observe que os códigos fontes estão devidamente comentados. Desta forma, a leitura do código fonte é importante para continuidade da compreensão do conteúdo.

Para acessá-los adequadamente, após a devida implantação no servidor de aplicação, basta realizar uma requisição HTTP, por meio do browser web, para a URL adequada que possui o arquivo estático "indexFormularioLivroVisitas.html"

18

2.4.1 indexFormularioLivroVisitas.html

```
<!DOCTYPE html>
<html>
	<head>
		<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
		<title>Livro de Visitas</title>
	</head>
	<body>
		<form action="BeanPessoaLivroVisitas.jsp" method="post">
			Nome: <input type="text" name="nome"><br />
			Data Nascimento: <input type="text"
name="dataNascimento"> (dd/MM/yyyy)<br />
			<input type="submit" value="enviar">
		</form>
	</body>
</html>
```

19

2.4.2 BeanPessoaLivroVisitas.jsp

```
<%@page import="java.util.Date"%>
<%@page import="java.text.SimpleDateFormat"%>
<%@page import="br.aiec.Pessoa"%>
<%@page language="java"
		contentType="text/html; charset=UTF-8"
	pageEncoding="UTF-8"
%>
<!DOCTYPE html>
```

```
<html>
	<head>
		<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
		<title>BeanPessoa - Livro de Visitas - JSP</title>
	</head>
	<body>

		<jsp:useBean id="pessoaBean" class="br.aiec.Pessoa"
scope="request">
			<%
				//Obtendo o nome do formulário HTML no formato
String
				String nomePessoa =
request.getParameter("nome");
			%>

			<%--Utilizando o elemento de ação para atribuir o
valor do nome que foi recebido --%>
			<jsp:setProperty name="pessoaBean" property="nome"
value="<%=nomePessoa%>"/>
			<%
				//Obtendo a data do formulário HTML no formato
String
				String strDtNascimento =
request.getParameter("dataNascimento");
				//Convertendo a data no formato String para o
tipo java.util.Date
				Date dtNascimento = new
SimpleDateFormat("dd/MM/yyyy").parse(strDtNascimento);
			%>
			<%--Utilizando o elemento de ação para atribuir o
valor da data, devidamente convertida, que foi recebida como parâmetro
--%>
			<jsp:setProperty name="pessoaBean"
property="dataNascimento" value="<%=dtNascimento%>"/>
		</jsp:useBean>
		<%-- Redirecionando para o servlet de historico de visitas
--%>
		<jsp:forward page="/historicoLivroVisitas.do" />
	</body>
</html>
```

2.4.3 HistoricoLivroVisitasDemoServlet.java

```
package br.aiec;

import java.io.IOException;
import java.util.LinkedList;
import java.util.List;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(name="HistoricoLivroVisitas",
urlPatterns="/historicoLivroVisitas.do")
public class HistoricoLivroVisitasDemoServlet extends HttpServlet {

	private List<Pessoa> historicoVisitantes = new
LinkedList<Pessoa>();
	@Override
	public void init() throws ServletException {
		//Compartilhando a lista de histórico no contexto da
aplicação
		getServletContext().setAttribute("historyVisits",
historicoVisitantes);
	}
	@Override
	protected void doPost(HttpServletRequest request,
HttpServletResponse response)
			throws ServletException, IOException {
		//Recuperando o bean criado na JSP
		Pessoa pessoaBean =
(Pessoa)request.getAttribute("pessoaBean");
		//Adicionando o bean ao historico de visitantes
		historicoVisitantes.add(pessoaBean);
		//Redirecionando para uma JSP exibir o histórico de
visitantes
		RequestDispatcher dispatcher =
request.getRequestDispatcher("ExibirHistorico.jsp");
		dispatcher.forward(request, response);
	}
}
```

2.4.4 Pessoa.java (Classe que representa o JavaBean)

```
package br.aiec;

import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;

public class Pessoa {

	private String nome;
	private Date dataNascimento;

	public String getNome() {
		return nome;
	}
	public void setNome(String nome) {
		this.nome = nome;
	}
	public Date getDataNascimento() {
		return dataNascimento;
	}
	public void setDataNascimento(Date dataNascimento) {
		this.dataNascimento = dataNascimento;
	}
	public int getIdade(){
		Calendar hoje = GregorianCalendar.getInstance();
		hoje.setTime(new Date());

		Calendar nascimento = GregorianCalendar.getInstance();
		nascimento.setTime(dataNascimento);

		int quantidadeAnos = hoje.get(Calendar.YEAR) -
nascimento.get(Calendar.YEAR);

		nascimento.add(Calendar.YEAR, quantidadeAnos);

		if(nascimento.after(hoje)){
			quantidadeAnos--;
		}
		return quantidadeAnos;
	}
}
```

22

2.4.5 ExibirHistorico.jsp

```
<%@page import="java.text.SimpleDateFormat"%>
<%@page import="br.aiec.Pessoa"%>
<%@page import="java.util.List"%>
<%@page language="java"
            contentType="text/html; charset=UTF-8"
      pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
      <head>
            <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
            <title>Histórico de Visitantes - JSP</title>
      </head>
      <body>
            <%
                  //Obtendo a lista de histórico do contexto da
aplicação
                  List<Pessoa> listHistorico =
(List<Pessoa>)getServletContext().getAttribute("historyVisits");

                  //Criando um objeto para formatar a data
                  SimpleDateFormat dateFormat = new
SimpleDateFormat("dd/MM/yyyy");
            %>
            <table border="1">
                  <thead>
                        <tr>
                              <td>Nome</td>
                              <td>Data Nascimento</td>
                              <td>Idade</td>
                        </tr>
                  </thead>
                  <tbody>
                        <%--Esse laço percorre a lista de visitantes,
exibindo cada um deles --%>
                        <% for(int i = 0; i < listHistorico.size();
i++){ %>
                        <% Pessoa pessoa = listHistorico.get(i); %>

                        <tr>
                              <td><%=pessoa.getNome()%></td>
```

```
      <td><%=dateFormat.format(pessoa.getDataNascimento())%></td>
                              <td><%=pessoa.getIdade()%></td>
                        </tr>
                        <% } %>
                    </tbody>
                </table>
      </body>
</html>
```

23

RESUMO

Esse módulo teve como objetivo definir, especificar, apresentar e utilizar a linguagem Java para escritas de páginas JSP. Além disso, demonstramos a sintaxe e a semântica necessárias para realização da escrita destas páginas de modo coerente. Esse módulo apresentou e utilizou os três tipos de elementos JSP: diretivas, script e ação, além dos *scriptlets*, das declarações e expressões. Por último foram demonstrados os *JavaBeans* bem como suas características e as formas de se escrever comentários em JSP.

UNIDADE 3 – JAVA SERVER PAGES (JSP)
MÓDULO 3 – EXPRESSION LANGUAGE (EL)

01

1 - MAP

A sintaxe e o alcance da linguagem de expressão são bem simples. Esse módulo (EL) e o próximo módulo (JSTL) abordam assuntos que trabalham de modo conjunto com o propósito de tornar os códigos de páginas JSP mais homogêneo e intuitivo, além de procurar separar regras da aplicação das regras de interface com o usuário. Como visto anteriormente, os *scriptlets* não oferecem um meio para alcançar tal propósito. Pelo contrário, os scriptlets tornam o código da página JSP confuso e complexo, pois mistura, basicamente, duas técnicas de programação distintas: a **técnica imperativa** e a **técnica declarativa**.

O objetivo da EL (expression language) é auxiliar na eliminação dos *scriptlets*. Porém, a EL não consegue eliminar todo o scriptlet sozinha. Deste modo, o uso de EL e JSTL é essencial para atingir tal propósito.

Porém, antes de começarmos a falar da EL propriamente dita, teremos de conhecer um tipo de dado abstrato denominado ***Map***. Veja a seguir.

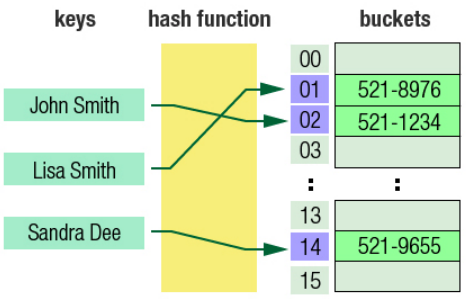
02

Um *Map* é uma simples estrutura de dados que associa uma chave (key) a um valor (value). Esse conceito extremamente flexível pode ser associado a diversas outras estruturas de dados, tais como uma árvore de pesquisa binária (Red-Black Tree).

Contudo, um *Map* é, na maioria das vezes, associado a outra estrutura de dados denominada de ***Hash Table***.

Uma *Hash Table* utiliza uma função de *hash* da *key* para definir o índice de um array cujo *value* da referida posição (bucket) estará, respectivamente, associado àquela *key*.

A imagem a seguir demonstra o referido conceito:



Por exemplo, uma função de hash qualquer utiliza a *key* de valor "Jonh Smith" como entrada. O hash produzido é o número 02 que servirá como índice para acessar o array.

Na referida posição apontada pelo indice (hash) existe o *value* "521-1234". Desta forma a *key* "Jonh Smith" está associada ao *value* "521-1234"

03

De um modo ideal, uma função de *hash* de uma determinada *key* somente poderá estar associada a uma única posição (*bucket*). Porém, existe a possibilidade de que duas ou mais *keys* diferentes gerem um mesmo índice provocando o que se chama de colisão. Ou seja, keys diferentes associadas a mesma posição (*bucket*).

Essa situação deve ser tratada de modo adequado, uma vez que sua ocorrência poderá provocar perda/sobreposição de valores/dados (*values*).

No caso da linguagem de programação Java, existe a interface ***java.util.Map*** que abstrai as operações que tal estrutura deve ter. Além disso, a classe *java.util.HashMap* implementa a referida estrutura de dados abstrata denominada de ***Hash Table***.

A compreensão das referidas estruturas irão auxiliar a compreensão no uso da *Expression Language* (EL) bem como de seus objetos implícitos.

04

2 - Expression Language (EL)

A sintaxe que envolve as ELs sempre virão dentro de um par de chaves “{}”, precedidas pelo símbolo de cifrão “$” conforme fórmula genérica abaixo:

```
${expr}
```

O termo “*expr*” deve corresponder a uma EL.

Uma EL utiliza-se de dois operadores básicos:

- o operador ponto “.” e
- o operador colchete “[ ]”.

Esses dois operadores permitem o acesso a vários atributos dos JavaBeans e dos objetos implícitos da EL. Neste ponto é importante observar que o operador colchete utilizado na EL é diferente do operador colchete da linguagem Java. Apesar da sintaxe de ambos serem próximas, sua semântica é considerada bem diferente. Parece um pouco confusa e contraditória essa relação, já que a EL, em sua essência, é código Java. Contudo, para que se possa compreender o operador colchete da EL de forma adequada, considere que o mesmo possua funcionalidades complementares, ou seja, considere-o como um operador sobrecarregado. Portanto, os colchetes em Java significam a representação de uma estrutura de dados denominada de Array cujas características são descritas como sendo **homogêneas**, **estáticas** e **contínuas** na memória.

Além disso, no caso do Java, o índice do array deve ser um **número natural**. Já para o operador colchete da EL, existem outros possíveis significados que serão explicados adiante.

05

As ELs suportam uma série de operadores lógicos, relacionais e aritméticos da linguagem Java. Abaixo, uma lista com os operadores mais usados:

Operador	Descrição
.	Acessar uma propriedade do bean ou uma entrada do *Map*
[ ]	Acessar um array ou uma lista de elementos
( )	Agrupar uma subexpressão para mudar a ordem de avaliação
+	Adição

-	Subtração ou negação de um valor
*****	Multiplicação
div ou /	Divisão
mod ou %	Resto da divisão
eq ou ==	Teste de igualdade
ne ou !=	Teste de diferença
lt ou <	Teste menor que
gt ou >	Teste maior que
le ou <=	Teste menor que ou igual
ge ou >=	Teste maior que ou igual
and ou &&	Teste para E lógico
or ou ||	Teste para o OU lógico
not ou !	Negação
empty	Teste para valores vazios de variáveis

06

2.1 - Objetos Implícitos da EL

Uma EL possui os objetos implícitos abaixo:

Objeto Implícito	Descrição
pageScope	Variáveis com escopo desta página
requestScope	Variáveis com escopo desta requisição
sessionScope	Variáveis com escopo desta sessão
applicationScope	Variáveis com escopo desta aplicação (contexto)
param	Parâmetros desta requisição como Strings

paramValues	Parâmetros desta requisição como uma coleção de Strings
header	Cabeçalhos desta requisição HTTP como Strings
headerValues	Cabeçalhos desta requisição HTTP como coleção de Strings
initParam	Parâmetros de inicialização do contexto
cookie	Valores dos Cookies
pageContext	Uma referencia real ao objeto pageContext da JSP.

Com exceção do *pageContext*, todos os demais objetos implícitos são objetos de mapeamento (*java.util.Map*). Isso significa que os objetos implícitos de uma JSP e de uma EL não são os mesmos, com exceção para o *pageContext*.

07

2.2 - O operador ponto "."

O operador ponto pode ser utilizado para acessar propriedades de objetos *JavaBeans* ou valores de um *Map*. Observe a fórmula genérica abaixo:

```
${firstThing.secondThing}
```

A variável genérica firstThing deve se referir ou a um objeto implícito ou a um atributo do tipo *JavaBean*.

Nos casos de se referir a um atributo, ele pode ser o nome de um atributo armazenado em qualquer um dos quatro escopos disponíveis (page, request, session ou application).

Já a variável genérica secondThing pode se referir ou a uma *key* do *Map (*Objetos Implícitos*)* ou a uma propriedade do objeto JavaBean.

08

2.3 - O operador colchete "[ ]"

O operador colchete é mais flexível que o operador ponto. Além de poder ser utilizado para manipular os mesmos elementos do operador ponto, o operador colchete também pode ser utilizado para acessar um *java.util.List* ou um *Array*.

Observe as duas fórmulas genéricas abaixo:

```
a)                    ${thirdThing [ fourtThing ] }
```

```
b)                    ${thirdThing [ "fourtThing" ] }
```

A variável genérica **thirdThing** deve se referir:

- ou a um objeto implícito
- ou a um atributo do tipo JavaBean
- ou a um atributo do tipo *java.util.List*
- ou a um atributo do tipo a*rray.*

Nos casos de se referir a um atributo, ele pode ser o nome de um atributo armazenado em qualquer um dos quatro escopos disponíveis (page, request, session ou application).

Já a variável genérica **fourtThing** pode se referir:

- ou a uma *key* do *Map (*Objetos Implícitos*)*
- ou a uma propriedade do objeto JavaBean
- ou a um índice de um *java.util.List*
- ou a um índice de um *array*.

Observe que a variável genérica **fourtThing** pode estar entre aspas duplas, ou seja, pode ser uma variável que possui um valor do tipo String. Lembre-se que para os casos em que estiver acessando um *java.util.List* ou um *array*, a String deve ser passível de conversão para números naturais (índices das estruturas) ou será lançada uma exceção do tipo *NumberFormatException*.

09

3 - Exemplos de páginas JSP

Como forma de demonstrar o uso de algumas ELs descritas neste módulo, abaixo seguem os códigos de exemplo. Observe que os códigos fontes estão devidamente comentados. Desta forma, a leitura do código fonte é importante para continuidade da compreensão do conteúdo.

Para acessá-los adequadamente, após a devida implantação no servidor de aplicação, basta realizar uma requisição HTTP, por meio do browser web, para a URL adequada que possui o arquivo estático "indexFormularioLivroVisitas.html"

10

3.1 indexFormularioLivroVisitas.html

```
<!DOCTYPE html>
<html>
	<head>
		<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
		<title>Livro de Visitas</title>
	</head>
	<body>
		<form action="BeanPessoaLivroVisitas.jsp" method="post">
			Nome: <input type="text" name="nome"><br />
			Data Nascimento: <input type="text"
name="dataNascimento"> (dd/MM/yyyy)<br />
			<input type="submit" value="enviar">
		</form>
	</body>
</html>
```

11

3.2 BeanPessoaLivroVisitas.jsp

```
<%@page import="java.util.Date"%>
<%@page import="java.text.SimpleDateFormat"%>
<%@page import="br.aiec.Pessoa"%>
<%@page language="java"
		contentType="text/html; charset=UTF-8"
	pageEncoding="UTF-8"
%>

<!DOCTYPE html>
<html>
	<head>
		<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
		<title>BeanPessoa - Livro de Visitas - JSP</title>
	</head>
	<body>

		<jsp:useBean id="pessoaBean" class="br.aiec.Pessoa"
scope="request">
			<%--Utilizando o elemento de ação para atribuir o
```

```
valor do nome que foi recebido como parâmetro --%>
                <jsp:setProperty name="pessoaBean" property="nome"
value="${param.nome}"/>
                <%--Utilizando o elemento de ação para atribuir o
valor da data que foi recebida como parâmetro --%>
                <jsp:setProperty name="pessoaBean"
property="dataNascimentoString" value="${param.dataNascimento}"/>

           </jsp:useBean>

           <%-- Redirecionando para o servlet de historico de visitas
--%>
           <jsp:forward page="/historicoLivroVisitas.do" />

     </body>
</html>
```

12

3.3 HistoricoLivroVisitasDemoServlet.java

```
package br.aiec;

import java.io.IOException;
import java.util.LinkedList;
import java.util.List;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(name="HistoricoLivroVisitas",
urlPatterns="/historicoLivroVisitas.do")
public class HistoricoLivroVisitasDemoServlet extends HttpServlet {

     private List<Pessoa> historicoVisitantes = new
LinkedList<Pessoa>();

     @Override
     public void init() throws ServletException {
          //Compartilhando a lista de histórico no contexto da
```

```
aplicação
            getServletContext().setAttribute("historyVisits",
historicoVisitantes);
      }

      @Override
      protected void doPost(HttpServletRequest request,
HttpServletResponse response)
                  throws ServletException, IOException {

            //Recuperando o bean criado na JSP
            Pessoa pessoaBean =
(Pessoa)request.getAttribute("pessoaBean");

            //Adicionando o bean ao historico de visitantes
            historicoVisitantes.add(pessoaBean);

            //Redirecionando para uma JSP exibir o histórico de
visitantes
            RequestDispatcher dispatcher =
request.getRequestDispatcher("ExibirHistorico.jsp");
            dispatcher.forward(request, response);
      }
}
```

13

3.4 Pessoa.java (Classe que representa o JavaBean)

```
package br.aiec;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;

public class Pessoa {

      private String nome;
      private Date dataNascimento;

      public String getNome() {
            return nome;
```

```
	}

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

	public Date getDataNascimento() {
		return dataNascimento;
	}

	public void setDataNascimento(Date dataNascimento) {
		this.dataNascimento = dataNascimento;
	}

	/**
	 * Método que recebe a dataNascimento no formato String
dd/MM/yyyy e transforma no tipo adequado.
	 *
	 * @param dataNascimento
	 * @throws ParseException
	 */
	public void setDataNascimentoString(String dataNascimento)
throws ParseException {
		SimpleDateFormat sf = new SimpleDateFormat("dd/MM/yyyy");
		this.dataNascimento = sf.parse(dataNascimento);
	}

	/**
	 * Esse método retorna a data formatada (tipo String)
	 *
	 * @return Data no formato dd/MM/yyyy
	 * @throws ParseException
	 */
	public String getDataNascimentoString() throws ParseException {
		SimpleDateFormat sf = new SimpleDateFormat("dd/MM/yyyy");
		return sf.format(dataNascimento);
	}


	public int getIdade(){
		Calendar hoje = GregorianCalendar.getInstance();
		hoje.setTime(new Date());

		Calendar nascimento = GregorianCalendar.getInstance();
```

```
            nascimento.setTime(dataNascimento);

            int quantidadeAnos = hoje.get(Calendar.YEAR) -
nascimento.get(Calendar.YEAR);

            nascimento.add(Calendar.YEAR, quantidadeAnos);

            if(nascimento.after(hoje)){
                  quantidadeAnos--;
            }

            return quantidadeAnos;
      }

}
```

Observe que a classe Pessoa teve que incorporar os métodos de conversão/formatação da data. Isso foi necessário, pois todo o código de scriptlet foi retirado da página “BeanPessoaLivroVisitas.jsp”. No próximo módulo, utilizaremos as tags específicas da JSTL para formatar a data sem scriptlets. Por ora, manteremos, temporariamente, a referida funcionalidade na classe “Pessoa”.

14

3.5 ExibirHistorico.jsp

```
<%@page import="java.text.SimpleDateFormat"%>
<%@page import="br.aiec.Pessoa"%>
<%@page import="java.util.List"%>
<%@page language="java"
            contentType="text/html; charset=UTF-8"
      pageEncoding="UTF-8"%>

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

<!DOCTYPE html>
<html>
      <head>
            <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
            <title>Histórico de Visitantes - JSP</title>
      </head>
      <body>
            <table border="1">
```

```
                <thead>
                      <tr>
                            <td>Nome</td>
                            <td>Data Nascimento</td>
                            <td>Idade</td>
                      </tr>
                </thead>
                <tbody>
                      <%--Esse laço percorre a lista de visitantes,
exibindo cada um deles --%>
                      <c:forEach var="pessoa"
items="${historyVisits}">
                            <tr>
                                  <td>${pessoa.nome}</td>

      <td>${pessoa.dataNascimentoString}</td>
                                  <td>${pessoa.idade}</td>
                            </tr>
                      </c:forEach>
                </tbody>
           </table>
      </body>
</html>
```

Esse último código da página “ExibirHistorico.jsp” faz uso da tag JSTL *ForEach*, que será tratado mais adiante na disciplina. Para que seja possível executar a referida página corretamente, faz-se necessária a instalação de duas bibliotecas extras:

- *jstl-api-1.2.1.jar*
- *jstl-1.2.1.jar*

A instalação das bibliotecas consiste em copiar os dois arquivos para a pasta WEB-INF/lib do contexto da sua aplicação web.

jstl-api-1.2.1.jar
Esta biblioteca está disponível em
http://search.maven.org/remotecontent?filepath=javax/servlet/jsp/jstl/javax.servlet.jsp.jstl-api/1.2.1/javax.servlet.jsp.jstl-api-1.2.1.jar

jstl-1.2.1.jar
Esta biblioteca está disponível em
http://search.maven.org/remotecontent?filepath=org/glassfish/web/javax.servlet.jsp.jstl/1.2.1/javax.servlet.jsp.jstl-1.2.1.jar

15

RESUMO

Vimos neste módulo o modelo de funcionamento da expression language (EL). O objetivo da EL (expression language) é auxiliar na eliminação dos *scriptlets*. Porém, a EL não consegue eliminar todo o scriptlet sozinha. Deste modo, o uso de EL e JSTL é essencial para atingir tal propósito. A compreensão da sintaxe e a da semântica envolvidas na leitura e na escrita das expressões é essencial para permitir uma alternativa ao uso de *scriptlets* com o objetivo de criar páginas JSP com código homogêneo e intuitivo, auxiliando também na separação lógica do aplicativo web em camadas.

UNIDADE 3 – JAVA SERVER PAGES (JSP)
MÓDULO 4 – JSP STANDARD TAG LIBRARY (JSTL)

01

1- A JSTL - STANDARD TAG LIBRARY

A JSTL é uma coleção de tags JSP que encapsula diversas funcionalidades de uso contínuo e comum presentes em inúmeras aplicações web. Alguns dos objetivos que motivaram o desenvolvimento da JSTL foram aqueles ligados, por exemplo, a clareza e a legibilidade do código presente nas páginas JSP.

Como já dito anteriormente, o uso de *scripts* torna o código das páginas JSP complexo e confuso em função da mistura de dois estilos/técnicas de programação diferentes: imperativo e declarativo. Desta forma, a API **JSTL** encapsulou em tags simples, toda a funcionalidade de construção dinâmica de páginas Web tais como:

- instruções de repetição (fors),
- instruções de decisão (ifs, elses, switchs),
- instruções de manipulação de dados XML,
- instruções de internacionalização/formatação de dados da sua aplicação, dentre outras.

Existe um total de seis taglibs para a JSTL:

a) *Core library*
b) *i18n-capable formating library*
c) *Functions library*
d) *Database library*
e) *TLVs library*
f) *XML library*

02

Para esta disciplina, serão estudadas apenas as três primeiras taglibs (*Core library, i18n-capable formating library, Functions library*). As demais não serão objeto de estudo, pois contradizem (*Database* e *TLVs*) ou extrapolam (*XML*) os objetivos desta disciplina.

A taglib ***Database*** deve ser evitada, uma vez que a mesma vai contra o propósito de separação lógica das camadas de construção de um *software* que está sendo estudado nesta disciplina. A referida taglib possui tags nativas para facilitar o acesso a banco de dados relacional por meio da linguagem SQL.

No caso desta disciplina, mais adiante, será apresentado um padrão de projeto denominado de Data Access Object (DAO) cuja finalidade é encapsular o acesso as bases de dados tornando o acoplamento entre a aplicação web e o banco de dados extremamente flexível.

Já as tags **TLVs** são utilizadas para permitir a validação de visões XML como páginas JSP. As TLVs permitem aos seus autores imporem restrições sobre o uso de elementos de script e bibliotecas de tags em páginas JSP.

Isso significa estabelecer restrições em como as páginas JSP serão escritas e, consequentemente, traduzidas e executadas. Deste modo, contraria o nosso propósito nesta disciplina cujo objetivo é, exatamente, o de estudar os diversos recursos disponíveis em todos os seus aspectos e dimensões.

E por último, a tag ***XML library*** que, embora útil, extrapola o conteúdo desta disciplina.

03

1.1 - Instalando a JSTL

Para que seja possível executar as tags da JSTL, faz-se necessário a instalação de duas bibliotecas extras:

- *jstl-api-1.2.1.jar*
- *jstl-1.2.1.jar*

A instalação das bibliotecas consiste em copiar os dois arquivos para a pasta “WEB-INF/lib” do contexto da sua aplicação web.

jstl-api-1.2.1.jar
Para instalar esta biblioteca, acesse http://search.maven.org/remotecontent?filepath=javax/servlet/jsp/jstl/javax.servlet.jsp.jstl-api/1.2.1/javax.servlet.jsp.jstl-api-1.2.1.jar

jstl-1.2.1.jar
Para instalar esta biblioteca, acesse
http://search.maven.org/remotecontent?filepath=org/glassfish/web/javax.servlet.jsp.jstl/1.2.1/javax.servlet.jsp.jstl-1.2.1.jar

04

2 - UTILIZANDO A JSTL

Para que seja possível fazer uso da JSTL, faz-se necessário importar, por meio da diretiva *taglib*, o conjunto de bibliotecas específicas estabelecendo um valor único e exclusivo para os dois atributos: ***prefix*** e ***uri***.

A diferença nestes valores é que o primeiro é de livre escolha do programador e o segundo não. Porém, mesmo para o primeiro valor, será utilizado o recomendado pela especificação. Abaixo, seguem os exemplos de uso das diretivas para cada taglib que será estudada:

Core library

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

i18n-capable formating library

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

Functions library

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

É importante lembrar que o atributo *uri* especifica um valor fixo (pré-definido) que não pode ser mudado sob o risco de problemas em tempo de tradução/compilação/execução. O referido valor de cada taglib será utilizado para procurar pelos arquivos que compõem a taglib dentro do contexto da aplicação web. Ou seja, não existe a realização de nenhum *download* automático de recursos da web, apesar de seus valores serem, aparentemente, endereços web. O funcionamento está condicionado à instalação, conforme item anterior.

05

2.1 - Core Library

A tabela abaixo possui uma listagem das core JSTL tags.

Tag	Descrição
<c:out>	Escrever dados na saída padrão
<c:set>	Inicializar variáveis em um escopo
<c:remove>	Remover as variáveis de um escopo
<c:catch>	Capturar qualquer Throwable que acontecer durante a execução e opcionalmente, exibi-la
<c:if>	Instrução de decisão simples
<c:choose>	Instrução de decisão para valores mutuamente exclusivos. Essa instrução é usado em conjunto com <c:when> e <c:otherwise>
<c:when>	Subtag de <c:choose> que inclue um corpo para valores 'true'
<c:otherwise>	Subtag de <c:choose> que deve seguir, sempre após, a última subtag <c:when>. Essa tag somente será executada se todas as condições anteriores forem 'false'
<c:import>	Recupera uma URI relativa ou absoluta e adiciona seu conteúdo à página JSP
<c:forEach>	Instrução de repetição simples
<c:forTokens>	Instrução de repetição simples para iterar sobre *tokens* (delimitadores)
<c:param>	Adiciona um parâmetro para o conteúdo importando por meio da tag <c:import>
<c:redirect>	Realiza um redirecionamento externo do browser com reescrita de URL
<c:url>	Formata uma URL e, quando necessário, realiza a reescrita de URL. Além disso, permite, opcionalmente, a definição de parâmetros.

06

2.2 - i18n-capable formating library

A tabela abaixo possui uma listagem das formating JSTL tags.

Tag	Descrição
<fmt:formatNumber>	Formatar números reais (ponto flutuante) com precisão especifica.
<fmt:parseNumber>	Converter a representação String de um número
<fmt:formatDate>	Formatar a date/time usando padrões
<fmt:parseDate>	Converter a representação String de uma date/time
<fmt:bundle>	Carregar um recurso de um pacote para ser usado no corpo da tag. Trabaha em conjunto com as tags <fmt:setLocale> e <fmt:setBundle>. É utilizado para exibição de mensagens padronizadas/traduzidas
<fmt:setLocale>	Utilizado para armazenar a configuração de determinado local (*locale*)
<fmt:setBundle>	Carrega um recurso de um pacote e armazena em uma variável ou em uma variável de configuração de pacote. É utilizado para exibição de mensagens padronizadas/traduzidas
<fmt:timeZone>	Especifica o timezone para qualquer formato de date/time ou converte actions aninhados em seu corpo
<fmt:setTimeZone>	Armazena o timezone em uma variável
<fmt:message>	Exibir mensagens padronizadas/internacionalizadas/traduzidas
<fmt:requestEncoding>	Usada para especificar o encoding type das requisições HTTP que serão feitas ao servidor de aplicação

07

2.3 - Functions library

As funções padrão definidas nesta biblioteca são para o tipo de dado String. Deste modo, a tabela abaixo possui uma listagem de algumas dessas funções.

Tag	Descrição
fn:contains()	Testa se a String de entrada contém a substring especificada
fn:containsIgnoreCase()	Testa se a String de entrada contém a substring especificada ignorando o case sensitive
fn:endsWith()	Testa se a String de entrada termina com o sufixo determinado
fn_indexOf()	Retorna o indice da String que possui a primeira ocorrencia da substring especificada
fn:length()	Retorna o número de itens de uma coleção ou o número de caracteres de uma String
fn:trim	Remove os espaços em branco das bordas(começo e fim) de uma String

08

3 - Exemplos de páginas JSP

Como forma de demonstrar o uso de algumas JSTLs descritas neste módulo, a seguir apresentamos os códigos de exemplo. Observe que os códigos fontes estão devidamente comentados. Desta forma, a leitura do código fonte é importante para continuidade da compreensão do conteúdo.

Para acessá-los adequadamente, após a devida implantação no servidor de aplicação, basta realizar uma requisição HTTP, por meio do browser web, para a URL adequada que possui a página dinâmica "FormularioLivroVisitas.jsp"

3.1 - Arquivos Texto de Mensagens Padronizadas (i18n)

O referido exemplo demonstra, dentre outras funcionalidades, a padronização de mensagens de uma aplicação web. Desta forma, é necessário criar um arquivo texto ASCII puro que contenha as mensagens a serem exibidas na interface web cuja extensão seja "*.properties*". Para o referido exemplo, foram criados dois arquivos contendo as mensagens em idiomas diferentes:

a) português brasil (pt_BR) e

b) inglês americano (en_US).

Os referidos arquivos devem ser instalados no pacote "*br.aiec.i18n*". Veja as instruções a seguir.

09

a) text_pt_BR.properties

```
# Esse arquivo deve ser instalado no pacote "br.aiec.i18n"
#
# Nome do arquivo: text_pt_BR.properties
#

site.titulo = Sistema i18n (Internacionalização)
site.titulo.historico = Histórico de Visitantes
site.saudacao = Seja bem Vindo

site.campo.indice = Indice
site.campo.nome = Nome
site.campo.idade = Idade
site.campo.dataNascimento = Data de Nascimento
```

b) text.properties

```
# Esse arquivo deve ser instalado no pacote "br.aiec.i18n"
#
# Esse é considerado o arquivo DEFAULT de idiomas (en_US)
#
# Nome do arquivo: text.properties
#

site.titulo = i18n System
site.titulo.historico = History of Visits
site.saudacao = Welcome

site.campo.indice = Index
site.campo.nome = Name
site.campo.idade = Age
site.campo.dataNascimento = Birthday
```

10

3.2 - FormularioLivroVisitas.jsp

```
<%-- Está página é a primeira página da aplicação web de exemplo.
---- É aquela que constrói o formulário HTML de entrada dos dados
--%>
<%@page language="java"
            contentType="text/html; charset=UTF-8"
            pageEncoding="UTF-8"%>

<%--Importando as taglibs core e formating JSTL --%>
```

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

<%--Definindo a variável idioma por sessão --%>
<c:set var="idioma" value="${not empty param.language ? param.language
: not empty language ? language : pageContext.request.locale}"
scope="session" />

<%--Alterando o locale --%>
<fmt:setLocale value="${idioma}" />

<%--Definindo o pacote.nomeArquivo (fqdn) das mensagens --%>
<fmt:setBundle basename="br.aiec.i18n.text" />

<!DOCTYPE html>
<html lang="${idioma}">
      <head>
            <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
            <title>
                  <fmt:message key="site.titulo" />
            </title>
      </head>
      <body>
            <h1 align="center">
                  <fmt:message key="site.saudacao"/>
            </h1>
            <form action="BeanPessoaLivroVisitas.jsp" method="post">
                  <fmt:message key="site.campo.nome" />
                  <input type="text" name="nome"><br />

                  <fmt:message key="site.campo.dataNascimento" />
                  <input type="text" name="dataNascimento">
(dd/MM/yyyy)<br />

                  <input type="submit">
            </form>
      </body>
</html>
```

11

3.3 - BeanPessoaLivroVisitas.jsp

```
<%-- Esta página não produz qualquer resultado para a interface do
usuário.
---- O objetivo da mesma ter sido utilizada foi, apenas, para fins
didáticos.
---- Nesta página são demonstrados quatro coisas:
```

```
----
---- 1) uso de tags de ação padrão
---- 2) criação do bean pessoa a partir dos parâmetros do formulário
html
---- 3) uso da taglib de formatação para converter a data
---- 4) redirecionamento interno da requisição
--%>
<%@page language="java"
          contentType="text/html; charset=UTF-8"
          pageEncoding="UTF-8"%>
<%@page import="br.aiec.Pessoa"%>
<%--Importando a taglibs formating JSTL --%>
<%@taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
<!DOCTYPE html>
<html lang="${idioma}">
     <head>
          <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
          <title></title>
     </head>
     <body>
          <jsp:useBean id="pessoaBean" class="br.aiec.Pessoa"
scope="request">
               <%--Utilizando o elemento de ação para atribuir o
valor do nome que foi recebido como parâmetro --%>
               <jsp:setProperty name="pessoaBean" property="nome"
                    value="${param.nome}" />
               <%--Utilizando a taglib parseDate para converter o
parâmetro date recebido em formato String --%>
               <fmt:parseDate pattern="dd/MM/yyyy"
var="dtNascimento"
                    value="${param.dataNascimento}" />
               <%--Utilizando o elemento de ação para atribuir o
valor da data convertida --%>
               <jsp:setProperty name="pessoaBean"
property="dataNascimento"
                    value="${dtNascimento}" />
          </jsp:useBean>

          <%-- Redirecionando para o servlet de historico de visitas
--%>
          <jsp:forward page="/historicoLivroVisitas.do" />
     </body>
</html>
```

3.4 - HistoricoLivroVisitasDemoServlet.java

```
package br.aiec;

import java.io.IOException;
import java.util.LinkedList;
import java.util.List;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(name="HistoricoLivroVisitas",
urlPatterns="/historicoLivroVisitas.do")
public class HistoricoLivroVisitasDemoServlet extends HttpServlet {

	private List<Pessoa> historicoVisitantes = new
LinkedList<Pessoa>();
	@Override
	public void init() throws ServletException {
		//Compartilhando a lista de histórico no contexto da
aplicação
		getServletContext().setAttribute("historyVisits",
historicoVisitantes);
	}
	@Override
	protected void doPost(HttpServletRequest request,
HttpServletResponse response)
			throws ServletException, IOException {
		//Recuperando o bean criado na JSP
		Pessoa pessoaBean =
(Pessoa)request.getAttribute("pessoaBean");
		//Adicionando o bean ao historico de visitantes
		historicoVisitantes.add(pessoaBean);
		//Redirecionando para uma JSP exibir o histórico de
visitantes
		RequestDispatcher dispatcher =
request.getRequestDispatcher("ExibirHistorico.jsp");
		dispatcher.forward(request, response);
	}
}
```

13

3.5 - Pessoa.java

```
package br.aiec;

import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;

public class Pessoa {

	private String nome;
	private Date dataNascimento;

	public String getNome() {
		return nome;
	}

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

	public Date getDataNascimento() {
		return dataNascimento;
	}

	public void setDataNascimento(Date dataNascimento) {
		this.dataNascimento = dataNascimento;
	}

	public int getIdade(){
		Calendar hoje = GregorianCalendar.getInstance();
		hoje.setTime(new Date());

		Calendar nascimento = GregorianCalendar.getInstance();
		nascimento.setTime(dataNascimento);

		int quantidadeAnos = hoje.get(Calendar.YEAR) -
nascimento.get(Calendar.YEAR);

		nascimento.add(Calendar.YEAR, quantidadeAnos);

		if(nascimento.after(hoje)){
			quantidadeAnos--;
		}

		return quantidadeAnos;
	}

}
```

Observe que, diferentemente do módulo anterior, onde a classe Pessoa possuía as regras de conversão de datas, agora as referidas regras fazem parte da camada lógica de apresentação, conforme poderá ser visualizado na página logo a seguir: “ExibirHistorico.jsp”.

14

3.6 ExibirHistorico.jsp

```
<%@page language="java"
        contentType="text/html; charset=UTF-8"
        pageEncoding="UTF-8"%>

<%@page import="br.aiec.Pessoa"%>

<%--Importando as taglibs core e formating JSTL --%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<%--Alterando o locale --%>
<fmt:setLocale value="${idioma}" />

<%--Definindo o pacote.nomeArquivo (fqdn) das mensagens --%>
<fmt:setBundle basename="br.aiec.i18n.text"/>

<!DOCTYPE html>
<html lang="${idioma}">
      <head>
            <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
            <title>
                  <fmt:message key="site.titulo"/>
            </title>
      </head>
      <body>
            <h1>
                  <fmt:message key="site.titulo.historico"/>
            </h1>

            <table border="1">
                  <thead>
                        <tr>
                              <td><fmt:message
key="site.campo.indice"/></td>
                              <td><fmt:message
key="site.campo.nome"/></td>
                              <td><fmt:message
key="site.campo.dataNascimento"/></td>
                              <td><fmt:message
key="site.campo.idade"/></td>
```

```
                    </tr>
                </thead>
                <tbody>
                    <%--Esse laço percorre a lista de visitantes,
exibindo cada um deles --%>
                    <c:forEach var="pessoa" items="${historyVisits}"
varStatus="i">
                        <tr>
                            <td>${i.count}</td>
                            <td>${pessoa.nome}</td>
                            <td><fmt:formatDate
pattern="dd/MM/yyyy" value="${pessoa.dataNascimento}" /></td>
                            <td>${pessoa.idade}</td>
                        </tr>
                    </c:forEach>
                </tbody>
            </table>
        </body>
</html>
```

15

RESUMO

Vimos neste módulo o modelo de funcionamento das tags padronizadas denominadas de JSTL. Vimos também que a JSTL é uma coleção de tags JSP que encapsula diversas funcionalidades de uso contínuo e comum presentes em inúmeras aplicações web. O uso de *scripts* torna o código das páginas JSP complexo e confuso em função da mistura de dois estilos/técnicas de programação diferentes: imperativo e declarativo. Desta forma, a API JSTL encapsulou em tags simples, toda a funcionalidade de construção dinâmica de páginas Web. Aprendemos como descrever a sintaxe e a semântica dos diversos elementos definidas pelas três, dentre as seis disponíveis, taglibs especificadas: core, i18n formating e functions.