

## **Curso de JSP**

- 1. O que   JSP**
- 2. Simbologia - Parte 1**
- 3. Simbologia - Parte 2**
- 4. Comandos Condicionais**
- 5. Comandos de Repeti o**
- 6. Formul rios**
- 7. Tratando Cookies**
- 8. Tratamento de Erros**
- 9. JavaBeans**
- 10. M todos**

**Ap ndice A: Servlets**

**Ap ndice B: Tags**

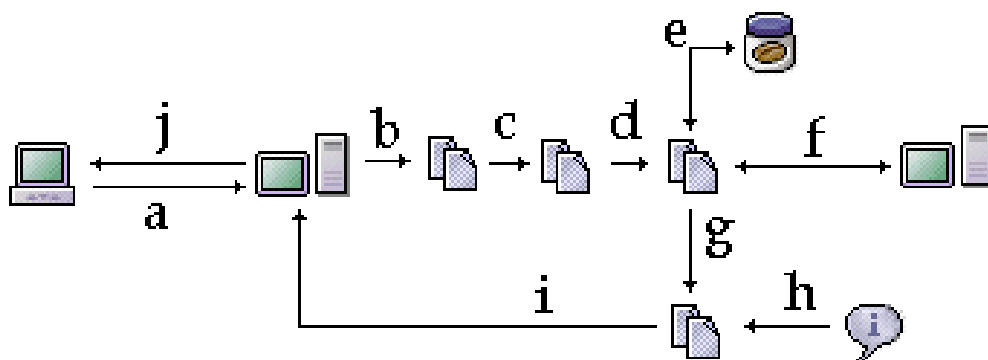
## 1. O que é JSP ?

A Java Server Pages (JSP) é uma tecnologia para o desenvolvimento de aplicações WEB, criando páginas chamadas dinâmicas semelhante ao Active Server Pages (ASP), porém tem a vantagem da portabilidade de plataforma podendo ser executado em outros Sistemas Operacionais além dos da Microsoft e da compilação das páginas permitindo que elas rodem muito mais rapidamente. Permite também que o desenvolvedor de sites produzir aplicações que permitem acessos nativos aos mais diversos bancos de dados graças a tecnologia JDBC, acesso a arquivos texto, a captação de informações a partir de formulários, a captação de informações sobre o visitante e sobre o servidor entre diversas outras coisas.

Para aqueles que conhecem a tecnologia Java Servlets verá que o JSP não oferece nada que não se possa conseguir com as Servlets puras. O JSP, entretanto, oferece a vantagem de ser facilmente codificado, facilitando assim a elaboração e manutenção da página dinâmica. Além disso, essa tecnologia permite separar a programação lógica (parte dinâmica) da programação visual (parte estática), facilitando o desenvolvimento de aplicações mais robustas, onde programador e o designer podem trabalhar em um mesmo projeto, mas de forma independente. Outra característica da JSP é produzir conteúdos dinâmicos que podem ser reutilizados.

### Como funciona ?

Inicialmente o cliente faz uma requisição de uma página JSP através de um Browser, esta página então será processada pelo servidor, se for a primeira vez, esta página JSP é transformada em um programa Java (conhecido como Servlet) este é compilado e gerado um bytecode (conhecido por .class) e a partir deste é gerada uma página HTML que é enviada de volta ao browser do cliente, a partir da segunda vez que esta mesma página for acessada é verificado apenas se ocorreu ou não quaisquer mudanças, se não ocorrerem o bytecode é chamado para gerar a página HTML. A figura abaixo ilustra esse funcionamento:



---

a	Todo o processo começa quando um cliente faz uma solicitação de uma página JSP, neste momento, é enviado um objeto do tipo request para o WebServer
b	O WebServer localiza e envia a ação para a página JSP
c	E verificado mudanças nesta, se for a primeira vez é criada um programa Java Servlet
d	O programa Java Servlet é compilado e transformado em um ByteCode (.class)
e	Este .class pode ser auxiliados por pacotes .JAR que carregam JavaBeans
f	Durante a montagem da página HTML podem ocorrer também solicitações a Bancos de Dados
g	A página HTML é gerada
h	Chamadas a TAG's Personalizadas são transformadas neste momento em TAG's comuns para a geração final
i	A página final padrão HTML é enviada de volta ao servidor
j	Que a devolve para o cliente que fez sua solicitação

A JSP usa a linguagem Java como base para sua linguagem de Scripts, utilizando todo o potencial desta e é exatamente por este motivo que a tecnologia JSP se apresenta muito mais flexível e muito mais robusta do que outras plataformas.

### **Requisitos para entender JSP**

Para tirar proveito da JSP obviamente é preciso entender a linguagem Java. Isso porque, a parte lógica da JSP envolve JavaBeans, api JDBC, Tags Personalizadas entre outros componentes que interagem com a plataforma. Portanto, alertamos a todos aqueles que pretendem desenvolver uma aplicação mais sofisticada que é necessário um pouco de programação em Java.

O ideal é que se conheça muito bem HTML pois é comum em grandes aplicações que o Programador JSP e o Designer sejam pessoas diferentes. Portanto, se faz a necessidade de se unir os dois mundos em uma linguagem comum e fácil.

## 2. Simbologia – Parte 1

A maior parte do código de uma página JSP consiste em template text. O template text é similar ao HTML, obedecendo às mesmas regras de sintaxe e é simplesmente passado ao cliente por um servlet criado especialmente para manusear a página. Para construir uma aplicação usando JSP você geralmente escreve o texto HTML e inclui códigos JSP entre tags JSP. As tags normalmente começam com "<%" e terminam com "%>". Vamos mostrar o clássico exemplo do "Olá Mundo".

```
<html>
<body>
<h1><%= request.getParameter("text") %></h1>
</body>
</html>
```

Salve esse código com o nome "exemplo.jsp". Para executar esse código basta digitar em seu browser a URL:

```
http://localhost:8080/exemplo.jsp?text=Ola+Mundo
```

Perceba que foi possível passar o valor do objeto "text" como parâmetro na própria URL e o servidor de JSP montou o código. Se você verificar o código da página que apareceu no seu browser, verá o seguinte:

```
<html>
<body>
<h1>Ola Mundo</h1>
</body>
</html>
```

O que aconteceu foi que o servidor JSP executou os comandos entre as tags especiais "<%" e "%>", gerou e enviou para o browser um código HTML puro.

JSP fornece cinco categorias de tags: diretivas, declarações, expressões, scriptlets e comentários.

As diretivas serão tratadas na parte 2 desse módulo do tutorial. Vamos nos concentrar aqui no estudo das declarações, expressões, scriptlets e comentários. Os três primeiras têm sintaxes e usos similares, mas têm diferenças importantes. Vamos explicar, com alguns exemplos, as semelhanças e diferenças entre elas.

### **Declarações (entre <%! and %>):**

As declarações são usadas para definir variáveis e métodos específicos para uma página JSP. Os métodos e variáveis declaradas podem então ser referenciados por outros elementos de criação de scriptlets na mesma página. A cada declaração deve ser finalizada ou separada por "ponto-e-vírgula" e pode assumir a seguinte sintaxe:

```
<%! declaration %>
```

Veja alguns exemplos:

```
<%! int i = 0; %>  
<%! int a, b; double c; %>  
<%! Circle a = new Circle(2.0); %>
```

Você deve declarar uma variável ou um método antes de usá-lo. O escopo de uma declaração é geralmente o arquivo JSP, mas se for incluído outros arquivos com a diretiva `include`, o escopo se expande para o cover do arquivo incluído.

### Expressões (entre `<%= e %>`)

Podem conter alguma expressão válida da linguagem de script usada nessa página (o padrão é que a Linguagem seja Java), mas sem ponto-e-vírgula. A sintaxe para este elemento de criação de scripts é a seguinte:

```
<%= expression %>
```

Veja alguns exemplos:

```
<%= Math.sqrt(2) %>  
<%= items[i] %>  
<%= a + b + c %>  
<%= new java.util.Date() %>
```

A expressão Java é avaliada (da esquerda para a direita), convertida em String e depois inserida na página. Essa avaliação é feita em tempo de execução (quando a página é solicitada) permitindo fácil e rápido acesso a informação que foi requisitada. Por exemplo, uma exibição de data e hora em que a página é acessada.

Para construir uma expressão em JSP você pode colocar entre as tags qualquer expressão definida na Especificação da Linguagem Java. Ao contrário dos scriptlets (que veremos a seguir), uma expressão não aceita ponto e vírgula e define somente uma expressão da Linguagem.

Veja um exemplo de como ler seu hostname:

```
<%= request.getRemoteHost() %>
```

Para simplificar as expressões, existe um número de variáveis predefinidas que você pode usar e que veremos a seguir.

## Scriptlets (entre <% e %>)

Permitem você escrever trechos de código da Linguagem usada na página.

```
<html>
<body>
.
.
.
<%
String option;
int numberOption = Integer.parseInt(request.getParameter("option"));
if (numberOption == 1) {
    option = "Compra";
} else if (numberOption == 2) {
    option = "Venda";
} else {
    option = "Aluguel";
}
%>
<font face="verdana, arial" size=5>Opção Escolhida: <%= option %> </font>
.
.
.
</body>
</html>
```

Lembre-se que em um script você deve finalizar as expressões através do uso de ponto-e-vírgula. Quando você escreve um script, você pode usar algum dos objetos implícitos do JSP ou das classes importadas através da diretiva page, variáveis ou métodos (declarados entre as tags <%! e %> ou objetos nomeados através da tag <jsp:useBean>. Para testar a página acima basta salvá-la com o nome de "classificados.jsp" e digitar a url ".../classificados.jsp?option=2".

## Comentários

Existem dois tipos principais de comentários que podem ser usados em uma página JSP:

Comentário de Conteúdo: esses comentários são transmitidos de volta para o navegador como parte da resposta de JSP e são visíveis na visualização do código da página. O comentário de conteúdo tem a seguinte sintaxe:

```
<!-- comment -->
```

Aqueles familiarizados com HTML percebem que é a mesma sintaxe de comentário para essa linguagem de marcação. Tais comentários não produzem qualquer output visível, mas podem ser visualizados pelo usuário final através do item "view source" do navegador.

Comentários JSP: não são enviados para o cliente e são visíveis apenas nos arquivos fonte JSP originais. O corpo do comentário é ignorado pelo container JSP. Os comentários JSP podem assumir duas sintaxes:

```
<!-- comment -->
```

e

```
<% /* comment */ %>
```

Esse segundo comentário é introduzido dentro da página através de um scriptlets, usando a sintaxe de comentário nativa da linguagem de criação de scripts, no caso JAVA.

### 3. Simbologia - Parte 2

As diretivas são usadas para fornecer informações especiais ao container JSP sobre a página JSP quando esta é compilada para servlet. As diretivas JSP afetam a estrutura global da classe servlet.

Existem dois tipos principais de diretivas:

**page:** permite situações como importação de classes, customização de super classes servlet entre outras;

**include:** permite que seja inserido o conteúdo de um arquivo no servlet no momento em que o arquivo JSP é traduzido para servlet.

#### Diretiva page

A diretiva page tem a seguinte sintaxe:

```
<%@ page attribute1=valor1 attribute2=valor2 attribute3=... %>
```

Abaixo relacionamos os atributos mais utilizados nas diretivas page:

#### Atributo import

```
import="package.class" ou import="package.class1,...,package.classN".
```

Permite que seja especificado qual o pacote a ser importado. Por exemplo:

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

O atributo import attribute é o único que pode aparecer várias vezes.

#### Atributo isThreadSafe

```
isThreadSafe="true|false"
```

O valor de true (default) indica o processamento normal do servlet quando múltiplas requisições podem ser acessadas simultaneamente na mesma instância de servlet. O valor false indica que o servlet deve implementar SingleThreadModel, como requisição para cada requisição sinalizada ou com requisições simultâneas sendo uma em cada instância.

#### Atributo session

```
session="true|false"
```

O valor de true (default) indica que a variável predefinida session (do tipo HttpSession) deve estar ligada a sessão existente, caso não exista uma sessão, uma nova sessão deve ser criada para ligá-la. O valor false indica que sessões não devem ser usadas. Aqui está a implementação do contexto.

### **Atributo buffer**

```
buffer="sizekb|none"
```

Especifica o tamanho do buffer para o JspWriter out. O buffer padrão é definido pelo servidor.

### **Atributo autoFlush**

```
autoflush="true|false"
```

O valor de true (default) indica se o buffer deve ser esvaziado quando estiver cheio. O valor false, indica que uma exceção deve ser mostrada quando ocorrer overflows.

### **Atributo extends**

```
extends="package.class"
```

Define se a super classe do servlet deve ser gerada.

### **Atributo info**

```
info="message"
```

Define uma string que pode ser recuperada pelo método getServletInfo. Com esse atributo o autor pode adicionar uma cadeia de documentação à página que sumariza sua funcionalidade. O valor padrão para o atributo info é a cadeia vazia.

### **Atributo errorPage**

```
errorPage="url"
```

Especifica que a página JSP deve processar algum Throwable, mas não carregá-lo na página corrente.

### **Atributo isErrorPage**

```
isErrorPage="true|false"
```

Define se uma página pode atuar como uma página de erro para uma outra página JSP. O default é false.

### Atributo language

```
language="java"
```

Especifica a linguagem que está sendo usada. Por enquanto o JSP suporta somente Java.

### Diretiva include

A diretiva include permite que sejam incluídos arquivos na hora em que a página JSP é traduzida no servlet. Uma directive include é algo como:

```
<%@ include file="relative url" %>
```

Essa diretiva pode ser implementada de acordo com o seguinte exemplo: muitos sites têm uma barra de navegação em cada página. Devido a problemas com frames HTML isto é normalmente implementado com uma tabela repetindo o código HTML em cada página do site. Esta diretiva supre essa necessidade de minimizar os esforços de manutenção.

```
<html>
<body>
<%@ include file="navbar.html" %>
<!--Parte específica da página ... -->
</body>
</html>
```

navbar.html

```
<font face="verdana" size=1 color="#ffffcc">
<a href="home.jsp">HOME</a> -
<a href="secao_01.jsp">SEÇÃO 01</a>-
<a href="secao_02.jsp">SEÇÃO 02</a>-
<a href="secao_02.jsp">SEÇÃO 02</a>
</font>
```

```
<html>
<body>

<font face="verdana" size=1 color="#ffffcc">
<a href="home.jsp">HOME</a> -
<a href="secao_01.jsp">SEÇÃO 01</a> -
<a href="secao_02.jsp">SEÇÃO 02</a> -
<a href="secao_02.jsp">SEÇÃO 02</a>
</font>

<!--Parte específica da página ... -->

</body>
</html>
```

**Resumo geral da sintaxe do JSP**

Declarações	Declara variáveis e métodos válidos no script daquela página.	<%! declaração; [declaração;]+ ... %>
Expressões	Contém uma expressão válida no script daquela página.	<%= expressão %>
Scriptlet	Contém um fragmento de código válido no script daquela página.	<% fragmento de um código com uma ou mais linhas %>
Comentário HTML	Cria um comentário que é enviado para o cliente viabilizar no código da página.	<!-- comentário [<%= expressão %>] ->
Comentário JSP	É visto apenas no fonte do JSP mas não é enviado para o cliente.	<%-- comentário --%> ou <% /* comentário */ %>
Diretiva "Include"	Inclue um arquivo estático, analisando os elementos JSP daquela página.	<%@ include file="URL relativa" %>
Diretiva "Page"	Define atributos que serão aplicados a uma página JSP.	<%@ page [ atributo = valor(es) ] %> atributos e valores: - language="java" - extends = "package.class" - import = "{package.class"   package.* }, ..." ] - session = "true   false" - buffer = "none   8kb   sizekb" - autoFlush = "true   false" - isThreadSafe = "true   false" - info = "text" - errorPage="relativeURL" - contentType = "{mimeType [; charset = characterSet ] text/html; charset = isso-8859-1}" - isErrorPage = "true   false"
Diretiva Taglib	Define uma biblioteca tag e um prefixo para uma tag padrão usada na página JSP.	<%@ taglib uri="URIToTagLibrary" prefix="tagPrefix" %>
<tagPrefix:name>	Acessa um padrão de funcionalidade de uma tag.	<tagPrefix:name attribute="value" + ... />  <tagPrefix:name attribute="value" + ... > other tags and data </tagPrefix:name>
<jsp:forward>	Redireciona uma requisição para um arquivo HTML, JSP ou servlet para processar.	<jsp:forward page="{relativeURL   <%= expressão %>}"

## 4. Comandos Condicionais

Uma das grandes vantagens de qualquer linguagem de programação é a utilização de controles de fluxo (condicionais e de repetição) para executar diferentes partes de um programa baseado em condições definidas.

Como foi dito em lições anteriores, o JSP nos permite construir páginas unindo HTML e comandos Java. Mostraremos nessa lição como utilizar as rotinas condicionais do Java em páginas JSP.

### Comando if

O comando if contém a palavra-chave if, seguida de um teste booleano, obrigatoriamente entre parênteses, e de uma instrução (ou bloco de instruções) para ser executada caso a condição seja verdadeira, e não obrigatoriamente a palavra-chave else, com uma instrução (ou bloco de instruções) para ser executada caso a condição seja falsa.

```
<html>
<body>

<!-- AS DUAS PROXIMAS LINHAS CRIA UMA VARIÁVEL
QUE RECEBE O VALOR DA DATA ATUAL DO SERVIDOR --%>

<%
java.util.Date dateNow = new java.util.Date();
int hourNow = dateNow.getHours();
%>

<!-- AS PROXIMAS LINHAS MISTURA HTML E JAVA
E PRODUZ COMO RESULTADO UMA SAUDAÇÃO QUE DEPENDE DA HORA --%>

<% if ((hourNow >= 5) && (hourNow < 13)) { %>
<font face="verdana">Bom Dia!!!</font>

<% } else if ((hourNow >= 13) && (hourNow < 19)) { %>
<font face="verdana">Bom Tarde!!!</font>

<% } else if ((hourNow >= 19) && (hourNow < 24)) { %>
<font face="verdana">Bom Noite!!!</font>

<% } else { %>
<font face="verdana">Boa Madrugada!!!</font>
<% } %>

</body>
</html>
```

### Operador Condicional

Uma alternativa para o uso das palavras-chaves if e else em uma instrução condicional é usar o operador condicional, algumas chamado de operador ternário.

O operador ternário é uma expressão, significando que ele devolve um valor (diferentemente do if mais geral, o qual pode resultar em qualquer instrução ou bloco sendo executado). O operador ternário é muito útil para condicionais muito curtos ou simples, e tem a seguinte formato:

```
condicao ? caso-verdadeiro : caso-falso
```

Vamos como podemos usar o operador condicional para obter o mesmo efeito do código anterior:

```
<html>
<body>

<!-- AS DUAS PROXIMAS LINHAS CRIA UMA VARIÁVEL
QUE RECEBE O VALOR DA DATA ATUAL DO SERVIDOR --%>

<%
java.util.Date dateNow = new java.util.Date();
int hourNow = dateNow.getHours();
%>

<%
String mensagem;
mensagem = ((hourNow < 12)? "Onde você vai almoçar hoje?" : "Onde você almoçou
hoje?");
%>
<font face="verdana">Olá, Tudo bem? <%= mensagem %></font>

</body>
</html>
```

Apesar deste código parecer, a uma primeira vista, um pouco menos fácil de entender que o anterior, ele é menor e portanto mais prático.

## Comando switch

No comando switch, o teste (qualquer tipo primitivo em Java) é comparado com cada valor em questão. Se um valor coincidente é achado, a instrução (ou instruções) depois do teste é executada. Se nenhum valor for encontrado, a instrução default é executada. Vamos analisar o exemplo abaixo:

```
<html>
<body>

<!-- AS DUAS PROXIMAS LINHAS CRIA UMA VARIÁVEL
QUE RECEBE O VALOR DA DATA ATUAL DO SERVIDOR --%>

<%
java.util.Date dateNow = new java.util.Date();
int monthNow = (dateNow.getMonth()+1);
%>
```

```
<%  
String mes;  
  
switch(monthNow){  
case 1: mes="Janeiro"; break;  
case 2: mes="Fevereiro"; break;  
case 3: mes="Março"; break;  
case 4: mes="Abril"; break;  
case 5: mes="Maio"; break;  
case 6: mes="Junho"; break;  
case 7: mes="Julho"; break;  
case 8: mes="Agosto"; break;  
case 9: mes="Setembro"; break;  
case 10: mes="Outubro"; break;  
case 11: mes="Novembro"; break;  
default: mes="Dezembro"; break;  
}  
%>  
  
<font face="verdana"> Nós estamos em <%= mes %></font>  
  
</body>  
</html>
```

Esse código atribui a variável `monthNow` o valor do mês atual. Note que na instrução nós incrementamos o mês em uma unidade porque o método `"getMonth()"` retorna 0 para o mês de janeiro, 1 para fevereiro e assim por diante.

Observe que a limitação significativa no Java é que os testes e valores podem ser somente de tipos primitivos. Você não pode usar tipos primitivos maiores (`long`, `float`) ou objetos dentro de um `switch`, nem pode testar para nenhuma outra relação senão a igualdade. Isso limita a utilidade do `switch` para todos os casos exceto os mais simples, `if`'s aninhados podem funcionar para qualquer espécie de teste em qualquer tipo.

## 5. Comandos de Repetição

Essa lição trata as rotinas de repetição do Java que poderão ser usadas em páginas JSP.

### Comando for

O comando for, repete uma instrução ou um bloco de instruções algum número de vezes até que a condição seja satisfeita. Comandos for são freqüentemente usados para simples iterações na qual você repete um bloco de instruções um certo número de vezes e então pára; mas você pode usar o for para qualquer espécie de laço de repetição. O for no Java possui a seguinte sintaxe:

```
for (inicial; condição; incremento) {  
    Corpo do Laço  
}
```

Onde:

inicial - é uma expressão que inicia o começo do laço;

condição - é o teste que ocorre depois de cada passo do laço;

incremento - é qualquer chamada de expressão ou função.

O código abaixo mostra um simples exemplo da utilização do comando for:

```
<html>  
<body>  
  
<% for (int i = 2; i < 8 ; i++) { %>  
<font size=<%= i %>>Bom Dia!!! - Fonte: <%= i %></font>  
<% } %>  
  
</body>  
</html>
```

### Comando While

O while é usado para repetir uma instrução ou bloco de instruções até que uma condição particular seja verdadeira. Comandos while possuem a seguinte sintaxe:

```
while (condition) {  
    Corpo do laço  
}
```

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

```
<%  
int i = 0;  
while (i < 8) { %>  
<font size=<%= i %>>Bom Dia!!! - Fonte: <%= i %></font><br>  
<%  
i++;  
} %>  
  
</body>  
</html>
```

Observe que se a condição é inicialmente falsa na primeira vez que é testada o corpo do while não será executado. Se você precisa executar o corpo, pelo menos uma vez, você pode usar o comando do..while que será visto a seguir.

## Comando Do...While

O comando é exatamente como o while, exceto pelo fato de o laço executar uma dada instrução ou bloco até que uma condição seja falsa. A diferença principal entre os dois é que o comando while testa primeiro a condição antes de iniciar o laço. Enquanto que o comando do...while executa pelo menos uma vez antes de testar a condição.

O do...while possui a seguinte sintaxe:

```
do {  
    Corpo do laço  
} while ( condição );
```

```
<html>  
<body>  
  
<%  
int i = 0;  
do { %>  
<font size=<%= i %>>Bom Dia!!! - Fonte: <%= i %></font><br>  
<%  
i++;  
}  
while (i < 8); %>  
  
</body>  
</html>
```

## Interrupção de Laços

Em todos os comandos (for, while e do) terminam quando a condição que você está testando é atingida. Porém, em alguma situação, você desejará sair do laço antes do seu término normal. Para isso, pode-se usar as palavras chaves break e continue. O comando break para imediatamente a execução do laço corrente. Se você tiver laços aninhados dentro de outros laços mais externo; caso contrário, o programa simplesmente continua a execução da próxima instrução após o laço.

```
<html>
<body>

<%
int j;
for (int i = 1; i < 8 ; i++) {
j = i%5;
if (j == 0){ break; }
}%>
<font size=<%= i %>>Bom Dia!!! - Fonte: <%= i %></font><br>
<% } %>

</body>
</html>
```

No exemplo acima, o código dentro do laço é executado até que a variável *i* assumira um valor que seja divisível por 5. Quando isso ocorrer o laço será finalizado e o programa continua a ser executado após o final do comando `for`. Nesse caso o resultado obtido foi uma a geração de quatro linhas de código html (ao invés de 7). Após gerar a página, o servidor JSP enviará para o navegador o seguinte código:

```
<html>
<body>
<font size=1>Bom Dia!!! - Fonte: 1</font><br>
<font size=2>Bom Dia!!! - Fonte: 2</font><br>
<font size=3>Bom Dia!!! - Fonte: 3</font><br>
<font size=4>Bom Dia!!! - Fonte: 4</font><br>
</body>
</html>
```

O comando `continue` interrompe o laço e começa novamente na próxima iteração. `continue` é útil quando você quer ter situações de casos especiais dentro do laço.

```
<html>
<body>

<%
int j;
for (int i = 1; i < 8 ; i++) {
j = i%5;
if (j == 0){ continue; }
}%>
<font size=<%= i %>>Bom Dia!!! - Fonte: <%= i %></font><br>
<% } %>

</body>
</html>
```

Esse código é semelhante ao anterior, contudo, ao invés de finalizar a execução do comando `for` ele será reiniciado em uma nova interação. O resultado obtido é a geração de 6 linhas de código html (a linha cujo teste do `if`

foi verdadeiro, ou seja, onde a variável *i* assumiu um valor divisível por 5, não será gerada). O servidor JSP irá enviar para o navegador o seguinte código:

```
<html>
<body>
<font size=1>Bom Dia!!! - Fonte: 1</font><br>
<font size=2>Bom Dia!!! - Fonte: 2</font><br>
<font size=3>Bom Dia!!! - Fonte: 3</font><br>
<font size=4>Bom Dia!!! - Fonte: 4</font><br>
<font size=6>Bom Dia!!! - Fonte: 6</font><br>
<font size=7>Bom Dia!!! - Fonte: 7</font><br>
</body>
</html>
```

## 6. Formulários

Os formulários são ferramentas úteis e muito usadas em diversas aplicações, tais como: cadastro registros em um banco de dados, validação de senhas, envio de e-mail, envio de dados de uma pesquisa, entre outros. Hoje em dia é muito difícil desenvolver uma aplicação para WEB que não exija seu uso. Nesta lição aprenderemos como manipular formulários em aplicações JSP.

### Formulários em HTML

Apresentamos abaixo um código para mostrar o formato de um formulário HTML e de seus objetos.

```
<html>
<body>

<!-- cabeçalho do formulário -->
<form name="nomedoformulario" action="paginajsp.jsp" method="get">

<!-- caixa de texto -->
<input type="text" name="variavel1" size=40 maxlength=40>

<!-- caixa de texto para senha -->
<input type="password" name="variavel2" size=40 maxlength=40>

<!--objeto do tipo radio -->
<input type="radio" name="variavel2" value="valordavariavel">Texto da Varivavel
2

<!--objeto do tipo checkbox -->
<input type="checkbox" name="variavel3" value="xxxxx"> Texto da Varivavel 3

<!--objeto do tipo select -->
<select name="variavel4">
<option value="valor1">Valor 1
<option value="valor2">Valor 2
<option value="valor3">Valor 3
</select>

<!-- area de texto -->
<textarea name="variavel5" cols=40 rows=2>
Texto da Variavel 5
</textarea>

<!-- objeto hidden, para enviar dados que o usuário não vê no formulário -->
<input type="hidden" name="asd" value="asd">

<!-- botão -->
<input type="button" value="textodobotao">

<!-- botao de limpar informações do formulário -->
<input type="submit" value="limpar">

<!-- botao de enviar formulário -->
<input type="submit" value="ok">
```

```
<!-- imagem colocada para funcionar com botao de envio de formulário -->
<input type="image" src="pathdaimage/image.gif">

<!-- objeto para anexar arquivo -->

<input type="file" name="asdas" accept="asd">

</form>

</body>
</html>
```

É importante fazermos algumas observações a cerca do código acima:

- no cabeçalho do formulário, indicamos através de `action="pathdoarquivo/paginajsp.jsp"` o arquivo JSP que receberá os seus dados.

- cada objeto do formulário recebe um nome. Deve-se tomar bastante cuidado ao nomear tais objetos, isto porque, como sabemos, as variáveis Java são sensíveis maiúscula/minúscula. Portanto, os objetos:

```
<input name="variavel1" type="text" value="">
<input name="Variavel1" type="text" value="">
```

São objetos diferentes pois receberam nomes diferentes (`variavel1` e `Variavel1`).

## Envio de dados

Neste exemplo (bastante simples) veremos como enviar dados a partir de um formulário a uma página JSP.

```
<html>
<body>

<center><h1> <%= request.getParameter("teste") %> </h1></center>

<form action="teste.jsp" method=get>
<input type="text" name="teste" size=40 maxlength=40><br>
<input type="submit" value="enviar">
</form>

</body>
</html>
```

A página jsp acima contém um formulário que envia para ela mesma. O valor digitado em uma caixa de texto será mostrado como título da página. Observe como fizemos isso:

A página para qual nós enviaremos os dados do formulário é designada no cabeçalho do formulário:

```
<form action="teste.jsp" method=get>
```

O nome do objeto caixa de texto caixa de texto ("teste") é usado na expressão `request.getParameter("teste")`. Note que se usássemos `request.getParameter("Teste")` (com T maiúsculo), a página não iria retornar o valor digitado na caixa de texto.

O próximo exemplo é formado por dois arquivos. O primeiro pode contém apenas códigos HTML e o segundo contém códigos HTML e JSP.

```
<html>
<body>

<h3>Qual o mês do seu aniversário?</h3>
<form action="recebe_mes.jsp" method=get>
<select name="mesNasceu">
<option value="1">Janeiro
<option value="2">Fevereiro
<option value="3">Março
<option value="4">Abril
<option value="5">Maio
<option value="6">Junho
<option value="7">Julho
<option value="8">Agosto
<option value="9">Setembro
<option value="10">Outubro
<option value="11">Novembro
<option value="12">Dezembro
</select>

<input type="submit" value="enviar">
</form>

</body>
</html>
```

```
<%@ page import=java.util.Date %>
<%@ page import=java.lang.String %>
<%
String msg = "";
String mesString = request.getParameter("mesNasceu");
int mes = Integer.parseInt(mesString);
Date dateNow = new Date();
int monthNow = dateNow.getMonth() + 1;
mes -= monthNow;
if (mes == 1)
    msg = "Falta apenas"+ mes +" mês para o seu aniversário.";
if (mes == -1)
    msg = "Seu aniversário foi no mês passado";
if (mes > 1)
    msg = "Faltam "+ mes +" meses para o seu aniversário.";
if (mes == 0)
    msg = "Oba... estamos no mês do seu aniversário.";
else if (mes < 1) {
    mes *= -1;
    msg = "Seu aniversário foi a " + mes + " meses atrás.";
```

```
}
%>
<html>
<body>
<center>
<h3><%= msg %></h3>
<br><br><br>
<a href="Javascript:history.back(-1)">voltar</a>
</center>
</body>
</html>
```

O exemplo acima é um pouco menos simples que o primeiro. O arquivo "envia\_mes-jsp" contém um formulário com um objeto select que envia o mês que a pessoa nasceu. Após escolher o mês e clicar no botão "ok", o formulário chama a página "recebe\_mes.jsp" e envia seus dados para ela. Esta segunda página é um pouco menos simples que a primeira. Vamos analisar o que ela faz.

Nas primeiras linhas utilizamos as tags "page import" para indicar quais classes iremos utilizar em nossa página:

```
<%@page import=java.util.Date %>
<%@page import=java.lang.String %>
```

Cinco objetos são criados e inicializados.

Usamos o método "request.getParameter('nomedoparametro')" com a finalidade de pegar o valor passado para a página através de algum formulário ou passando diretamente em sua URL. O segundo objeto foi inicializado utilizando esse método para pegar o valor passado pelo formulário:

```
String mesString = request.getParameter("mesNasceu");
```

O valor passado através de um formulário ou através da URL da página é sempre do tipo String. Ao inicializarmos o terceiro objeto, o método "Integer.parseInt(variavelString)" transformou o valor contido na variável mesString em Inteiro.

```
int mes = Integer.parseInt(mesString);
```

O penúltimo objeto criado é do tipo Date (daí a importância de termos importado a classe java.util.Date na primeira linha de nossa página). Ele é inicializado com a hora local do servidor.

```
Date dateNow = new Date();
```

Na inicialização do último objeto utilizamos o método "dateNow.getMonth()" que retorna um inteiro indicando o valor da variável. Somamos 1 ao valor

retornado a partir desse método porque ele retorna 0 para janeiro, 1 para fevereiro e assim por diante.

```
int monthNow = dateNow.getMonth() + 1;
```

Cinco teste são efetuados dentro de um script (<% e %>). Eles são usados para definir o valor que a variável "msg" terá, ou seja, a partir dos testes, decidiremos qual mensagem será exibida na tela.

Após efetuar os testes, o texto HTML é inserido na página.

Uma expressão (<%= %>) é usada para exibir o valor da variável "msg":

```
<%= msg %>
```

## 7. Tratando Cookies

Cookie é um mecanismo padrão fornecido pelo protocolo HTTP e que permite gravarmos pequenas quantidades de dados persistentes no navegador de um usuário. Tais dados podem ser recuperados posteriormente pelo navegador. Esse mecanismo é usado quando queremos recuperar informações de algum usuário. Com os cookies, pode-se reconhecer quem entra num site, de onde vem, com que periodicidade costuma voltar. Para se ter uma idéia de como eles fazem parte da sua vida, dê uma olhada na sua máquina. Se você usa o Internet Explore 5.1, vá a `c:\windows\cookies`. No Communicator 4.7, os cookies ficam em `c:\arquivos de programas\netscape\users`. Os cookies em si não atrapalham ninguém, se propriamente usados.

Como padrão, os cookies expiram tão logo o usuário encerra a navegação naquele site, porém podemos configurá-los para persistir por vários dias. Além dos dados que ele armazena, um cookie recebe um nome; um servidor pode então definir múltiplos cookies e fazer a identificação entre eles através dos seus nomes.

Os cookies são associados ao URL da página que os manipula.

### Gerenciando Cookies

Os cookies são definidos por um servidor da web. Quando um usuário solicita um URL cujo servidor e diretório correspondam àqueles de um ou mais de seus cookies armazenados, os cookies correspondentes são enviados de volta para o servidor. As páginas JSP acessam os seus cookies associados através do método `getCookies()` do objeto implícito `request`. De forma similar, as páginas JSP podem criar ou alterar cookies através do método `addCookie()` do objeto implícito `response`. Esses métodos são resumidos na tabela abaixo:

Objeto Implícito	Método	Descrição
<code>request</code>	<code>getCookies()</code>	retorna uma matriz de cookies acessíveis da página
<code>response</code>	<code>addCookie()</code>	envia um cookie para o navegador para armazenagem/modificação

### A classe Cookie

Manipulamos um cookie através de instâncias da classe `javax.servlet.http.Cookie` (calma, você não precisa se preocupar com isso. É que o container JSP insere o comando `"import javax.servlet.http.*;"` automaticamente no servlet associado que é gerado na compilação da página JSP).

Essa classe fornece apenas um tipo de construtor que recebe duas variáveis do tipo String, que representam o nome e o valor do cookie. O cookie tem a seguinte sintaxe:

```
Cookie cookie = new Cookie("nome da fera" , "valor da fera");
```

Abaixo apresentamos os métodos fornecidos pela classe Cookie:

Método	Descrição
getName()	retorna o nome do cookie
getValue()	retorna o valor armazenado no cookie
getDomain()	retorna o servidor ou domínio do qual o cookie pode ser acessado
getPath()	retorna o caminho deURL do qual o cookie pode ser acessado
getSecure()	indica se o cookie acompanha solicitações HTTP ou HTTPS.
setValue()	atribui um novo valor para o cookie
setDomain()	define o servidor ou domínio do qual o cookie pode ser acessado
setPath(nome do path)	define o caminho de URL do qual o cookie pode ser acessado
setMaxAge(inteiro)	define o tempo restante (em segundos) antes que o cookie expire
setSecure(nome)	retorna o valor de um único cabeçalho de solicitação como um número inteiro

Depois de construir uma nova instância, ou modificar uma instância recuperada através do método `getCookies()`, é necessário usar o método `addCookie( )` do objeto `response`, com a finalidade salvar no navegador do usuário as alterações feitas no cookie. Para apagar um cookie utilizamos a seguinte técnica: chamamos o método `"setMaxAge(0)"` com valor zero e depois mandamos gravar chamando o método `"addCookie( )"`. Isso faz com que o cookie seja gravado e imediatamente (após zero segundos) expira.

### Definindo um cookie

O primeiro passo, então, ao usar um cookie dentro de uma página, é defini-lo. Isto é feito criando uma instância da classe `Cookie` e chamando os métodos "sets" para definir os valores de seus atributos.

```
<%
String email = request.getParameter("email");
String cookieName = "cookieJSP";
Cookie cookieJSP = new Cookie(cookieName, email);
```

```

cookieJSP.setMaxAge(7 * 24 * 60 * 60); //define o tempo de vida como 7 dias
(604800 segundos)
cookieJSP.setVersion(0); //versão 0 da especificação de cookie
cookieJSP.setSecure(false); //indica que o cookie deve ser transferido pelo
protocolo HTTP padrão
cookieJSP.setComment("Email do visitante"); //insere um comentário para o cookie
response.addCookie(cookieJSP); //grava o cookie na máquina do usuário
%>
<html>
<head>
<title>Grava Cookie</title>
</head>
<body>
<h1>Grava Cookie</h1>
Esta página grava um cookie na sua máquina.<br>
<a href='readcookie.jsp'>Lê conteúdo do cookie</a>
</body>
</html>

```

Vamos criar uma página html com um formulário que irá fornecer um email que será gravado pela página "addcookie.jsp":

```

<html>
<body>
<form action="addcookie.jsp">
<input type="text" name="email">
<input type="submit" value="ok">
</body>
</html>

```

No exemplo acima o cookie é identificado pelo nome cookieJSP e recebe o valor passado pelo usuário através de um formulário.

## Recuperando um Cookie

A página jsp vista anteriormente tem a finalidade de receber um valor (email) passado através de um formulário de uma página html. Este valor é armazenado de forma persistente em um cookie, e pode ser acessado pelas outras páginas JSP que compartilham o domínio e o caminho originalmente atribuídos ao cookie. Os cookies são recuperados através do método `getCookies()` do objeto implícito `request`. A página abaixo mostra um exemplo de recuperação do valor de um cookie.

```

<%
String cookieName = "cookieJSP";
Cookie listaPossiveisCookies[] = request.getCookies();
Cookie cookieJSP = null;
if (listaPossiveisCookies != null) {
//quando não existe cookies associados o método getCookies() retorna um valor
null
    int numCookies = listaPossiveisCookies.length;
    for (int i = 0 ; i < numCookies ; ++i) {
        if (listaPossiveisCookies[i].getName().equals(cookieName)) { //procura pelo
cookie
            cookieJSP = listaPossiveisCookies[i];

```

```
        break;
    }
}
}
%>
<html>
<body>
<h1>Lê Cookie</h1>
<% if (cookieJSP != null) { %>
A pagina "addcookie" gravou o seguinte email: <%= cookieJSP.getValue() %>
<% }
else { %>
O cookie não gravou ou o prazo do cookie expirou.
<% } %>
</body>
</html>
```

O primeiro scriptlet nesta página recupera os cookies associados aquela página e depois tenta encontrar um cookie identificado pelo nome "cookieJSP".

### Considerações finais sobre cookies

Apesar da praticidade de se utilizar os cookies oferecidos pelo protocolo HTTP, devemos fazer algumas considerações quanto a sua utilização.

- O tamanho dos dados armazenados (nome e valor) não devem ultrapassar 4K.
- O navegador pode armazenar múltiplos cookies, contanto obedece um certo limite. Um navegador armazena até 20 cookies por configuração de domínio. O navegador armazena também 300 cookies em geral. Se qualquer um destes dois limites forem alcançados, espera-se que o navegador exclua cookies menos utilizados.
- O domínio atribuído a um cookie deve ter pelo menos dois pontos. Quando nenhum domínio é especificado na criação de um cookie através do método `setDomain(nome do domínio)`, então ele só poderá ser lido apenas pelo host que originalmente o definiu.

## 8. Tratamento de Erros

Esta última categoria dos objetos implícitos tem apenas um membro, o objeto `exception`. Este objeto implícito é fornecido com o propósito de tratamento de erros dentro de uma página JSP.

### Objeto Exception

O objeto `exception` não está automaticamente disponível em todas as páginas JSP. Este objeto está disponível apenas nas páginas que tenham sido designadas como páginas de erro, usando o atributo `isErrorPage` configurado com `true` na diretiva `page`. O objeto `exception` é uma instância da classe `java.lang.Throwable` correspondente ao erro não capturado que fez com que o controle fosse transferido para a página de erro. Os principais métodos da classe `java.lang.Throwable` que são utilizados dentro das páginas JSP são listados na tabela abaixo:

Método	Descrição
<code>String getMessage()</code>	Retorna a mensagem de erro descritiva associada com a exceção quando ela foi lançada.
<code>void printStackTrace(PrintWriter out)</code>	Imprime a pilha de execução em funcionamento quando a exceção foi lançada para o fluxo de saída especificado pelo parâmetro <b>out</b> .
<code>String toString()</code>	Retorna uma cadeia combinando o nome da classe da exceção com sua mensagem de erro, se houver alguma.

O trecho abaixo ilustra o uso do objeto `exception` em uma página de erro JSP:

```
<@ page isErrorPage=true %>
<h1>Erro Encontrado</h1>
O seguinte erro foi encontrado:<br>
<b><%= exception %></b><br>
<%= exception.printStackTrace(out); %>
```

## 9. JavaBeans

JavaBeans são componentes de software que são projetados para serem unidades reutilizáveis, que uma vez criados podem ser reusados sem modificação de código, e em qualquer propósito de aplicação, seja um applet, um servlet ou qualquer outra.

Um modelo de componente é definido como um conjunto de classes e interfaces na forma de pacotes Java que deve ser usado em uma forma particular para isolar e encapsular um conjunto de funcionalidades.

Os componentes JavaBeans são também conhecidos como Beans. Passaremos a usar esta nomenclatura no restante deste tutorial.

Neste tutorial não falaremos sobre todos os conceitos de JavaBeans, daremos atenção àqueles conceitos que mais são utilizados nas páginas JSP.

### Regras para Escrever Beans

Para que esses componentes possam ser reconhecidos de forma geral, sua implementação deve seguir um conjunto de regras que serão usadas pelas ferramentas para introspecção da classe, ou seja, o processo de descobrir quais as funcionalidades do Bean e disponibilizá-la para o usuário. Cabe ao usuário fazer a interface entre o componente e o restante da aplicação, criando assim um novo tipo de programador, o programador de componentes.

```
import java.io.Serializable;
public class MyBean implements Serializable {
    private int property1;
    private boolean property2;
    public MyBean() {

    }
    public void setProperty1(int property1) {
        this.property1 = property1;
    }

    public void setProperty2(boolean property2) {
        this.property2 = property2;
    }
    public int getProperty1() {
        return property1;
    }
    public boolean isProperty2() {
        return property2;
    }
}
```

Esta classe é um exemplo do que devemos seguir para tornar uma classe um bean, ou seja, o conjunto de regras que devemos obedecer para tornar o componente reconhecível por ferramentas e usável.

Um bean, como modelo de componente, usa a idéia de encapsulamento. Portanto, as suas variáveis devem ser acessadas somente através de métodos. As variáveis de um Bean são suas propriedades.

A primeira regra que devemos respeitar é o construtor. O construtor de um bean deve ser sem parâmetros.

Outra regra que devemos seguir diz respeito ao nome dos métodos que darão acesso as propriedades do bean. Os métodos podem ser divididos em duas categorias:

### **Métodos de Leitura**

Os métodos de leitura de propriedades devem seguir a seguinte convenção:

```
public TipoDaPropriedade getNomeDaPropriedade()
```

Como mostrado no código acima a propriedade chamada property1 tem o método de acesso getProperty1 . Note que a primeira letra da propriedade deve ser minúscula enquanto a primeira letra depois do get deve ser em maiúscula. A palavra TipoDaPropriedade deve ser substituída por o tipo da propriedade. Para variáveis booleanas vale o mesmo conceito, mas, ao invés do get usamos a palavra is. No exemplo temos o acesso a propriedade property2 que é booleana como:

```
public boolean isProperty2()
```

Observação: Você pode não definir um método de leitura para a variável. Se isto for feito você não permitirá o acesso a ela.

### **Métodos de escrita**

Os métodos de escrita permitem ao usuário do componente modificar o conteúdo da propriedade. Ele segue a seguinte terminologia:

```
public void setNomeDaPropriedade(TipoDaPropriedade varName)
```

Nos métodos de escrita não há caso especial para variáveis booleanas que também podem ser modificadas através de métodos set.

Você pode não definir métodos de escrita para uma determinada variável, fazendo com que o usuário não seja capaz de modificar o conteúdo da propriedade. Outra regra que um Bean deve seguir é a implementação da interface Serializable. A implementação desta interface fornece ao bean a propriedade chamada de persistência.

O conceito de persistência é permitir que o usuário faça uso do componente em um determinado momento e possa salvar o seu estado para o uso posterior partindo do mesmo ponto. A tecnologia que possibilita essa propriedade é a Serialização de Objetos. Esta tecnologia permite salvarmos o objeto em um fluxo para posterior recuperação. Quando houver a recuperação, o objeto deve se comportar como se fosse exatamente o mesmo de quando foi salvo, se olharmos o objeto como uma máquina de estado, então podemos dizer que o objeto é recuperado no mesmo estado de quando foi salvo.

A tecnologia de serialização de objetos é muito importante em Java pois permite a linguagem atividades complexas como computação distribuída e além de muitas outras funcionalidades interessantes.

Para sermos capazes de serializar objetos basta fazer a classe implementar a interface Serializable e nada mais.

## Propriedades de um Bean

Os beans possuem diversos tipos de propriedades, que são:

**Simples:** Propriedades simples alteram a aparência ou comportamento do bean e são acessadas através dos métodos de escrita e leitura mostrados.

**Ligadas:** As propriedades deste tipo quando alteradas provocam uma notificação para algum evento.

**Reprimidas:** As propriedades podem ter sua mudança vetada pelo próprio bean ou por objetos externos.

**Indexadas:** Propriedades indexadas representam coleções de valores acessados por índices, como arrays.

Para as propriedades indexadas temos:

Métodos de leitura do array inteiro:

```
public TipoDaPropriedade[] getNomeDaPropriedade();  
public void setNomeDaPropriedade(TipoDaPropriedade[] value);
```

Métodos de leitura de elementos individuais:

```
public TipoDaPropriedade getNomeDaPropriedade(int index);  
public void setNomeDaPropriedade(int index, TipoDaPropriedade value);
```

## Como as ferramentas lêem os Beans ?

JavaBeans são desenvolvidos para serem usados por terceiros (ou não). As ferramentas de desenvolvimento em Java são capazes de ler beans e prover a funcionalidade do componente para o usuário. Mas como as ferramentas são capazes de descobrir a funcionalidade de um Bean já que ele não é uma classe conhecida, ou seja, pode ser qualquer componente?

Para a realização desta tarefa, que chamamos introspecção, é usada uma API disponível no Ambiente Runtime da plataforma Java, a Java Reflection API . Usando a funcionalidade desta API a ferramenta procura os métodos públicos do bean e faz a ligação dos métodos com as propriedades. Após a ferramenta ter obtido todas as informações que ela necessita sobre a classe então as propriedades e os métodos são disponibilizados para uso.

A introspecção é realizada pela classe `java.beans.Introspector` que usa a Reflection API para descobrir as funcionalidades do bean por casamento de padrão. No entanto, alternativamente você pode definir as informações sobre o Bean em separado, em uma classe que implementa a interface `BeanInfo`, porém não entraremos em detalhes sobre isso.

É importante saber que um bean também pode implementar outros métodos que não aqueles que lêem ou gravam as propriedades. Esses métodos não diferem em nada de outros métodos de qualquer outra classe em Java.

### **JavaBeans x Enterprise JavaBeans**

Já conhecemos o que é um JavaBean, então agora podemos compará-los com o Enterprise JavaBeans (EJB). Quais as diferenças?

Os JavaBeans assim como o EJB são modelos de componente, porém apesar dos dois compartilharem o termo JavaBeans os modelos não são relacionados. Uma parte da literatura tem referenciado EJB como uma extensão dos JavaBeans, mas isto é uma interpretação errada. As duas APIs servem para propósitos bem diferentes.

O JavaBean tem a intenção de ser usado em propósitos do mesmo processo (intraprocesso) enquanto EJB é projetado para ser usado como componentes interprocessos. Em outras palavras, os JavaBeans não têm a intenção de ser usado como componente distribuído assim como o EJB.

## 10. Métodos

Como foi visto em módulos anteriores, é possível definir funções na própria página JSP através das tags `<%! e %>`. Veja o exemplo abaixo:

```
<%!
String jogueMoeda(){
    String retorno = "";
    int i = (int)(Math.random()*10); // um número entre 0 e 10
    i = i%2;                          // resto da divisao de i por 2
    if (i==0) {
        retorno = "cara";
    } else {
        retorno = "coroa";
    }
    return retorno;
}
%>
<html>
<body>
<h1>Deu <%= jogueMoeda() %>!</h1>
</body>
</html>
```

Nesse trecho de código, uma função é definida dentro de uma página JSP. Essa função é bastante simples: simula o lançamento de uma moeda para cima e escolhe, aleatoriamente, entre cara ou coroa.

Imagine agora que queremos que a função simule o lançamento da moeda várias vezes seguidas. Podemos alterar o código anterior e chamarmos a função várias vezes:

```
....
<html>
<body>

<h1>Deu <%= jogueMoeda() %>,
<%= jogueMoeda() %> ...
<%= jogueMoeda() %>!</h1>

</body>
</html>
```

Essa, solução, apesar de funcionar, não é a mais adequada. Podemos reescrever a função para receber como parâmetros a quantidades de jogadas que deverão ser efetuadas e fornecer um meio para que a função possa escrever o conteúdo na página JSP:

```
<%!
void jogueMoeda(int numeroJogadas, javax.servlet.jsp.JspWriter o){
    try{
        for(int k=1; k <= numeroJogadas; k++){
            int i = (int)(Math.random()*10); //número entre 0 e 10
            i = i%2;
            if(i==0){
```

```

        o.print("cara");
    } else {
        o.print("coroa");
    }
    if(k==numeroJogadas){
        o.print("!");
    } else {
        o.print(", ");
    }
}
} catch(Exception e){}
}
%>

<html>
<body>

<h1>Deu <% jogueMoeda(10,out); %></h1>

</body>
</html>

```

Essa segunda versão, apesar de ter uma funcionalidade bem parecida com a anterior traz algumas novidades.

Passamos como parâmetro um objeto da interface `javax.servlet.jsp.JspWriter`. É através desse objeto que iremos enviar para a página JSP o resultado da "jogada".

Na chamada da função passamos, além do número de jogadas desejadas, uma referência ao objeto implícito `out`. É através dessa referência que a função irá imprimir na saída da página JSP o resultado de sua jogada.

É importante aprender que não podemos chamar o objeto `out` (ou qualquer outro objeto implícito) diretamente em uma função declarada dentro de uma página JSP. Portanto, o seguinte código está errado e apresentará um erro em tempo de compilação:

```

<%!
void jogueMoeda(int numeroJogadas){
    for(int k=1; k <= numerovezes; k++){
        int i = Math.random();
        i = i*10;
        i = i%2;
        if(i==0){
            out.print("cara");
        } else {
            out.print("coroa");
        }
        if(k==numeroJogadas){
            out.print("!");
        } else {
            out.print(", ");
        }
    }
}
}

```

```

}
%>
<html>
<body>

<h1>Deu <% jogueMoeda(50) %></h1>

</body>
</html>

```

A tabela a seguir exhibe como criar e chamar funções que usam os nove objetos implícitos:

Objeto	Assinatura da Função	Chamada da Função na página JSP
page	<pre>void f(javax.servlet.jsp.HttpJspPage p) { ... p.algumaFuncaoDoObjetoPage(); ... }</pre>	<pre>&lt;% f(page); %&gt;</pre>
config	<pre>void f(javax.servlet.Config c){ ... c.algumaFuncaoDoObjetoConfig(); ... }</pre>	<pre>&lt;% f(config); %&gt;</pre>
request	<pre>void f(javax.servet.http.HttpServletRequest rq){ ... rq.algumaFuncaoDoObjetoRequest(); ... }</pre>	<pre>&lt;% f(request); %&gt;</pre>
response	<pre>void f(javax.servet.http.HttpServletResponse rp){ ... rp.algumaFuncaoDoObjetoResponse(); ... }</pre>	<pre>&lt;% f(response); %&gt;</pre>
out	<pre>void f(javax.servet.jsp.JspWriter o){ ... o.algumaFuncaoDoObjetoOut(); ... }</pre>	<pre>&lt;% f(out); %&gt;</pre>
session	<pre>void f(javax.servet.http.HttpSesseion s){ ... s.algumaFuncaoDoObjetoSession(); }</pre>	<pre>&lt;% f(session); %&gt;</pre>

	... }	%>
application	void f(javax.servlet.ServletContext a){ ... a.algumaFuncaoDoObjetoApplication(); ... }	<% f(application); %>
pageContext	void f(javax.servlet.jsp.PageContext pc){ ... pc.algumaFuncaoDoObjetoPageContext(); ... }	<% f(pageContext); %>
exception	void f(java.lang.Throwable e){ ... e.algumaFuncaoDoObjetoException(); ... }	<% f(exception); %>

Observação: As funções acima exemplificadas recebem apenas um parâmetro e não têm valor de retorno. Contudo, podemos criar funções que recebem vários parâmetros e tenham algum tipo de retorno.

## **Apêndice A:Servlets**

Servlets oferecem uma maneira alternativa a CGI para estender as funcionalidades de um servidor Web. Na verdade, a API de servlet de Java oferece mecanismos adequados à adaptação qualquer servidor baseado em requisições e respostas, mas é em aplicações Web que servlets têm sido mais utilizados.

Com o uso de servlets, a arquitetura da Web torna-se um base atrativa para o desenvolvimento de aplicações distribuídas em Java integrando outros serviços. A utilização de browsers HTML simplifica o desenvolvimento das aplicações cliente. Servidores Web suportam os mecanismos básicos de conexão ao cliente. Assim, o desenvolvimento irá se concentrar na extensão dos serviços através dos servlets.

### **Fundamentos da API**

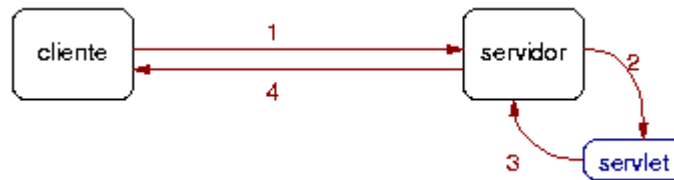
A especificação de servlets foi definida como uma das extensões padronizadas ao pacote Java, não sendo parte da distribuição básica de Java (J2SE). As classes associadas à especificação de servlets podem ser obtidas na página da Java Servlet Technology ou então instalar a plataforma J2EE, que complementa o J2SE com diversas funcionalidades para desenvolvimento de aplicações distribuídas em Java.

A API de servlets está concentrada nos pacotes `javax.servlet` e `javax.servlet.http`. Classes desses pacotes são utilizadas para desenvolver servlets que serão integrados a servidores.

### **Ciclo de vida de servlet**

A execução de um servlet não difere muito de uma aplicação CGI em sua forma de interação com o servidor. As quatro principais etapas nessa interação são:

- (1) cliente envia solicitação ao servidor;
- (2) servidor invoca (através do seu container) o servlet indicado para a execução do serviço solicitado;
- (3) servlet gera o conteúdo em resposta à solicitação do cliente, eventualmente acessando outros serviços acessíveis através da plataforma Java;
- (4) servidor repassa o resultado gerado pelo servlet para o cliente como uma resposta HTTP convencional.



Quando um servlet é carregado pela primeira vez para a máquina virtual Java do servidor, o método `init()` é invocado. Esse método tipicamente prepara recursos para a execução do serviço (por exemplo, abrir arquivos ou ler o valor anterior de um contador de número de acessos) ou estabelece conexão com outros serviços (por exemplo, com um servidor de banco de dados). O método `destroy()` permite liberar esses recursos (fechar arquivos, escrever o valor final nessa sessão do contador de acessos), sendo invocado quando o servidor estiver concluindo sua atividade.

Uma diferença fundamental entre um servlet e uma aplicação CGI é que a classe que implementa o servlet permanece carregada na máquina virtual Java após concluir sua execução. Um programa CGI, ao contrário, inicia um novo processo a cada invocação -- por este motivo, CGI deve utilizar mecanismos adicionais para manter o estado entre execuções, sendo a forma mais comum a utilização de arquivos em disco. Com um servlet, tais mecanismos são necessários apenas na primeira vez que é carregado e ao fim da execução do servidor, ou eventualmente como um mecanismo de checkpoint.

Servlets também oferecem como vantagem o fato de serem programas Java. Assim, eles permitem a utilização de toda a API Java para a implementação de seus serviços e oferecem adicionalmente portabilidade de plataforma.

## Apêndice B: Tags e JSTL (JSP Standard Tag Library)

### Qual o problema ?

- Dificuldade de construir páginas JSPs bem organizadas internamente; Páginas JSPs com muito código Java
- Web designers não sabem programar em Java
- Problemas na interação entre desenvolvedores e web designers.

### O que queremos ?

- Criar páginas dinâmicas bastante complexas sem escrever código Java dentro delas
- Fornecer tags que tornem fáceis tarefas que exigiriam várias linhas de código Java, como formatação de números e datas seguindo configurações regionais do usuário
- Facilitar a interação entre desenvolvedores e web designers.

### Qual a solução ?

- Utilizar JSTL

## JSTL

JSTL consiste em uma coleção de bibliotecas, tendo cada uma um propósito bem definido, que permitem escrever páginas JSPs sem código Java, aumentando assim a legibilidade do código e a interação entre desenvolvedores e web designers.

Uma página JSTL é uma página JSP contendo um conjunto de tags JSTLs. Cada tag realiza um determinado tipo de processamento (equivalente a código Java dentro de JSP). Cada tag JSTL, faz parte uma biblioteca JSTL. Uma página JSTL pode utilizar várias bibliotecas JSTLs.

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<html>
<body bgcolor="#FFFFFF">
<jsp:useBean id="agora" class="java.util.Date"/><br>
Versão Curta: <fmt:formatDate value="{agora}" /><br>
Versão Longa: <fmt:formatDate value="{agora}" dateStyle="full"/>
</body>
</html>
```

O exemplo apresenta a data atual em dois formatos: um curto e outro longo. Observe que não existe nenhum código Java. Na primeira linha, temos o seguinte:

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

Essa declaração informa ao compilador JSPs para utilizar um biblioteca de tags, cujos tags serão reconhecidos pelo prefixo "fmt", de modo a evitar possíveis conflitos com tags de mesmo nome de outras bibliotecas.

A biblioteca de tags é identificada pelo atributo uri. A tag padrão `<jsp:useBean>` é utilizada para instanciar um objeto `java.util.Date`, inicializado por padrão com a data e hora atuais do sistema. A variável agora, criada pelo tag `<jsp:useBean>`, é depois referenciada dentro do atributo value dos tags `<fmt:formatDate>`, que aparece duas vezes na página.

Observe que o atributo `dateStyle` do tag `<fmt:formatDate>` define se será utilizado um formato resumido da data ou um formato longo.

Versão Curta: 25/07/2005

Versão Longa: Segunda-feira, 25 de Julho de 2005

EL - Expression Language

O valor de qualquer expressão pode ser acessado da seguinte forma:

```
${expressão}
```

Operador	Descrição	Exemplo	Resultado
<code>==</code> <code>eq</code>	Igualdade	<code>\${5 == 5}</code>	true
<code>!=</code> <code>ne</code>	Desigualdade	<code>\${5 != 5}</code>	false
<code>&lt;</code> <code>lt</code>	Menor que	<code>\${5 &lt; 7}</code>	true
<code>&gt;</code> <code>gt</code>	Maior que	<code>\${5 &gt; 7}</code>	false
<code>&lt;=</code> <code>le</code>	Menor ou igual que	<code>\${5 le 5}</code>	true
<code>&gt;=</code> <code>ge</code>	Maior ou igual que	<code>\${5 ge 6}</code>	false
<code>empty</code>	Checa se um parâmetro está vazio	<code>\${user.lastname}</code>	depende
<code>and</code> <code>&amp;&amp;</code>	E	<code>\${param.month == 5 and param.day == 25}</code>	depende
<code>or</code> <code>  </code>	OU	<code>\${param.month == 5 or param.month == 6}</code>	depende
<code>+</code>	soma	<code>\${4 + 5}</code>	9
<code>!</code> <code>not</code>	Negação	<code>\${not true}</code>	false

## Bibliotecas padrão

Biblioteca JSTL	Prefixo	URI	Tipos de uso	Exemplo de tag
<i>Core</i>	c	<a href="http://java.sun.com/jstl/core">http://java.sun.com/jstl/core</a>	Acessar e modificar dados em memória Comandos condicionais Loop	<c:forEach>
Processamento de XML	x	<a href="http://java.sun.com/jstl/xml">http://java.sun.com/jstl/xml</a>	Parsing (leitura) de documentos Impressão de partes de documentos XML Tomada de decisão baseado no conteúdo de um documento XML	<x:forEach>
Internacionalização e formatação	fmt	<a href="http://java.sun.com/jstl/fmt">http://java.sun.com/jstl/fmt</a>	Leitura e impressão de números Leitura e impressão de datas Ajuda a sua aplicação funcionar em mais de uma lingua	<fmt:formatDate>
Acesso a banco de dados via SQL	sql	<a href="http://java.sun.com/jstl/sql">http://java.sun.com/jstl/sql</a>	Leitura e escrita em banco de dados	<sql:query>

**Tags básicas****Tags de Iteração**

A biblioteca core do JSTL fornece tags para executar trechos repetidamente, de maneira similar aos comandos for e while da linguagem Java.

**Tag <c:forEach>**

Permite realizar um loop

Exemplo: Imprime os valores entre 2 e 5.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
<body bgcolor="#FFFFFF">
<c:forEach var="i" begin="2" end="5">
<c:out value="{i}"/>;</c:forEach>
</body>
</html>
```

Saída

```
2;3;4;5;
```

Tag <c:forEach>

Quebra uma string em substrings, de acordo com o delimitador indicado como atributo

Exemplo: Imprime os valores entre dois e cinco.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
<body bgcolor="#FFFFFF">
<c:forEach var="i" delims="," items="2,3,4,5">
<c:out value="{i}"/>;</c:forEach >
</body>
</html>
```

Saida

```
2;3;4;5;
```

## Tags condicionais

### Tag <c:if>

Equivalente ao comando if

Atributo test realiza o teste condicional

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<html>
<body bgcolor="#FFFFFF">
Elementos pares:
<c:forEach var="i" delims="," items="2,3,4,5">
  <c:if test="{i} % 2 == 0">
    <c:out value="{i}"/>;
  </c:if>
</c:forEach >
</body>
</html>
```

## Saída

```
Elementos pares: 2; 4;
```

Uma falha da biblioteca é a inexistência do complemento do comando if, ou seja, o comando else (caso se deseje criar fluxos alternativos, deve-se utilizar <c:choose>).

### Tag <c:choose>

Equivalente ao comando switch

Tags utilizadas:

**<c:when>**, realiza o teste condicional

**<c:otherwise>**, se todos os testes condicionais falharem, ele será utilizado.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<html>
<body bgcolor="#FFFFFF">
<c:forTokens var="i" delims="," items="2,3,4,5">
  <c:choose>
    <c:when test="\${i % 2 == 0}">\${i} (par)</c:when>
    <c:otherwise>\${i} (impar)</c:otherwise>
  </c:choose>
  ;
</c:forTokens >
</body>
</html>
```

## Saída

```
2 (par) ; 3 (impar) ; 4 (par) ; 5 (impar) ;
```

## Tags de atribuição e importação

### Tag <c:import>

Permite importar páginas web do mesmo contexto web, de contextos diferentes e até mesmo de máquinas diferentes.

Atributo Descrição Requerido? Default

Atributo	Descrição	Requerido?	Default
url	URL a ser importada	Sim	Nenhum
context	"/" seguido do nome da aplicação web local	Não	Contexto corrente
var	Nome do atributo onde será armazenado o conteúdo da página importada	Não	Nenhum
scope	Escopo do atributo onde será	Não	page

	armazenado o conteúdo da página importada Pode ser: page, request, session, application		
--	--	--	--

## Tag <c:set>

Permite a atribuir valores a variáveis em um determinado escopo.

Atributo	Descrição	Requerido?	Default
value	Expressão a ser processada	Não	Nenhum
var	Nome do atributo onde será armazenado o resultado do processamento do atributo "value"	Não	Nenhum
scope	Escopo do atributo. Pode ser: page, request, session, application	Não	page

## Exemplo

Nesse exemplo, a variável "title" é criada com o valor "Welcome to Page 1" com escopo "request". Em seguida, a página "header.jsp" é carregada, e seu conteúdo é armazenado na variável "headerText". Finalmente, imprimimos o conteúdo da variável "title" e da variável "headerText".

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<html>
<body>
<c:set scope="request" var="title" value="Welcome to Page 1"/><c:import
var="headerText" url="header.jsp"/><br>Minha página:${title}<br>Texto
importado:${headerText}
</body>
</html>
```

Arquivo "header.jsp"

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
#${title}#
```

## Saída

```
Minha página: Welcome to Page 1
Texto importado: #Welcome to Page 1#
```

## Acessando objetos Java

Os exemplos anteriores se detiveram em apresentar apenas as tags JSTL básicas. Vamos agora mostrar alguns exemplos mais elaborados utilizando objetos Java.

## Exemplo 1

Acessando uma coleção de objetos Java. Seja a classe Java ColecaoDeNomes:

```
package test;

import java.util.*;
import java.io.*;

public class ColecaoDeNomes implements Serializable{

    private Collection nomes = new ArrayList();
    public ColecaoDeNomes() {
        nomes.add("Maria");
        nomes.add("Zeca");
        nomes.add("Carlos");
    }

    public Collection getNomes() {
        return nomes;
    }
}
```

Queremos iterar a coleção de nomes existente dentro da classe ColecaoDeNomes (veja o exemplo abaixo). Observe que o objeto da classe ColecaoDeNomes foi instanciado utilizando a tag <jsp:useBean>. Além disso, o nome da instância criada é "colecacao". Note que o atributo items da tag <c:forEach> faz referência à instância criada (batizada como "colecacao"). Os objetos existentes dentro da coleção nomes (java.util.Collection) dentro da classe ColecaoDeNomes está sendo acessado através de `#{colecacao.nomes}`:

`#{colecacao.nomes}`: colecacao é o nome da instância da classe ColecaoDeNomes recém criada

`#{colecacao.nomes}`: nomes faz referência ao nome do método getNomes()

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

<jsp:useBean id="colecacao" class="teste.ColecaoDeNomes"/>

<c:forEach var="nome" items="#{colecacao.nomes}">
    <br>#{nome}
</c:forEach >

</body>
</html>
```

## Saída

```
Maria
Zeca
Carlos
```

## Exemplo 2

Agora utilizando mais recursos da tag `<c:forEach>`.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<html>
<body bgcolor="#FFFFFF">
<jsp:useBean id="colecacao" class="teste.ColecaoDeNomes"/>
<table border="0" align="center" cellpadding="2" cellspacing="2">
<tr>
<td bgcolor="#e0e0e0">Id</td>
<td bgcolor="#e0e0e0">Nome</td>
</tr>
<c:forEach var="nome" items="{colecacao.nomes}" varStatus="status">
<tr>
<td bgcolor="#f0f0f0">
<c:choose>
<c:when test="{status.first}">Primeiro</c:when>
<c:when test="{status.last}">Último</c:when>
<c:otherwise>Número {status.count}</c:otherwise>
</c:choose>
</td>
<td bgcolor="#f0f0f0">${nome}
</td>
</tr>
</c:forEach >
</table>
</body>
</html>
```

### Saída

```
Id Nome
Primeiro Maria
Número 2 Zeca
Último Carlos
```

## Instalação de uma aplicação que utilize JSTL

- Colocar os jars (jstl.jar e standard.jar) dentro de WEB-INF/lib.
- Editar o arquivo web.xml

### Exemplo de um arquivo web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Copyright (c) 2002 by ObjectLearn. All Rights Reserved. -->
<web-app xmlns=http://java.sun.com/xml/ns/j2ee
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-app_2_4.xsd"
version="2.4">
<welcome-file-list>
<welcome-file>index.htm</welcome-file>
</welcome-file-list>
<error-page>
<error-code>404</error-code>
```

```
<location>/error.jsp</location>  
</error-page>  
</web-app>
```

- Copiar os .class para a pasta WEB-INF/classes
- Copiar os arquivos JSPs com as tags jstl do diretório da principal da aplicação