

# Desenvolvimento Web I

## Aula 08 - JSTL – Internacionalização, Taglib functions e expression language

# Apresentação

---

Olá, caro aluno. Hoje, continuaremos com a nossa disciplina falando mais um pouco sobre JSTL, mais especificamente sobre sua taglib de internacionalização chamada de I18N. Você já pensou em desenvolver um software capaz de ser utilizado por pessoas de diferentes países? Se sim, você deve ter se deparado com o desafio de implementar um sistema multilíngue, ou seja, que apresente suas mensagens em diferentes línguas, de acordo com a localidade (origem/região) do usuário. Mas você não quer ficar reestruturando seu código toda vez que precisar trabalhar com uma linguagem de usuário diferente (português, inglês, espanhol etc.), certo?

As tags de internacionalização da I18N vão nos ajudar a estruturar nossos sistemas Web, de forma que ele possa funcionar em diferentes idiomas e padrões (como formação de datas, números e moedas) específicos de determinadas regiões. Além disso, finalizaremos nossas aulas de JSTL, estudando a taglib *functions*, e veremos um pouco do que é EL (expression language).

Tenha uma boa leitura!

## Objetivos

- Construir páginas cujo conteúdo seja sensível à localização do usuário.
- Aplicar a taglib *functions*.
- Definir o uso de EL ao invés de scriptlets.

# A taglib I18N

---

A biblioteca I18N implementa funcionalidades que nos ajuda a construir sistemas Web internacionalizáveis, ou seja, cujas mensagens mostradas são configuradas de acordo com a localidade ou preferências do usuário. Essas funcionalidades são importantes, quando se deseja construir uma aplicação que será usada em diferentes idiomas, como veremos nos exemplos mais adiante.

O quadro a seguir mostra as principais funções da I18N.

Área	Funcionalide	Tags	Prefixo
I18N	Definição de Localidade	setLocale	fmt
	Mensagens	bundle message param setBundle	
	Formatação	formatNumber formatDate	

**Quadro 1** - Principais funções definidas pela I18N.

## Curiosidade

O nome I18N vem de *Internationalization* (em português, internacionalização), que começa com I, termina com N e tem 18 letras intermediárias. Nome grande, não é?

Começamos nossos estudos pela tag de definição de localidade `fmt:setLocale`. Essa tag permite definir qual *locale* (combinação de idioma e região) será usado para exibir os itens sensíveis à localidade. O *Locale* será definido dentro dos seguintes escopos: a página (escopo padrão); a requisição; a sessão; ou a aplicação. A tag `fmt:setLocale` sobrescreve o *Locale* definido pelo browser do usuário, ou seja, mesmo que o browser do usuário esteja configurado com o *Locale* `en_US` (inglês dos Estados Unidos), por exemplo, nós podemos usar a tag `fmt:setLocale` para definir a localidade para `pt_BR` (português do Brasil) e a configuração do browser será ignorada. Note que a String da localidade é formada por 5 caracteres, dois que indicam o idioma, um separador “\_” e o país. Isso é necessário, pois o mesmo idioma pode variar de país para país (português do Brasil e de Portugal, inglês dos Estados Unidos e da Inglaterra, etc.).

A Listagem 1 apresenta o uso da tag `fmt:setLocale` da taglib `fmt`

```
1 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
2 <%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
3 <fmt:setLocale value="en_US" />
4 <html>
5 <body>
6 </body>
7 </html>
```

**Listagem 1** - Código JSP que define o idioma da página JSP para inglês americano.



**Vídeo 01** - Introdução ao jstl

Um detalhe que vale a pena mencionar é que a tag `fmt:setLocale` é uma tag de configuração, logo, o seu corpo deve ser deixado vazio e seu escopo pode ser mais abrangente que apenas uma página. Note também que é necessário utilizar no início da página o código: `<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>` para indicar que também se deseja utilizar a taglib “fmt”, necessária para o i18N.

Uma vez que explicamos a funcionalidade dessa tag, vamos, agora, mostrar e explicar cada um dos seus atributos.

Atributo	Descrição	Obrigatório	Valor default
value	String ou expressão que, quando avaliada, resulta em uma String ou java.util.Locale.	Sim	
scope	Escopo onde a variável que representa o Locale será armazenada.	Não	Página

**Quadro 2** - Atributos da tag `fmt:setLocale`

Na Listagem 1, vimos o uso do atributo `value` que pode ser uma String, como, por exemplo, `pt_BR`, ou uma expressão que será avaliada para `Java.lang.String` ou `Java.util.Locale`. O atributo `scope` já é um velho conhecido nosso e tem a mesma característica que vimos antes.

## A taglib I18N II

Continuando com as tags de internacionalização, uma vez que já sabemos como indicar qual a localidade (ou preferência) do usuário (ex: `pt_BR`), vamos agora ver as tags de mensagem. A tag **`fmt:bundle`** define o contexto de localização, baseado em um arquivo de propriedades (resource bundle) que será usado apenas dentro do corpo da tag. Um arquivo de propriedades é um arquivo de texto com a extensão `".properties"`. Esse arquivo contém dados específicos de uma localidade (idioma/região). Os arquivos de propriedade são os mecanismos para termos um sistema multilíngue, em que cada idioma/região suportada está associada a um arquivo de propriedades como as mensagens do sistema naquela linguagem. Dessa maneira, cada linha de um arquivo de propriedades contém uma palavra-chave e um valor, por exemplo:

`MSG_ERRO_SISTEMA=Ocorreu um erro inesperado no sistema!`

Já adiantando, a ideia é que o código do sistema trabalhe com a chave MSG\_ERRO\_SISTEMA e que as tags de internacionalização troquem a chave pelo texto correspondente.

A tag `fmt:setBundle` também pode ser utilizada para identificar um arquivo de propriedades, associando-o a um determinado escopo. Se deseja usar, por exemplo, um arquivo de propriedades para a toda a sua aplicação, você pode usar a tag `fmt:setBundle`, como mostrado na Listagem 2. Porém, se em determinada página quisermos outro arquivo de propriedades, podemos usar a tag `fmt:bundle` e aninhar dentro dela o conteúdo que desejamos usar em outro arquivo, como mostra a Listagem 3.

```
1 <fmt:setBundle basename="messages" scope="application"/>
```

**Listagem 2** - Código JSP que define o resource bundle para ser usado pela aplicação.

```
1 <fmt:bundle basename="messages2">
2   <fmt:message key="message.helloWorld" />
3 </fmt:bundle>
```

**Listagem 3** - Código JSP que define um resource bundle específico para ser usado dentro do corpo da tag.

Aqui, é importante algumas explicações. Na linha 01 da Listagem 2, a tag **basename** define qual o arquivo de propriedades da aplicação, ou seja, qual o nome do arquivo `.properties` que será usado. No caso, é o de nome `messages`. Na linha 01 da Listagem 3, estamos dizendo que usaremos um outro arquivo chamado `messages2` como arquivo de propriedades. Na linha 02, estamos dizendo que existe uma propriedade neste arquivo cujo nome (ou chave) é `message.helloWorld` e que o seu valor será usado dentro do corpo da tag `fmt:bundle`. Para ficar mais claro, vamos a um exemplo.

Para exemplificar, foi criado um projeto chamado “aula08”, um pacote dentro dele também chamado “aula08” e dentro desse pacote foi adicionado um servlet que, por enquanto, só redireciona a requisição para o nosso JSP, como mostrado a seguir.

**Figura 01** - Exemplo de Servlet que redireciona requisições.



```
1 package aula08;
2
3 import java.io.IOException;
4
5
6
7
8
9
10 @WebServlet("/HelloWorldServlet")
11 public class HelloWorldServlet extends HttpServlet {
12     private static final long serialVersionUID = 1L;
13     protected void doGet(HttpServletRequest request, HttpServletResponse response)
14         throws ServletException, IOException {
15         request.getRequestDispatcher("/helloWorld.jsp").forward(request, response);
16     }
17 }
18
```

O nosso JSP define o arquivo de propriedades como messages e solicita que o valor associado à chave message.helloWorld seja exibido.

**Figura 02** - Exemplo de código JSP que utiliza um arquivo de propriedades.

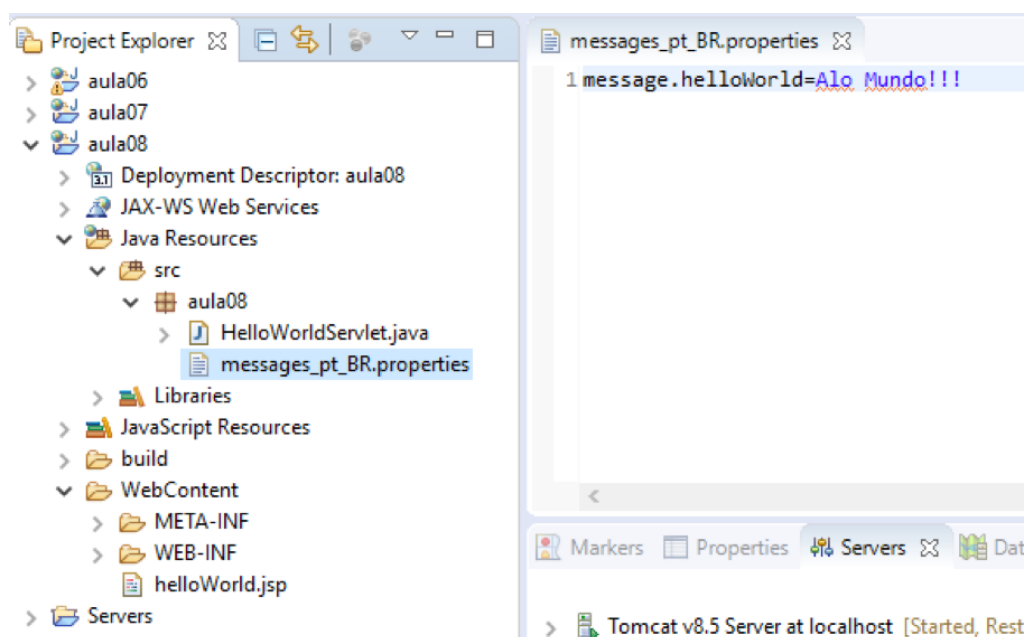


```
1 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
2 <%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
3 <!DOCTYPE html>
4 <html>
5 <body>
6     <fmt:setBundle basename="aula08.messages" scope="application"/>
7     <fmt:message key="message.helloWorld" />
8 </body>
9 </html>

```

O nosso arquivo de propriedades messages (mostrado a seguir) possui uma chave que é message.helloWorld cujo valor é Olá Mundo!!!. Note, no JSP acima, que essa é a mesma chave que nós estamos usando na tag fmt:message (falaremos sobre essa tag mais a frente). Note que a versão em português do arquivo tem nome "messages\_pt\_BR.properties", ou seja, o arquivo procurado pelo sistema será sempre o nome indicado (ex: messages) concatenado com "\_", a localidade (ex: pt\_BR) e a extensão ".properties". O arquivo messages\_pt\_BR.properties pode ser adicionado dentro de um pacote, por exemplo o pacote aula08 criado nesse projeto. Isso é importante pois o JSP faz referência ao arquivo de propriedades utilizando o nome do pacote que ele se encontra (basename="aula08.messages"). Para criar um arquivo dentro de um pacote basta clicar com o botão direito do mouse sobre o pacote (aula08, por exemplo), escolher a opção New->Other... e depois escolher a opção "File", adicionando o nome desejado ao arquivo na sequência.

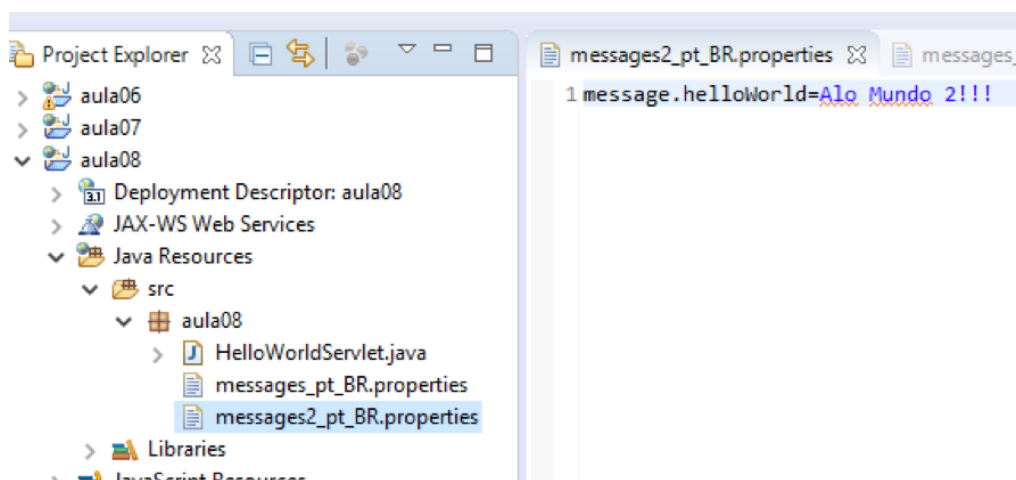
**Figura 03** - Estrutura do projeto e localização do arquivo messages\_pt\_BR.properties (a esquerda) e o seu conteúdo (a direita).



## A taglib I18N III

Considerando que para uma determinada parte do nosso JSP, poderíamos não querer usar o arquivo de propriedades definido pela tag `fmt:setBundle`, mas sim um outro arquivo de mensagens específico somente para essa parte do JSP, por exemplo chamado `messages2`. Para isso ele deve ser criado dentro de um pacote com o nome `messages2_pt_BR.properties` com seus valores próprios para cada chave, como mostrado a seguir

**Figura 04** - Arquivo de propriedades alternativo





Dessa forma, a mensagem final não seria mais Alô Mundo!!!, e sim Alô Mundo 2!!!, como pode ser visto no JSP modificado.

**Figura 05** - Uso de arquivo de propriedades diferente em parte específica do JSP.



```
1 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
2 <%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
3 <!DOCTYPE html>
4 <html>
5 <body>
6   <fmt:setBundle basename="aula08.messages" scope="application" />
7   <fmt:message key="message.helloWorld" />
8   <br />
9   <fmt:bundle basename="aula08.messages2">
10     <fmt:message key="message.helloWorld" />
11   </fmt:bundle>
12 </body>
13 </html>
```

Perceba que no JSP acima, na Figura 5, estamos utilizando inicialmente o `messages_pt_BR.properties`, como definido na Linha 6 e a mensagem “Alo Mundo!!!” deve aparecer no HTML gerado resultando do Linha 7. Entretanto logo em seguida (Linha 9) usamos o `<fmt:bundle>` com o `basename="aula08.messages2"` o que implica que qualquer `<fmt:message>` que estiver dentro dessa tag bundle usará o arquivo `messages2_pt_BR.properties` como fonte de dados.

Uma vez que explicamos a funcionalidade da tag `fmt:bundle` e `fmt:setBundle`, vamos, agora, mostrar e explicar cada um dos seus atributos.

Atributo	Descrição	Obrigatório	Valor default
basename	Nome do arquivo de propriedades (sem localidade e extensão “.properties”).	Sim	
prefix	String que será pré-concatenada ao valor da chave da mensagem (com o “.” no final).	Não	

**Quadro 3** - String que será pré-concatenada ao valor da chave da mensagem (com o “.” no final).

Atribuição	Descrição	Obrigatório	Valor default
basename	Nome do arquivo de propriedades (sem localidade e extensão “.properties”).	Sim	
var	Nome do arquivo de propriedades (sem localidade e extensão “.properties”).	Não	
scope	Escopo da variável armazenada em var.	Não	Página

**Quadro 4** - Atributos da tag `fmt:setBundle`

Em ambas as tags, `basename` é o atributo que define o nome do arquivo de propriedades. Para a tag `fmt:bundle`, o atributo `prefix` define uma String que será concatenada no início de cada String usada como chave pela tag `fmt:message`. No exemplo anterior, nós usamos como chave “`message.helloWorld`”. Entretanto, se tivéssemos definido o atributo `prefix` na tag `fmt:bundle` como “`message.`”, poderíamos ter usado como chave apenas a String “`helloWorld`”. Os demais atributos, mais uma vez, são nossos velhos conhecidos.

A próxima tag é a tag `fmt:message`, que possui uma tag auxiliar, que é a tag `fmt:param`. Nós já vimos, brevemente, a tag `fmt:message` em uso quando falamos de `fmt:bundle` e `fmt:setBundle`. Agora, veremos a tag `fmt:message` usando a tag `fmt:param`. A tag `fmt:message` procura por mensagens localizadas dentro de um arquivo de propriedades. Essa tag, como dito, pode conter a tag `fmt:param` no seu corpo que especificam valores para os parâmetros definidos na mensagem. Isso quer dizer que podemos ter mensagens parametrizadas.

Na Listagem 4, podemos ver um exemplo de uso da tag `fmt:message`, como a tag `fmt:param`.

```

1 <fmt:message key="message.goodMorning">
2   <fmt:param value="Professor" />
3 </fmt:message>
```

**Listagem 4** - Código JSP que procura por uma chave chamada `message.goodMorning` com um parâmetro que será substituído pelo valor definido por `fmt:param`.

Aqui, vale explicar o que seria substituir o parâmetro da mensagem da chave “message.goodMorning” (linha 01) pelo valor do parâmetro especificado pela tag fmt:param (linha 02). Para que isso seja possível, o valor da chave “message.goodMorning”, no arquivo, deve ser algo como:

```
message.goodMorning=Bom dia {0}!!!
```

Isso permite que no lugar de {0} terei o valor que foi definido na tag fmt:param. Um exemplo mais prático seria usar como parâmetro o nome do usuário logado no sistema. Uma mensagem pode ter vários parâmetros, cujos valores são indexados por um número entre os “{” e “}”.

Uma vez que explicamos a funcionalidade da tag fmt:message e fmt:param, vamos, agora, mostrar e explicar cada um dos seus atributos.

Atribuito	Descrição	Obrigatório	valor default
key	Nome da chave a ser procurada no arquivo de propriedades.	Não	Corpo
var	Nome da variável que armazenará a mensagem.	Não	
scope	Escopo da variável que armazenará a mensagem (indicada no atributo var).	Não	Página

**Quadro 5** - Atributos da tag fmt:message

Atributo	Descrição	Obrigatório	Valor default
value	Valor que será substituído em uma mensagem parametrizada.	Sim	Corpo

**Quadro 6** - Atributos da tag fmt:param

O atributo key é o nome da chave no arquivo de propriedades que corresponde à mensagem a ser exibida. Key pode ser uma String ou uma expressão que resulte em uma String. Já o atributo value da tag `fmt:param` define o valor que será usado para substituir os parâmetros da mensagem. É importante notar que a tag `fmt:param` só possui um value. Como uma mensagem pode ter N parâmetros ( {0}, {1}, {2}, ..., {n} ), pode ser necessário que haja mais de uma utilização dessa tag para preencher todos os parâmetros da mensagem, como visto na Listagem 5. A ordem da tag `fmt:param` também deve obedecer à ordem definida no arquivo de propriedades para que seja feita a substituição correta. O atributo value pode ser uma String ou uma expressão.

```
1 <fmt:message key="message.goodMorning">
2   <fmt:param value="Professor" />
3   <fmt:param value="Francisco" />
4 </fmt:message>
```

**Listagem 5** - Código JSP que procura por uma chave chamada `message.goodMorning` com dois parâmetros, que serão substituídos pelos valores definidos pelas tags `fmt:param`, na ordem em que aparecem.

No caso da Listagem 5, o valor da chave “`message.goodMorning`” no arquivo de propriedades seria algo como:

```
message.goodMorning=Bom dia {0} e {1}!!!
```

## Atividade 01

---

1. Crie um arquivo “`messages_en_US.properties`” e defina a chave “`message.helloWorld`” com o valor “`Hello World!!!`”. Em seguida, use a tag `fmt:setLocale` para mudar o Locale para “`en_US`” e confirme que o novo arquivo está sendo usado no lugar do anterior.
2. Retire a tag `fmt:setLocale` e mude o idioma diretamente no seu browser para “`en_US`” e reporte qual mensagem é mostrada, se a em inglês ou em português.
3. Use, novamente, a tag `fmt:setLocale`, mantendo a língua do seu browser em “`en_US`”, com o locale “`pt_BR`” e veja o que acontece.

# Formatação de datas e números

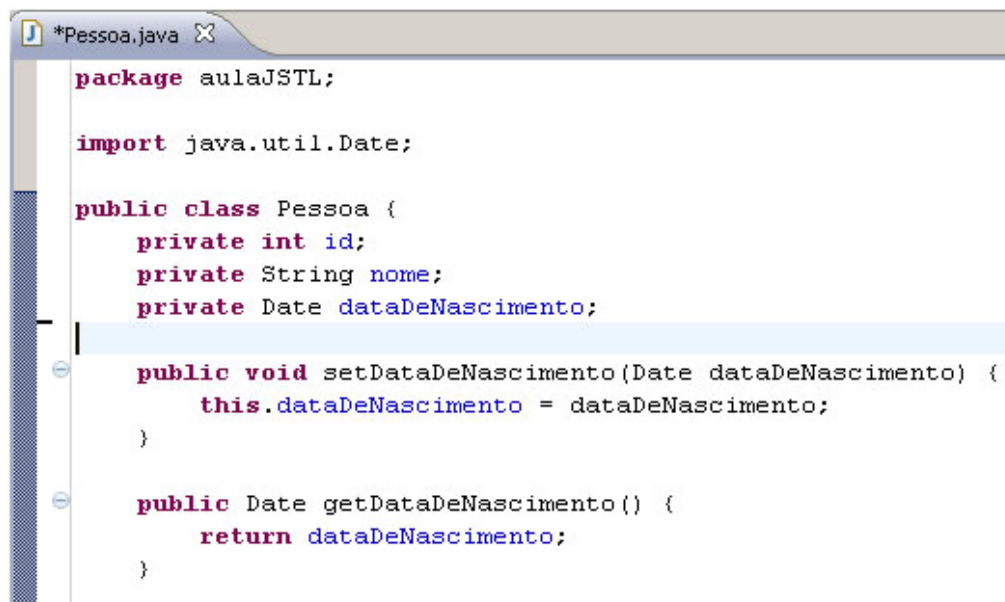
---

Agora, veremos as tags de formatação de datas e números que são sensíveis à localização. Por exemplo, nos Estados Unidos, o formato padrão de data é mês/dia/ano, enquanto no Brasil esse formato é dia/mês/ano. Da mesma forma, valores numéricos são formatados diferentemente, pois lá eles usam como separador de casas decimais o caractere '.' e nós usamos a ','. Assim, as tags de formatação nos fornecem mecanismos necessários para exibir essas informações sensíveis de localidade (data, números etc.), de acordo com a localidade do usuário.

A tag de formatação de data, `fmt:formatDate`, usa as informações de localização do browser do usuário ou a definida pela tag `fmt:setLocale`. Você também pode customizar a formatação dessa data usando alguns atributos da tag, como veremos mais adiante. Adicionalmente, essa tag formata informações de data e de hora.

Vamos começar alterando o exemplo da aula passada, que lista um conjunto de pessoas. Vamos alterar a classe `Pessoa` para ter mais um atributo, que é a data de nascimento da pessoa, como mostrada na figura a seguir.

**Figura 06** - Código da classe `Pessoa`



```
*Pessoa.java X
package aulaJSTL;

import java.util.Date;

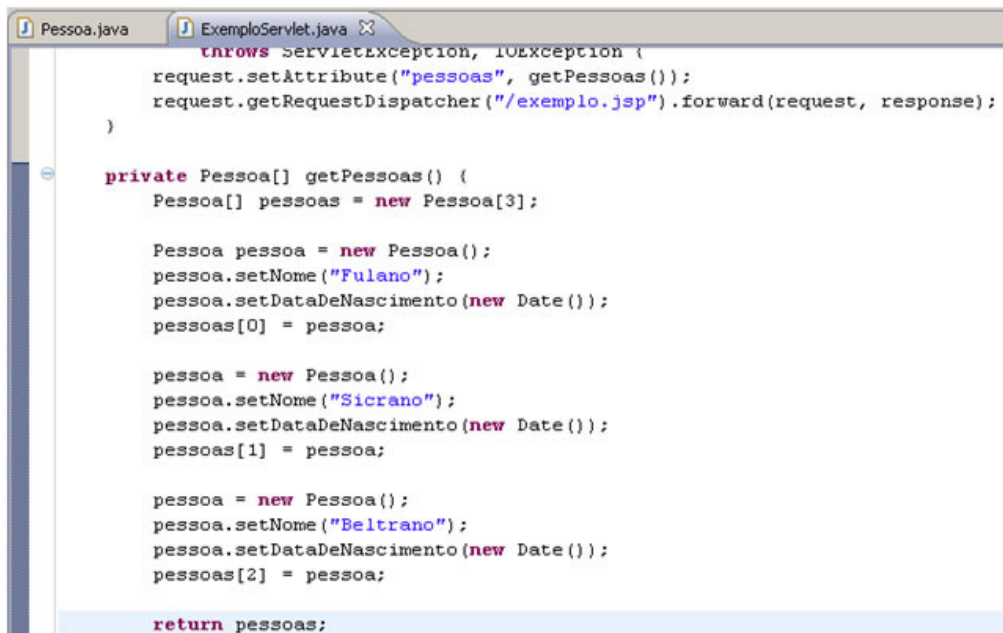
public class Pessoa {
    private int id;
    private String nome;
    private Date dataDeNascimento;

    public void setDataDeNascimento(Date dataDeNascimento) {
        this.dataDeNascimento = dataDeNascimento;
    }

    public Date getDataDeNascimento() {
        return dataDeNascimento;
    }
}
```

Agora, vamos alterar o servlet de exemplo que lista as pessoas do sistema para incluir o valor do atributo `dataDeNascimento`, conforme a figura a seguir.

**Figura 07** - Método da classe Pessoa que retorna um conjunto de instâncias.



```
Pessoa.java  ExemploServlet.java X
throws ServletException, IOException {
    request.setAttribute("pessoas", getPessoas());
    request.getRequestDispatcher("/exemplo.jsp").forward(request, response);
}

private Pessoa[] getPessoas() {
    Pessoa[] pessoas = new Pessoa[3];

    Pessoa pessoa = new Pessoa();
    pessoa.setNome("Fulano");
    pessoa.setDataDeNascimento(new Date());
    pessoas[0] = pessoa;

    pessoa = new Pessoa();
    pessoa.setNome("Sicrano");
    pessoa.setDataDeNascimento(new Date());
    pessoas[1] = pessoa;

    pessoa = new Pessoa();
    pessoa.setNome("Beltrano");
    pessoa.setDataDeNascimento(new Date());
    pessoas[2] = pessoa;

    return pessoas;
}
```

Agora, vamos alterar a o JSP que exibe os dados das pessoas para exibir, além do nome, também, a data de nascimento, como mostra a figura a seguir.

**Figura 08** - Código JSP de apresentação dos dados de cada pessoa cadastrada no sistema.

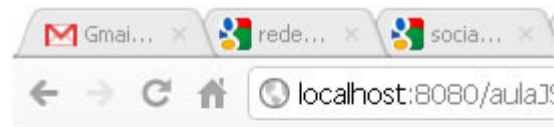


```
Pessoa.java  ExemploServlet.java  exemplo.jsp X  RequestFacade.class
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<html><body>
<strong> Lista de pessoas:</strong>
<br><br>
<table>
<c:forEach var="pessoa" items="${pessoas}" >
<tr>
<td>
<c:out value="${pessoa.nome}" /> - <fmt:formatDate value="${pessoa.dataDeNascimento}" />
</td>
</tr>
</c:forEach>
</table>
</body></html>
```

Veja o resultado:

**Figura 09** - Resultado da tela de apresentação dos dados das pessoas cadastradas.



**Lista de pessoas:**

Nome	Data de Nascimento
Fulano	23/09/2010
Sicrano	23/09/2010
Beltrano	23/09/2010

Perceba que a data ficou num formato legível, de acordo com o padrão de formatação que usamos no Brasil.

## Atividade 02

---

1. Implemente as alterações mostradas relativas à inclusão da data de nascimento de uma pessoa. Reporte se você encontrou dificuldades.
2. Troque a tag `fmt:formatDate` por `c:out`, ou seja, onde tem `<fmt:formatDate value="pessoa.dataDeNascimento"/>`, mude para `<c:out value="pessoa.dataDeNascimento"/>`, mude para `<c:out value="{pessoa.dataDeNascimento}" />` e reporte o que mudou.

## Formatação de datas e números II

---

Percebeu a diferença trabalhada na questão 2 da Atividade 02? O que acontece quando usamos a tag `c:out` é que, quando a expressão `${pessoa.dataDeNascimento}` é avaliada, o resultado é um objeto do tipo `java.util.Date`, mas essa tag trabalha com objetos do tipo `java.lang.String`, então, a tag chama o método `toString()` da classe `java.util.Date`, cujo resultado é o que você viu. Já a tag `fmt:formatDate` espera um objeto do tipo `java.util.Date` e usa seu valor para formatar a data da forma que estamos acostumados a usar, no formato reduzido de dia/mês/ano.

Vamos, agora, examinar todos os atributos da tag `fmt:formatDate` para entendermos seu comportamento padrão e o que podemos fazer para customizar a formatação de uma data e de uma hora.

Atributo	Descrição	Obrigatório	Valor default
<code>type</code>	Define o tipo do valor especificado pelo atributo <code>value</code> . Os tipos válidos são: <code>time</code> , <code>date</code> , <code>both</code> .	Não	Date
<code>dataStyle</code>	Estilos de formatos predefinidos para uma data. É usado se o <code>type</code> for <code>"date"</code> . Os estilos válidos são: <code>default</code> , <code>short</code> , <code>medium</code> , <code>long</code> e <code>full</code> .	Não	Default
<code>timeStyle</code>	Estilos de formatos predefinidos para uma hora. É usado se o <code>type</code> definido for <code>"time"</code> . Os estilos válidos são: <code>default</code> , <code>short</code> , <code>medium</code> , <code>long</code> e <code>full</code> .	Não	Default
<code>pattern</code>	Expressão customizada para formatação de data e hora. Se for definido, sobrescreve os atributos <code>type</code> , <code>dataStyle</code> e <code>timeStyle</code> .	Não	
<code>timeZone</code>	Fuso horário da data a ser exibida.	Não	Fuso default
<code>value</code>	O valor da data a ser formatada.	Não	
<code>var</code>	Nome da variável que armazenará a data/hora formatada.	Não	
<code>scope</code>	Escopo da variável armazenada em <code>var</code> .	Não	Página

**Quadro 7** - Atributos da tag `fmt:formatDate`

Começaremos explicando o atributo `type`. Como vimos no exemplo anterior, se usarmos a tag `c:out` para exibir uma data, são exibidas informações de data e hora. Entretanto, na maioria dos casos, nós estamos apenas interessados na data ou na



hora. Com o uso do atributo `type`, nós podemos definir qual informação será exibida, se a data, hora ou ambos.

Os atributos `dateStyle` e `timeStyle` fornecem algumas formatações básicas para data e hora. Como vimos na tabela anterior, existem cinco tipos predefinidos de estilos para data e hora. Quando usamos o tipo `default`, o resultado para uma data é dia/mês/ano com quatro dígitos. Já para hora, o resultado é hora:minuto:segundo. Para o estilo `short`, temos os seguintes resultados: dia/mês/ano com dois dígitos e a hora igual ao estilo `default`. Como exercício, você pode verificar como ficam os demais estilos.

O atributo `timeZone` indica qual o fuso horário deve ser utilizado para aquela data. Caso o atributo não seja utilizado, o fuso padrão do sistema será o escolhido. Caso seja preenchido, a data obedecerá ao fuso selecionado no atributo, o que permite criar páginas sensíveis aos diferentes horários do planeta.

O atributo `pattern` prevê uma forma alternativa de formatação de data e hora, quando nenhum dos estilos predefinidos pode atender às nossas necessidades. Para entender quais são os possíveis valores desse campo, é necessário entender a classe `java.text.SimpleDateFormat`. Só a título de exemplo, imaginemos que seja necessário exibir apenas a data com seu dia e mês. Nenhum dos tipos predefinidos suporta essa formatação, mas podemos usar no atributo `pattern` o valor `dd/MM`, que, como vimos na tabela, sobrescreve os atributos `type`, `dateStyle` e `timeStyle`.

Os demais atributos já são nossos velhos conhecidos.

## Importante!

Se você tiver alguma dificuldade com padrões de formatação de data e hora, veja [este link](http://download.oracle.com/javase/1.5.0/docs/api/java/text/SimpleDateFormat.html): `<http://download.oracle.com/javase/1.5.0/docs/api/java/text/SimpleDateFormat.html>`.

## Atividade 03

---

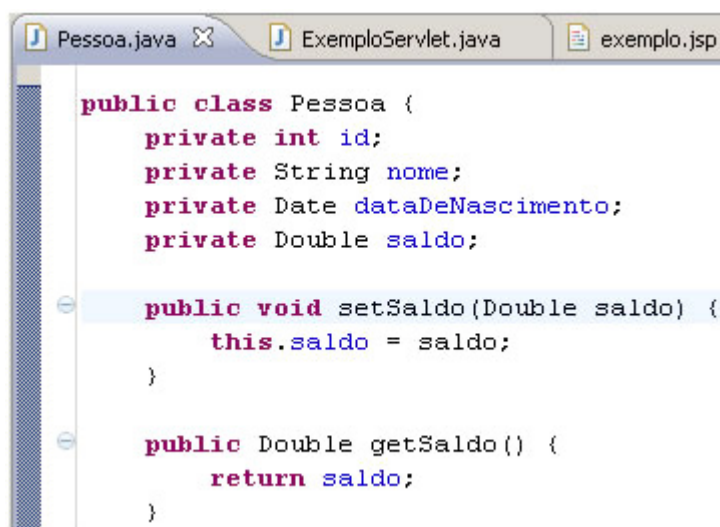
1. Manipule os valores dos atributos `type`, `timeStyle` e `dateStyle` para entender melhor o comportamento deles. Relate o que você entendeu deles.
2. Use o atributo `format` para formatar uma data como dia/mês hora:minuto.
3. Utilizando o conhecimento adquirido nas aulas de JTSL, crie uma página e nela declare três variáveis (nome, data de nascimento, endereço), inicialize-as e apresente o valor dessas variáveis usando JTSL, sempre que possível.

## Formatação de datas e números III

---

Vamos, agora, alterar o nosso exemplo, de forma que a classe `Pessoa` tenha mais um atributo, que é o saldo da conta corrente da pessoa, como mostrado na figura a seguir.

**Figura 10** - Adição do atributo saldo na classe `Pessoa`



```
Pessoa.java X ExemploServlet.java exemplo.jsp

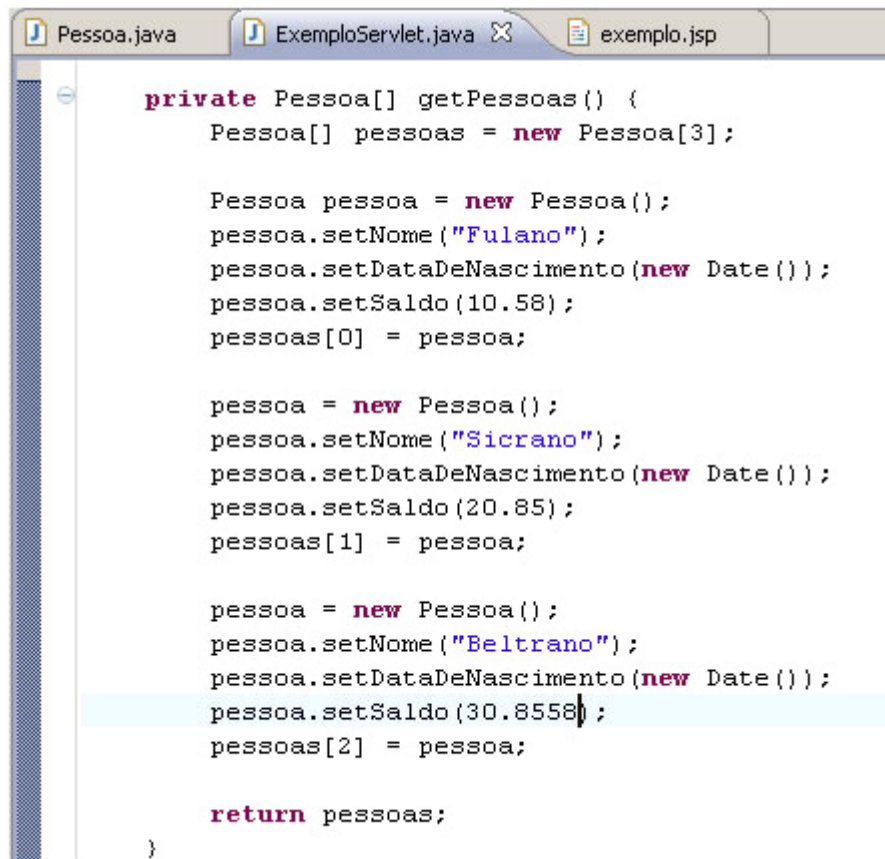
public class Pessoa {
    private int id;
    private String nome;
    private Date dataDeNascimento;
    private Double saldo;

    public void setSaldo(Double saldo) {
        this.saldo = saldo;
    }

    public Double getSaldo() {
        return saldo;
    }
}
```

Agora, vamos alterar o Servlet de exemplo, que lista as pessoas do sistema, para incluir o valor do atributo `saldo`, conforme a figura a seguir.

**Figura 11** - Adaptação do restante do código da classe Pessoa, dado o novo atributo saldo.



```
private Pessoa[] getPessoas() {
    Pessoa[] pessoas = new Pessoa[3];

    Pessoa pessoa = new Pessoa();
    pessoa.setNome("Fulano");
    pessoa.setDataDeNascimento(new Date());
    pessoa.setSaldo(10.58);
    pessoas[0] = pessoa;

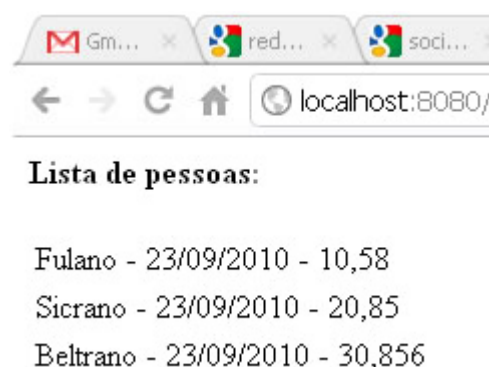
    pessoa = new Pessoa();
    pessoa.setNome("Sicrano");
    pessoa.setDataDeNascimento(new Date());
    pessoa.setSaldo(20.85);
    pessoas[1] = pessoa;

    pessoa = new Pessoa();
    pessoa.setNome("Beltrano");
    pessoa.setDataDeNascimento(new Date());
    pessoa.setSaldo(30.8558);
    pessoas[2] = pessoa;

    return pessoas;
}
```

Devemos alterar, também, o JSP que exibe os dados das pessoas para exibir, além do nome, a data de nascimento e, agora, o saldo, utilizando a tag `fmt:formatNumber`. Veja o resultado:

**Figura 12** - Novo resultado com a apresentação dos valores referentes ao atributo saldo de cada pessoa.



Perceba que o saldo ficou formatado, inclusive com vírgula separando as casas decimais. Repare, entretanto, a última casa decimal do último valor foi arredondada de 30,8558 para 30,856.

## Atividade 04

---

1. Implemente as alterações mostradas relativas à inclusão da data de nascimento de uma pessoa. Reporte se você encontrou dificuldades.
2. Troque a tag `fmt:formatNumber` por `c:out`, ou seja, onde temos `<fmt:formatNumber value="pessoa.saldo"/>` vamos mudar para `<c:out value="pessoa.saldo"/>` vamos mudar para `<c:out value="{pessoa.saldo}" />`.

## Formatação de datas e números IV

---

Percebeu a diferença trabalhada pela troca do `fmt:formatNumber` por `c:out` na questão 2 da Atividade 04? O que acontece quando usamos a tag `c:out` é que perdemos o separador de decimais, porém a última casa decimal do saldo de “Beltrano” foi exibida, ou seja, não há formatação com o `c:out`. O objeto que a tag `c:out` recebe é um `java.lang.Double` e seu método `toString()` retorna o texto do número sem formatação. Já a tag `fmt:formatNumber` espera receber um objeto do tipo `java.lang.Number` (`Double` é uma subclasse de `Number`) para, então, aplicar as formatações adequadas.

Vamos, agora, examinar todos os atributos da tag `fmt:formatNumber` para entendermos seu comportamento padrão e o que podemos fazer para customizar a formatação de um número.

Atributo	Descrição	Obrigatório	Valor default
value	O valor numérico a ser formatado.	Não	Valor definido no body da tag
type	Define o tipo do valor especificado pelo atributo value. Os tipos válidos são: number, currency, percent.	Não	<i>number</i>
pattern	Expressão customizada para formatação. Se for definido, sobrescreve os demais atributos de formatação.	Não	
currencyCode	Código da moeda usado para formatar valores monetários, tais como USD, EUR.	Não	De acordo com o Locale
currencySymbol	Símbolo da moeda (\$, F).	Não	De acordo com o Locale
maxIntegerDigits	Número máximo de dígitos a ser exibido da parte inteira do número.	Não	
minIntegerDigits	Número mínimo de dígitos a ser exibido da parte inteira do número.	Não	
maxFractionDigits	Número máximo de dígitos a ser exibido da parte fracionário do número.	Não	
minFractionDigits	Número mínimo de dígitos a ser exibido da parte fracionário do número.	Não	

Atributo	Descrição	Obrigatório	Valor default
var	Nome da variável que armazenará o número formatado.	Não	
scope	Escopo da variável armazenada em var.	Não	Página

**Quadro 8** - Atributos da tag `fmt:formatNumber`

Os atributos `value`, `var` e `scope` dispensam apresentação, pois são utilizados da mesma maneira que nas outras tags que já vimos. O atributo `currencySymbol` é o prefixo utilizado antes da descrição da moeda e pode receber valores diferentes, como `U` ou simplesmente para dólares. Já o `currencyCode` é um atributo que segue as especificações definidas pelo ISO 4217 ([http://pt.wikipedia.org/wiki/ISO\\_4217](http://pt.wikipedia.org/wiki/ISO_4217)) e define o prefixo da moeda de acordo com as especificações. Ou seja, se o `currencyCode` for definido como “BRL”, por exemplo, automaticamente o prefixo R, padrão para o Brasil, será utilizado. Perceba que esse atributo sobrescreve o atributo `currencySymbol`. Aqui, vale a seguinte explicação: se o atributo `type` for definido como `currency` e o `Locale` estiver definido como `pt BR`, automaticamente, o valor formatado será exibido com R, padrão para o Brasil, será utilizado. Perceba que esse atributo sobrescreve o atributo `currencySymbol`. Aqui, vale a seguinte explicação: se o atributo `type` for definido como `currency` e o `Locale` estiver definido como `pt BR`, automaticamente, o valor formatado será exibido com R. Da mesma forma, se o atributo `type` for definido como `percent`, o símbolo de % será exibido, automaticamente, para você.

Os demais atributos definem a quantidade de dígitos que serão exibidos na parte inteira e fracionária do número. Isso ajuda a, por exemplo, limitar o número de casas decimais a ser exibido. Se `minFractionDigits` for definido para 2 e o número a ser formatado for, por exemplo, 23,2, o valor exibido será 23,20. Por outro lado, se `maxFractionDigits` for igual a 2 e o número a ser exibido for 23,285, o resultado será 23,29. Perceba que houve um arredondamento, e não um truncamento do número, para o número mais próximo. Os atributos `minIntegerDigits` e `maxIntegerDigits` funcionam de forma similar.

Então, mais uma vez, muito fácil, não é?! Nada que um pouco de prática não resolva. Bom, chegamos ao fim da nossa explanação sobre a taglib fmt. Veremos agora a taglib functions (fn), que trata de manipulação de Strings e tamanho de coleções, e EL, que mostra como podemos acessar as variáveis de escopo de uma forma elegante e padronizada.

## A taglib functions

---

Acabamos de ver algumas das tags de internacionalização disponíveis na JSTL. Vamos agora continuar nossos estudos mostrando as funções de JSTL que manipulam Strings e descobrem o tamanho das coleções e Strings, bem como EL (*Expression Language*), que é uma alternativa mais elegante ao uso de scriptlets. Vamos lá, então, começando pelas funções JSTL.

As funções JSTL possuem, basicamente, dois conjuntos de funcionalidades. A primeira trata das funções básicas de String, como trim(), toUpperCase(), substring() etc. A segunda possui a única finalidade de, dada uma coleção ou uma String, descobrir qual o seu tamanho. O quadro a seguir mostra o conjunto de funcionalidades existentes.

Área	Funcionalidade	Tag	Prefixo
Funções	Tamanho de coleções.	Manipulação de Strings.	fn
	Length	toUpperCase, toLowerCase, substring, substringAfter, substringBefore, trim, replace, indexOf, startsWith, endsWith, contains, containsIgnoreCase, split, join, escapeXml	

**Quadro 9** - Funcionalidades detalhadas da biblioteca functions

Cada uma dessas funcionalidades tem a mesma semântica dos métodos da classe java.lang.String, exceto no caso dos tags substringAfter, substringBefore, containsIgnoreCase e join, visto que eles não possuem métodos correspondentes para suas funcionalidades na classe String. Por isso, o nosso foco será justamente

nessas tags. Veremos, também, a funcionalidade `length`. Caso tenha alguma dúvida em relação às funções da classe `Java.lang.String`, busque informações no manual da linguagem.

Antes de começarmos a explicar a taglib functions, é necessário explicar uma pequena diferença entre essa tag e as demais. As taglibs que vimos antes, `core` e `l18N`, definem um conjunto de atributos que são usados para que a tag execute adequadamente. Entretanto, a taglib functions não funciona assim. Na verdade, ela define uma assinatura de método, como seus parâmetros e tipo de retorno, que devem ser invocados de forma similar à invocação de método numa classe Java.

Vamos a um exemplo, ao invés de chamar a funcionalidade *contains*

```
1 <fn:contains var="variavel" value="algumValor" />
```

Você deve chamar assim:

```
1 ${fn:contains(variavel, algumValor)}
```

Percebeu a diferença?

Preparamos um exemplo a seguir, que mostra o uso das taglibs que não possuem função correspondente na classe `String`.

```
1 <%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>
2 <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
3 <html>
4   <body>
5     <c:set var="tempStr" value="Functions e Tags tem sintaxe diferente"/>
6     O tamanho da String tempStr é: ${fn:length(tempStr)}<br/>
7     A substring depois da palavra "Tags" é: ${fn:substringAfter(tempStr,"Tags")}<br />
8     A substring antes da palavra "tem" é: ${fn:substringBefore(tempStr,"tem")}<br />
9     Esta string contém a palavra "tags"? ${fn:containsIgnoreCase(tempStr,"tags")}<br />
10    <br />
11  </body>
12 </html>
```

**Listagem 6** - Código JSP que define o idioma da página JSP para inglês americano.

Nesse exemplo, vemos o uso das tags `substringAfter`, `substringBefore` e `containsIgnoreCase`. Daqui a pouco, veremos um exemplo de `join`. Na linha 05, definimos uma variável `tempStr` usando a tag `c:set`. Na linha 06, pedimos que seja



exibido o tamanho da String referenciada por essa variável. Nesse caso, a função `length` contará o número de caracteres dessa variável.

Na linha 07, será exibida uma String que é uma substring da variável `tempStr` a partir da palavra "Tags". O resultado disso será "tem sintaxe diferente". Já na linha 08, temos um exemplo similar. A diferença é que estamos interessados em exibir apenas a substring que vem antes da palavra "tem", no caso "Functions e Tags". Na linha 09, vemos o uso da funcionalidade `containsIgnoreCase`. Essa funcionalidade verifica se existe dentro de `tempStr` a substring "tags", independente se ela está escrita como "TAGS", "TaGs" etc. Nesse o caso, o resultado será *true*.

A tag `join` tem o comportamento de exibir os elementos de um array como String, onde cada elemento do array será convertido para texto, concatenando-os por meio de um separador indicado do tipo String. Veja o exemplo a seguir.

O nosso servlet de exemplo define um array de Strings como atributo da requisição:

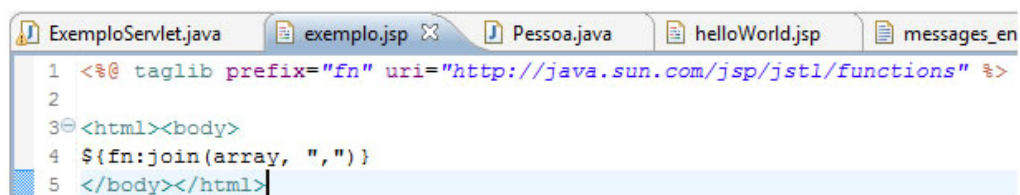
**Figura 13** - Servlet de exemplo que seta atributo e redireciona chamada.



```
1 package aulaJSTL;
2
3 import java.io.IOException;
4
5 import javax.servlet.ServletException;
6 import javax.servlet.http.HttpServlet;
7 import javax.servlet.http.HttpServletRequest;
8 import javax.servlet.http.HttpServletResponse;
9
10 public class ExemploServlet extends HttpServlet {
11     @Override
12     protected void doGet(HttpServletRequest request, HttpServletResponse response)
13         throws ServletException, IOException {
14
15         request.setAttribute("array", new String[] { "Fulano", "Sicrano", "Beltrano" });
16         request.getRequestDispatcher("/exemplo.jsp").forward(request, response);
17     }
18 }
```

No arquivo `exemplo.jsp`, usamos o array definido no Servlet da Figura 13 (linha 15) como parâmetro para a função `join` (linha 4), bem como o parâmetro adicional ",".

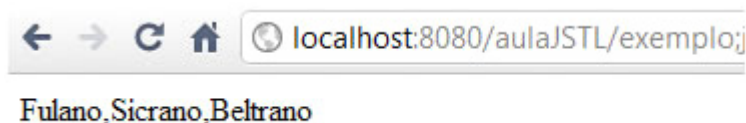
**Figura 14** - Arquivo JSP que manipula array enviado pelo Servlet.



```
1 <@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" >
2
3 <html><body>
4     ${fn:join(array, ",")}
5 </body></html>
```

O resultado é que os elementos do array são exibidos separados por “,”.

**Figura 15** - Valores do array sendo impressos na tela.



Com esse último exemplo, terminamos nossa explicação sobre como usar as funções de JSTL. Agora, veremos EL, considerado um excelente recurso na construção de páginas JSP.

## A taglib functions II

EL (*Expression Language*) é uma forma elegante de não usar scriptlets, porém isso não significa que não existam situações que o uso de scriptlets não se justifique. De qualquer forma, EL deve ser usado no lugar de scriptlets sempre que possível. A principal vantagem em usar EL no lugar de scriptlets é que o código JSP fica mais legível, fácil de entender e manter. As expressões em EL possuem a seguinte sintaxe: `${expressao}`.

Já vimos um pouco de EL enquanto estudamos JSTL, porém sem entrar no detalhe de contexto e funcionamento. Com EL podemos acessar objetos que estão no escopo da página, da requisição, da sessão e da aplicação, sem precisar indicar onde esses objetos estão, pois, automaticamente, a EL procura o objeto em todos esses escopos na ordem apresentada. Por exemplo, digamos que existe um atributo chamado “usuarioLogado” na sessão da minha aplicação. Suponha que queremos exibir o **nome** desse usuário na página. Para isso, temos que usar o seguinte código no JSP:

```
1  ${usuarioLogado.nome}
```

Agora, como faríamos isso usando scriptlets? Seria algo como:

```
1  <%= request.getSession().getAttribute("usuarioLogado").getNome() %>
```

Perceba que, no caso do scriptlet, foi necessário indicar, explicitamente, em qual contexto estava o objeto referente a “usuarioLogado”. Além disso, esse código não é tão robusto, pois se o usuário logado não estiver na sessão, fatalmente, esse código levantará uma exceção `NullPointerException`. Quando usamos EL, o comportamento é procurar por `usuarioLogado`, em cada um dos escopos, e, se o objeto não for encontrado em nenhum deles, o restante da expressão não é avaliada, evitando, por exemplo, a tentativa de avaliar a propriedade `nome` em uma variável nula.

Perceba que a sintaxe do EL é muito mais sucinta que a do scriptlet, o que melhora o entendimento do código. Perceba, também, que seria possível escrever um código mais robusto usando scriptlets, porém seria muito mais trabalhoso e menos produtivo, como mostra o exemplo a seguir.

```
1 <%@ page import="aulaJSTL.Usuario" %>
2 <%
3 String nome = "";
4 if (session != null && session.getAttribute("usuarioLogado") != null) {
5     Usuario usuario = (Usuario) session.getAttribute("usuarioLogado");
6     nome = usuario.getNome();
7 }
8 %>
9 <%= nome %>
```

Esse exemplo mostra porque é preferível usar EL ao invés de scriptlets. Desde a implementação 2.2 da EL, que é a que estamos utilizando nesse curso, juntamente aos Servlets 3.0 e o Tomcat 7, passou a ser possível utilizar também métodos que não são *getters* ou *setters* através de EL. Essa chamada ocorre de maneira similar às chamadas regulares. Supondo que tivéssemos uma classe `JavaBean` chamada **Pessoa** e essa classe tivesse um método declarado como **public String falar()**, poderíamos invocar esse método usando EL através do comando **\$(pessoa.falar())** – ‘pessoa’ sendo um objeto em memória do tipo **Pessoa**. Isso quebrou as últimas barreiras para a utilização de EL no lugar de scriptlets.

Observemos, ainda, que para acessar a propriedade `nome` do objeto `usuarioLogado`, precisamos apenas usar `usuarioLogado.nome`. Mas essa não é a única maneira de acessar a propriedade `nome`. Podemos, também, usar anotação `usuarioLogado.nome`. Mas essa não é a única maneira de acessar a propriedade `nome`. Podemos, também, usar a notação `{usuarioLogado["nome"]}`. Essa forma é possível e pode ser utilizada tanto para um `Java Bean` que contém o atributo “nome” quanto para um `java.util.Map` ‘usuarioLogado’ que possui a chave “nome”. Vamos

supor que na sessão o objeto associado a `usuarioLogado` fosse um `Map`, ao invés de um `Usuario`. A forma de acessar os valores desse mapa seria algo como `nomeDoMapa["nomeDaChave"]`. Da mesma forma, imaginemos agora que o atributo na sessão fosse um `java.util.List` ou um `array`. A forma de acessar os valores nessa lista seria `nomeDoMapa["nomeDaChave"]`. Da mesma forma, imaginemos agora que o atributo na sessão fosse um `java.util.List` ou um `array`. A forma de acessar os valores nessa lista seria `{nomeDaLista[indice]}`. Legal, não é mesmo?

## A taglib functions III

---

Uma vez que vimos como podemos acessar objetos, mapas, listas, arrays e suas propriedades, vamos agora entender mais um detalhe sobre EL. Da mesma forma que dentro dos códigos scriptlets temos algumas variáveis que estão implícitas, em EL isso também é verdade. Por padrão, `algumaCoisa` procura por uma variável com nome `algumaCoisa`, em algum dos escopos possíveis. Porém, é possível deixar explícito qual o escopo usando um nome de variável que representa cada um dos escopos. Por exemplo, `algumaCoisa` procura por uma variável com nome `algumaCoisa`, em algum dos escopos possíveis. Porém, é possível deixar explícito qual o escopo usando um nome de variável que representa cada um dos escopos. Por exemplo, `{sessionScope.usuarioLogado}` ou `${pageScope.pessoas}`, pois `sessionScope` e `pageScope` são variáveis implícitas do contexto EL. O quadro a seguir exhibe os contextos implícitos disponíveis.

Objeto	Descrição
<code>pageScope</code>	Define as variáveis que estão no escopo da página.
<code>requestScope</code>	Define as variáveis que estão no escopo da requisição.
<code>sessionScope</code>	Define as variáveis que estão no escopo da sessão.
<code>applicationScope</code>	Define as variáveis que estão no escopo da aplicação.
<code>param</code>	Mapa que contém o valor parâmetros da requisição.

Objeto	Descrição
paramValues	Mapa que contém o valor dos parâmetros da requisição como array de Strings.
header	Mapa que contém o nome e valor dos headers da requisição.
headerValues	Mapa que contém o nome e o array de valores dos headers da requisição.
cookie	Mapa do nome dos cookies e o cookie.
initParam	Mapa com o nome dos parâmetros iniciais e seus valores.

**Quadro 10** - Variáveis de contexto implícitas que se encontram disponíveis para o JSP.

Cada uma dessas variáveis representa um objeto Java já visto por nós. Por exemplo, quando usamos `${param.code}`, estamos, na verdade, fazendo `request.getParameter("code")`. Bem mais fácil que usar scriptlets, não é?!

Outra coisa muito legal de EL é o uso de operadores de aritmética, de lógica e relacionais. A seguir, mostraremos quais são esses operadores e como usá-los.

Operador	Descrição
+	Adição.
-	Subtração.
*	Multiplicação.
/ (div)	Divisão.
% (mod)	Módulo da divisão (resto).

Operador	Descrição
== (eq)	Igualdade.
!= (ne)	Desigualdade.
< (lt)	Menor que.
> (gt)	Maior que.
<= (le)	Menor ou igual.
>= (ge)	Maior ou igual.
&& (and)	Verdadeiro, se ambos operandos são verdadeiros; Falso, caso contrário.
(or)	Verdadeiro, se um ou ambos operandos forem verdadeiros; Falso, caso contrário.
! (not)	Verdadeiro, se operando for verdadeiro; Falso, caso contrário.
Empty	Verdadeiro, se o operando for nulo, uma String vazia, um mapa vazio ou uma lista vazia; Falso, caso contrário.

#### **Quadro 11** - Operadores suportados pela EL

Bom, operações é que não faltam. Basicamente, essas são as principais operações de lógica e aritmética da linguagem Java. Lembram-se da atividade que listava as pessoas do sistema? Pois bem, e se quiséssemos pintar as linhas ímpares da tabela de uma cor e as pares de outra, como poderíamos fazer? Abaixo, mostramos uma possível maneira fazer isso.

```

1 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
2 <html>
3   <body>
4     <strong>Lista de pessoas:</strong>
5     <%
6       request.setAttribute("pessoas", new String[] {"joao", "maria", "joaquim", "Luana", "eduardo"});
7     %>
8     <br><br>
9     <table>
10      <c:forEach var="pessoa" items="${pessoas}" varStatus="status">
11        <c:choose>
12          <c:when test="${status.index % 2 == 0}">
13            <c:set var="bgColor" value="GRAY" />
14          </c:when>
15          <c:otherwise>
16            <c:set var="bgColor" value="WHITE" />
17          </c:otherwise>
18        </c:choose>
19        <tr bgcolor="<c:out value="${bgColor}" />">
20          <td><c:out value="${pessoa}" /></td>
21        </tr>
22      </c:forEach>
23    </table>
24  </body>
25 </html>

```

**Listagem 7** - Exemplo de uso de operadores lógicos.

Observe que no `c:forEach` usamos o atributo `varStatus` para ter acesso ao contexto de loop da tag. Isso permite que possamos utilizar a variável "status" como uma forma de saber, por exemplo, qual o número da iteração que estamos. Dentro do loop, temos a tag `c:choose`, na linha 07, para escolher se a linha será cinza ou branca, de acordo com o índice da iteração atual. Se o `status.index % 2` for zero, a linha será cinza; caso contrário, será branca. Aqui vale comentar uma coisa: no lugar do operador `%` poderíamos usar `mod`, e no lugar do operador `==` poderíamos usar `eq`.

Então, viu como é fácil, poderoso e claro usar EL no lugar de scriptlets? Espero que daqui pra frente scriptlets seja coisa do passado e você comece a usar JSTL e EL nas suas páginas JSP. Você perceberá como o seu código ficará mais legível e mais fácil de manter.

Bom, pessoal, terminamos de comentar o que existe de mais importante no uso de JSTL e EL. Espero que vocês tenham aproveitado e entendido.

## Atividade 05

---

1. Quais são as duas maneiras de testar se uma coleção é vazia?
2. Veja o que acontece ao tentar usar uma expressão da seguinte forma:  
`${usuarioLogado.nome + 1}` .
3. Usando os conceitos trabalhados nesta aula, siga os seguintes passos:
  - a. Escreva uma página JSP `novaMensagem.jsp` que mostre formulário na tela com dois campos: e-mail e mensagem.
  - b. Escreva outra página `gravarMensagem.jsp` que receba parâmetros: e-mail e mensagem, e grave esses dois parâmetros na sessão do usuário.
  - c. Faça com que a primeira página aponte para a segunda.
  - d. Crie uma terceira página `listarMensagens.jsp` que mostre mensagens criadas até o momento, indicando, inclusive, seus tamanhos (quantidade de caracteres).



## Leitura Complementar

---

Para complementar seu aprendizado, leia sobre os comandos aprendidos nesta aula por meio das referências a seguir. A primeira apresenta um guia de referência rápida do JSTL, poderá ser bastante útil para você ao longo do curso. A segunda referência apresenta exemplos de uso da biblioteca *functions* JSTL. Já a terceira é o tutorial oficial de Java para JSTL, sendo um ótimo recurso para reforço dos seus conhecimentos. Analise essas referências de acordo com o conteúdo já visto nas aulas.

- <<http://cs.roosevelt.edu/eric/books/JSP/jstl-quick-reference.pdf>>
- <<http://www.java2s.com/Code/Java/JSTL/UsingtheJSTLfunctions.htm>>
- <<http://download.oracle.com/javasee/5/tutorial/doc/bnakc.html>>

## Resumo

---

Hoje, concluímos a taglib de internacionalização I18N do JSTL. Você viu que podemos configurar nossa aplicação com as tags `fmt:setLocale` e `fmt:setBundle`. Adicionalmente, viu que podemos sobrescrever o arquivo de propriedades definido pela tag `fmt:setBundle` usando a tag `fmt:bundle`. Aprendeu como usar as mensagens que estão definidas dentro do arquivo de propriedades e como passar parâmetros para as mensagens parametrizadas. Com o conhecimento dessa taglib, já é possível fazer uma aplicação que suporte várias localizações, sendo bastante reduzido o esforço para isso, trabalhando-se, principalmente, com a criação de novos arquivos de propriedades.

Vimos também taglib `functions` para manipulação de Strings e a linguagem de expressão (EL) para acesso a valores e expressões de uma forma mais simples. Com esse conhecimento, já é possível fazer bastante coisa de forma organizada e de fácil manutenção.

# Autoavaliação

---

Descreva com as suas palavras:

- a. O que pode ser feito com a taglib *functions*?
- b. Como a linguagem de expressões (EL) ajuda na escrita do código JSP?

## Referências

---

AHMED, K. Z.; UMRYSH, C. E. **Desenvolvendo aplicações comerciais em Java com Java J2EE e UML**. Rio de Janeiro: Ciência Moderna, 2003. 324 p.

BASHAM, Bryan; SIERRA, Kathy; BATES, Bert. **Head First Servlets and JSP: passing the Sun Certified Web Component Developer Exam (SCWCD)**. 2. ed. O'Reilly Media, 2008.

CATTELL, Rick; INSCORE, Jim. **J2EE: criando aplicações comerciais**. Rio de Janeiro: Campus, 2001.

HALL, Marty. **More Servlets and JavaServer Pages (JSP)**. New Jersey: Prentice Hall PTR (InformIT), Sun Microsystems Core Series, 2001. 752 p. Disponível em: <<http://pdf.moreservlets.com/>>. Acesso em: 16 ago. 2012.

HALL, Marty; BROWN, Larry. **Core Servlets and JavaServer Pages (JSP)**. 2. ed. New Jersey: Prentice Hall PTR (InformIT), Sun Microsystems Core Series, 2003. 736 p. (Core Technologies, 1). Disponível em: <<http://pdf.coreservlets.com/>>. Acesso em: 16 ago. 2012.

HYPERTEXT. **Transfer Protocol -- HTTP/1.1: methods definition**. Disponível em: <<http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html>>. Acesso em: 16 ago. 2012.

ORACLE. **Java EE Reference at a Glance**. Disponível em: <<http://java.sun.com/javaee/reference/tutorials/>>. Acesso em: 16 ago. 2012.