JSTL Tag-Library

http://www.tutego.com/

Custom-Tags und Tag-Library

- JSPs bestehen im Kern aus Template-Code.
  - Der JSP-Servlet-Übersetzer kopiert sie die Ausgabeseite.
- Der JSP-Compiler kann jedoch bei gewissen Tags programmierte Aktionen vornehmen.

```
<s:SkypePresence skypeId="c.ullenboom" />
```

  - Der Servlet-Container schreibt bei <s:SkypePresence > (Tag SkypePresence im Namensraum s) den Skype-Status (etwa ONLINE) in den Ausgabestrom.
- Da man grundsätzlich eigene Tags mit speziellem Verhalten definieren kann, spricht man von **Custom-Tags**.
  - Custom-Tags werden in einer **Tag-Library** (kurz **Tag-Lib**) zusammengefasst.

Warum Tag-Libraries?

- **Trennen von Logik und Visualisierung**
  - Ein Custom-Tag hält die Web-Seite von eingebettetem Java-Quellcode (Scriplets) frei. Tools können sich nicht an der Seiten-Syntax „verschlucken“.
- **Gute Lesbarkeit**
  - Hinter der einfachen Syntax kann beliebige Logik stehen. Web-Designer werden nicht überfordert.
- **Wiederverwendbarkeit**
  - Es gibt Tag-Libraries für fast alles. Eigene Tag-Libs lassen sich in unterschiedlichen Projekten wiederverwenden.
- **Einfache Entwicklung**
  - Da der Kern einer Tag-Lib eine Java-Klasse ist, wird die Entwicklung über die IDE erleichtert.

Was können Custom-Tags?

- be customized via attributes passed from the calling page, either staticly or determined at runtime;
- have access to all the objects available to JSP pages including request, response, in and out;
- modify the response generated by the calling page;
- communicate with each other; you can create and initialize a JavaBeans component, create a variable that refers to that bean in one tag, and then use the bean in another tag;
- be nested within one another, allowing for complex interactions within a JSP page; and

JavaServer Pages Standard Tag Library

http://www.tutego.com/

JavaServer Pages Standard Tag Library

- JSP-Programmierer benötigen oft Tags für alltägliche Aufgaben.
  - Fallunterscheidungen, ob ein Benutzer angemeldet ist.
  - Durchlaufen einer Ergebnisliste für eine Tabelle.
- Die **JavaServer Pages Standard Tag Library (JSTL)** ist eine Standard Tag-Bibliothek.
  - Sie ist von Sun standardisiert, gut dokumentiert und sehr verbreitet.

JSTL-Beispiel

```
<fmt:formatDate type="both" timeStyle="short"
                value="${date}" />

<table>

<c:forEach var="i" begin="1" end="10" >
 <tr>
   <td> ${i} </td>
   <td> ${i*i} </td>
 </tr>
</c:forEach>
</table>
```

In den JSTL-Tags können EL-Ausdrücke verwendet werden.

Inhalt der JSTL

- Die Abkürzung steht zwar für JavaServer Pages Standard Tag Library, doch die JSTL steht nicht für **eine** Tag-Library, sondern für eine Sammlung von fünf Tag-Bibliotheken.

Kern-Funktionalität (Core)

- Variablen, Ein-/Ausgabe, Fallunterscheidungen, Iteration

Formatierung/I18n

- Message-Bundles, Zahlen, Datum

XML-Operationen

- Parsen, XSLT-Transformationen, XPath

Datenbankoperationen

- Aufbau von Verbindungen

String-Funktionen für die EL

Versionen der JSTL

- Von der JSTL gibt es drei Versionen:
  - **JSTL 1.0**. Basiert auf JSP 1.2-Spezifikation.
  - **JSTL 1.1**. Basiert auf JSP 2.0- und Servlet 2.4-Spezifikation (realisiert etwa von Tomcat 5).
  - **JSTL 1.2**. Kleine Überarbeitung (Maintenance Release) von JSTL 1.1. Vereinheitlicht mit der „Unified Expression Language (EL)“ die Nutzung aus JavaServer Faces.
    (Uns reicht im Kurs JSTL 1.1).
- Laufen Web-Anwendungen auf einem Application-Server (JBoss, …), bringt dieser eine JSTL-Implementierung mit.
  - JSTL 1.1 ist Teil von J2EE 1.4
  - JSTL 1.2 ist Teil von Java EE 5
- Um Web-Anwendungen auf einem „normalen“ Servlet-Container zu entwickeln, ist eine Implementierung nötig.

Jakarta Taglibs

- Die Referenz-Implementierung für JSTL 1.1 liegt unter
  - *http://jakarta.apache.org/taglibs/index.html*

The Jakarta Taglibs Project - Mozilla Firefox

Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe

http://jakarta.apache.org/taglibs/index.html

Google

The Jakarta Project
http://jakarta.apache.org

Taglibs

Jakarta Taglibs

Home
Downloads
Using
Taglib Tutorial
Reporting Bugs

Digging into Jakarta Taglibs

Jakarta Taglibs

This project is an open source repository for JSP custom tag libraries. For a general description, please see the official java.sun.com page on JSP(tm) Tag Libraries.

Jakarta Taglibs also hosts the development tree of the reference implementation (RI) for the JSTL specification - the Jakarta Standard Taglib. Versions 1.0 and 1.1 of JSTL (JSP Standard Tag Library) are implemented here, currently there are no plans for a 1.2 implementation and you are advised to investigate the Glassfish implementation.

News

08/04/2005	[RDC] The Reusable Dialog Components (RDC) Taglib has migrated to Taglibs-Proper after its graduation from Taglibs-Sandbox.

Fertig

Adblock

jakarta-taglibs-standard-1.1.2.zip

- Die Taglib-Dateien gibt es unter der URL *jakarta.apache.org/site/downloads/downloads_taglibs-standard.cgi*

The Jakarta Site - Standard 1.1 Taglib Downloads - Mozilla Firefox

Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe

http://jakarta.apache.org/site/downloads/downloads_taglibs-standard.cgi

The Apache Jakarta Project
http://jakarta.apache.org/

ApacheCon US November 12-16 2007 Atlanta Georgia, USA

Support

- Downloads
- Subversion
- Mailing Lists
- Bug Database
- Jakarta Wiki

Subprojects

- BCEL
- BSF
- Cactus
- Commons
- ECS
- HttpComponents
- JCS
- JMeter
- ORO
- Regexp
- Slide
- Taglibs

Standard 1.1 Taglib Downloads

We recommend you use a mirror to download our release builds, but you **must** verify the integrity of the downloaded files using signatures downloaded from our main distribution directories. Recent releases (48 hours) may not yet be available from the mirrors.

You are currently using **http://www.internet.bs/apache.org**. If you encounter a problem with this mirror, please select another mirror. If all mirrors are failing, there are *backup* mirrors (at the end of the mirrors list) that should be available.

Other mirrors: http://www.internet.bs/apache.org [Change]

The KEYS link links to the code signing keys used to sign the product. The PGP link downloads the OpenPGP compatible signature from our main site.

For more information concerning Standard 1.1 Taglib, see the Standard 1.1 Taglib site.

KEYS

- 1.1.2.tar.gz
  - [pgp]
- 1.1.2.zip
  - [pgp]

About Jakarta

- Welcome
- News
- Contacting Us
- Who We Are
- Charter
- Quarterly Reports
- Retired Projects

Reference

- ASF Website
- Sponsors of the ASF
- How The ASF Works
- Developer Resources
- Get Involved
- Project Guidelines

Search Jakarta

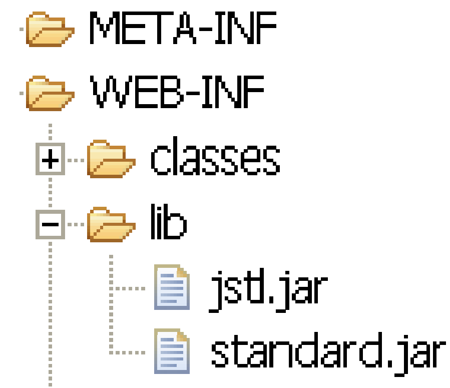
[Search]

Fertig Adblock

Kopieren

- *jakarta-taglibs-standard-1.1.2.zip* ist das Archiv, was man zunächst auspacken muss.
- Der Ordner *lib* enthält die beiden nötigen Java-Archive.
  - *jstl.jar* API
  - *standard.jar* Implementierung
- Die beiden Jar-Dateien kopiert man in den eigenen *WEB-INF/lib*-Ordner.

Das Ziel: WEB-INF/lib



Die Core-Tags

Core-Tags einbinden

- Die Core-Tags werden über den Tag Lib Deskriptor *c.tld* beschrieben. Sie bekommen den Namensraum »c« zugewiesen.
  - Es gibt kaum einen Grund, diesen Namensraum zu ändern!

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
           prefix="c"%>
<c:out value="${100 + 100}" />
```

Dokumentation der JSTL-Tags

out (TLDDoc Generated Documentation) - Mozilla Firefox

Datei Bearbeiten Ansicht Gehe Lesezeichen Extras Hilfe

http://java.sun.com/products/jsp/jstl/1.1/docs/tlddocs/

TLDDoc Generated

All Tags / Functions

Tag Libraries

JSTL core
JSTL fmt

All Tags / Functions

c:catch
c:choose
c:forEach
c:forTokens
c:if
c:import
c:otherwise
c:out
c:param
c:redirect
c:remove
c:set
c:url
c:when
fmt:bundle
fmt:formatDate
fmt:formatNumber

Overview Library Tag Help

FRAMES NO FRAMES

JSTL core

Tag out

Like <%= ... >, but for expressions.

Tag Information	
Tag Class	org.apache.taglibs.standard.tag.rt.core.OutTag
TagExtraInfo Class	*None*
Body Content	JSP
Display Name	*None*

Attributes

Name	Required	Request-time	Type	Description
value	true	true	java.lang.String	Expression to be evaluated.
default	false	true	java.lang.String	Default value if the resulting value is null.
escapeXml	false	true	java.lang.String	Determines whether characters <,>,&,'," in the resulting string should

Überblick über die Core-Tags

- Core-Tags dienen der Ausgabe, URL-Behandlung, imperatives Programmieren. Allgemeine Tags sind:
  - `<c:out>`
  - `<c:set>`
  - `<c:remove>`
  - `<c:catch>`
- Tags für Fallunterscheidungen
  - `<c:if>`, `<c:choose>`, `<c:when>`, `<c:otherwise>`
- Tags zur Iterationen
  - `<c:forEach>`, `<c:forTokens>`
- Tags zur URL-Behandlung
  - `<c:url>`, `<c:param>`, `<c:redirect>`, `<c:import>`

<c:out>

- Mit der EL lässt sich ein Ausdruck ausgeben, aber es gibt auch ein JSTL-Tag dafür: <c:out>.

```
<c:out   value="value"
       [ default="defaultValue" ]
       [ escapeXml="bool" ] />
```

- Die Funktionalität ist mit EL-Ausdrücken und einem Aufruf von <jsp:getProperty> für Properties vergleichbar.
- Liefert das Attribut value den Wert null, so kann man mit dem Attribut default einen Alternativwert angeben.
  - Der Default-Wert kann auch im Body vorgegeben werden.

Beispiele von <c:out>

```
<c:out value="100" />
```
100

```
<c:out value="${100+100}" />
```
200

```
<c:out value="${nix}" />
```

```
<c:out value="${nix}" default="Nix da" />
```
Nix da

```
<jsp:useBean id="sb" scope="page"
  class="java.util.Date" />
```

```
<c:out value="${sb}" />
```
Thu Aug 28 10:57:24 CEST 2003

```
<c:out value="${sb.year + 1900}" />
```
2003

<c:out> mit escapeXml

- Mit dem Tag kann man etwas machen, was mit ${} nicht funktioniert:
  - Das Attribut escapeXml steuert, ob Zeichen in XML-Entities umgewandelt werden sollen. Der Standard ist true.

```
<c:out value="<" />                          ⇨ &lt;
<c:out value="<" escapeXml="false" />        ⇨ <
```

Die Umwandlungen sind:

- < ⇨ <
- > ⇨ >
- & ⇨ &
- ’ ⇨ '
- ” ⇨ "

<c:set> Tag

- Mit dem Tag <c:set> lässt sich ein Ausdruck auswerten und das Ergebnis einer Variablen im Scope zuweisen.

```
<c:set   var="varName"
         value="value"
       [ scope="varScope" ] />
```

- Der Scope einer Variablen ist entweder page (Standard), request, session oder application.
- Der Wert der Variablen kann auch im Body spezifiziert werden.

Beispiele für Konvertierung

```
<c:set var="gruß" value="Moin" />
${gruß}
```

Moin

```
<c:set var="gruß" value="Moin" />
<c:out value="${gruß + 1}" />
```

Tomcat-Fehler

```
<c:set var="zahl" value="1000" />
<c:out value="${zahl + 1}" />
```

1001

<c:set> Tag für Maps oder Beans

- Vom <c:set>-Tag gibt es noch eine Variante.
  - Sie wertet das Ergebnis aus und setzt eine Map oder eine Bean-Eigenschaft.
  - Sie kommt der Benutzung von <jsp:setProperty> gleich.

```
<c:set   target="beanOrMap"
         property="propertyOrKey"
         value="value" />
```

- Das value kann auch wieder im Body stehen.

<c:remove>

- Das Tag <c:remove> entfernt benannte Variablen aus dem Scope.

```
<c:remove    var="varName"
           [ scope="varScope" ] />
```

- Der Scope ist entweder page (Standard), request, session oder application.

<c:if>

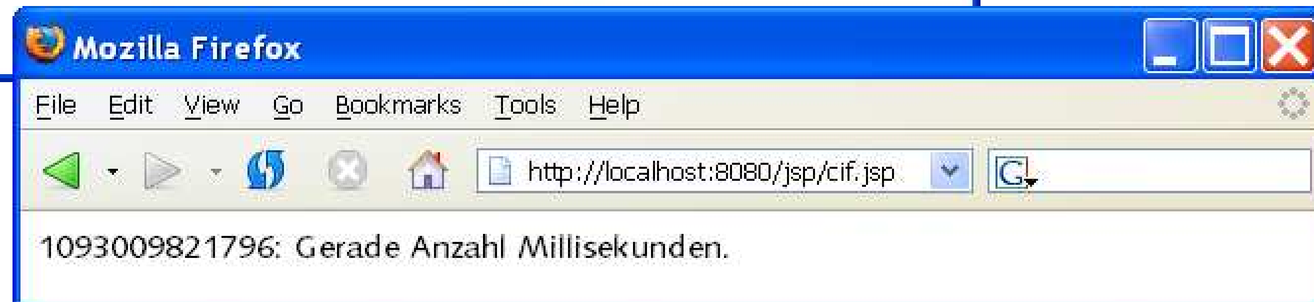
- Fallunterscheidungen müssen nicht mit Java-Code gemacht werden.
  - Dafür gibt es das Tag <c:if>.

```
<c:if   test="condition"
      [ scope="varScope" ]  >
 Körper
</c:if>
```

- Wenn eine Bedingung gilt, so wird der Köper ausgewertet.

Beispiel für <c:if> (1/2)

```
<%@ taglib prefix="c" uri="/WEB-INF/tld/c.tld" %>
<jsp:useBean id="datum" class="java.util.Date" />
${datum.time}:
<c:if test="${datum.time mod 2 == 0}">
 Gerade Anzahl Millisekunden.
</c:if>
```



Beispiel für <c:if> (2/2)

- Strings lassen sich mit == vergleichen.

```
<c:set var="s" value="Moin" />
<c:if test="${s == 'Moin'}">
 s ist Moin
</c:if> <br>
<c:if test="${s != 'Hund'}">
  s ist nicht Hund
  </c:if> <br>
<c:if test="${! (s == 'Hund') }">
  s ist nicht Hund
</c:if> <br>
```

Fallunterscheidung mit Variable

- Man kann den Körper auch weglassen und das Ergebnis des Ausdrucks einer Variable zuweisen.

```
<c:if   test="condition"
        var="varName"
        scope="varScope">
</c:if>
```

- Das Folgende setzt false in die Variable ergebnis:

```
<c:if test="${2 < 2}" var="ergebnis" />
${ergebnis}
```

- Körper und Variable können auch beide gleichzeitig angegeben sein!
  - Aber: Gibt es keinen Körper, muss es eine Variable geben.

<c:choose><c:when><c:otherwise>

- Bei Fallunterscheidungen kommt es oftmals zu kaskadierten if-Anweisungen. Doch das <c:if> kennt kein else!
- <c:choose> realisiert eine Fallunterscheidung mit JSTL.

```
<c:choose>
 <c:when test="Bedingung">
 Körper
 </c:when>
[ <c:otherwise>
  Körper
 </c:otherwise> ]
</c:choose>
```

Auch noch andere when möglich.

Choose und when

- In einem <c:choose> kann eine beliebige Anzahl von <c:when>-Anweisungen liegen.
  - Eine muss es aber immer geben.
- Wenn die erste Bedingung gültig ausgeführt wurde, ist <c:choose> zu Ende. Keine weiteren Tests werden gemacht.
  - Es gibt also nicht so etwas wie ein anti-break bei switch.
- Falls die Bedingungen alle nicht gelten, kann ein <c:otherwise> ausgeführt werden.
  - Davon darf es höchstens einen geben.
  - Es muss am Ende nach allen <c:when> stehen! (Es gibt zwar keinen Compilerfehler, aber zumindest bei Tomcat eine fasche Auswertung.)
- Mit choose und otherwise lässt sich ein einfaches if/else nachbauen.

Alles abhängig vom Alter…

```
<c:set var="alter" value="1" />
<c:choose>
 <c:when test="${alter < 16}">
 Kind
 </c:when>
 <c:when test="${alter >= 16 and alter < 18}">
  Jugendlich
 </c:when>
 <c:when test="${alter >= 18 and alter < 60}">
  Volljährig
 </c:when>
 <c:otherwise> Das reife Alter </c:otherwise>
</c:choose>
```

<c:forEach>

- Schleifen bauen das Tag <c:forEach> auf.
  - Es gibt einen Schleifenzähler, einen Start und Endwert und optional eine Schrittweite.

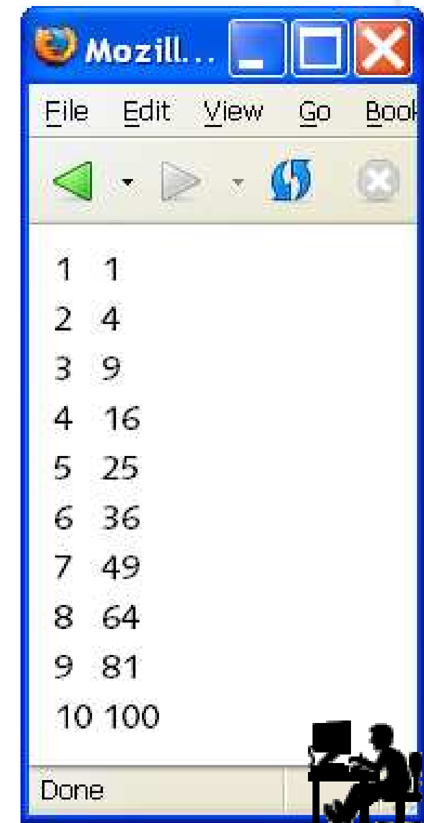
```
<c:forEach  var="varName"
            begin="begin" end="end"
          [ varStatus="varStatusName" ]
          [ step="step" ] >
  Körper
</c:forEach>
```

- Das Attribut var definiert den Schleifenzähler. Über ihn kann man im Rumpf auf das aktuelle Element zugreifen. Dieser ist nur in der Schleife selbst sichtbar und definiert.

Quadrattabelle

- Mit der Schleife soll eine Quadratzahlentabelle geschrieben werden.
  - Die erste Spalte soll die Zahlen, die zweite Spalte die Quadratzahlen enthalten.

```
<table>
<c:forEach var="i" begin="1" end="10">
 <tr>
   <td> ${i} </td>
   <td> ${i*i} </td>
 </tr>
</c:forEach>
</table>
```



<c:forEach> über Collections

- Das <c:forEach>-Tag kann noch mehr: Mit ihm kann man über Sammlungen iterieren.

```
<c:forEach  var="varName"
            items="collection"
          [ varStatus="varStatusName" ]
          [ begin="begin" ] [ end="end" ]
          [ step="step" ] >
 Körper
</c:forEach>
```

- Bei diesem Iterator ist die Angabe von begin, end und step optional.

item-Typen bei <c:forEach>

- Der Wert für items kann ganz unterschiedlich sein:
  - Felder, auch Felder mit Primitiven.
  - java.util.Collection (und damit List, LinkedList, ArrayList, Vector, Stack, Set, …)
  - java.util.Map (und somit HashMap, Hashtable, Properties, Provider, Attributes, …)
  - java.util.Iterator, java.util.Enumeration
- Im Fall der Map zeigt die Variable auf ein Map.Entry-Objekt.
  - Entry hat die Properties key und value.

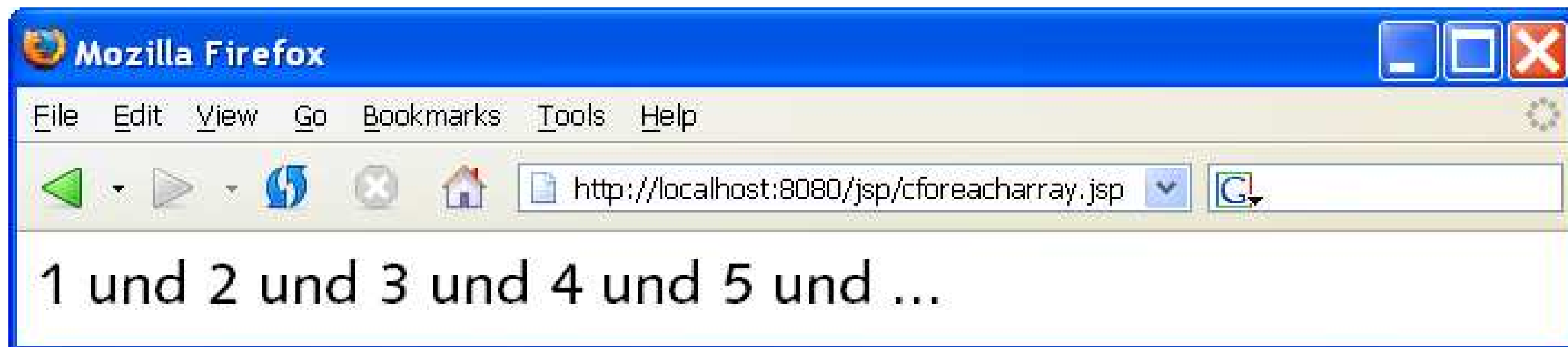
Beispiel für <c:forEach> über Felder

```
<jsp:useBean id="format"
             class="java.text.SimpleDateFormat" />
<c:forEach var="w"
      items="${format.dateFormatSymbols.weekdays}">
  ${w}
</c:forEach>
```

<c:forEach> über kommasep. Strings

- Auch kann das Item ein String sein, der seine Komponenten mit Komma trennt.
  - Das ist aber deprecated!

```
<c:forEach var="i" items="1, 2, 3, 4, 5">
  ${i} und
</c:forEach>
…
```

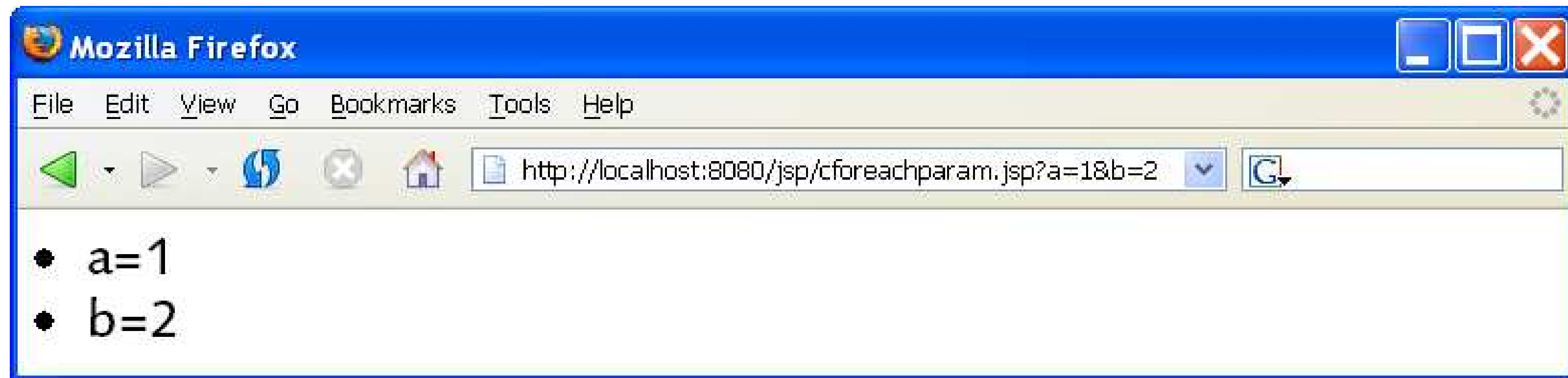


Mit <c:forEach> über Parameter

- Wir erinnern uns: Hinter der impliziten Variablen param verbirgt sich eine Sammlung von übergebenen Parametern.
- Iterieren über alle Parameter.

```
<%@ taglib prefix="c" uri="/WEB-INF/tld/c.tld" %>
<c:forEach var="val" items="${param}">
 <li> ${val}
</c:forEach>
```

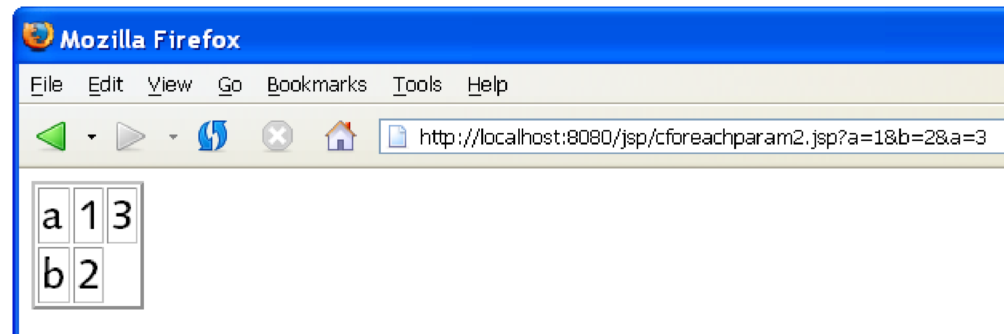
cforeachparam.jsp?a=1&b=2



Auslesen aller Parameter

```
<%@ taglib prefix="c" uri="/WEB-INF/tld/c.tld" %>
<table border="2">
<c:forEach var="current" items="${param}">
 <tr>
  <td> ${current.key} </td>
   <c:forEach var="aVal"
              items="${paramValues[current.key]}">
    <td> ${aVal} </td>
   </c:forEach>
   </tr>
</c:forEach>
</table>
```

Aufruf mit cforeachparam2.jsp?a=1&b=2&a=3



Attribut varStatus bei <c:forEach>

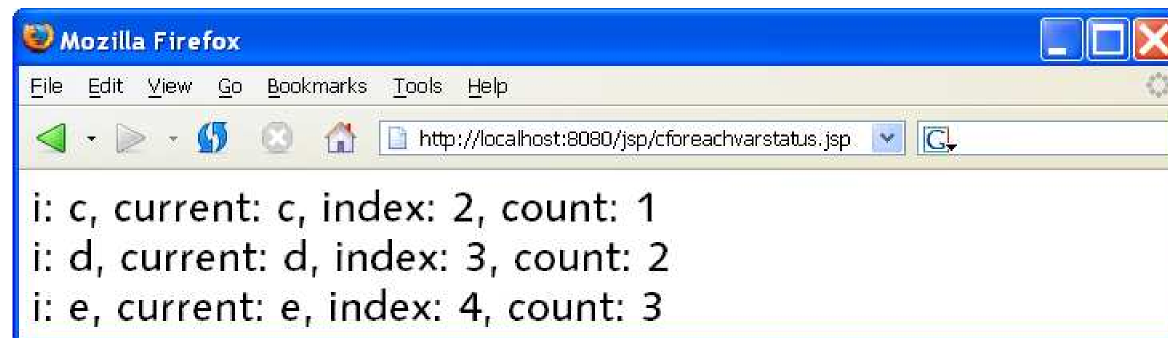
- Bei Schleifen lässt sich das Attribut varStatus nutzen.
  - Mit ihr kann man genaueres über den aktuellen Iterations-Status herausfinden.
  - Die Variable ist vom Typ javax.servlet.jsp.jstl.core.LoopTagStatus und ist auch nur im Schleifenbereich sichtbar.
- LoopTagStatus definiert die Funktionen:
  - getCurrent() aktuelles Element (die Variable).
  - getIndex() aktueller Index (Start bei 0, Initialisiert durch das Attribut begin.)
  - getCount() Zähler. Beginnt bei 1. Unabhängig von begin.
  - isFirst(), isLast()
  - getBegin(), getEnd(), getStep()

Beispiel für varStatus

```
<%@ taglib prefix="c" uri="/WEB-INF/tld/c.tld" %>
<c:forEach var="i" items="a, b, c, d, e, f, g"
   varStatus="status" begin="2" end ="4">
  i: ${i},
  current: ${status.current},
  index: ${status.index},
  count: ${status.count}
  <br>
</c:forEach>
```

Praktisch ist count für Aufzählungen der Art

1. abcdefghi
2. jklmnopq



<c:forTokens>

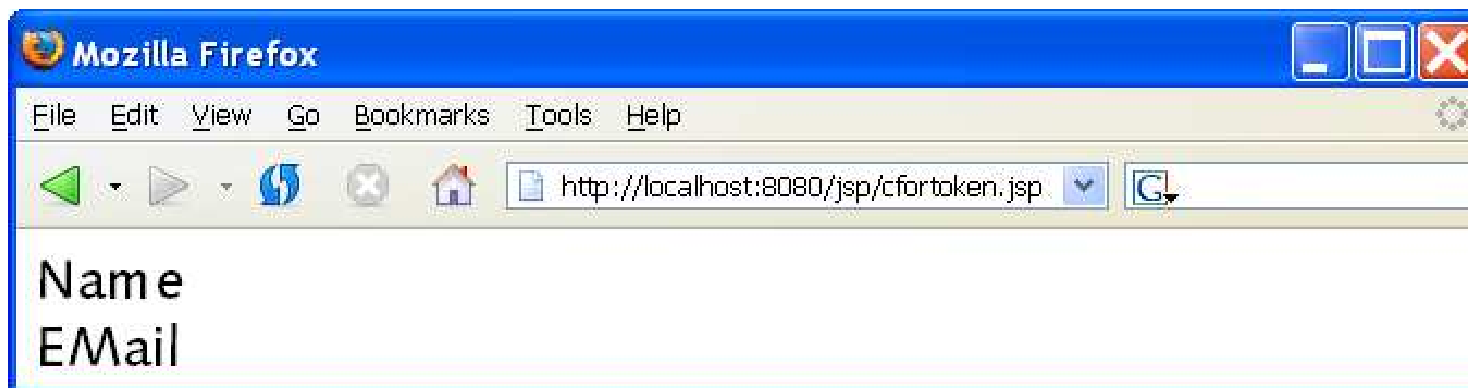
- Ist bei <c:forEach> das Attribut items ein String, so wird immer ein Tokenizer mit Delimiter Komma angewendet.
- Flexibler ist das Tag <c:forTokens>. Es nutzt zur Zerlegung einen java.util.StringTokenizer mit beliebigem Trenner.

```
<c:forTokens    [ var="varName" ]
                items="stringOfTokens"
                delims="delimiters"
                [ varStatus="varStatusName" ]
                [ begin="begin" end="end" ]
                [ step="step" ] >
  Körper
</c:forEach>
```

Beispiel für <c:forTokens>

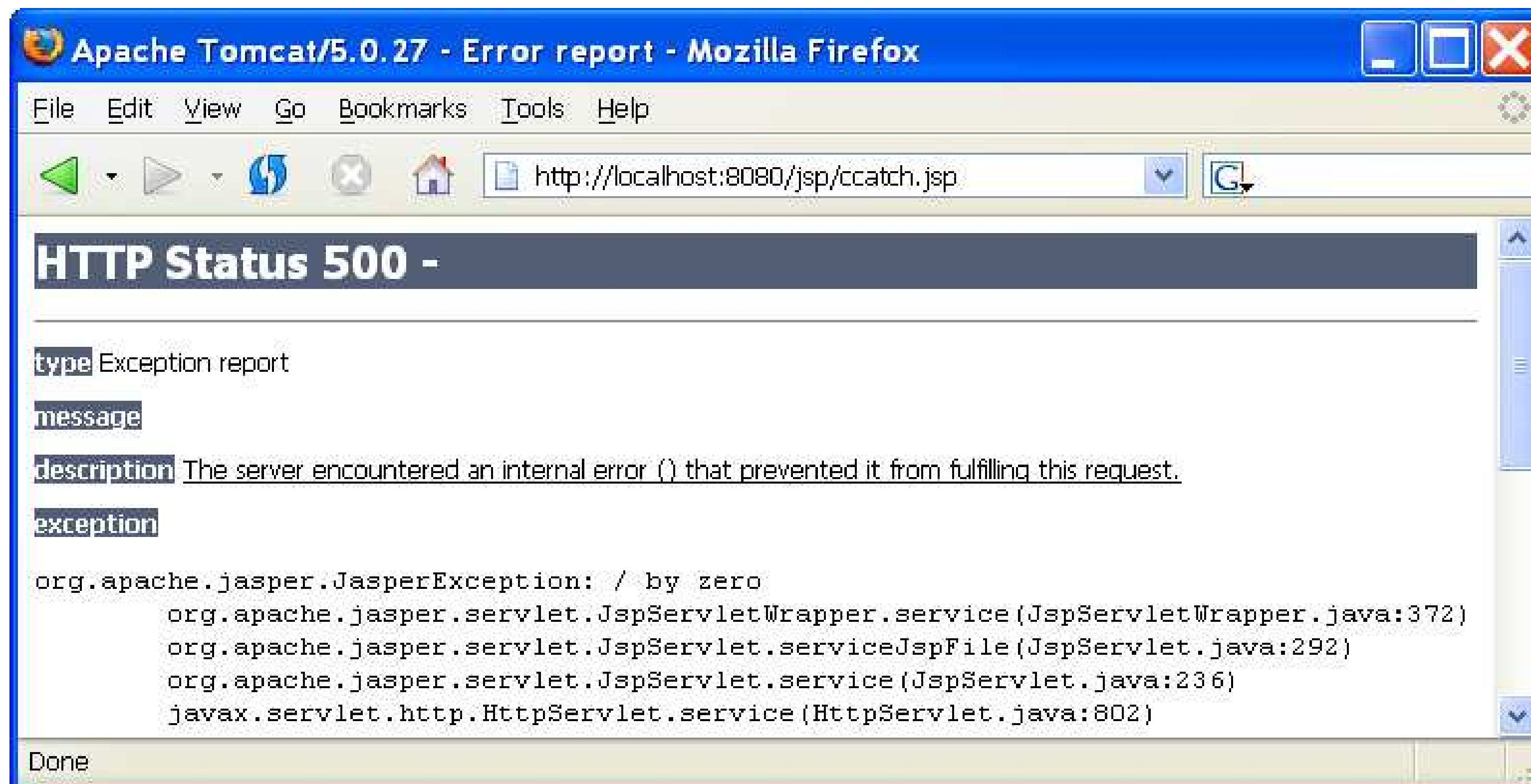
- Ein String soll Name und E-Mail beinhalten.
  - Die Werte sind mit : getrennt. : ist also der Delimiter.
  - Beispiel: **Name:EMail**

```
<c:forTokens var="elem" items="Name:EMail" delims=":">
  ${elem} <br/>
</c:forTokens>
```



Ausnahmen

- Was ist, wenn Anweisungen/Ausdrücke eine Exception produzieren, wie <jsp:expression>12/0</jsp:expression>?
  - Dann gibt es eine Exception und die Abarbeitung endet.



Ausnahmen mit <c:catch> auffangen

- Wenn Ausnahmen aufgefangen werden sollen, so kann man sie in einem <c:catch>-Block auffangen.

```
<c:catch [ var="varName" ] >
  Aktionen
</c:catch>
```

- Für unser Beispiel heißt das

```
<c:catch>
  <jsp:expression> 12 / 0 </jsp:expression>
</c:catch>
```

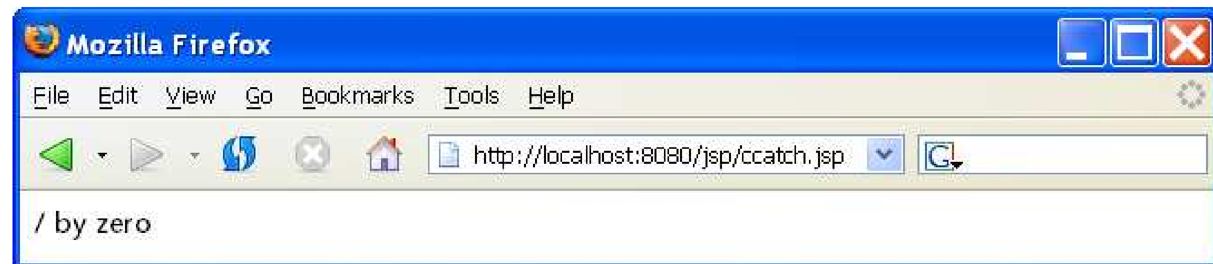
Exception-Objekt behalten

- Das Exception-Objekt ist an die optionale Variable var im page-Kontext gebunden.

```
<c:catch var="varName">
  Aktionen
</c:catch>
```

- Vom Exception-Objekt können wir den Grund erfragen.

```
<c:catch var="e">
  <jsp:expression> 12 / 0 </jsp:expression>
</c:catch>
${e.message}
```



<c:url>

- Mit dem Tag <c:url> lässt sich eine URL mit Parametern aufbauen. Nehmen wir eine JSP *curl.jsp* an:

  ```
  <c:url value="http://www.google.de/search?" var="url">
    <c:param name="q" value="${param.name}"/>
  </c:url>
  <a href='${url}'>Suche ${url}</a>.
  ```

- Rufen wir das Skript mit dem Parameter name=Lego auf

  http://localhost:8080/jsp/curl.jsp?name=Lego

  so wird die folgende URL generiert:

  http://www.google.de/search?q=Lego&

- Sonderzeichen werden URL-Encoded.

JSTL Formatierung und Funktionen

Formatier-Tags

Tags zum Formatieren

- Die Formatierungs-Bibliothek bindet man ein mit

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

- Tags zur Internationalisierung
  - `<fmt:message>`, `<fmt:param>`
  - `<fmt:bundle>`, `<fmt:setBundle>`
  - `<fmt:setLocale>`
  - `<fmt:requestEncoding>`
- Datum/Zeit Formatieren
  - `<fmt:formatNumber>`, `<fmt:parseNumber>`
  - `<fmt:formatDate>`, `<fmt:parseDate>`
  - `<fmt:timeZone>`, `<fmt:setTimeZone>`

<fmt:formatNumber>-Tag

- Das Tag <fmt:formatNumber> dient zur Darstellung von numerischen Werten unter Berücksichtigung der landestypischen Formatierungen.
- Das Tag schreibt
  - allgemeine Zahlen
  - Währungen
  - Prozentangeben

  und erinnert damit an die Klasse java.text.NumberFormat.
- Die allgemeinste Form des Tags ist

```
<fmt:formatNumber value="numericValue" />
```

- Beispiel:

```
<fmt:formatNumber value="-12.34444" />
```
-12.344

<fmt:formatNumber>

- Das <fmt:formatNumber>-Tag erlaubt viel „Feintuning“ mit vielen Attributen.

```
<fmt:formatNumber value=”numericValue”
       [type="{number|currency|percent}"]
       [pattern="customPattern"]
       [currencyCode="currencyCode"]
       [currencySymbol="currencySymbol"]
       [groupingUsed="{true|false}"]
       [maxIntegerDigits="maxIntegerDigits"]
       [minIntegerDigits="minIntegerDigits"]
       [maxFractionDigits="maxFractionDigits"]
       [minFractionDigits="minFractionDigits"]
       [var="varName"]
       [scope="{page|request|session|application}"]/>
```

Die wichtigsten Attribute

- Die darzustellende Zahl lässt sich auch einer Variablen zuweisen.
  - Dann ist das Attribut var zu setzen.
- Das Attribut type lässt die Werte „number“, „currency“ oder „percentage“ zu.
- Mit dem Attribut pattern kann man ganz genau die Ausgabe steuern.
  - Der Formatierungsstring ist genauso aufgebaut wie ein String aus java.text.DecimalFormat.

<fmt:formatDate>

- Zum Formatieren von Datumswerten wird das Attribut <fmt:formatDate> verwendet.
  - Damit lässt sich Datum oder eine Zeit darstellen oder einer Variablen zuweisen.
- Wie auch bei den numerischen Werten ist die Visualisierung abhängig vom Land und der Umgebung.
- Die einfachste Schreibweise ist

```
<fmt:formatDate value="numericValue" />
```

<fmt:formatDate> vollständig

- Die Attribute sind zwar einfacher im Vergleich zu <fmt:formatNumber>, aber immer noch umfangreich.

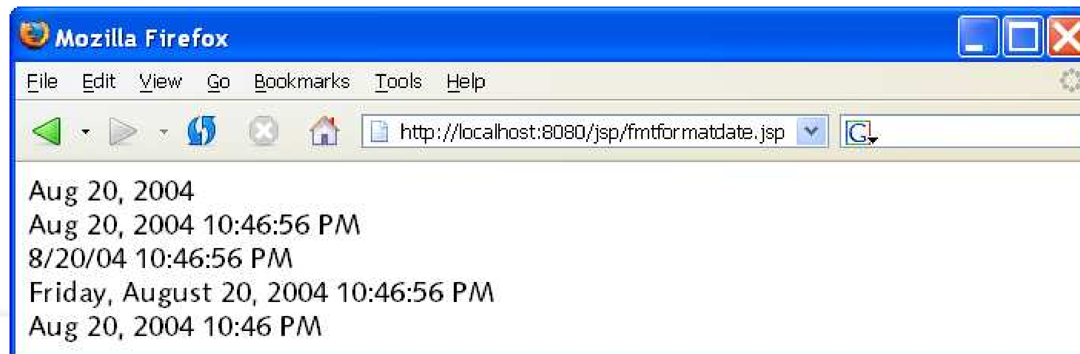
```
<fmt:formatDate value="numericValue"
               [ type="type"
                 pattern="pattern"
                 dateStyle="dateStyle"
                 timeStyle="timeStyle"
                 timeZone="timeZone"
                 var="varName"
                 scope="varScope"] />
```

Die Attribute von <fmt:formatDate>

- Das Format-Tag kann über type unterschiedliche Formatierungs-Arten vorschreiben.
  - „time“, „date“, „both“
- Eine Angabe kann unterschiedliche komplex, bzw. vollständig sein. Das bestimmt das Attribut dateStyle und timeStyle.
  - „default“, „short“, „medium“, „long“, „full“
- Die Werte werden mit einem java.text.DateFormat-Objekt in einer Zeichenkette konvertiert. Mit dem Attribut pattern lässt sich ein spezieller Ausgabestil noch weiter vorschreiben.
  - Er folgt den Konventionen vom java.text.SimpleDateFormat.

Beispiel für <fmt:formatDate>

```
<%@ taglib prefix="fmt" uri="/WEB-INF/tld/fmt.tld" %>
<jsp:useBean id="date" class="java.util.Date" />
<fmt:formatDate value="${date}" /><br>
<fmt:formatDate type="both" value="${date}" /><br>
<fmt:formatDate type="both" dateStyle="short"
                value="${date}" /><br>
<fmt:formatDate type="both" dateStyle="full"
                value="${date}" /><br>
<fmt:formatDate type="both" timeStyle="short"
                value="${date}" /><br>
```



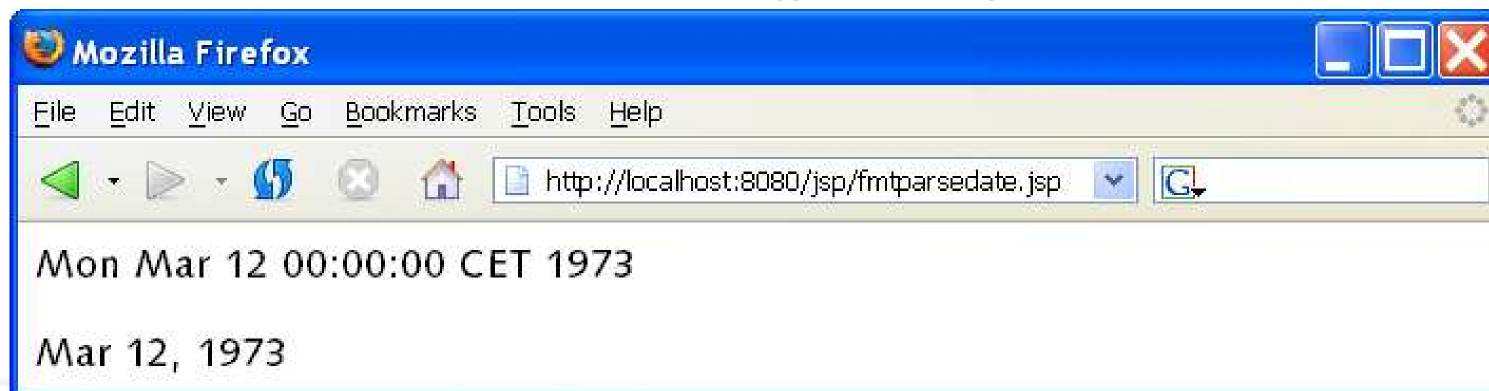
Parsen. Der Weg zurück

- Beim **Formatieren** von numerischen Werten oder Datumswerten entsteht ein String.
- Beim **Parsen** entsteht aus einer String-Repräsentation des Datenwertes der eigentlich Datentyp (Zahl, Datum).
- Zum Parsing bietet JSTL zwei Tags, die landesabhängig einen String auseinander nehmen:
  - `<fmt:parseNumber>`
  - `<fmt:parseDate>`
- Beim Parsen wird die Zeichenfolge in einen numerischen Wert konvertiert und entweder ausgegeben oder einer Variablen zugewiesen.

Beispiel für <fmt:parseDate>

- Ist das Datentyp String, so muss er vorher konvertiert werden:

```
<fmt:parseDate pattern="dd/MM/yyyy">
 12/3/1973
</fmt:parseDate> <p>
<fmt:parseDate value="12.03.1973"
               parseLocale="de" var="parsed"/>
<fmt:formatDate value="${parsed}" />
```



Function Tags

String-Funktionen der fn-TagLib

- Die TagLib Function bietet in erste Linie String-Funktionen an:
  - Verändern der Groß-/Kleinschreibung
  - Liefern von Substrings, Ersetzen von Zeichen
  - Abschneiden der Leerzeichen vorne und hinten
  - Testen auf einen Teilstring
  - Spliten und Zusammenführen eines Strings
  - Ausmaskieren von XML-Zeichen

Einbinden

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

```
${fn:toUpperCase(fn:substring("Hallo Willi", 0, 5))}
```

Die Format-Funktionen

- fn:contains(string, substring) → boolean
- fn:containsIgnoreCase(string, substring) → boolean
- fn:escapeXml(string) → String
- fn:indexOf(string, substring) → int
- fn:join(array, separator) → String
- fn:length(input) → integer
- fn:replace(inputString, beforeSubstring, afterSubstring) → String
- fn:split(string, delimiters) → String[]
- fn:startsWith(string, prefix) → boolean
- fn:endsWith(string, suffix) → boolean
- fn:substringAfter(string, substring) → String
- fn:substringBefore(string, substring) → String
- fn:toLowerCase(string) → String
- fn:toUpperCase(string) → String
- fn:trim(string) → String

Die Funktion length

- Die length-Funktion hat verglichen zu allen anderen Funktionen zwei Aufgaben:
  - Ermitteln der Stringlänge
  - Bei einer Datenstruktur Anzahl der Elemente.
- Beispiel für das Ermitteln der Stringlänge.

  ```
  ${fn:length("Hallo")}
  ```

- Bei einer Datenstruktur (alles, was als Attribut items bei <c:forEach> eingesetzt werden kann) so liefert length die Anzahl Elemente der Datenstruktur.

  ```
  ${fn:length(param)}
  ```

  liefert im Fall von fn_length.jsp?a=1&b=2 die Ausgabe 2.

Eigene Funktionen definieren

Entwickeln von Funktionen

- Mit der Expression Language können Funktionen aufgerufen werden, die man einfach vorher definieren kann.
- Die Funktionen müssen
  - öffentlich,
  - statisch und
  - in einer öffentlichen Klasse liegen.
- Dann muss für die Funktion ein **Function Descriptor** (eine XML-Datei) geschrieben sein.

- Ein Beispiel soll zeigen, wie man das Maximum von Zahlen bilden kann.

Statischen Funktion implementieren

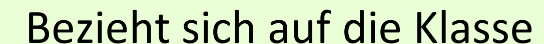
- Die öffentliche Klasse Max im Paket com.tutego.tags soll die statische public Funktion max() definieren.

```
package com.tutego.tags;
public class Max {
  public static int max( String x, String y ) {
    int a = 0, b = 0;
    try {
      a = Integer.parseInt(x); b = Integer.parseInt(y);
    } catch ( Exception e ) { }
    return Math.max( a, b );
  }
}
```

Function Descriptor

- Der **Function Descriptor** ist eine XML-Datei, die die statische Funktion beschreibt.
  - Wir nennen die Datei *jsp-max-taglib.tld*.

```
<?xml ...><taglib ...>
  <function>
    <description>Bildet Maximum von x und y</description>
    <name>max</name>
    <function-class>com.tutego.tags.Max</function-class>
    <function-signature>
   int max( java.lang.String, java.lang.String )
 </function-signature>
  </function>
</taglib>
```



Nutzendes Beispiel

- M

ERROR: syntaxerror
OFFENDING COMMAND: --nostringval--

STACK:

519
3724
5