Thomas Stark

MASTER CLASS

Java EE 5

» Einstieg für Anspruchsvolle

Lerntest, Beispiele,
Java EE 5, JBoss, Tools

DVD

ADDISON-WESLEY [in Kooperation mit] PEARSON Studium

4

Tag-Bibliotheken

Die vorangegangenen Kapitel haben erläutert, wie Sie durch die Verbindung von HTML und Javacode flexible, dynamische Webseiten erstellen, und mit JavaServer Pages, Servlets und JavaBeans haben Sie bereits drei wichtige Technologien des Java Enterprise Framework kennen gelernt. Dabei spielte Markup in Form von Tags eine große Rolle, doch wurden auch immer grundlegende Java-Kenntnisse benötigt, um die resultierenden Dokumente zu verstehen. Hinzukommt, dass eine Portierung einmal geschriebenen JSP- oder Servlet-Codes in ein anderes Projekt nur schwer möglich ist, da JSPs und Servlets in der Regel fest in einen Kontext integriert sind.

Tipp

Um die Beispiele in diesem Kapitel nachzuvollziehen, können Sie entweder eine neue Webanwendung konfigurieren oder das bereits bestehende Projekt aus Kapitel 2 weiterverwenden.

Dieses Kapitel wird Ihnen zeigen, wie Sie eigene Tags schreiben, um oft benötigte Funktionalität zu kapseln. Durch solche Tags werden die JSPs nicht nur von lästigem Java-Code befreit, der sie häufig aufbläht und unübersichtlich macht. Sie können Ihren Code darüber hinaus auch leichter wiederverwenden und ohne Mühe selbst über Projektgrenzen hinweg portieren. Außerdem existieren inzwischen viele frei verfügbare Tag-Bibliotheken, die einen riesigen Satz an vordefinierter Funktionalität bereitstellen.

Tipp

Das *Jakarta Apache Project* hostet ein eigenes Subprojekt, das sich ausschließlich mit Tag-Bibliotheken befasst, von Benchmark-Tests bis zu Mail-Funktionalität. Hier finden Sie nahezu alles: *http://jakarta.apache.org/taglibs/*.

Zu den bekanntesten gehören:

- Die JSP Standard Tag Library (JSTL)

  Ein grundlegendes Set an Tags zur Manipulation von Datenbanken, zum Lesen von XML-Dateien, Internationalisierung (i18n) etc.; *http://java.sun.com/products/jsp/taglibraries.html*.

- Die Struts Tag Library

  Dieser Tag-Bibliothek werden Sie im Struts-Kapitel natürlich wiederbegegnen. In ihr finden Sie nützliche Tags, mit denen Sie das Model-View-Controller-Prinzip (MVC) gut in JSPs umsetzen können; *http://jakarta.apache.org/struts/*.

- Die Regexp Tag-Library

  Bei dieser Tag-Bibliothek ist der Name Programm: Sie ermöglicht es, *reguläre Ausdrücke* (Regular Expressions) im Perl-Stil in Ihre JSPs zu integrieren. Mehr dazu finden Sie unter *http://jakarta.apache.org/taglibs/doc/regexp-doc/*.

4.1 Ein eigenes Tag

Die ersten Beispiele der vorangegangenen Kapitel veranschaulichen ihre jeweilige Funktionalität anhand der Ausgabe eines Datums und auch dieses Kapitel setzt mit dieser Tradition fort. Hier werden Sie die Ausgabe nun allerdings in einem eigenem Tag namens `<date/>` kapseln.

Bei der Erstellung eigener Tag-Bibliotheken (Taglibs) sind, ähnlich wie bei den Servlets, verschiedene Schritte notwendig, die Ihnen beim ersten Mal vielleicht aufwändig und umständlich erscheinen, später jedoch schnell zur Routine werden.

4.1.1 Erstellen der Java-Klasse

Als Erstes benötigen Sie eine Java-Klasse, die die gewünschte Funktionalität aufnimmt, den so genannten *Tag-Handler*. Java stellt Ihnen dabei im Paket `javax.servlet.jsp.tagext` verschiedene Interfaces bereit, die Sie verwenden können, um Tag-Handler mit unterschiedlichen Fähigkeiten zu implementieren. Außerdem stellt Ihnen das Package mit den beiden Klassen `TagSupport` und `BodyTagSupport`, analog zum `HttpServlet`, zwei Basisklassen zur Verfügung, die bereits alle Methoden mit Standardfunktionalität implementieren und von Ihnen überschrieben werden können.

Der Lebenszyklus eines Tag vom Typ TagSupport

Info

Im Folgenden werden wir das Tag `<tag/>` als symbolischen Platzhalter für alle von Ihnen entwickelten Tags verwenden.

Enthält eine aufgerufene JSP eines Ihrer Tags, so wird eine Instanz der zugehörigen Java-Klasse aufgerufen, um das gewünschte Verhalten bzw. Resultat zu erzeugen. Diese Klasse durchläuft dabei einen bestimmten Lebenszyklus – in Form von Methoden, die nacheinander gerufen werden:

- `doStartTag()`
  Diese Methode wird bei einem öffnenden Tag (`<tag>`) aufgerufen. Mit ihr haben Sie die Möglichkeit, Ihr Tag zu initialisieren und festzustellen, ob das Tag an der richtigen Stelle innerhalb des Dokuments verwendet wurde.

  Am Ende der Methode können Sie durch Rückgabe einer entsprechenden Konstante entscheiden, ob der Rumpf des Tag verarbeitet oder übersprungen werden soll. Ist Letzteres der Fall, endet die Bearbeitung des Tag an dieser Stelle und der Rumpf, inklusive des schließenden Tag (`</tag>`), wird schlicht ignoriert.

Tipp

Sie können das Überspringen des Rumpfs dabei auch als Vorsichtsmaßnahme verstehen: Sie verhindern damit, dass das Resultat eines Tag, das keinen Rumpf besitzen sollen – wie z.B. Ihr Datums-Tag – durch versehentlich im Rumpf eingefügten Code etwa unschön formatiert wird.

- `doInitBody()`
  Diese Methode wird nur gerufen, wenn das Tag auch tatsächlich einen Rumpf besitzt. Sie dient z.B. der Initialisierung von JSP-Variablen, die innerhalb des Tag-Rumpfs gültig sind (siehe Absatz 4.5).
- `doAfterBody()`
  Diese Methode wird immer unmittelbar *vor* dem schließenden Tag (`</tag>`) aufgerufen und gibt Ihnen die Möglichkeit, den Rumpf des Tag zu modifizieren. An dieser Stelle können Sie außerdem entscheiden, ob Sie die Bearbeitung des Tag beenden wollen und zu `doEndTag()` übergehen oder den (möglicherweise modifizierten) Inhalt des Tag noch einmal bearbeiten wollen. In diesem Fall beginnt die Bearbeitung des Tag und seines Inhalts wieder von vorn, unmittelbar nach dem öffnenden Tag (dort, wo Sie standen, *nachdem* wir `doStartTag()` beendet hatten – Punkt 2 in Abbildung 4.1). Anschließend wird `doAfterBody()` erneut gerufen.

  `doAfterBody()` wird nur ausgeführt, wenn das Tag tatsächlich einen Rumpf besitzt. Bei leeren Tags (`<tag />`) wird diese Methode übersprungen.

- `doEndTag()`

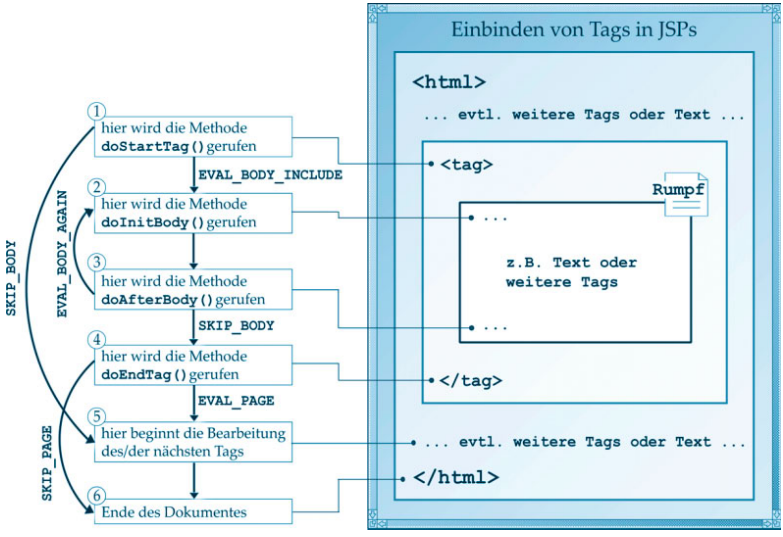
  Wenn Sie die Bearbeitung eines Tag nicht am Ende von `doStartTag()` abgebrochen haben, wird der Lebenszyklus des Tag durch die Methode `doEndTag()` abgeschlossen. Hier haben Sie die Möglichkeit, eventuell benötigte Ressourcen wieder freizugeben oder den Zustand des Tag in einer Datei zu speichern.

Tipp

Als Rumpf eines Tag werden *alle Zeichen* (Markup und Text) bezeichnet, die sich zwischen dem öffnenden Tag (<tag>) und dem schließenden End-Tag (</tag>) befinden. Tags der Art <tag /> besitzen also keinen Rumpf. Bei ihnen wird die Methode `doAfterBody()` nicht gerufen. Tags ohne Rumpf werden auch als leere Elemente bezeichnet.

Ob die Bearbeitung des aktuellen Tag abgebrochen oder fortgesetzt wird, teilen Sie dem Servlet-Container durch Rückgabe bestimmter Konstanten mit, deren Bedeutung Sie beim Durcharbeiten dieses Kapitels kennen lernen werden. Abbildung 4.1 verdeutlicht den Lebenszyklus eines `TagHandler`.

Abbildung 4.1
Lebenszyklus eines Tag-Handler



Der Tag-Handler

Da das Tag <date/> im Beispiel zunächst nur durch das aktuelle Datum ersetzt werden soll, genügt es, die Methode `doStartTag()` zu implementieren. Danach wird die Bearbeitung abgebrochen. Der Rumpf sowie ein eventuell vorhandenes End-Tag werden nun aus der JSP entfernt.

Listing 4.1
Ein einfacher Tag-Handler zum Einfügen des Datums (SimpleDateTag.java)

```
package de.akdabas.jee.tags;
import java.util.Date;
import java.io.IOException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.tagext.Tag;
import javax.servlet.jsp.tagext.TagSupport;

/**
 * Dieser Tag-Handler ersetzt alle Vorkommen des Tags '<date/>'
 * durch das aktuelle Datum des Servers
 */
public class SimpleDateTag extends TagSupport implements Tag {

   /**
    * Fügt das aktuelle Datum in die JSP ein. Diese Methode wird
    * gerufen, wenn bei der Verarbeitung der JSP ein öffnendes
    * Tag '<date>' gefunden wird.
    */
   public int doStartTag() {
      try {
         // Binden des JspWriters 'out' zur Ausgabe
         JspWriter out = pageContext.getOut();

         // Einfügen des Datums an der aktuellen Position
         out.print(new Date());
      } catch (IOException ioe) {
         ioe.printStackTrace();
      }

      // Verarbeitung des Rumpfs überspringen (engl. to skip), d.h.
      // eventuell vorhandener Rumpf bzw. End-Tags werden ignoriert
      return SKIP_BODY;
   }
}
```

Die Klasse `SimpleDateTag` leitet von der Basisklasse `TagSupport` ab und muss deshalb nur die Methoden überschreiben, deren Verhalten von der Standardimplementierung abweichen – in diesem Fall also `doStartTag()`. Dank der Superklasse verfügen Sie auch automatisch über die dort definierte Variable `pageContext` vom Typ `javax.servlet.jsp.PageContext`. Diese Variable ermöglicht es Ihnen z.B., den Request weiterzuleiten (*Forward*) bzw. externe Ressourcen einzubinden (*Include*) und auf wichtige Objekte wie das Request- oder Response-Objekt, die Session oder den `PrintWriter out` zuzugreifen.

Tipp

Die Basisklasse `TagSupport` implementiert bereits alle vom Interface `Tag` geforderten Methoden und eignet sich deshalb gut als Basisklasse für Ihre eigenen Tag-Handler.

Im oben stehenden Listing erzeugen Sie einfach ein neues `Date`-Objekt und fügen es über den `PrintWriter` in die JSP ein. Um den Servlet-Container danach anzuweisen, den eventuell vorhandenen Rumpf zu überspringen (engl. *to skip*) und die Bearbeitung zu beenden, geben Sie anschließend die vom Interface `Tag` deklarierte Konstante `SKIP_BODY` zurück.

Info

Die Klassen und Interfaces, die Sie zum Übersetzen der Beispiele in diesem Kapitel benötigen, finden Sie zum Beispiel in den Dateien *javax.servlet.jar* und *javax.servlet.jsp.jar*, im Verzeichnis *$JBOSS_HOME/server/default/lib*.

4.1.2 Konfiguration des Tag-Handler

Nachdem Sie das Tag erfolgreich übersetzt haben, ist dem Servlet-Container natürlich noch mitzuteilen, für welches Tag er die Methode `doStartTag()` aufrufen soll. Analog zu den Servlets, denen Sie über den Web Deployment Descriptor die zugehörigen URLs zugeordnet haben, existiert für Tag-Bibliotheken der so genannte *Tag Library Descriptor (TLD)*.

Ein TLD besteht aus einer meist auf das Kürzel *.tld* endenden XML-Datei, die im Wesentlichen die Aufgabe hat, Ihre Tags den jeweiligen Tag-Handlers zuzuordnen.

Listing 4.2 Ein einfacher Tag Library Descriptor (masterclass.tld)

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib PUBLIC
    "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
    "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">

<taglib>

   <!-- Unterstuetzter Taglib-Standard -->
   <tlib-version>1.0</tlib-version>

   <!-- Unterstuetzter JSP-Standard -->
   <jsp-version>1.2</jsp-version>

   <!-- Namensraum der definierten Tags dieses TLDs -->
   <short-name>masterclass</short-name>

   <!-- Mit dem Namensraum verknuepfte URL -->
   <uri>www.akdabas.de/javaee</uri>

   <!-- Verknuepfung des Tags 'date' mit dem Tag-Handler -->
   <tag>
      <name>date</name>
      <tag-class>de.akdabas.javaee.tags.SimpleDateTag</tag-class>
   </tag>

</taglib>
```

Einleitende Tags

Zuerst teilen Sie dem Servlet-Container mit, welche Taglib- bzw. JSP-Version die von Ihnen geschriebenen Tags unterstützen und welchem Namensraum die Tags angehören sollen. Die Angabe des Namensraums ist notwendig, weil innerhalb einer Webapplikation verschiedene Tag-Bibliotheken parallel zum Einsatz kommen können. Dabei könnte beispielsweise das Tag `<date>` in zwei Bibliotheken mit jeweils unterschiedlichem Verhalten definiert sein. Damit der Servlet-Container dennoch sicher feststellen kann, welches der beiden Tags gemeint ist, werden diese über die Namens-

räume zu so genannten Tag-Familien zusammengefasst. Jede Tag-Familie besitzt dadurch einen einheitlichen (Nach-)Namen, über den ein Tag sicher zugeordnet werden kann.

Info

Ein *Namensraum* (*Namespace*) ist ein Kontext, in dem ein konkreter Name nur ein einziges Mal existiert. Durch die Aufteilung von Ressourcen in einzelne Namensräume brauchen Sie sich keine Gedanken mehr darüber zu machen, ob der gewählte Name vielleicht mit einer Bezeichnung in einem anderen Teil der Applikation kollidiert.

Beispiele für Namensräume sind etwa die Packages in Java: Dort enthalten die Packages `java.sql` und `java.util` je eine Klasse `Date`, doch durch die Angabe des Namensraums kann der Compiler beide voneinander unterscheiden.

Info

In Markup-Dokumenten, wie HTML oder dem später besprochenen XML, erfolgt die Angabe des Namensraums durch ein Präfix, welches durch einen Doppelpunkt (:) vom Namen getrennt wird.

Achtung

Die Namensräume `java`, `javax`, `jsp`, `jspx`, `servlet`, `sun` und `sunw` sind bereits reserviert und können nicht für eigene verwendet werden. Auch leere Präfixe sind wegen der Verwechslungsgefahr mit HTML-Tags nicht zulässig.

Ein Beispiel für solche Namensräume haben Sie in Kapitel 3 schon einmal verwendet: Erinnern Sie sich, dass Sie statt `<forward>` und `<include>` stets `<jsp:include>` und `<jsp:forward>` schreiben mussten? Damit haben wir dem Servlet-Container mitgeteilt, dass diese Tags in der Tag-Bibliothek mit dem Namensraum `jsp` zu finden sind und nicht etwa Bestandteil von HTML sind.

Da sich auch Namensräume, aufgrund der wenigen Zeichen, überschneiden können, wird jeder Namensraum mit einem URL verknüpft. Denn ein URL ist schon per se eineindeutig, so dass es zu keinen Überschneidungen kommen kann. Die unter `<uri>` angegebene Ressource kann frei gewählt sein und muss nicht wirklich existieren. Es ist jedoch sinnvoll, unter dieser Adresse eine kurze Beschreibung des Namensraums, wie zum Beispiel die entsprechende Document Type Definition (DTD), zu hinterlegen.

Info

Prinzipiell muss es sich beim Wert des Elementes `<uri>` in Listing 4.2 nicht um eine URL, ja noch nicht einmal um einen URI handeln, da diese Beziehung zu keiner Zeit überprüft wird. Die Bezeichnung muss lediglich eindeutig sein.

Das Mapping

Auf diese stets gleich bleibende Einleitung zu Beginn eines TLD folgt das eigentliche *Mapping* von Tags auf Java-Klassen. Wichtig dafür sind ein konkreter Name für das Tag sowie die Angabe der zugehörigen Tag-Handler-Klasse.

Listing 4.3
Mapping eines Tag

```
...
   <!-- Verknuepfung des Tags 'date' mit dem Tag-Handler -->
   <tag>
      <name>date</name>
      <tag-class>de.akdabas.javaee.tags.SimpleDateTag</tag-class>
   </tag>
...
```

Dieses Beispiel definiert, dass der Tag-Handler `SimpleDateTag` Tags der Form `<date>` behandeln soll.

4.1.3 Einbinden der Tag-Bibliothek

Es gibt zwei Möglichkeiten, den TLD und damit auch die Tag-Bibliothek in Ihre Webanwendungen zu integrieren, damit die darin enthaltenen Tags verwendet werden können: entweder über einen öffentlichen URL oder über einen Eintrag in den Web Deployment Descriptor (*web.xml*).

Einbinden über einen öffentlichen URL

Um den Tag Library Descriptor unter einem öffentlichen URL verfügbar zu machen, können Sie beispielsweise ein Verzeichnis */tlds* parallel zu ihren Verzeichnissen für die JSPs oder HTML-Seiten erzeugen und ihn darin ablegen.

Info

Öffentliche TLDs werden oberhalb des vom Servlet-Container gesperrten Verzeichnisses *WEB-INF* abgelegt und können auch extern abgerufen werden.

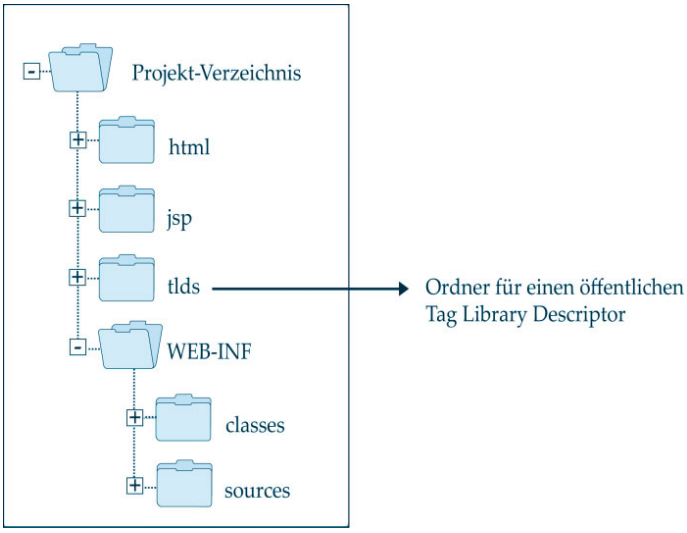


Abbildung 4.2
Veröffentlichen des Tag Library Descriptor

Damit ist Ihr TLD im gesamten Netz unter dem URL *http://IhrServer.Ihre-Domain/kap04/tlds/masterclass.tld* verfügbar. Zwar werden alle Tags bereits vor der Übertragung an den Client durch die Ausgabe des jeweils zugeordneten Tag ersetzt, so dass der Browser nicht bemerkt, dass Ersetzungen vorgenommen wurden. Allerdings kann nun jedermann nachvollziehen, welches Tag mit welcher Klasse verknüpft wird, und dies bietet einen Angriffspunkt für potenziellen Missbrauch.

Einbinden über den Web Deployment Descriptor

Um den TLD vor allzu neugierigen Blicken zu verbergen und Ihre Applikation damit prinzipiell sicherer zu machen, können Sie den Ordner *tlds* beispielsweise in den Ordner *WEB-INF* verschieben:

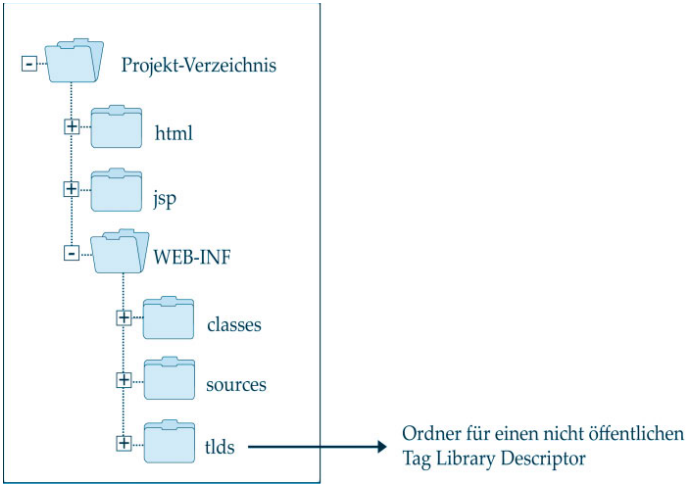


Abbildung 4.3
Indirektes Einbinden des Tag Library Descriptor

Um die im TLD beschriebenen Tags später dennoch verwenden zu können, fügen Sie folgende Zeilen am Ende Ihres Web Deployment Descriptor (*web.xml*) ein und machen den TLD damit nur innerhalb der Webanwendung und unter einer symbolischen Adresse verfügbar.

Listing 4.4 Binden des TLD im Web Deployment Descriptor (web.xml)

```
<web-app>
...
  <!-- Definition von Servlet-Mappings (Vlg. Kap. 3) -->
  <servlet-mapping />

  <!-- Einbinden des Tag Library Descriptors unter einer
       symbolischen URL -->
  <taglib>
   <taglib-uri>/masterclass-tags</taglib-uri>
   <taglib-location>/WEB-INF/tlds/masterclass.tld</taglib-location>
  </taglib>
</web-app>
```

Achtung

Die Servlet-Spezifikation beschreibt genau, in welcher Reihenfolge die verschiedenen Elemente des Web Deployment Descriptor auftauchen dürfen. Während einige Servlet-Container dies nicht so genau nehmen, verweigern andere dagegen bei Missachtung vollkommen den Dienst. Achten Sie also darauf, die Einträge für `<taglib>` immer *nach* dem `<servlet-mapping>` zu platzieren.

Jetzt können Sie den Tag Library Descriptor innerhalb Ihrer JSPs unter dem symbolischen URL */masterclass-tags* erreichen und die darin beschriebenen Tags verwenden.

4.1.4 Verwenden des Tag in einer JSP

Erinnern Sie sich noch an die JSP-Direktiven aus Kapitel 2? Dort haben Sie bereits die Page-Direktive und die Include-Direktive kennen gelernt. Hier folgt nun die dritte Direktive `taglib`. Sie ermöglicht es Ihnen, eigene Tags in Ihren JSPs zu verwenden:

Listing 4.5 Einbinden des Tag Library Descriptor in eine JSP

```
<!-- Einsatz der 'taglib'-Direktive in JSPs -->
<%@ taglib uri="/masterclass-tags" prefix="masterclass" %>
```

Durch diese Zeile weisen Sie den Servlet-Container an, für alle Tags des Namensraums `masterclass` im TLD unter dem URL */masterclass-tags* nach einer entsprechenden Tag-Handler-Klasse zu suchen und deren Servicemethode auszurufen. Dabei kann es sich sowohl um einen öffentlichen als auch um einen symbolischen URL handeln.

Jetzt können Sie das Tag `<date/>` beliebig innerhalb der JSP verwenden. Fast so, als wäre es ein Bestandteil der HTML-Spezifikation.

```
<%@ taglib uri="/masterclass-tags" prefix="masterclass" %>

<html>
   <body style="text-align:center; color:green;">
      Eigene Tag-Bibliotheken verwenden. <br/>
      Es ist jetzt genau: <masterclass:date />
   </body>
</html>
```

Listing 4.6
Verwenden des <date>-Tag (simpleDate.jsp)

Nun ist lediglich noch ein Neustart des Webservers notwendig, damit dieser die neu hinzugekommenen Einträge der Datei *web.xml* auslesen und verarbeiten kann. Und schon sehen Sie Ihr erstes eigenes Tag in Aktion.

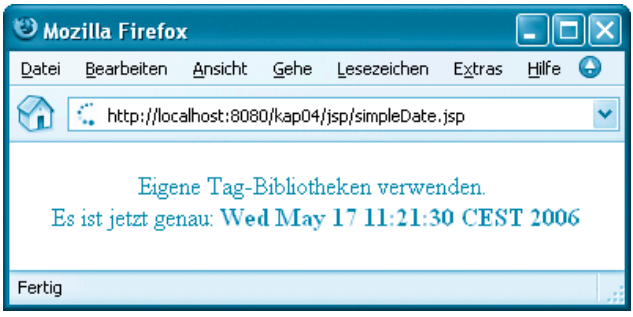


Abbildung 4.4
Das erste eigene Tag

4.2 Verwendung von Attributen

Beim Datenbank-Servlet des vorangegangenen Kapitels haben Sie gesehen, wie Sie Ihre Klassen allgemein entwickeln und anschließend mit Parametern entsprechend den jeweiligen Anforderungen anpassen. Auf ähnliche Weise können Sie nun auch Ihre Tags mit Attributen versehen, um diese flexibel und erweiterbar zu machen. Zur Veranschaulichung soll das obige Beispiel nun um ein Attribut `format` erweitert werden, um die Ausgabe des jeweiligen Datums anzupassen.

4.2.1 Der erweiterte Tag-Handler

Die Übertragung von Attributen eines Tag auf den Tag-Handler macht – analog zu den JavaBeans – die Deklaration passender Setter-Methoden erforderlich, über die der Servlet-Container den jeweiligen Wert des Attributs setzt.

```
package de.akdabas.javaee.tags;
import java.util.Date;
import java.io.IOException;
import java.text.SimpleDateFormat;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.tagext.Tag;
import javax.servlet.jsp.tagext.TagSupport;

/**
 * Dieser TagHandler fügt für das Tag '<date format="???"/>' das
 * aktuelle Datum ein und verwendet ein optionales Ausgabeformat.
 */
```

Listing 4.7
Der erweiterte Tag-Handler (FormattedDateTag.java)

Listing 4.7 (Forts.)
Der erweiterte Tag-Handler (FormattedDateTag.java)

```
public class FormattedDateTag extends TagSupport implements Tag {

   /** Speichert den Wert des Tag-Attributes 'format' */
   private String formatPattern;

   /** Setzt den Wert des Tag-Attributes 'format' */
   public void setFormat(String value) {
      formatPattern = value;
   }

   /** Fügt das formatierte Datum in die JSP ein */
   public int doStartTag() {

      try {
         // Binden des JspWriters 'out'
         JspWriter out = pageContext.getOut();

         // Wenn das Attribut 'format' nicht angegeben wurde, wird
         // das Datum unformatiert eingefügt.
         if (formatPattern == null) {
            out.print (new Date());
         } else {
            SimpleDateFormat dateFormat =
               new SimpleDateFormat(formatPattern);
            out.print(dateFormat.format(new Date()));
         }
      } catch (IOException ioe) {
         ioe.printStackTrace();
      }

      // Verarbeitung des Rumpfes überspringen (engl. to skip)
      return SKIP_BODY;
   }

   /**
    * Diese Methode setzt den Zustand des TagHandlers zwecks
    * Wiederverwendung (durch den Server) zurück.
    */
   public void release() {
      formatPattern = null;
   }
}
```

Achtung

Viele Servlet-Container verwalten ihre Tags über einen Pool! Das heißt, dass beim Start der Webanwendung oder beim ersten Aufruf eines Tags der Tag-Bibliothek eine Reihe von Instanzen des Tag-Handlers erzeugt werden, die bei nachfolgenden Aufrufen wiederverwendet werden. Aus diesem Grund sollten Tag-Handler, die einen Zustand besitzen, über die Methode `release()` in den Ausgangszustand zurückgesetzt werden. Die Methode `release()` wird vom Servlet-Container nach Beendigung des Tag-Lebenszyklus aufgerufen.

4.2.2 Registrieren des Tag-Handler im Tag Library Descriptor

Anschließend fügen Sie den folgenden Eintrag dem Tag Library Descriptor (*masterclass.tld*) einfach hinzu.

Listing 4.8
TLD-Eintrag für das <formatted-date>-Tag (masterclass.tld)

```
...
   <!-- Definition des 'formatted-date'-Tags und des zugehoerigen,
        optionalen Attributs 'format' -->
   <tag>
      <name>formatted-date</name>
     <tag-class>de.akdabas.javaee.tags.FormattedDateTag</tag-class>

      <!-- Definition des optionalen Attibutes 'format' -->
      <attribute>
         <!-- Name des Attributs in der JSP und im Tag-Handler -->
         <name>format</name>

         <!-- Muss dieses Attribut angegeben werden;
              optional (default: false) -->
         <required>false</required>

         <!-- Kann der Wert dieses Attributs dynamisch gesetzt
              werden; optional (default: true) -->
         <rtexprvalue>true</rtexprvalue>

         <!-- Typ des Attributs im Tag-Handler;
              optional (default: java.lang.String) -->
         <type>java.lang.String</type>
      </attribute>
   </tag>
...
```

Info

Zwingend erforderlich ist bei der Deklaration eines Attributs nur der Name (<name>).

Die verschiedenen Konfigurationsparameter haben dabei folgende Funktionen:

- `name`

  Die Angabe des Attributnamens ist als einziges Pflicht. Für alle anderen Attribute sind in der Java-EE-Spezifikation auch Standardwerte hinterlegt. Es versteht sich dabei fast von selbst, dass der hier angegebene Name für dieses Tag eindeutig sein muss.

- `required` (optional; default: `false`)

  Das optionale Flag `required` (dt. *benötigt, erforderlich*) gibt an, ob das jeweilige Attribut für die Arbeit des Tag essenziell ist. Wenn Sie den Wert auf `true` setzen und das Tag der JSP kein solches Attribut enthält, generiert der Servlet-Container eine entsprechende Fehlermeldung und bricht die Verarbeitung der Seite ab.

- `rtexpression` (optional; default: `false`)

  Mit dem Attribut `rtexpression` (Abkürzung für Runtime Expression) geben Sie an, ob der Wert des Attributs bei der Verwendung des Tag in der JSP dynamisch gesetzt werden kann, also beispielsweise auch über eine JSP-Expression.

Info

Die Reihenfolge, in der Sie die Attribute eines Tag-Handler definieren, ist egal und es spielt auch keine Rolle, in welcher Reihenfolge Sie die Attribute in der JSP angeben, solange alle benötigten Attribute vorhanden sind.

- `type` (optional; default: `java.lang.String`)

  Der Servlet-Container muss natürlich wissen, welchen Parametertyp die korrespondierende Methode im Tag-Handler erwartet. Standardmäßig werden hierfür Strings verwendet.

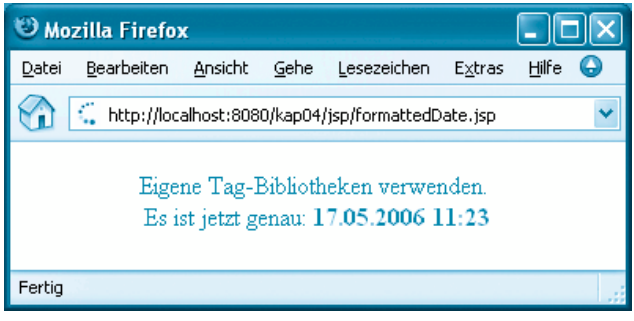
4.2.3 Verwendung innerhalb der JSP

Nun brauchen Sie dem neuen Tag `<formatted-date>` nur noch das Attribut `format` mitzugeben und schon wird das aktuelle Datum wie gewünscht ausgegeben:

Listing 4.9 Formatierte Ausgabe des aktuellen Datums via Tag (formattedDate.jsp)

```
<%@ taglib uri="/masterclass-tags" prefix="masterclass" %>
<html>
   <body style="text-align:center; color:green; ">
      Eigene Tag-Bibliotheken verwenden. <br/>
      Es ist jetzt genau:
         <masterclass:formatted-date format="dd.MM.yyyy HH:mm"/>
   </body>
</html>
```

Abbildung 4.5 Ausgabe des formatierten Datums



Da Sie in Listing 4.8 die Runtime-Expressions für den Attributwert `format` aktiviert haben, können Sie das tatsächliche Datumsformat mit einer kleinen Modifikation beispielsweise auch über einen Parameter steuern.

Listing 4.10
Verwendung von Runtime-Expressions für Tag-Attribute

```
<%@ taglib uri="/masterclass-tags" prefix="masterclass" %>
<%
   String format = request.getParameter("format");
   if (format == null) {
      format = "dd.MM.yyyy HH:mm";
   }
%>
<html>
   <body style="text-align:center; color:green; ">
      Eigene Tag-Bibliotheken verwenden. <br/>
      Es ist jetzt genau:
         <masterclass:formatted-date format="<%= format %>"/>
   </body>
</html>
```

4.3 Vordefinierte Variablen in Tag-Handlers

Häufig benötigen Sie innerhalb der Tag-Handler die gleichen Variablen wie in Ihren JSPs und Servlets. Während diese Variablen in JSPs bereits vordefiniert waren, mussten Sie sie bei Servlets über das `ServletRequest`- bzw. `ServletResponse`-Objekt selbst binden.

Tipp

Die in der Klasse `TagSupport` definierte Variable `pageContext` gestattet Ihnen den Zugriff auf das Request- und das Response-Objekt.

Wenn Ihre Tag-Handler Klassen von `TagSupport` oder `BodyTagSupport` ableiten, können Sie auf die dort definierte Variable `pageContext` zurückgreifen, die Ihnen ebenfalls Zugriff auf das `ServletRequest`- und `ServletResponse`-Objekt und damit auch auf alle anderen Variablen bietet. Auf diese Weise können Sie beispielsweise Werte von Request-Parametern auslesen oder Objekte in der Benutzer-Session ablegen.

4.3.1 Ein Tag-Handler für HTTPS-Verbindungen

Um die Arbeit mit dem `pageContext`-Objekt zu demonstrieren, soll nun ein Security-Tag entwickelt werden, welches seinen Body nur dann in die resultierende Seite einfügt, wenn diese Seite über eine verschlüsselte Verbindung HTTPS übertragen wird. Anderenfalls wird der Inhalt einfach übersprungen und so sicher aus der resultierenden Seite entfernt.

Listing 4.11
FilterSecureTag.java

```
package de.akdabas.javaee.tags;
import javax.servlet.ServletRequest;
import javax.servlet.jsp.tagext.Tag;
import javax.servlet.jsp.tagext.TagSupport;

/**
 * Dieser Tag-Handler filtert den Inhalt des Bodys aus, wenn dieser
 * nicht per HTTPs (HTTP Secure) übertragen wird.
 */
public class FilterSecureTag extends TagSupport implements Tag {
```

Listing 4.11 (Forts.)
FilterSecureTag.java

```
    /** Stellt sicher, dass der Body verschlüsselt übertragen wird */
    public int doStartTag() {

        // Binden des ServletRequest-Objekts aus dem Page-Context
        ServletRequest request = pageContext.getRequest();

        // Überprüfen, ob die Seite verschlüsselt übertragen wird
        if (request.isSecure()) {
            // Body des Tags '<is-secure>' einfügen (Include)
            return EVAL_BODY_INCLUDE;
        } else {
            // Body des Tags '<is-secure>' überspringen (Skip)
            return SKIP_BODY;
        }
    }
}
```

Über die Methode `isSecure()` des `ServletRequest`-Objekts können Sie ermitteln, ob zwischen Server und Client eine Verschlüsselung der Daten stattfindet. Ist die Verbindung sicher, gibt die Methode `doStartTag()` die Konstante `EVAL_BODY_INCLDUE` zurück, anderenfalls die schon bekannte Konstante `SKIP_BODY`.

Tipp

HTTPS steht für *Hypertext Transfer Protocol Secure* und ermöglicht eine auf Zertifikaten basierende gesicherte HTTP-Verbindung zwischen zwei Rechnern. Die größte Schwachstelle dieses Protokolls liegt dabei in der Verwendung nicht signierter Zertifikate, die einen so genannten »man in the middle«-Angriff ermöglichen. Mehr zu HTTPS findet sich im RFC 2818 z.B. unter *http://www.rfc-editor.org*.

Wie Sie an diesem Beispiel sehen, sind auch Tag-Handler nicht auf HTTP-Requests beschränkt, sondern können prinzipiell für verschiedene Request-Typen eingesetzt werden. Dafür müssen Sie allerdings auch in Kauf nehmen, dass die vom `pageContext` zurückgegebenen Objekte »nur« dem Typ der jeweiligen Basisklassen `ServletRequest` bzw. `ServletResponse` entsprechen und bei Bedarf eigenständig in `HttpServletRequest` bzw. `HttpServletResponse` gecastet werden müssen.

Info

Unter einem *Cast*, oder genauer einem *Typecast* (dt. Typumwandlung), versteht man die Zuweisung eines Objekts zu einem bestimmten Typ. Wie Sie wissen, kann ein Objekt der Klasse `HTTPServletRequest` auch an eine Variable des Typs `ServletRequest` gebunden werden, da jeder `HTTPServletRequest` auch ein `ServletRequest` (upcast) ist. Dabei ändern Sie nicht den Typ des Objekts selbst, sondern nur die Sichtweise darauf. Wenn Sie sicher wissen, dass ein bestimmtes Objekt eine bestimmte Unterklasse (downcast) oder ein spezifisches Interface (sidecast) implementiert, können Sie es natürlich auch in diese Richtung casten.

Einbinden des Rumpfs oder Überspringen

In Listing 4.11 entscheiden die Konstanten `EVAL_BODY_INCLUDE` und `SKIP_BODY` darüber, ob der Rumpf des Tag vom Servlet-Container in die resultierende Seite eingefügt oder übersprungen wird. Durch sie teilen wir dem Servlet-Container das gewünschte Verhalten mit. Gültige Konstanten für die Methode `doStartTag()` sind dabei:

- `SKIP_BODY`
  Überspringt den Rumpf des Tag und setzt die Bearbeitung der Seite nach dem Tag fort. Die Methode `doEndTag()` wird nicht gerufen.
- `EVAL_BODY_INCLUDE`
  Setzt die Bearbeitung des Tag mit dem Rumpf fort. Eventuell dort enthaltene Tags werden gegebenenfalls ausgeführt. Abschließend wird die Methode `doEndTag()` aufgerufen.

4.3.2 Die Konfiguration im Tag Library Descriptor

In der Konfiguration unterscheidet sich das `FilterSecureTag` nicht von den vorhergehenden. Wie Sie außerdem sehen, können Sie innerhalb der Tag-Namen auch einen Bindestrich verwenden:

Listing 4.12
Konfiguration des Filter-Secure-Tag (masterclass.tld)

```
...
   <!-- Definition des 'is-secure'-Tag -->
   <tag>
      <name>is-secure</name>
      <tag-class>de.akdabas.javaee.tags.FilterSecureTag</tag-class>
   </tag>
...
```

4.3.3 Eine Test-JSP

Um das Verhalten des Tag-Handler zu testen, können Sie die folgende JSP einmal über HTTP und einmal über HTTPS aufrufen. Dazu muss Tomcat allerdings für eine verschlüsselte Übertragung konfiguriert sein.

Listing 4.13
Anwendung des <is-secure>-Tag (secure.jsp)

```
<%@ taglib uri="/masterclass-tags" prefix="masterclass" %>
<html>
   <body style="color:green;">
      Diese Seite wird nur über sichere Verbindungen
      vollständig dargestellt. <br />

      <masterclass:is-secure>
         <!-- Dies ist der Body des Tags 'is-secure' -->
         <p> Dies ist eine sichere Verbindung. </p>
      </masterclass:is-secure>
   </body>
</html>
```

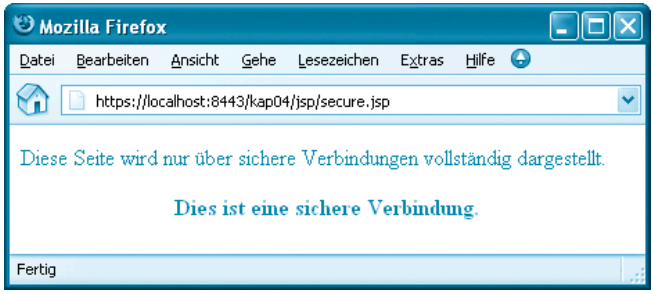


Abbildung 4.6
Aufruf über eine verschlüsselte Verbindung (HTTPS)

Eine Einführung, wie Sie eine HTTPs-Verbindung für den Apache Tomcat konfigurieren und das dafür benötigte SSL-Zertifikat erstellen, finden Sie unter *http://tomcat.apache.org/tomcat-5.5-doc/ssl-howto.html*.

4.4 Den Rumpf eines Tag manipulieren

Bisher haben Sie Ihre Tag-Handler lediglich dazu verwendet, Inhalte in Ihre JSPs einzufügen und den Rumpf des Tag bei Bedarf zu überspringen. Nun werden Sie lernen, wie Sie den Inhalt des Rumpfs auch differenziert bearbeiten. Dazu erweitert dieser Tag-Handler nicht mehr nur die Klasse `TagSupport`, sondern auch die Klasse `BodyTagSupport`, die eine abgeleitete Klasse von `TagSupport` darstellt.

4.4.1 Die Klasse BodyTagSupport

Die Klasse `BodyTagSupport` erweitert die Klasse `TagSupport` und implementiert damit das Interface `Tag`. Außerdem implementiert diese Basisklasse noch das Interface `BodyTag` und stellt Ihnen so über die Methode `getBodyContent()` ein Objekt vom Typ `javax.servlet.jsp.tagext.BodyContent` zur Verfügung, über dessen Methoden Sie den Rumpf des Tag gezielt manipulieren können:

- `getString()` – Diese Methode liefert Ihnen den Inhalt des Rumpfs (Text und Markup) als `String`.
- `clearBody()` – Diese Methode löscht den vorhandenen Inhalt aus der resultierenden Seite.
- `writeOut(Writer out)` – Schreibt den aktuellen Rumpfinhalt in den übergebenen `Writer` (z.B. den `JspWriter` der JSP).

4.4.2 Ein Iteration-Tag

Um die Arbeit mit der Klasse `BodyTagSupport` kennen zu lernen, soll nun ein Iteration-Tag entwickelt werden, welches vergleichbar einer `for`-Schleife den Inhalt seines Rumpfs für eine vorher festgelegte Anzahl wiederholt. Die verwendete Methode `doAfterBody()` stammt übrigens bezeichnenderweise vom Interface `javax.servlet.jsp.tagext.IterationTag`, welches ebenfalls von `BodyTagSupport` implementiert wird.

Listing 4.14
Ein Tag-Handler zum Iterieren des Rumpfs (IterationTag.java)

```
package de.akdabas.javaee.tags;
import java.io.IOException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.tagext.BodyTag;
import javax.servlet.jsp.tagext.BodyContent;
import javax.servlet.jsp.tagext.BodyTagSupport;

/** Wiederholt den Inhalt des Rumpfes 'n'-mal */
public class IterationTag extends BodyTagSupport implements BodyTag{

  /**
   * Anzahl der Wiederholungen, gesetzt über das Attribut 'times'
   */
  private int iterations = 0;

  /** Setzt die Anzahl der gewünschten Wiederholungen */
  public void setTimes(Integer value) {
    iterations = value;
  }

  /** Wiederholt den Inhalt des Tags 'iterate' n-mal */
  public int doAfterBody() {

    // Binden der Variable body
    BodyContent body = getBodyContent();

    // Test: Hat das Tag überhaupt einen Rumpf ?!
    if (body != null) {

      // Extrahieren des JspWriters (entspricht 'out' in JSPs)
      JspWriter out = body.getEnclosingWriter();

      try {
        // Fügt den Rumpf 'n'-mal in die JSP ein
        for (int i = 0; i < iterations; i++) {
          out.print(body.getString());
        }
      } catch (IOException ioe) {
        ioe.printStackTrace();
      }
    }

    // Am Ende wird der Rumpf übersprungen, da er bereits
    // 'n'-mal eingefügt wurde
    return SKIP_BODY;
  }

  /**
   * Diese Methode setzt den Zustand des TagHandlers zwecks
   * Wiederverwendung zurück
   */
  public void release() {
    iterations = 0;
  }
}
```

Achtung

Bevor Sie mit dem `BodyContent`-Objekt arbeiten, sollten Sie immer sicherstellen, dass dessen Wert *nicht* `null` ist.

Hinter dem vom `BodyContent`-Objekt über die Methode `getEnclosingWriter()` zurückgegebenen `PrintWriter` verbirgt sich nichts anderes, als die

vordefinierte Variable `out` Ihrer JSPs, über die Sie in diesem Beispiel den Rumpf des Tag ausgeben. Dazu extrahieren Sie dessen Inhalt (Text und Markup) über die Methode `getString()` als Zeichenkette.

Da Sie den Rumpf bereits in der `for`-Schleife n-Mal ausgeben, überspringen Sie die weitere Bearbeitung des Tag am Ende der Methode mit der bereits vertrauten Konstanten `SKIP_BODY`.

4.4.3 Konfiguration im Tag Library Descriptor

Es ist egal, ob Sie Ihren Tag-Handler nun von der Klasse `TagSupport` oder der Klasse `BodyTagSupport` ableiten oder auch alle vom Interface geforderten Methoden »zu Fuß« implementieren. Die Konfiguration im TLD bleibt stets die gleiche und so lautet der erforderliche Eintrag:

Listing 4.15
Eintrag im Tag Library Descriptor (masterclass.tld)

```
...
   <!-- Definition des Tags 'iterate' vom Typ BodyTag -->
   <tag>
      <name>iterate</name>
      <tag-class>de.akdabas.javaee.tags.IterationTag</tag-class>
      <attribute>
         <name>times</name>
         <required>true</required>
         <type>java.lang.Integer</type>
      </attribute>
   </tag>
...
```

Das Tag `<iterate>` macht ohne die Angabe, wie oft sein Rumpf wiederholt werden soll, wenig Sinn, weswegen Sie dieses Attribut diesmal als `required` deklarieren. Und da Sie in diesem Fall eine Zahl statt einer Zeichenkette erwarten, geben Sie auch den entsprechenden Typ an.

Tipp

Eine Beschreibung der Attribute finden Sie unter Abschnitt 4.2.2.

4.4.4 Test mit einer JSP

Jetzt können Sie das `<iterate>`-Tag dazu verwenden, sowohl Text als auch Markup beliebig oft zu vervielfältigen:

Listing 4.16
<iterate>-Tag in Aktion (iterate.jsp)

```
<%@ taglib uri="/masterclass-tags" prefix="masterclass" %>
<html>
   <body style="text-align:center; color:green;">
      <masterclass:iterate times="3">
         <p> Carpe Diem: Genieße den Tag ! </p>
      </masterclass:iterate>
   </body>
</html>
```

Abbildung 4.7
Das Resultat des Tag <iterate>

4.5 Definition von Script-Variablen

Achtung

Die hier vorgestellte Technik funktioniert erst ab der JSP-Spezifikation 1.2.

Sie können mit Ihren Tag-Handlers nicht nur auf die im jeweiligen Tag gesetzten Attributwerte zurückgreifen und diese so z.B. mit lokalen Variablen der JSP initialisieren. Der Tag-Handler ermöglicht auch die Manipulation von in JSPs erzeugten Variablen, so dass Sie direkt auf JSP-Scriptlets und -Ausdrücke einwirken können.

4.5.1 Der Gültigkeitsbereich von Script-Variablen

Jede Variable besitzt einen fest definierten Gültigkeitsbereich (Scope), innerhalb dessen sie erreichbar ist und referenziert werden kann. Außerhalb des Gültigkeitsbereichs ist die Variable unbekannt. Für Script-Variablen, die durch Tag-Handler erzeugt werden, gibt es dabei drei Gültigkeitsbereiche:

Tabelle 4.1
Gültigkeitsbereiche für Script-Variablen

Scope	Beschreibung	Gesetzt in
`AT_BEGIN`	Diese Variable ist nach dem öffnenden Tag `<tag>` bis zum Ende der JSP gültig.	`doInitBody()` / `doAfterBody()`, wenn `BodyTagSupport` bzw. `doStartTag()` / `doEndTag()`
`NESTED`	Diese Variable ist im Rumpf des Tag gültig.	`doInitBody()` / `doAfterBody()`, wenn `BodyTagSupport`, sonst in `doStartTag()`
`AT_END`	Diese Variable existiert erst nach dem schließenden Tag `</tag>`.	Definiert in `doEndTag()`

4.5.2 Ein Tag-Handler für Script-Variablen

Der folgende Tag-Handler setzt Variablen mit verschiedenen Gültigkeitsbereichen, die Sie anschließend in der JSP verwenden können:

Listing 4.17 Tag-Handler für Script-Variablen

```
package de.akdabas.javaee.tags;
import java.io.IOException;
import javax.servlet.jsp.tagext.BodyTag;
import javax.servlet.jsp.tagext.BodyTagSupport;

/**
 * Erzeugt Script-Variablen mit verschiedenen Gültigkeitsbereichen
 * innerhalb der JSP
 */
public class ScriptVariableTag
   extends BodyTagSupport implements BodyTag {

   /** Erzeugt die Variablen 'atBegin' und 'nested' */
   public void doInitBody() {
      pageContext.setAttribute("atBegin","Set in doInitBody()");
      pageContext.setAttribute("nested","Set in doInitBody()");
   }

   /** Aktualisiert den Wert der Variablen am Ende des Rumpfs */
   public int doAfterBody() {

      // Aktulisieren der Variablen 'atBegin' und 'nested'
      pageContext.setAttribute("atBegin","Set in doAfterBody()");
      pageContext.setAttribute("nested","Set in doAfterBody()");

      // Inhalt des Rumpfs in die JSP einfügen, weil dieser sonst
      // übersprungen wird. Dies ist die Default-Implementierung.
      try {
         bodyContent.writeOut(bodyContent.getEnclosingWriter());
      } catch (IOException ioe) {
         ioe.printStackTrace();
      }
      return SKIP_BODY;
   }

   /** Erzeugt die Variable 'atEnd' die nach dem Rumpf gültig ist */
   public int doEndTag() {
      pageContext.setAttribute("atEnd","Set in doEndTag()");
      return EVAL_PAGE;
   }
}
```

Lokale Variablen innerhalb der JSP

Dieser Tag-Handler setzt die drei Variablen `atBegin`, `nested` und `atEnd`, die später mit den gleichnamigen Gültigkeitsbereichen ausgestattet werden, in verschiedenen Phasen des Tag-Lebenszyklus. Um eine Variable zu setzen, fügen Sie sie einfach in den `PageContext` ein:

Listing 4.18 Setzen einer Variablen innerhalb der JSP

```
...
   // Initialisiert oder aktualisiert eine JSP-Variable
   pageContext.setAttribute("VariablenName", VariablenWert);
...
```

Die Methode doInitBody()

Listing 4.17 macht Sie zusätzlich mit einer weiteren Methode des Tag-Lebenszyklus bekannt, welche diesmal `void` zurückgibt:

```
...
   // Wird gerufen, wenn das Tag tatsächlich einen Rumpf besitzt
   // und dient z.B. der Initialisierung von Script-Variablen
   public void doInitBody() {...}
...
```

Listing 4.19
Signatur der Methode doInitBody()

`doInitBody()` wird vom Interface `BodyTag` deklariert und soll der Initialisierung des Rumpfs dienen. Diese Methode wird nur ausgeführt, wenn das damit verknüpfte Tag auch tatsächlich einen Rumpf besitzt und die zuvor gerufene Methode `doStartTag()` nicht `SKIP_BODY` zurückgegeben hat.

4.5.3 Konfiguration des Tag-Handler

Im TLD (*masterclass.tld*) teilen Sie dem Servlet-Container nun alle benötigten Informationen wie den Name der Variablen, ihren Typ und natürlich den gewünschten Scope mit.

```
...
   <!-- Dieses Tag erzeugt die JSP-Variablen 'atBegin', 'nested' und
        'atEnd' vom Typ String -->
   <tag>
     <name>script-variable</name>
     <tag-class>de.akdabas.javaee.tags.ScriptVariableTag</tag-class>
     <variable>
        <name-given>atBegin</name-given>
        <variable-class>java.lang.String</variable-class>
        <declare>true</declare>
        <scope>AT_BEGIN</scope>
     </variable>
     <variable>
        <name-given>nested</name-given>
        <variable-class>java.lang.String</variable-class>
        <declare>true</declare>
        <scope>NESTED</scope>
     </variable>
     <variable>
        <name-given>atEnd</name-given>
        <variable-class>java.lang.String</variable-class>
        <declare>true</declare>
        <scope>AT_END</scope>
     </variable>
   </tag>
...
```

Listing 4.20
Konfiguration der Script-Variablen

Außer dem Namen des Tag und der Klasse des Tag-Handler fügen Sie für jede Variable ein `<variable>`-Tag ein, welches folgende Informationen enthält:

```
<variable>
   <name-given>Name Der Variablen</name-given>
   <variable-class>Qualifizierenden Name Des Typs</variable-class>
   <scope>Gültigkeitsbereich der Variablen</scope>
</variable>
```

Listing 4.21
Schematische Konfiguration einer JSP-Variablen

Das oben zusätzlich angegebene Flag `<declare>` zeigt an, ob der Servlet-Container die Variable bei Bedarf neu erzeugen soll (Standard: `true`).

4.5.4 Eine Beispiel-JSP

Nun schreiben wir eine einfache JSP, um die Funktionalität zu testen. Dabei ist darauf zu achten, stets nur auf die Variablen zuzugreifen, die im entsprechenden Scope sichtbar sind. Anderenfalls kommt es zu einer entsprechenden Fehlermeldung:

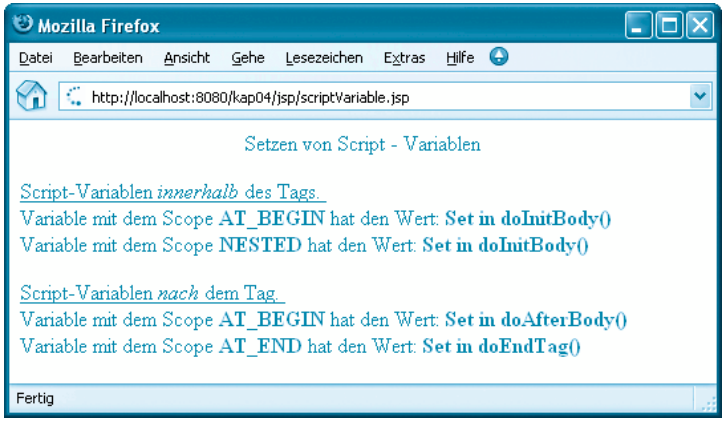
Listing 4.22 JSP zum Test der Script-Variablen

```
<%@ taglib uri="/masterclass-tags" prefix="masterclass" %>
<html>
   <body style="color:green;">
      <center>Setzen von Script - Variablen</center>

      <masterclass:script-variable>
        Script-Variablen innerhalb des Tags. <br/>
       Variable mit dem Scope AT_BEGIN hat den Wert: <%= atBegin %>
        Variable mit dem Scope NESTED hat den Wert: <%= nested %>
      </masterclass:script-variable>

        Script-Variablen nach dem Tag. <br/>
     Variable mit dem Scope AT_BEGIN hat den Wert: <%= atBegin %>
        Variable mit dem Scope AT_END hat den Wert: <%= atEnd %>
   </body>
</html>
```

Wie Sie in Listing 4.22 sehen, deklariert die JSP keine der Variablen selbst und kann in deren Gültigkeitsbereichen dennoch problemlos darauf zugreifen.

Die Variablen `atBegin` und `nested` sind innerhalb des Rumpfs sichtbar und können dort zum Beispiel ausgegeben oder auch für Berechnungen verwendet werden. Nach dem schließenden Tag `</masterclass:script-variable>` sind nur noch Variablen mit dem Gültigkeitsbereich `AT_BEGIN` oder `AT_END` sichtbar.

Abbildung 4.8 Resultat des Tag <script-variable>



Das Resultat in Abbildung 4.8 zeigt Ihnen deutlich, wie der Wert der Variablen `atBegin` in der Methode `doInitBody()` überschrieben wird.

4.6 Mit dem Kontext arbeiten

Bisher haben Sie Ihre Tags und die damit verknüpften Tag-Handler nur für sich betrachtet und ihren jeweiligen Kontext außer Acht gelassen. Einige Tags ergeben jedoch nur in Kombination mit anderen eine sinnvolle Struktur. So beschreibt das `<td>`-Tag in HTML beispielsweise die Zelle einer Tabelle (Table Data), was nun wirklich nur sinnvoll ist, wenn es sich tatsächlich in einer solchen (`<table>`) befindet. In diesem Abschnitt soll es um diese geschachtelten Tags gehen.

4.6.1 Wie realisiert man den Zugriff auf Eltern-Tags?

Tags, die das aktuelle Element umschließen, werden auch als *Eltern-Tags* (Parent-Tags) bezeichnet. Analog sind alle Elemente innerhalb eines Rumpfs *Kind-Tags* (Child-Tags). Aus dieser vermenschlichten Interpretation leitet sich auch die Methode ab, mit der Sie die umschließenden Elemente (Vorfahren) Ihres aktuellen Tag finden können: `findAncestorWithClass()` (engl. Ancestor, dt.: Vorfahre, Ahne).

Diese sowohl von `TagSupport` als auch von `BodyTagSupport` unterstützte Methode liefert Ihnen stets die »jüngsten« Vorfahren eines Tag zurück. Dieses Tag muss dabei auch in einer Tag-Library definiert und an einen Tag-Handler gebunden sein. Auf das Beispiel mit der HTML-Klasse gemünzt, könnten Sie also innerhalb der Methoden des `<td>`-Handler die Methode:

```
...
  // Bindet eine Referenz auf das Eltern-Tag an die Variable parent
  TableHandler parent =
    (TableHandler) findAncestorWithClass(this, TableHandler.class);
...
```

Listing 4.23
Schematisches Beispiel, um das Elterntag TableHeader zu lokalisieren

ausführen, um die umschließende Instanz (`<table>`) dieser Klasse (`<td>`) zu erhalten. Da die Methode `findAncestorWithClass()` nur ein verallgemeinertes `Object` zurückgibt, müssen Sie den entsprechenden Tag-Handler schließlich noch casten.

Tipp

Um einen weiter entfernten (»älteren«) Vorfahren unseres Tag zu erhalten, können Sie beispielsweise die Methode `findAncestorWithClass()` auf das zurückgegebene Elternobjekt anwenden und so die Ahnenreihe »rückwärts« abarbeiten.

Die Methode `findAncestorWithClass(from, type)` erwartet folgende Parameter:

- `javax.servlet.jsp.tagext.Tag from`

  Diese Klasse gibt das Ausgangs-Tag an, bei dem Sie die Suche nach einem Vorfahren beginnen wollen. Dies kann der aktuelle Tag-Handler (`this`) oder zum Beispiel ein bereits gefundener Vorfahr sein.

- `java.lang.Class type`

  Dieser Parameter gibt die Klasse des zu suchenden Objekts bzw. Tag-Handler an. Ein Cast des zurückgegebenen Tag auf diesen Typ ist in jedem Fall erfolgreich.

Tipp

Leider stellt Ihnen Java bislang keine Methode zum Auffinden bestimmter Kindelemente bereit. Sie können diese jedoch beispielsweise durch das Parsen (dt. »den Inhalt grammatikalisch bestimmen«) des `BodyContent`-Objekts ermitteln (siehe auch Kapitel 11).

4.6.2 Entscheidungen (if-then-else)

Um die Arbeit mit Tag-Handlers und die Suche nach dem richtigen Vorfahren kennen zu lernen, werden Sie in diesem Abschnitt eine Reihe von Tags entwickeln, mit denen Sie Entscheidungen (if-then-else) analog zu Java implementieren können. Das folgende Listing verdeutlich den Einsatz der folgenden Tags schon einmal schematisch:

Listing 4.24 Schematischer Aufbau tag-basierter Entscheidungen

```
...
   <if condition="Boolean">
      <then> Nomen est omen! </then>
      <else> Der optionale Else-Zweig. </else>
   </if>
...
```

Dabei sollen die einzelnen Tags folgende Aufgaben erfüllen:

- `<if>`

  Das umschließende `<if>`-Tag hält den gesamten Verbund zusammen und hat die Aufgabe, den Zustand der `condition` zu speichern.

  Des Weiteren zeigt das Vorhandensein eines `<if>`-Tag den anderen drei Tags an, dass sie im richtigen Kontext verwendet worden sind.

- `<then>`

  In dieses Tag kommt der Markup, der ausgegeben werden soll, wenn der Wahrheitswert `true` ist. Ist der Wert `false`, wird der Inhalt übersprungen.

- `<else>`

  Dieses Tag ist (wie das `<then>`-Tag) optional. Es ermittelt zur Laufzeit den Zustand der Entscheidung und gibt seinen Inhalt aus, wenn der Zustand nicht wahr (`false`) ist.

4.6.3 Überprüfen des Tag-Kontexts

Die Tags `<then>` und `<else>` ergeben nur einen Sinn, wenn sie innerhalb eines `<if>`-Tag verwendet werden. Deshalb überprüfen sie zu Beginn (`doStartTag()`), ob ein solches Eltern-Tag existiert, und speichern es gegebenenfalls zur späteren Verwendung in einer Instanz-Variablen.

Listing 4.25
Überprüfen des Kontexts

```
...
   /**
    * Überprüft, ob dieses Tag von einem Eltern-Tag umschlossen wird
    */
   public int doStartTag() throws JspException {

      // Test, ob der Kontext stimmt und das Eltern-Tag vorhanden ist
      ifTag = (IfTag) findAncestorWithClass(this, IfTag.class);
      if (ifTag == null) {
         throw new JspTagException("Could only used inside if!");
      }
      return EVAL_BODY_BUFFERED;
   }
...
```

Wurden die Tags nicht innerhalb eines `<if>`-Tag verwendet, kann die Methode `findAncestorWithClass()` keinen entsprechenden Elter finden und die Tags quittieren ihren Dienst mit einer für diese Fälle vorgesehenen `JspTagException`.

4.6.4 Der <if>-Handler

Der Handler für das Tag `<if>` entspricht, bis auf die Methode `doStartTag()`, einer JavaBean, welche den Zustand des `<condition>`-Tag speichert. Sie müssen allerdings die Methode `doStartTag()` überschreiben, da die Standardimplementierung der Basisklasse `TagSupport` `SKIP_BODY` zurückgibt.

Listing 4.26
Der <if>-Handler

```
package de.akdabas.javaee.tags;
import javax.servlet.jsp.tagext.Tag;
import javax.servlet.jsp.tagext.TagSupport;

/** Dieses Tag umschließt die anderen und speichert die Bedingung */
public class IfTag extends TagSupport implements Tag {

   /** Wert des Attributs 'condition' */
   private Boolean condition;

   /** Speichert den Wert des Attributs 'condition' */
   public void setCondition(Boolean condition) {
      this.condition = condition;
   }

   /** Gibt den Wert des Attributs 'condition' zurück */
   public Boolean getCondition() {
      return condition;
   }

   /**
    * Initialisiert das Tag und überschreibt das Verhalten der
    * Klasse TagSupport, die standardmäßig SKIP_BODY zurückgibt.
    */
   public int doStartTag() {
      return EVAL_BODY_INCLUDE;
   }
}
```

4.6.5 <then> und <else>

<then> und <else> unterscheiden sich von der Logik her nur durch eine einzige Negation (!). Dementsprechend ähnlich sind sich die beiden doAfterBody()-Methoden:

Listing 4.27 Der <then>-Handler

```
package de.akdabas.javaee.tags;
import java.io.IOException;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import de.akdabas.javaee.tags.IfTag;

/** Dieses Tag umschließt den <then> - Zweig */
public class IfThenTag extends BodyTagSupport implements BodyTag {

   /** Referenz auf das Eltern-Tag (<if>)*/
   private IfTag ifTag;

   /**
    * Überprüft, ob dieses Tag von einem Eltern-Tag umschlossen wird
    */
   public int doStartTag() throws JspException {

      // Test, ob der Kontext stimmt und das Eltern-Tag vorhanden ist
      ifTag = (IfTag) findAncestorWithClass(this, IfTag.class);
      if (ifTag == null) {
         throw new JspTagException("Could only used inside if!");
      }
      return EVAL_BODY_BUFFERED;
   }

   /** Gibt den Inhalt aus, wenn die Bedingung 'true' war */
   public int doAfterBody() {

      if (ifTag.getCondition()) {
         try {
            BodyContent body = getBodyContent();
            JspWriter out = body.getEnclosingWriter();
            out.print(body.getString());
         } catch (IOException ioe) {
            pageContext.getServletContext().log("Error", ioe);
         }
      }

      // Der Inhalt wurde an dieser Stelle ggf. eingefügt.
      return SKIP_BODY;
   }
}
```

Um den Leser nicht zu langweilen, zeigt das nächste Listing lediglich die beiden Stellen des Codes, in denen sich der <else>-Handler unterscheidet:

Listing 4.28 Unterschiede zwischen <then>- und <else>-Handler

```
public class IfElseTag extends BodyTagSupport implements BodyTag {
...
   public int doAfterBody() {
      if (! ifTag.getCondition()) {
...
}
```

Diese Tag-Handler quittieren die Methode doStartTag() mit der Konstanten EVAL_BODY_BUFFERED. Diese kann von BodyTag-Objekten zurückgegeben werden und bewirkt, dass der Rumpf des Tag in einem Puffer (engl. Buffer) zwischengespeichert wird und nachfolgend bearbeitet werden kann.

4.6.6 Konfiguration der Tags im Tag Library Descriptor

Natürlich müssen Sie jedes der drei Tags im Tag Library Descriptor einzeln konfigurieren:

Listing 4.29
Konfiguration der Tags

```
...
   <!-- 3 Tags für if-then-else Entscheidungen -->
   <tag>
      <name>if</name>
      <tag-class>de.akdabas.javaee.tags.IfTag</tag-class>
      <attribute>
         <name>condition</name>
         <required>true</required>
         <rtexprvalue>true</rtexprvalue>
         <type>java.lang.Boolean</type>
      </attribute>
   </tag>
   <tag>
      <name>then</name>
      <tag-class>de.akdabas.javaee.tags.IfThenTag</tag-class>
   </tag>
   <tag>
      <name>else</name>
      <tag-class>de.akdabas.javaee.tags.IfElseTag</tag-class>
   </tag>
...
```

4.6.7 Ein Beispiel

Die Erstellung der `<if>`-, `<then>`- und `<else>`-Handler mag zwar umfangreich gewesen sein, doch die Arbeit hat sich gelohnt. Ab jetzt können Sie Ihre JSPs mit klaren Entscheidungsstrukturen versehen, die auch ein Java-Unkundiger mühelos versteht.

Listing 4.30
Eine Beispiel-JSP

```
<%@ taglib uri="/masterclass-tags" prefix="masterclass" %>
<%
   Boolean isParamSet =
         new Boolean(request.getParameter("Test") != null);
%>
<html>
   <body style="text-align:center; color:green;">
      Der Parameter 'Test' ist
      <masterclass:if condition="<%= isParamSet %>">
         <masterclass:then>vorhanden.</masterclass:then>
         <masterclass:else>nicht vorhanden.</masterclass:else>
      </masterclass:if>
   </body>
</html>
```

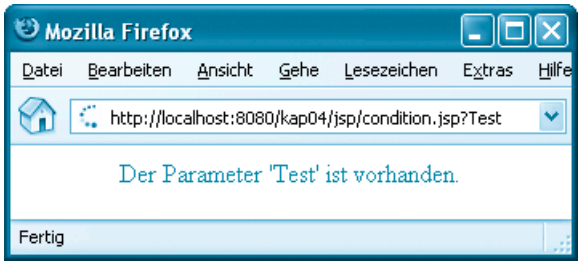


Abbildung 4.9
Tag-gesteuerte If-Else-Entscheidungen in JSPs

4.7 Zusammenfassung

Tag-Bibliotheken erweitern Ihre Fähigkeiten, dynamische Webanwendungen zu erstellen, und bieten einen flexiblen Ansatz, um wiederverwendbaren Code zu kapseln. Einmal erstellt, ermöglichen sie auch Java-Unkundigen den Einsatz entsprechender Funktionalität und erleichtern auf diese Weise die Zusammenarbeit zwischen Webdesigner und Java-Entwickler.

Es existieren bereits unzählige, frei verfügbare Tag-Bibliotheken für nahezu jede Aufgabe und die `taglib`-Direktive macht deren Einsatz in Ihren JSPs denkbar einfach. Die folgenden beiden Portale bilden einen guten Ausgangspunkt, um das Web nach geeigneten Bibliotheken zu durchsuchen:

- *JavaServer Pages Standard Tag Library (JSTL)*
  http://java.sun.com/products/jsp/jstl/.
- *Apache Jakarta Taglibs*
  http://jakarta.apache.org/taglibs.

Trotzdem ist es manchmal notwendig, eigene Tags selbst zu definieren. Sei es, dass Sie komplexe HTML-Strukturen immer und immer wieder benötigen und diese durch ein standardisiertes Tag einfügen wollen, eine sehr spezielle Java-Funktionalität benötigen oder ganz einfach Ihr ureigenes Tag entwickeln wollen.

Für diese Fälle haben Sie gelernt, wie Sie entsprechende Klassen (*Tag-Handler*) erstellen, mit einem Tag Library Descriptor (TLD) zu einer Bibliothek zusammenzufassen und in Ihre Applikation integrieren.