PEARSON Education

Markt+Technik

jetzt lerne ich

J2EE

Der einfache Einstieg in die Programmierung mit der Java 2 Enterprise Edition

THOMAS STARK

Markt+Technik

Kapitel 3

jetzt lerne ich

JavaServer Pages einsetzen

Nachdem Ihnen das letzte Kapitel die in JSPs verwendete Syntax nahe gebracht hat, geht es in diesem Kapitel nun darum, diese sinnvoll einzusetzen. Sie werden dazu vordefinierte Variablen und Seitendirektiven kennen lernen und diese verwenden, um das Request-Objekt auszulesen, Seiten zur Laufzeit dynamisch einzubinden und in einem kleinen Ratespiel benutzerspezifische Daten über dessen Sitzung (*Session*) zu verwalten. Abschließen wollen wir dieses Thema dann mit einem Absatz über JavaBeans, bei deren intensiver Verwendung Sie beinahe vollständig auf die spezielle JSP-Syntax des letzten Kapitels verzichten können.

3.1 Die vordefinierten Variablen einer JSP

Wenn Sie das vorangegangene Kapitel durchgearbeitet haben, werden Sie feststellen, dass Sie – bis auf einige Spielereien mit dem Datum und dem Abarbeiten fest definierter Scripts – nur wenig wirklich sinnvolle Anwendungen für den Einsatz von JavaServer Pages kennen, was sich mit diesem Kapitel hoffentlich ändern wird. Dazu werden Sie als Erstes die vordefinierten Variablen kennen lernen, mit deren Hilfe Sie zum Beispiel den Request auslesen, die Response manipulieren oder auf die Benutzer-Session zugreifen können.

Eigentlich haben Sie schon mit vordefinierten Variablen gearbeitet! Denn bei einigen Beispielen des vorangegangenen Kapitels haben Sie

auf die Variable `out` zurückgegriffen, um Ausgaben in JSP-Scriptlets zu erzeugen (Listing 2.20).

Diese Variable musste dabei weder gebunden, noch initialisiert werden, sondern stand Ihnen ohne Weiteres zur Verfügung. Tatsächlich ist `out` eine von acht *vordefinierten Variablen*, die Ihnen z.B. den Zugriff auf den HTTP-Request oder die Benutzer-Session ermöglichen. Doch um diese und ihr Verhalten zu verstehen, erweitern wir zunächst unser Wissen um das Internet, HTTP und verschiedene Konventionen.

3.1.1 Die verschiedenen Kontexte des Webservers

HTTP ist ein *zustandsloses Protokoll (Stateless Protocol)*. Das bedeutet, dass der Server auf einen eingehenden Request mit einer Response antwortet, diese und damit auch den Client aber sofort wieder »vergisst«. Das ist sehr nützlich, da der Verwaltungsaufwand für den Server gering bleibt und die viel gelobte Anonymität des Internets quasi fest eingebaut ist. Auf der anderen Seite kann ein Server (sofern ihm der Client keinen Tipp gibt) nicht sagen, ob die aktuelle Anfrage zu einem bestimmten Anwender gehört oder nicht. Anwendungen, wie virtuelle Einkaufswagen oder Online-Überweisungen, welche in der Regel mehrere Requests erfordern, wären damit nicht realisierbar.

Da sich komplexe Anwendungen bei absoluter Anonymität der Benutzer nur schwer realisieren lassen, haben sich im Laufe der Jahre verschiedene Konventionen entwickelt, die es dem Server gestatten, einen Client innerhalb bestimmter Grenzen wieder zu erkennen. Dazu können Sie sich Ihren Webserver als eine Art Empfangsdame einer großen Firma vorstellen. Diese nimmt alle eingehenden Anfragen von Kunden entgegen, überprüft, ob der Kunde bereits registriert ist, und leitet den Request anschließend an den zuständigen Mitarbeiter (die JSP) weiter.

Zur Unterstützung hat die virtuelle Empfangsdame neben Ihrem Telefon drei Notizblöcke liegen, auf denen sie sich bestimmte Informationen (*Attribute*) notieren kann. Zu bestimmten Zeitpunkten reißt sie ein bestimmtes Blatt aus einem der Blöcke heraus und beginnt ein neues. Alle Dinge, die auf diesem Blatt standen, müssen dann erledigt (gestrichen bzw. gelöscht) sein oder werden schlicht vergessen. Die Informationen verlassen dann ihren Gültigkeitsbereich (*Scope*) und sind nicht weiter von Belang.

Der Application-Scope

Auf dem Ersten der Notizblöcke steht Applikation (*application*) und auf diesem notiert sich die Sekretärin all die Dinge, die den ganzen Arbeitstag, also bis der Server neu gestartet wird, gültig sind und für alle Mitarbeiter (JSPs) und Kunden (Clients) gleichermaßen gelten. Also z.B. Informationen über das Unternehmen selbst (die Web-Applikation) oder etwa die Buchhaltung, womit natürlich in diesem Fall die Datenbank gemeint ist.

Der Session-Scope

Der zweite Notizblock trägt den Titel *Session* (dt. Sitzung) und ist dafür gedacht, kundenspezifische Informationen zu speichern. Ruft ein Kunde (Client) bei unserer Sekretärin an, so fragt ihn diese, ob er bereits eine Kundennummer besitzt. Verneint er dies, so vergibt die Sekretärin eine neue (eindeutige) Nummer – die so genannte *Session-ID*, beginnt ein neues Blatt im Session-Block und notiert sich das Datum des Kontakts. Anschließend leitet sie den Kunden an die JSP weiter.

Besitzt ein Client bereits eine eindeutige Kundennummer und ist das zugehörige Blatt vorhanden, so wird das Datum des letzten Kontakts mit dem aktuellen Datum überschrieben. Von Zeit zu Zeit durchsucht der Server den Session-Block nach Kunden, die sich lange nicht mehr gemeldet haben, und entfernt deren Blatt. Die Session-ID wird damit ungültig und der Kunde bekommt beim nächsten Kontakt wieder eine neue Nummer.

Die Session-ID wird in der Regel über ein Cookie weitergegeben und ist damit an den Browser gebunden bzw. wird beim Schließen des Browsers gelöscht. Es ist also vollkommen egal, ob Ihre Verbindung zum Internet zwischen zwei Requests unterbrochen wird und Sie eine neue IP-Adresse bekommen haben, solange Sie zwischendurch Ihren Browser nicht schließen oder das Session-Timeout überschreiten.

Öffnen Sie hingegen einen neuen Browser und wiederholen den Request an den gleichen Server, so sind Sie für diesen stets ein neuer Kunde (Client). Sie können Ihren virtuellen Warenkorb also nicht von einem Browser in einen anderen transferieren.

Der Request-Scope

Das ist der Schmierblock unserer Sekretärin, auf dem sie sich Notizen zum aktuellen Anruf macht. Das sind z.B. Informationen darüber, mit wem der Client sprechen möchte (den angeforderten URL), mit wem er eventuell schon gesprochen hat (sind Cookies vorhanden?) oder in welcher Sprache er die Antwort am liebsten hätte.

Nachdem die virtuelle Sekretärin alle erforderlichen Informationen notiert hat, stellt sie diese den JSPs über vordefinierte Variablen zur Verfügung. Doch nicht nur der Client kann den Server dazu veranlassen, sich bestimmte Informationen zu notieren. Sie selbst können ebenfalls in Ihren JSPs Informationen, die Sie benötigen, auf jedem der drei Blöcke hinterlegen.

3.1.2 Acht Variablen zur Kontrolle von JSPs

In diesem Abschnitt lernen Sie die acht vordefinierten Variablen kennen, die Ihnen in jeder JSP von Anfang an zur Verfügung stehen. Leider können wir Sie aufgrund des beschränkten Umfangs dieses Buches in den folgenden Abschnitten nur in die Verwendung dieser Variablen einführen und nicht alle Möglichkeiten beleuchten, die diese Ihnen ermöglichen.

- `out` (`javax.servlet.jsp.JspWriter`)

  Diese Variable haben Sie bereits verwendet. Sie ähnelt dem `PrintStream` `System.out` und dient dazu, Ausgaben in die JSP zu erzeugen.

- `request` (`javax.servlet.http.HttpServletRequest`)

  Hinter dieser Variablen verbirgt sich nichts anderes als der gleichnamige »Schmierblock« der virtuellen Sekretärin. Über diese Variable können Sie Informationen über den aktuellen Request erhalten oder dessen Bearbeitung per *Forward* (dt. Weiterleiten) an eine andere JSP weiterleiten.

- `response` (`javax.servlet.http.HttpServletResponse`)

  Über diese Variable können Sie auf die Antwort (HTTP-Response) des Servers an den Client Einfluss nehmen.

- `session` (`javax.servlet.http.HttpSession`)

  Auch diese Variable entspricht dem gleichnamigen Kontextblock. Über die Methoden `setAttribute()` und `getAttribute()` haben Sie

dabei die Möglichkeit, die Sekretärin anzuweisen, bestimmte Informationen zum aktuellen Client zu hinterlegen bzw. wieder hervorzuholen.

- `application` (`javax.servlet.ServletContext`)

  Mit dieser Variablen erhalten Sie Zugriff auf den dritten Block unserer Sekretärin. Auch dieses Objekt kann über die Methoden `getAttribute()` und `setAttribute()` Objekte (`java.lang.Object`) speichern und restaurieren. Der Unterschied zum benutzerspezifischen Session-Objekt besteht darin, dass sich alle Clients ein und dasselbe `application`-Objekt teilen.

Verwenden Sie den Application-Scope also nur für Objekte, welche wirklich für alle Clients gleichermaßen gültig sind, wie z.B. das Datenbank-Interface. Benutzerspezifische Informationen sollten immer im Session-Kontext abgelegt werden. Sie werden dann nach Überschreiten des Timeouts automatisch aus dem Speicher des Servers gelöscht.

Damit haben Sie die fünf wichtigsten vordefinierten Variablen kennen gelernt. Die anderen drei spielen in JSPs eher eine untergeordnete Rolle und werden erst bei der Verwendung von Servlets (Kapitel 4) und Tag-Bibliotheken (Kapitel 5) interessant. Als da wären:

- `config` (`javax.servlet.ServletConfig`)

  Diese Variable gestattet Ihnen den Zugriff auf die serverseitigen Initialisierungsparameter, die Ihnen im nächsten Kapitel begegnen werden und die Sie zum Beispiel über den Web Deployment Descriptor (*web.xml*) vorbelegen können.

- `pageContext` (`javax.servlet.jsp.PageContext`)

  Über das `pageContext`-Objekt haben Sie Zugriff auf wichtige Seitenattribute der JSP. Es ermöglicht außerdem das Weiterleiten des Requests an eine andere JSP sowie das Einbinden (*Include*) von Seiten. Sie werden später in diesem Kapitel allerdings auch Techniken kennen lernen, mit denen dies leichter zu bewerkstelligen ist.

- `page` (`java.lang.Object`)

  Diese vordefinierte Variable ist für JSP-Programmierer sicherlich die uninteressanteste: Sie ist nämlich nichts anderes als ein Synonym des Schlüsselwortes `this` in Java und verweist auf Ihr JSP-Objekt.

3.2 Das Auslesen des HTTP-Requests

Eine der interessantesten vordefinierten Variablen ist der HTTP-Request (`request`). Über ihn können Sie innerhalb der JSP Informationen über die aktuelle Anfrage erhalten.

3.2.1 Request-Parameter

Wenn der Benutzer Daten in ein HTML-Formular eingibt und dieses anschließend abschickt, werden die eingetragenen Werte als Request-Parameter übertragen. Um die Werte auslesen zu können, stellt Ihnen die vordefinierte Variable `request` die Methode `getParameter()` zur Verfügung.

Listing 3.1: Ein dynamisches HTML-Formular (readParam.jsp)

```
<%
   // Auslesen des Request-Parameters 'user'
   String userName = request.getParameter("user");
%>
<html>
   <body>
      <!-- Dieses HTML-Formular verweist auf sich selbst, was dazu
           führt, dass die Seite auch nach dem Absenden (Submit)
           immer wieder erscheint. -->
      <form action="./readParam.jsp">

         <%-- Ausgabe des zuvor ausgelesenen Request-Parameters --%>
         Sie sind angemeldet als: <%= userName %> <br />

         <%-- Request-Parameter des nächsten Submit --%>
         Anmelden als: <input type="text" name="user" />
                       <input type="submit" value="Anmelden" />
      </form>
   </body>
</html>
```

Dieses Beispiel demonstriert das Auslesen des Request-Parameters `user` anhand eines Formulars, das sich nach dem Absenden (`submit`) selbst wieder aufruft, obwohl Sie die Daten eines Formulars natürlich prinzipiell an jeden beliebigen URL weiterleiten können.

```
...
   <form action="./readParam.jsp">
...
```

Im oberen JSP-Scriptlet lesen Sie den aktuellen Wert des Parameters aus und speichern ihn in der lokalen Variablen `userName`, um ihn weiter unter ausgeben zu können.

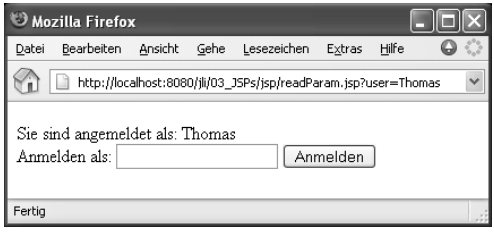


Abb. 3.1: Ein dynamisches HTML-Formular

Wenn Sie das Formular testen, werden Sie feststellen, dass das Eingabefeld zunächst immer leer ist. Sie können dieses nun auch auf einfache Weise vorbelegen:

```
...
   <!-- Vorbelegen des Eingabefeldes mit zuvor ausgelesenen Wert -->
   Anmelden als:<input type="text" name="user"
                      value="<%= userName %>" />
...
```

Listing 3.2: Vorbelegen von Input-Feldern

3.2.2 Auslesen aller übermittelten Request-Parameter

Neben der Methode `getParameter()` stellt die vordefinierte Variable `request` noch weitere Methoden zum Auslesen von Request-Parametern bereit, um beispielsweise alle gesetzten Parameter ohne Kenntnis über deren Anzahl oder Namen auszulesen.

Da Sie in der Regel allerdings wissen, welchen Wert Sie auslesen möchten, beschränken sich die Anwendungen in den meisten Fällen auf das Erzeugen von Debug-Informationen. Das folgende Listing demonstriert dies:

```
<%
   // Auslesen aller übermittelten Request-Parameter
   java.util.Enumeration params = request.getParameterNames();
%>
<html>
   <body>
```

Listing 3.3: Ausgabe aller Request-Parameter

```
      <!-- Ausgabe aller Request-Parameter zwecks Debugging -->
      Die folgenden Request-Parameter sind gesetzt:
      <ul>
         <%
            while (params.hasMoreElements()) {
               String paramName = params.nextElement().toString();
         %>
            <li>
               <%= paramName %> =
               <%= request.getParameter(paramName) %>
            </li>
         <% } %>
      </ul>

   </body>
</html>
```


Verwechseln Sie die bisher beschriebenen Request-*Parameter* nicht mit den eingangs Kapitel 2 erwähnten Request-*Headern* des HTTP-Protokolls. Request-Header enthalten vom Browser automatisch erzeugte Meta-Informationen, wie die bevorzugte Sprache und vom Browser darstellbare Dateiformate. Request-Parameter sind explizit erzeugte Informationen, die dazu dienen, einen Request zu personalisieren und beispielsweise eine HTML-Seite mit unterschiedlichen Werten zu füllen.

3.2.3 Auslesen von Request-Headern

Abschließend werden wir uns nun den Request-Headern widmen und damit die Betrachtung der vordefinierten Variablen `request` vorerst abschließen. Auf diese vom Browser erzeugten Meta-Informationen haben Sie in der Regel keinen Einfluss. Trotzdem kann die Auswertung dieser Information von Vorteil sein – beispielsweise um Inhalt und Übertragungsmodus der angeforderten Seite zu optimieren. Zunächst werden Sie die übermittelten Header jedoch analog zu den Request-Parametern (Listing 3.3) einfach ausgeben.

Listing 3.4: Auslesen aller Request-Header (readParams.jsp)

```
<%
   // Auslesen aller übermittelten Request-Header
   java.util.Enumeration headers = request.getHeaderNames();
%>
<html>
   <body>
      Die folgenden Request-Header wurden übermittelt:
      <ul>
```

```
        <% // Ausgabe der Request-Header via While-Schleife
           while (headers.hasMoreElements()) {
              String header = headers.nextElement().toString();
        %>
           <li><%= header %> = <%= request.getHeader(header)%></li>
        <% } %>
      </ul>

   </body>
</html>
```

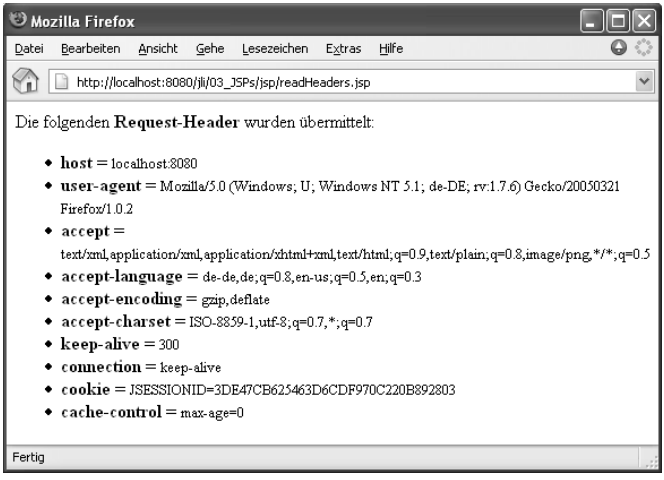


Abb. 3.2: Ausgabe der HTTP-Header. Wie geschwätzig ist Ihr Browser?

Keine Panik, für den erfolgreichen Einsatz von JSPs müssen Sie nicht alle HTTP-Header kennen, wenngleich sie Ihnen interessante Optimierungsansätze bieten. Sie werden später immer wieder verschiedenen HTTP-Headern begegnen und nach und nach lernen, wie Sie diese einsetzen können. Im Augenblick sollen Sie nur verstehen, wie Sie an die übermittelten Informationen gelangen.

Analog zur Methode `getParameterNames()` liefert Ihnen die Methode `getHeaderNames()` zunächst eine Aufzählung (*Enumeration*) aller übermittelten Request-Header:

```
...
   // Auslesen aller übermittelten Request-Header
   java.util.Enumeration headers = request.getHeaderNames();
...
```

Listing 3.5: Ermitteln aller übermittelten Request-Header

Request-Header unterscheiden *nicht* zwischen Groß- und Kleinschreibung. So führen die Anweisungen `getHeader("CONTENT-TYPE")` und `getHeader("content-type")` stets zum gleichen Resultat.

Je nach verwendeter HTTP-Version dürfen gleiche Header mehrmals (HTTP 1.0) bzw. nur ein einziges Mal auftauchen (HTTP 1.1). Bestimmte Header übermitteln dabei aber mehr als einen Wert, wie etwa `Accept` oder `Accept-Language` in Abbildung 3.2. Damit Sie auch in diesem Fall alle gesetzten Werte auslesen können, hält das `HttpServletRequest`-Objekt die Methode `getHeaders()` bereit, welche auch eine `Enumeration` zurückgibt, nur diesmal eben mit den Werten.

Listing 3.6: Auslesen von Headern mit mehreren Werten

```
// Auslesen der Werte mehrfach belegter Request-Header
java.util.Enumeration values = request.getHeaders("NameDesHeaders");
```

3.2.4 Besonders häufig verwendete Request-Header

Abschließend sei noch erwähnt, dass einige Request-Header so häufig verwendet werden, dass sie über eigene Methoden ausgelesen werden können. Deren Verwendung anstelle von `getHeader()` macht Ihren Java-Code dann wiederum lesbarer. Zu diesen Methoden gehören:

- `getContentType()`

  Über diesen Header teilt Ihnen der Browser die von ihm akzeptierten Dokumentformate (*MIME-Types*) in der von ihm favorisierten Reihenfolge mit. Bevor Sie den Client also mit *PNG-Grafiken* oder *PDF-Dokumenten* überschütten, können Sie so überprüfen, ob dieser damit überhaupt etwas anfangen kann.

- `getCookies()`

  Cookies (dt. Kekse) sind Name-Wert-Paare, über die Sie praktisch eigene Header erzeugen können. Diese werden zunächst vom Server erzeugt und bei späteren Requests vom Client zurückgesandt. Dadurch kann der Server verschiedene Clients voneinander unterscheiden.

  Cookies werden z.B. häufig für die Zuordnung von virtuellen Warenkörben und ihren Benutzern oder in Chat-Räumen verwendet.

- `getLocale()` und `getLocales()`

  Schließlich bevorzugen viele Anwender Dokumente in ihrer Muttersprache. Über diese Methoden können Sie die `java.util.Locale` des

Clients in absteigender Wertigkeit ermitteln und so bei Dokumenten, welche in unterschiedlichen Sprachen vorliegen, das jeweils günstigste wählen.

Wird dieser Request-Header wider Erwarten einmal doch nicht übermittelt, so ist das Resultat die Standard-Locale des Servers.

Sollten die Werte des ausgelesenen Headers vom Typ `int` sein oder einen Zeitstempel (`long`) repräsentieren, können Sie diese auch über die beiden folgenden Methoden auslesen und sich die Konvertierung sparen:

```
// Liest einen Header vom Typ 'int' aus.
public int getIntHeader(String "NameDesHeaders");

// Dient zum Auslesen eines Datums, der zurückgegebene Wert ist
// der zum Datum gehörende Zeistempel ('long').
public long getDateHeader(String "NameDesHeaders");
```

Listing 3.7: Zusätzliche Methoden zum Auslesen von Request-Headern

3.3 Direktiven – Eigenschaften einer JSP

In diesem Abschnitt geht es um die Eigenschaften Ihrer JSPs. Sie werden lernen, wie Sie Packages importieren, den Typ Ihres Dokuments festlegen und auf Fehler reagieren. Die Eigenschaften (Attribute) sind dabei in drei Kategorien aufgeteilt: `page`, `include` und `taglib`, die jeweils unterschiedliche Blickwinkel auf JSPs ermöglichen.

- `page`

  Über diese Direktive beschreiben Sie die Eigenschaften Ihrer JSP wie zum Beispiel Import, Vererbung oder die Pufferung im Speicher.

- `include`

  Diese Direktive erleichtert das Einfügen anderer JSPs.

- `taglib`

  Diese Direktive ermöglicht die Definition von eigenen Tags, in denen Sie immer wiederkehrende Anweisungen kapseln können. Die Direktive ist dabei jedoch so umfassend und wichtig, dass ihr ein eigenes Kapitel (Kapitel 5) gewidmet wird.

Alle drei Direktiven haben stets die gleiche Form:

Listing 3.8: Allgemeine Form von Direktiven

```
<%-- Allgemeine Form von JSP-Direktiven --%>
<%@ NameDerDirektive attribut="Wert" %>          <%-- bzw. --%>
<%@ NameDerDirektive attribut1="Wert1" attribut2="Wert2" ... %>
```

Sie beginnen mit der Zeichenkette »<%@«, gefolgt vom Namen der Direktive. Anschließend können Sie verschiedene Attribute der jeweiligen Direktive definieren, z.B.:

Listing 3.9: Beispiele für JSP-Direktiven

```
<%-- Beispiele für JSP-Direktiven --%>
<%@ include file="./Banner.jsp" %>
<%@ page session="true" autoflush="true" %>
```

3.3.1 Die Seitendirektive page

Die Attribute der Seitendirektive `page` werden unabhängig vom Java-Code interpretiert und nehmen nur indirekt Einfluss auf den Java-Code Ihrer JSP. Sie können damit prinzipiell überall in der JSP platziert werden. Da einige jedoch interpretiert werden müssen, bevor ein Inhalt an den Client gesendet wird, und auch um die Lesbarkeit Ihrer JSPs zu erhöhen, ist es ratsam, alle Attribute im oberen Teil der JSP zusammenzufassen.

Mögliche Attribute sind `import`, `extends`, `isThreadSafe`, `session`, `buffer`, `autoflush`, `contentType`, `pageEncoding`, `isELIgnored`, `info`, `errorPage`, `isErrorPage` und `language`, deren wichtigste Funktionen Sie in diesem Abschnitt kennen lernen werden.

Die Seitendirektive `page` hat übrigens nichts mit der gleichnamigen vordefinierten Variablen zu tun. Im Gegensatz zu dieser werden Sie die Seitendirektive bald sehr zu schätzen wissen.

Importieren von Klassen und Packages

In den bisherigen Listings waren Sie gezwungen, stets den vollständigen Pfad (wie z.B. `java.util.Date`) der verwendeten Klassen anzugeben, denn Import-Anweisungen, wie Sie sie aus der Applet- oder Anwendungsprogrammierung kennen, schlagen sowohl als JSP-Deklaration, wie auch als JSP-Scriptlet fehl.

Über das `page`-Attribut `import` haben Sie nun die Möglichkeit anzugeben, welche Klassen importiert werden sollen. Diese können dann ohne Pfad verwendet werden:

```
<%-- Importieren der Java-Klasse 'Date' --%>
<%@ page import="java.util.Date" %>
...
<%-- Verwenden der Klasse 'Date' ohne vollständigen Pfad --%>
<% Date myDate = new Date() %>
```

Listing 3.10: Import java.util.Date

Statt einzelner Klassen können Sie natürlich auch alle Klassen eines Packages zusammen importieren oder mehrere Import-Anweisungen zusammenfassen:

```
<%@ page import="java.util.*" %>                <%-- oder --%>
<%@ page import="java.util.Date, java.text.SimpleDateFormat" %>
```

Listing 3.11: Beispiele für den Klassen-Import

`import` ist dabei *das einzige Attribut* der Seitendirektive, welches innerhalb einer JSP *mehrmals vorkommen* darf.

Benutzer-Sitzungen vermeiden

Das Session-Objekt, das Sie im nachfolgenden Abschnitt kennen lernen werden, ist dafür gedacht, benutzerspezifische Daten von einem Request zum nächsten zu speichern. Doch in einigen Fällen können Sie sich die dafür benötigten Ressourcen des Webservers sparen.

```
<%-- Direktive, die die Unterstützung für Sessions ausschaltet --%>
<%@ page session="false">
```

Listing 3.12: Vermeiden von Sitzungen

Mit dem `session`-Attribut teilen Sie dem Webserver mit, dass *diese JSP* nicht an Sitzungen teilnimmt. Der Webserver überprüft also nicht, ob bereits eine Session existiert (was durchaus der Fall sein kann), und erzeugt auch keine neue. Das Session-Objekt wird dabei keinesfalls gelöscht, sondern lediglich für diese Webseite gesperrt. Ruft der Benutzer anschließend eine Seite mit Session-Unterstützung auf, steht sie ihm anschließend wieder zur Verfügung.

Nimmt eine JSP nicht an Sitzungen (engl. *Sessions*) teil, so hat sie auch keinen Zugriff auf die vordefinierte Variable `session`. Verwenden Sie diese dennoch, produzieren Sie unweigerlich eine Fehlermeldung.

Verschiedene Dokument-Typen erzeugen

Bisher haben Sie mit Ihren JSPs ausschließlich HTML-Seiten erzeugt, welche anschließend vom Browser dargestellt wurden. Doch woher

weiß dieser eigentlich, dass es sich bei dem empfangenen Inhalt um eine HTML-Seite und nicht beispielsweise um ein PDF-Dokument handelt? An der Endung *.jsp* kann er es schließlich nicht erkennen.

Die Antwort ist der Response-Header *Content-Type*. Dieser wird dem Browser in Form eines Response-Headers übermittelt und kann von Ihnen z.B. auch über die vordefinierte Variable `response` gesetzt werden.

Listing 3.13: Der Content-Type-Header

```
...
   // Setzen des Dokument-MIME-Types über die Variable 'response'
   response.setContentType("text/html");
...
```


Der Response-Header `Content-Type` teilt dem Browser die Art des nachfolgenden Dokuments, den so genannten *MIME-Type*, (z.B. HTML) mit. Anhand dieser Information entscheidet der Browser, wie er das Dokument rendert und ob zur Darstellung etwa ein PlugIn, wie der Acrobat Reader, benötigt wird. Das folgende Listing zeigt einige häufig verwendeten MIME-Typen. Eine (nahezu) vollständige Liste finden Sie unter *http://de.selfhtml.org/diverses/mimetypen.htm*.

Listing 3.14: Einige häufig verwendete MIME-Typen

```
text/html
text/plain
image/jpeg
application/pdf
application/vnd.ms-excel
```

Von nun an können Sie mit Hilfe von JSPs also weitaus mehr als reine HTML-Seiten, sondern zum Beispiel auch dynamische XML-Dokumente erzeugen:

Listing 3.15: Dynamisches XML per JavaServer Page

```
<%-- Zuerst setzen Sie den Dokument-Type (MIME-Type) ... --%>
<%@ page contentType="text/xml" %>

<%-- ... und hier erzeugen Sie die XML-Struktur --%>
<map>
   <entry key="key" value="<%= request.getParameter("key")%>" />
   <entry key="date" value="<%= new java.util.Date() %>" />
</map>
```

Wie Sie sehen, können Sie auch weiterhin alle vordefinierten Variablen verwenden und auch beliebigen Java-Code einbetten. Sie ändern schließlich nur das Darstellungsformat.

Abbildung 3.3 belegt auch, dass der Browser die Art der Darstellung nur über den Mime-Typen bestimmt, denn obwohl Ihr Dokument ebenfalls die Endung *.jsp* hat, wird es ganz anders dargestellt.

Abb. 3.3: Ein dynamisch erzeugtes XML-Dokument

Der Response-Header `content-Type` darf natürlich nur ein *einziges Mal* gesetzt werden und *muss* (wie alle anderen Header) vor dem Inhalt des Dokuments übermittelt werden. Mehr zum Aufbau und der Übertragung einer HTTP-Response finden Sie im Kapitel 2.

Einstellen des zu verwendenden Zeichensatzes

Neben dem Dokument-Typ legt das `page`-Attribut `contentType` auch den zu verwendenden Zeichensatz fest. Im so genannten *Character Set* wird definiert, welche binäre Codefolge welchem darzustellenden Zeichen zugeordnet ist. So wird die Folge `11101010` (`0xEA`) im hierzulande gebräuchlichen Lateinischen Zeichensatz (ISO-8859-1, Latin-1) als »e mit Circumflex« gerendert. Mit dem Zeichensatz für kyrillische Buchstaben (ISO-8859-5) hingegen würde die gleiche Folge als »kleines Härtezeichen« interpretiert werden.

Über die folgende Seitendirektive können Sie den zu verwendenden Zeichensatz festlegen, doch Vorsicht vor allzu gewagten Experimenten, denn die wenigsten Browser unterstützen auf Anhieb das japanische *Kanji* :-)

Listing 3.16: Definition des zu verwendenden Zeichensatzes (schematisch)

```
<%-- Setzen von MIME-Type und Zeichensatz --%>
<%@ page contentType="MIME-Type; charset=Encoding" %>
```

oder konkret:

Listing 3.17: Beispiel zur Manipulation des Zeichensatzes

```
<%-- Eine Einstellung für Westeuropäische XML-Dokumente --%>
<%@ page contentType="text/xml; charset=ISO-8859-1" %>
```

Da das in der westlichen Welt verbreitete ISO-System wegen der Fülle der zu kodierenden Zeichen bei asiatischen Sprachen versagt, wurde 1991 das Unicode-Konsortium gegründet (*http://www.unicode.org*) und mit der Aufgabe betraut, einen Zeichensatz zu entwickeln, der in der Lage ist, alle existierenden Zeichen in einem einheitlichen System abzubilden. Herausgekommen ist das *Unicode-System* (Abkürzung UTF-8), in dem alle Zeichen grundsätzlich mit 2 Byte kodiert werden und das auch von Java intern verwendet wird. In Ihren JSPs sollten Sie also nach Möglichkeit UTF-8 als Zeichensatz verwenden.

Listing 3.18: Zeichensatz für JSP-Entwickler

```
<%-- Verwendung von UTF-8 --%>
<%@ page contentType="text/html; charset=UTF-8" %>
```


Der Standard-Zeichensatz, wenn Sie kein `contentType`-Attribut bestimmen, ist übrigens `text/html` mit dem Zeichensatz `ISO-8859-1`.

Festlegen der Fehlerseite

Sie bedenken alles und der Benutzer gibt beim Ausfüllen Ihres Formulars trotzdem als Mengenangabe einen Buchstaben statt einer Zahl ein ...

Wenn Sie Java-Code zur Ausführung bringen und insbesondere wenn dieser Benutzereingaben verarbeitet, kann es immer mal wieder zu nicht abgefangenen Ausnahmen kommen. Und da der Missetäter (der Benutzer) in der Regel nichts mit der daraus resultierenden Fehlermeldung anfangen kann, sollten Sie ihn über das `page`-Attribut `errorPage` auf eine von Ihnen bereitgestellte Fehlerseite weiterleiten.

Listing 3.19: Festlegen einer Fehlerseite

```
<%-- Festlegen der für diese JSP zu verwendenden Fehlerseite --%>
<%@ page errorPage="NameDerFehlerseite.jsp" %>
```

Die Definition des Attributs `errorPage` bringt den Webserver dazu, im Fall der Fälle nicht die ursprüngliche, fehlerhafte Seite anzuzeigen, son-

dern die unter dem relativen URL `errorPage` angegebene Seite weiterzuleiten (*forward*).

Diese sollte dann wiederum die `page`-Direktive:

```
<%-- Markiert diese JSP als "Fehler-Behandlungs-Seite" --%>
<%@ page isErrorPage="true" %>
```

Listing 3.20: Deklarieren einer JSP als Fehlerseite

enthalten, um Zugriff auf die zusätzlich definierte Variable `exception` (Typ `java.lang.Throwable`) zu erhalten, hinter der sich nichts anderes als der aufgetretene Fehler verbirgt. Die folgenden beiden Listings verdeutlichen das Zusammenspiel der Attribute `errorPage` und `isErrorPage`.

```
<%-- Hier legen Sie die Fehlerseite fest --%>
<%@ page errorPage="HandleError.jsp" %>

<%
   // Dieser Code ist fehleranfällig für Falscheingaben
   if (request.getParameter("number") != null) {
      Double myDouble = new Double(request.getParameter("number"));
   }
%>
<html>
   <body>

      <!-- Dieses Formular verweist auf sich selbst -->
      <form action="ProduceError.jsp">
         <p>
            Wenn Sie in das folgene Feld keine Zahl eingeben,
            provozieren Sie einen Fehler:
         </p>

         Eingabe: <input type="text" name="number" />
                  <input type="submit" value="Absenden" />
      </form>
   </body>
</html>
```

Listing 3.21: Seite mit möglichem Fehler (ProduceError.jsp)

Zunächst legen Sie die Fehlerseite für diese JSP fest. Anschließend versuchen Sie, den Parameter `number` in einen `Double` zu konvertieren. Dabei kann es durch Fehleingaben z.B. zu einer `NumberFormatException` kommen.

Abb. 3.4: Seite mit potenziellem Fehler

Listing 3.22: Fehlerbehandlung (HandleError.jsp)

```
<%-- Hiermit qualifizieren Sie diese JSP als Fehlerseite --%>
<%@ page isErrorPage="true" %>

<%-- Zur einfacherer Ausgabe importieren Sie 'PrintWriter' --%>
<%@ page import="java.io.PrintWriter" %>

<html>
   <body>
      <%-- Hier verwenden Sie die zusätzliche Variable --%>
      Es ist zu folgender Ausnahme gekommen: <%= exception %> <br/>
      <a href="ProduceError.jsp"> Erneut versuchen ?! </a> <hr/>

      Der Stacktrace lautet:
      <% exception.printStackTrace(new PrintWriter(out)); %>

   </body>
</html>
```

Als Erstes importieren Sie die Klasse `java.io.PrintWriter`, um den *Stacktrace* der `Exception` in der JSP ausgeben zu können. Anschließend deklarieren Sie diese Seite als *ErrorPage*, um Zugriff auf die vordefinierte Variable `exception` zu erhalten.

Die vordefinierte Variable `exception` existiert nur, wenn es tatsächlich zu einer Ausnahme gekommen ist.

Abbildung 3.5 zeigt Ihnen außerdem, dass der URL (*HandleError.jsp*) der eigentlichen Fehlerseite vor dem Benutzer verborgen bleibt. Da die Weiterleitung auf die Fehlerseite intern vonstatten geht, zeigt der Browser weiterhin den ursprünglichen URL (*ProduceError.jsp*) an.

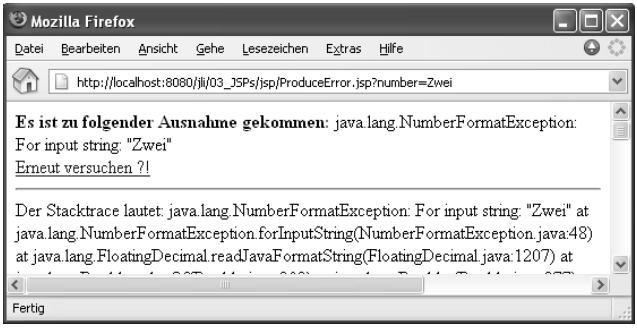


Abb. 3.5: Ein abgefangener Fehler

3.3.2 Einbinden von Dateien – die Direktive include

Dieser Abschnitt zeigt Ihnen verschiedene Techniken, mit denen Sie externe Ressourcen in Ihre JSPs integrieren oder die JSPs auf diese weiterleiten können (wobei es sich um HTML-Seiten oder sogar andere JSPs handeln kann).

Include per Direktive

Wie Sie wissen, wird der Java-Code Ihrer JSPs vor der Ausführung vom Webserver kompiliert und anschließend zur Ausführung gebracht. Wäre es dann nicht sinnvoll, `include`-Anweisungen bereits hier zu erkennen und die entsprechenden Seiten gleich zu integrieren, anstatt diese über einen weiteren (internen) Request vom eigenen Webserver anzufordern? Genau dazu dient die *Include-Direktive*. Das folgende Listing demonstriert Ihnen dies, indem es die Debug-Seite aus Listing 3.3 integriert.

```
<html>
   <body>
      <p style="text-align:center">
         Diese Seite ist um Debug-Informationen ergänzt!
      </p>

      <hr />

      <%-- Hiermit binden Sie den Code einer anderen JSP ein --%>
      <%@ include file="readParams.jsp" %>
   </body>
</html>
```

Listing 3.23: Dauerhaftes Einbinden der Debug-Seiten (includeDirektive.jsp)

In dieses Listing werden die einzubindenden Seiten während der internen Übersetzung des Java-Codes direkt integriert. Hierdurch haben sie Zugriff auf den umgebenden Java-Code, wie:

- zuvor in JSP-Scriptlets definierte lokale Variablen und
- in JSP-Deklarationen definierte globale Methoden und Variablen.

Als Merkhilfe können Sie sich vorstellen, dass die *Include-Direktive* durch die eigentliche Seite ersetzt wird, ganz so als würden Sie diese per *Copy-&-Paste* an dieser Stelle einfügen.

Diesen Komfort erkaufen Sie sich allerdings nicht ganz kostenlos: Vielleicht ist es Ihnen schon aufgefallen, aber mit der aktuellen Konfiguration erkennt der Webserver automatisch, wenn sich Ihre JSPs geändert haben und kompiliert diese bei Bedarf neu. Dadurch können Sie Ihre JSPs editieren und die Änderungen durch Drücken des Aktualisieren-Schalters Ihres Browsers betrachten. Im obigen Beispiel wird der Code der inkludierten Seite jedoch direkt als Kopie eingefügt. Wenn Sie nun das Original (*readParams.jsp*) anpassen, müssen Sie den Webserver dazu veranlassen, auch die Seite *includeDirective.jsp* (Listing 3.23) neu zu kompilieren, um die Änderungen zu übernehmen.

Um den Webserver zu veranlassen, die betreffende JSP erneut zu übersetzen, können Sie diese einfach öffnen und erneut speichern. Unter Linux steht Ihnen außerdem der elegante `touch`-Befehl zur Verfügung.

Diese Technik eignet sich daher eher für statische Seiten wie Menus, Copyright-Informationen und dergleichen. Für oft wechselnde Inhalte wie Banner, etc. ist sie ungeeignet.

Include zur Laufzeit

Offensichtlich ist die obige Methode nicht für alle Anwendungen gleichermaßen geeignet. Statt die einzufügende Seite direkt hinein zu kompilieren, können Sie auch einfach deren Resultat in Ihre Seite integrieren. Dabei schickt sich der Webserver beim Aufruf der ursprünglichen Seite gewissermaßen selbst einen neuen Request, in dem er die einzufügende Seite aufruft.

Der Vorteil dabei ist:

- Es wird stets der Code der einzufügenden Seite selbst aufgerufen und nicht etwa eine Kopie davon.

Hierdurch kann es nicht zu den oben beschriebenen Aktualisierungsproblemen kommen, allerdings hat auch diese Technik eine Kehrseite, da der Webserver die Seite über einen »ganz normalen« Request aufruft:

- Die einzufügende Seite muss »in sich geschlossen« sein, d.h., Sie haben innerhalb der Seite keinen Zugriff auf die Methoden oder Variablen der Seite, die sie einfügt.

Äußerlich unterscheiden sich die beiden Include-Varianten kaum. Statt der Direktive verwenden Sie nun einfach ein spezielles Tag:

Listing 3.24: Include per Tag (includeTag.jsp)

```
<html>
   <body>
      <p style="text-align:center">
         Diese Seite ist um Debug-Informationen ergänzt!
      </p>

      <hr />

      <%-- Einbinden des Resultates einer anderen JSP --%>
      <jsp:include page="readParams.jsp" />
   </body>
</html>
```

Und es kommt noch besser. Sie können die einzubindenden JSPs auch problemlos mit neuen Parametern aufrufen und sogar vorhandene Request-Parameter überschreiben. Das folgende Beispiel verdeutlicht dies. Hierbei wird die JSP *caller.jsp* mit den Parametern `Param1` und `Param2` aufgerufen. Diese bindet die JSP *callee.jsp* ein und überschreibt dabei den zweiten Parameter mit einem festen Wert.

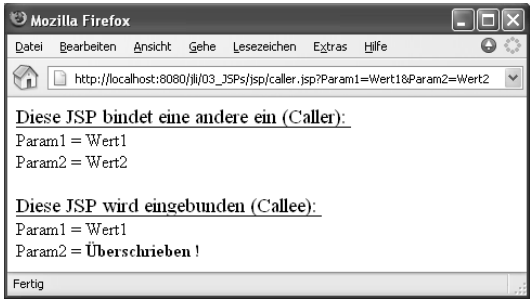
Listing 3.25: Bindet eine JSP dynamisch ein und überschreibt Parameter (caller.jsp)

```
<html>
   <body>
      Diese JSP bindet eine andere ein (Caller): <br/>
      Param1 = <%= request.getParameter("Param1") %> <br/>
      Param2 = <%= request.getParameter("Param2") %> <br/>

      <%-- Beim Einbinden auch Parameter überschreiben --%>
      <jsp:include page="callee.jsp">
         <jsp:param name="Param2" value="Überschrieben !" />
      </jsp:include>

    </body>
</html>
```

Listing 3.26: Wird eingebunden und gibt den Wert der Parameter aus (callee.jsp)

```
<html>
   <body>
      Diese JSP wird eingebunden (Callee): <br/>
      Param1 = <%= request.getParameter("Param1") %> <br/>
      Param2 = <b><%= request.getParameter("Param2") %> </b><br/>
    </body>
</html>
```

Abb. 3.6: Überschreiben von Request-Parametern



Forward zur Laufzeit

Mit dieser Technik können Sie nicht nur fremde Ressourcen einbinden, sondern auch auf diese weiterleiten (engl. *forward*). Auch hier können Sie die Parameter des ursprünglichen Requests überschreiben.

Listing 3.27: Alternative Definition eines Forwards

```
<html>
   <body>
      Auf dieser Seite werden Sie weitergeleitet !

      <%-- Hier leiten Sie den Request weiter --%>
      <jsp:forward page="callee.jsp">
         <jsp:param name="Param2" value="Überschrieben !" />
      </jsp:forward>
   </body>
</html>
```

3.4 Verwendung der Benutzer-Session

Nachdem Sie nun erste Erfahrungen mit vordefinierten Variablen und Direktiven gesammelt haben, ist es Zeit, dieses Wissen zu bündeln, um ein erstes Beispiel zu implementieren. Dabei werden Sie lernen:

- Wie Sie das benutzerspezifische Session-Objekt verwenden, um Daten abzulegen.
- Wie Sie URLs für Links kodieren.

3.4.1 Was ist die Benutzer-Session

Die Session (dt. Sitzung) ist, wie Sie bereits wissen, ein dem Benutzer zugeordnetes Objekt, in welchem Sie Informationen zum Benutzer ablegen können (Stichwort: Schreibblock der virtuellen Sekretärin, Abschnitt 3.1.1). Sie wird Ihnen in Form einer vordefinierten Variablen (`session`) zur Verfügung gestellt.

Die Session wird dabei über eine eindeutige Nummer identifiziert, welche, wie Sie sehen werden, entweder an den URL angehängt oder in Form eines Cookies übertragen wird. (Cookies werden Sie im nächsten Kapitel noch ausführlicher kennen lernen.)

Wird die Session eine längere Zeit nicht mehr benutzt (Session-Timeout), so verfällt sie und wird aus dem Speicher entfernt. Beim nächsten Aufruf erhält der Benutzer daraufhin eine neue (reinitialisierte) Session. Alle in der Session abgelegten Werte gehen dabei verloren.

3.4.2 Ein kleines Zahlenspiel

Um die Arbeit mit einer *Session* näher kennen zu lernen, werden Sie ein kleines Ratespiel implementieren, welches verschiedene Werte in der Session speichert. Das folgende Listing zeigt Ihnen die dafür benötigte JSP, die wir anschließend Stück für Stück analysieren werden:

Listing 3.28: Ein Ratespiel (numberGuess.jsp)

```
<%!
   /** Erzeugt eine Zufallszahl zwischen 0..100 */
   private Integer guessNewNumber() {
      int result = new Double(Math.random() * 100).intValue();
      return new Integer(result);
   }

   /** Erhöht den Zähler */
   private Integer inc(Integer counter) {
      int result = counter.intValue() + 1;
      return new Integer(result);
   }
%>

<%
   Integer guess = null;

   // Auslesen von Session-Attributen, ggf. 'null' wenn diese nicht
   // existieren
   Integer number = (Integer)session.getAttribute("number");
   Integer counter = (Integer)session.getAttribute("counter");
```

```
   // Auslesen eines Request-Parameters
   String param = request.getParameter("guess");
   if (param != null) {
      guess = new Integer(param);
   }
%>
<html>
   <body>
      <%-- Codieren eines URLs (siehe 3.4.4) --%>
      <form action='<%=response.encodeURL("numberGuess.jsp")%>'>

         Ich denke mir eine Zahl zwischen 0 und 100. Welche?

         <%
            // Neues Spiel oder läuft es bereits ?!
            if (number == null) {
               number = guessNewNumber();
               counter = new Integer(0);

               // Setzen von Session-Attributen
               session.setAttribute("number", number);
               session.setAttribute("counter", counter);
            } else {
               counter = inc(counter);
               session.setAttribute("counter", counter);
               int result = number.compareTo(guess);

               switch (result) {
                  case -1 : %> Die gesuchte Zahl ist kleiner.   <%
                     break;
                  case  0 : %> Richtig. Versuche <%= counter %> <%
                     // Löschen eines Session-Attributs
                     session.removeAttribute("number");
                     break;
                  case  1 : %> Die gesuchte Zahl ist größer.    <%
                     break;
               }
            }
         %>
         Eingabe: <input type="text" name="guess" />
                  <input type="submit" value="Versuchen" />
      </form>
   </body>
</html>
```

Listing 3.28 enthält fast alles bisher Gelernte und ist damit eine gute Zusammenfassung und zugleich ein Test für Sie.

Hilfsmethoden über JSP-Deklarationen

Im oberen Teil definieren Sie mittels einer JSP-Deklaration zwei Hilfsmethoden für den Umgang mit `Integer`-Objekten:

Listing 3.29: Definition von Hilfsmethoden

```
<%!
   /** Erzeugt eine Zufallszahl zwischen 0..100 */
   private Integer guessNewNumber() {
      int result = new Double(Math.random() * 100).intValue();
      return new Integer(result);
   }

   /** Erhöht den Zähler */
   private Integer inc(Integer counter) {
      int result = counter.intValue() + 1;
      return new Integer(result);
   }
%>
```

Die erste Methode dient der Erzeugung einer neuen Zufallszahl zwischen 0 und 100 und die untere Methode inkrementiert einen übergebenen `Integer` um 1.

Wir verwenden an dieser Stelle `Integer`-Objekte und keine elementaren Typen (`int`, `short` etc.), da die in der Session abgelegten Daten vom Typ `Object` sein müssen.

Zugriff auf Request-Parameter

Den Zahlenwert des aktuellen Versuchs lesen Sie als Request-Parameter aus und speichern ihn anschließend in einem `Integer`-Objekt:

Listing 3.30: Auslesen von Request-Parametern

```
...
   String param = request.getParameter("guess");
   if (param != null) {
      guess = new Integer(param);
   }
...
```

Und außerdem ...

... können Sie Anwendungsbeispiele für JSP-Scriptlets, Entscheidungen und einen Case-Verteiler in Aktion sehen.

3.4.3 Arbeiten mit der Session

Neben Altbekanntem zeigt Ihnen Listing 3.28 auch, wie Sie Objekte in Form von *Attributen* zu einer Session hinzufügen, aus dieser auslesen und schließlich wieder entfernen:

Listing 3.31: Arbeiten mit dem Session-Kontext

```
// Hinzufügen eines Session-Attributs
session.setAttribute("NameDesObjekts", Object object);

// Auslesen eines Session-Attributs
session.getAttribute("NameDesObjekts");

// Entfernen eines Session-Attributs
session.removeAttribute("NameDesObjekts");
```

Auf dieses Weise können Sie Benutzerdaten (z.B. Objekte seines virtuellen Warenkorbes oder seinen Login) von Request zu Request weiterreichen. Wie Sie sehen, werden Objekte in einer Session unter einem frei wählbaren, *symbolischen Namen* (z.B. `number` bzw. `counter`) abgelegt und wieder gefunden.

Listing 3.32: Neue Strukturen

```
...
   // Auslesen von Session-Attributen, ggf. 'null' wenn diese nicht
   // existieren
   Integer number = (Integer)session.getAttribute("number");
   Integer counter = (Integer)session.getAttribute("counter");
...
            if (number == null) {
               ...
               // Setzen von Session-Attributen
               session.setAttribute("number", number);
               session.setAttribute("counter", counter);
            } else {
               ...
               session.setAttribute("counter", counter);
               ...
               switch (result) {

                  case  0 : %> Richtig. Versuche <%= counter %> <%
                     // Löschen eines Session-Attributs
                     session.removeAttribute("number");
                  ...
               }
...
```

Damit gleicht die Session einer `java.util.HashMap`. Dies hat folgende drei Konsequenzen:

1. Der Name eines Objekts muss *eindeutig* sein. Das heißt, dass bei doppelter Verwendung ein und desselben symbolischen Namens das zuerst abgelegte Objekt der Session überschrieben wird und nicht mehr ausgelesen werden kann.
2. Es können nur vollwertige Objekte (Typ `java.lang.Object`) in der Session abgelegt werden. Elementare Typen (`int`, `long`, `double`, ...) werden nicht unterstützt. Deshalb arbeitet Listing 3.28 auch mit `Integer` statt mit dem an sich komfortableren `int`.
3. Alle Objekte, die Sie aus der Session auslesen, werden zunächst als `Object` zurückgegeben und müssen bei Bedarf anschließend *gecastet* werden (siehe auch Listing 3.33)

```
...
   // Casten eines ausgelesenen Session-Attributs zum Integer
   Integer number = (Integer) session.getAttribute("number");
...
```

Listing 3.33: Typecast nach Auslesen aus der Session

Ebenso wie in der Session können Sie auch im oben beschriebenen Request-Objekt (Request-Scope) über die Methoden `setAttribute()`, `getAttribute()` und `removeAttribute()` Objekte ablegen, um beispielsweise Daten bei der Weiterleitung von einer JSP auf eine andere weiterzugeben. Die Objekte existieren dann allerdings nur bis zum Abschluss des Requests und verfallen danach.

3.4.4 Kodieren von URLs

Die Funktionalität des obigen Beispiels stützt sich essenziell auf das Vorhandensein der benutzerspezifischen Session. In dieser wird festgehalten, welche Zahl der Webserver sich für diesen Anwender »gedacht« hat und wie viele Versuche dieser bereits gebraucht hat.

Die einzelnen Session-Objekte des Webservers müssen dabei eindeutig zugeordnet werden. Hierfür erzeugt der Webserver bei der Erzeugung einer neuen Session eine eindeutige ID, über die Benutzer und Session miteinander verknüpft werden. Damit dies gelingt, muss der Browser diese ID natürlich bei jedem Request übermitteln.

In der Regel werden die Session-IDs über Cookies realisiert. Diese Sonderform von HTTP-Headern werden Sie im nächsten Kapitel näher kennen lernen. Zunächst können Sie sich Cookies als frei definierbare HTTP-Header vorstellen. Wenn Sie Abbildung 3.2 genau betrachten, finden Sie dort die folgende Zeile:

Listing 3.34: Beispiel für ein Session-Cookie

```
...
   // Ausgabe des Beispielcookies in Abbildung 3.2
   cookie = JSESSIONID=3DE47CB625463D6CDF970C220B892803
...
```

Dies ist also die eindeutige ID dieser Session. Doch nicht alle Browser unterstützen Cookies, und einige Anwender schalten dieses Feature auch bewusst aus. Um die Session-Verwaltung auch ohne Cookie-Unterstützung zu ermöglichen, können Sie die entsprechende ID auch in Form eines besonderen Request-Parameters übertragen. Dabei unterstützt Sie die Methode `encodeURL()` der vordefinierten Variablen `response`.

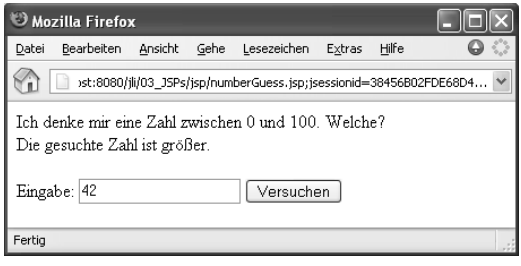
Listing 3.35: Kodieren eines URLs

```
...
   <!-- Sicherstellen der Session-ID, ggf. auch ohne Cookies -->
   <form action='<%= response.encodeURL("numberGuess.jsp") %>'>
...
```

Diese Methode übernimmt als Parameter den zu kodierenden URL und überprüft anschließend, ob die Session-ID über ein Cookie übertragen werden kann oder als Parameter übertragen werden muss.

Im ersten Fall ist die von der Methode `encodeURL()` zurückgegebene Zeichenkette gleich dem ursprünglichen URL. Anderenfalls fügt sie den erforderlichen Parameter hinzu. Um dies zu demonstrieren, wurde in Abbildung 3.7 die Unterstützung für Cookies deaktiviert.

Abb. 3.7: Ein kleines Zahlenratespiel



3.5 JSPs und JavaBeans

Je tiefer Sie in die Möglichkeiten von JSPs einsteigen, desto umfangreicher und komplexer wird auch Ihr Java-Code und desto schwieriger wird es, Darstellung und Logik voneinander zu unterscheiden. Das macht das Warten von JSP aufwändig und fehleranfällig. Hier können *JavaBeans* Abhilfe schaffen.

JavaBeans sind laut Sun konfigurierbare Container, welche Variablen speichern (persistent machen) und darauf basierende Logik kapseln. Je besser die Trennung vollzogen wird – JSPs für die Darstellung von Inhalten und JavaBeans für die Speicherung –, desto besser können Sie beide Bereiche unabhängig voneinander warten. So können Sie Webanwendungen ein vollkommen neues Aussehen geben, ohne befürchten zu müssen, die darin enthaltene Logik zu beeinflussen.

In Kapitel 9 über *Java Data Objects (JDO)* werden Sie lernen, wie Sie den Zustand Ihrer JavaBeans und damit auch den Zustand Ihrer Anwendung dauerhaft z.B. in einer Datenbank speichern.

3.5.1 Grundlagen

Da der Versuch, das JavaBean-Framework umfassend zu beschreiben, ein eigenes Buch füllen würde, wiederholen wir in diesem Absatz nur noch einmal die wichtigsten Merkmale von JavaBeans, wie Sie sie von klassischen Java-Anwendungen her kennen. Zum Selbststudium seien Ihnen an dieser Stelle die Java-Tutorials unter *http://java.sun.com/products/javabeans* empfohlen.

1. JavaBeans müssen einen *parameterlosen Konstruktor* besitzen, über den sie erzeugt werden.
2. JavaBeans sollten die persistenten Daten in Form von so genannten Attributen (engl. *properties*) verwahren. Einige Programmierer bevorzugen statt *Attribut* auch den Begriff *Eigenschaft*.
3. Die Attribute bzw. Eigenschaften bestehen aus einer nicht-öffentlichen (`private`) Instanzvariablen und öffentlichen Zugriffsmethoden (`public`), welche umgangssprachlich auch Getter und Setter genannt werden.
4. Die Zugriffsmethoden haben immer die Form `setXxx()` und `getXxx()`, bzw. `isXxx()` bei booleschen Attributen.

Neben den Gettern und Settern können JavaBeans auch noch weitere Methoden bereitstellen, die in der Regel ebenfalls auf den internen Variablen arbeiten.

3.5.2 Eine JavaBean

Zunächst benötigen Sie eine JavaBean, welche alle innerhalb der JSP benötigten Werte aufnehmen kann. Diese Werte werden als *Attribute* oder *Eigenschaften* der JavaBean (engl. Properties) bezeichnet. Im folgenden Beispiel soll es sich dabei um einen Namen und eine E-Mail-Adresse, z.B. für ein virtuelles Adressbuch, handeln.

Listing 3.36: Eine JavaBean (FormBean.java)

```
package de.akdabas.jli.j2ee.beans;

import java.util.List;
import java.util.LinkedList;

/** Speichert und validiert die Daten eines HTML-Formulars */
public class FormBean {

   /** Der Name */
   private String name;
   public String getName() {
      return name;
   }

   public void setName(String aName) {
      name = aName;
   }

   /** Die E-Mail-Adresse */
   private String eMail;
   public String getEmail() {
      return eMail;
   }

   public void setEmail(String aAddress) {
      eMail = aAddress;
   }

   /** Möglicherweise aufgetretene Eingabefehler */
   private List errors = new LinkedList();
   public List getErrors() {
      return errors;
   }
```

```
   /** Flag, ob alle Angaben korrekt sind */
   boolean valid = false;
   public boolean isValid() {
      return valid;
   }

   /** Validiert die Eingaben */
   public boolean validate() {

      // Zurücksetzen des Flags und der Fehlerliste
      valid = false;
      errors = new LinkedList();

      // Überprüfe den eingegebenen Namen
      if (name == null || name.length() == 0) {
         errors.add("Bitte geben Sie einen Namen ein.");
      }

      // Überprüfe die Syntax der E-Mail-Adresse
      if (eMail == null || eMail.length() == 0) {
         errors.add("Bitte geben Sie die eMail-Adresse ein.");
      }

      if (eMail != null && eMail.indexOf("@") == -1) {
         errors.add("Ungültiges Format der eMail-Adresse.");
      }

      // Wenn keine Fehler aufgetreten sind => valid = true
      if (errors.size() == 0) {
         valid = true;
      }

      return valid;
   }
}
```

Außer den Gettern und Settern für die Eigenschaften `name` und `email`, die die entsprechenden JSP-Attribute speichern, stellt diese JavaBean noch zwei weitere Eigenschaften `valid` und `errors` bereit. Diese werden von der JavaBean in Abhängigkeit von den Werten der anderen Eigenschaften gefüllt. Was dabei intern genau geschieht, bleibt dem Benutzer der Bean verborgen.

Die Methode `validate()` enthält dabei die so genannte *Geschäftslogik* der JavaBean. In dieser werden die Werte der Eigenschaften `name` und `email` überprüft und anhand dieser Werte die beiden anderen Eigenschaften `valid` und `errors` gesetzt.

Übersetzen der JavaBean

Bei der oben beschriebenen JavaBean handelt es sich nicht um ein JSP-Objekt, sondern um ein klassisches Java-Objekt, wie Sie es auch in anderen Applikationen finden. Es muss vor der Ausführung in einer *.java*-Datei gespeichert und vom *javac*-Compiler übersetzt werden.

Hierfür erweitern Sie Ihr Arbeitsverzeichnis zunächst um die folgenden Ordner.

Abb. 3.8: Erweitern des Arbeitsverzeichnisses für JavaBeans

Nachdem Sie die JavaBean ihrem Package entsprechend im Ordner *beans* abgelegt haben, übersetzen Sie diese ins Verzeichnis *classes*. Dazu verwenden wir in diesem Buch das freie OpenSource-Werkzeug *Ant* (*http://ant.apache.org*), welches Sie auch auf der beiliegenden CD finden. Kopieren Sie einfach das ebenfalls beiliegende Ant-Script in Form der Datei *build.xml* in das Arbeitsverzeichnis und führen Sie anschließend dort den Befehl *ant* aus. Sie können die Datei natürlich ebenso mit jedem anderen Java-Compiler übersetzen.

3.5.3 Eingabe der Daten in die JavaBean

Als Nächstes benötigen Sie eine JSP zur Eingabe und Validierung der Daten.

Listing 3.37: JSP zur Eingabe (beanInput.jsp)

```
<%-- Importieren der Java-Klasse 'Iterator' zur Fehler-Ausgabe --%>
<%@ page import="java.util.Iterator" %>

<%-- Binden der JavaBean an den symbolischen Name 'form' --%>
<jsp:useBean id="form"
             class="de.akdabas.jli.j2ee.beans.FormBean"
             scope="session"/>
```

```
<%-- Übernehmen der HTTP-Parameter in die JavaBean --%>
<jsp:setProperty name="form"
                 property="name"
                 value="<%= request.getParameter("name") %>"/>

<jsp:setProperty name="form"
                 property="email"
                 value="<%= request.getParameter("email") %>"/>

<%-- Aufruf der Geschäftslogik, ggf. Weiterleiten zur Ausgabe --%>
<%
   form.validate();
   if (form.isValid()) {
      // Weiterleiten des Request an die Ausgabeseite
      response.sendRedirect("beanOutput.jsp");
   }
%>

<html>
   <body>

      <%-- Kodieren des URLs (siehe 3.4.4) --%>
      <form action='<%= response.encodeURL("beanInput.jsp") %>'>
         Bitte füllen Sie das folgende Formular aus !

         <%-- ggf. Ausgabe von aufgetretenen Fehlern --%>
         <% if (! form.isValid()) { %>
            <ul>
            <%
               Iterator i = form.getErrors().iterator();
               while (i.hasNext()) {
             %>
                <li> <%= i.next() %> </li>
             <% } %>
         <% } %>

         Name : <input type="text" name="name" /> <br/>
         eMail: <input type="text" name="email" /> <br/>

         <input type="reset" value="Reset" />
         <input type="submit" value="Absenden" />
      </form>
   </body>
</html>
```

Verheiraten von JSP und JavaBean

Zunächst einmal müssen Sie die JSP und die JavaBean miteinander bekannt machen. Die einfachste Syntax hierfür ist:

Listing 3.38: Einbinden einer JavaBean in die JSP

```
<!-- Einbinden der JavaBean in die JSP -->
<jsp:useBean id="form"
             class="de.akdabas.jli.j2ee.beans.FormBean"
             scope="session" />
```

Hierdurch wird ein Objekt vom Typ `de.akdabas.jli.j2ee.beans.FormBean` erzeugt und an den Namen `form` gebunden. Dieses Listing ist äquivalent zu:

Listing 3.39: Äquivalentes Einbinden einer JavaBean

```
<% de.akdabas.jli.j2ee.beans.FormBean form =
                 new de.akdabas.jli.j2ee.beans.FormBean(); %>
```

Das obige Listing hat dabei den Vorteil, dass kein direkter Java-Code zum Einsatz kommt und diese Form auch von Java-Laien schnell erlernt werden kann. Auf diese Weise können Java-Programmierer und Web-Designer effizient zusammenarbeiten: Der Java-Entwickler entwickelt und wartet die JavaBeans, während der Web-Designer das Layout verändern kann, ohne den Java-Code zu beeinflussen.

Das Tag `<jsp:useBean>` übernimmt folgende vier Attribute:

- `id`

  Dieses Attribut enthält den symbolischen (Variablen-)Namen der JavaBean, über den sie anschließend referenziert werden kann.

- `class`

  Über dieses Attribut spezifizieren Sie den Namen und den vollständigen Pfad der JavaBean.

- `scope` (optional)

  Über dieses Attribut bestimmen Sie den Gültigkeitsbereich der JavaBean und binden diese an einen der oben genannten Kontexte. Mögliche Werte sind `page` (Standard), `session` und `application`.

- `type` (optional)

  Dieses Attribut verwenden Sie, wenn Sie Ihre JavaBean über ein Interface oder eine von ihr implementierte Basisklasse ansprechen möchten. Äquivalent könnten Sie auch schreiben:

  ```
  <% type name = new class() %>
  ```

Verschiedene Gültigkeitsbereiche einer JavaBean

Die Definition einer Variablen über das Tag `<jsp:useBean>` ist gleichwertig zur Definition einer Variablen via JSP-Scriptlet.

Diese (per Scriptlet) »zu Fuß« erzeugten Variablen können Sie über die Methoden `setAttribute()` der vordefinierten Objekte `request`, `session` und `application` an deren Gültigkeitsbereich knüpfen (siehe Abschnitt 3.1.1). Das Gleiche erreichen Sie auch durch das Setzen des Attributs `scope` in `<jsp:useBean>`.

- `page` (Standardwert)

  Diese Variablen sind, analog zu lokal definierten Variablen, nur innerhalb der aktuellen Seite gültig und werden wie diese verwendet, um Parameter auszuwerten oder Zwischenergebnisse zu speichern.

- `session`

  Variablen mit diesem Scope werden an das `session`-Objekt (siehe Abschnitt 3.1 zu den vordefinierten Variablen) gebunden und sind ebenso lange gültig wie die Sitzung, d.h. bis z.B. der Browser geschlossen oder das serverspezifische Timeout überschritten wird.

- `application`

  Während die vordefinierte Variable `session` benutzerspezifisch ist, ist das `application`-Objekt für alle Sessions dasselbe. Über diesen Scope können Variablen auch benutzerübergreifend verwendet werden. Ein Objekt, welches innerhalb dieses Gültigkeitsbereichs abgelegt wird, kann bis zum nächsten Start des Webservers oder dessen expliziter Löschung verwendet werden.

- `request`

  Diese Variablen werden für die Dauer eines Requests gebunden. Der Unterschied zum Scope `page` (Default) besteht darin, dass Aktionen wie `<jsp:include>` und `<jsp:forward>` als *ein* Request behandelt werden und diese Variablen daher beteiligten (d.h. ursprünglichen wie weitergeleiteten) Seiten zur Verfügung stehen.

Setzen von Variablenwerten

Nachdem die JavaBean erzeugt ist, können Sie auf deren Methoden und Attribute zugreifen. Bisher realisierten Sie diese z.B. durch:

Listing 3.40: Manuelles Setzen von Attributen

```
...
   <%
      // Setzen von Attributen einer JavaBean (klassisch)
      form.setName(request.getParameter("name"));
      form.setEmail(request.getParameter("email"));
   %>
...
```

Auch für das Setzen von JavaBean-Attributen existiert ein entsprechendes Tag:

Listing 3.41: Setzen von Attributen via <jsp:setProperty>

```
...
   <%-- Setzt den Wert des Attributs 'name' --%>
   <jsp:setProperty name="form"
                    property="name"
                    value="<%= request.getParameter("name") %>" />

   <%-- Setzt den Wert des Attributs 'email' --%>
   <jsp:setProperty name="form"
                    property="email"
                    value="<%= request.getParameter("email") %>" />
...
```

Das Tag `<jsp:setProperty>` benötigt drei Parameter, um den Wert einer eingebundenen JavaBean zu setzen:

- `name`

  Der Variablenname der JavaBean. Diese muss zuvor über das Tag `<jsp:useBean>` *oder* ein äquivalentes JSP-Scriptlet erzeugt worden sein.

- `property`

  Der Name des Attributs, *nicht* der Name des Setters! Der übergebene Wert `email` wird vom Tag selbstständig in `setEmail()` umgewandelt.

- `value`

  Der Wert, den das Attribut annehmen soll. Dabei kann es sich z.B. um eine konstante Zeichenkette oder ein per JSP-Ausdruck eingefügten Parameter handeln.

```
<%-- Setzen eines JavaBean-Attributs (Schematisch) --%>
<jsp:setAttribute name="SymbolischerNameDerJSP"
                  property="PropertyNameOhneSet"
                  value="Wert" />
```

Listing 3.42: Schematische Form des <jsp:setAttribute>-Tags

Setzen aller vorhandenen Request-Parameter

Im obigen Beispiel benötigen Sie zum Setzen der vorhandenen Request-Parameter je einen JSP-Ausdruck. Es gibt jedoch auch eine kürzere Schreibweise zum gleichzeitigen Setzen *aller* Parameter: Sind HTML-Parameter und zugehöriges JavaBean-Attribut gleichnamig, so können Sie diese durch folgendes Tag synchronisieren:

```
<%-- Setzen aller mit Request-Parametern gleichnamigen Attribute --%>
<jsp:setProperty name="SymbolischerNameDerJSP" property="*" />
```

Listing 3.43: Setzen aller vorhandenen Request-Parameter

Das Ergebnis

Die restlichen Elemente aus Listing 3.37 sind Ihnen bereits bekannt. Auch die beiden verbleibenden JSP-Scriptlets (Weiterleiten und Fehlerausgabe) lassen sich durch Tags ersetzen, wodurch die JSP frei von Java-Code wird. Diese sind jedoch Kandidaten für eigene Tags in Form von Tag-Bibliotheken, die Sie in Kapitel 5 kennen lernen werden.

Nachdem Name und E-Mail-Adresse der JavaBean gesetzt sind, überprüfen Sie diese über die Methode `isValid()`. Sind alle Werte korrekt, leiten Sie den Request an die Seite *beanOutput.jsp* weiter. Anderenfalls geben Sie die in der JavaBean erzeugten Fehlermeldungen aus und ermöglichen dem Benutzer, diese zu korrigieren,

Abb. 3.9: Eingabe und Validierung von Parametern

3.5.4 Ausgabe von Attributen einer JavaBean

Das bloße Erzeugen von JavaBeans und das Setzen von Attributwerten wären natürlich sinnlos ohne die Möglichkeit zur Ausgabe der Daten. Dies wird ebenfalls durch ein Tag ermöglicht. Nachdem Sie die JavaBean über ein Tag oder Scriptlet erzeugt und eingebunden haben, können Sie auch auf die dort vorhandenen Getter zugreifen und diese beispielsweise über einen JSP-Ausdruck ausgeben:

Listing 3.44: Ausgabe von Attributen via JSP-Scriptlet

```
...
   <%-- Ausgabe der Attribute einer JavaBean (klassisch) --%>
   <%= form.getName() %>            <%-- bzw. --%>
   <%= form.getEmail() %>
...
```

Eleganter realisieren Sie dies allerdings mit folgenden Tags:

Listing 3.45: Ausgabe von Attributen via Tag

```
...
   <%-- Ausgabe von JavaBean-Attributen über Tags --%>
   <jsp:getProperty name="form" property="name" />  <%-- bzw. --%>
   <jsp:getProperty name="form" property="email" />
...
```

Die Attribute haben dabei dieselbe Form wie beim analogen `<jsp:setProperty>`-Tag.

Ausgabe der Daten

Im obigen Beispiel (Listing 3.37) leiten Sie den Request nach einer erfolgreichen Validierung der Eingaben auf die JSP *beanOutput.jsp*:

Listing 3.46: beanOutput.jsp

```
<%-- Binden der JavaBean an den symbolischen Namen 'form' --%>
<jsp:useBean id="form"
             class="de.akdabas.jli.j2ee.beans.FormBean"
             scope="session" />

<html>
   <body>
      Erfolgreich eingegeben !

      <%-- Ausgabe der JavaBean-Werte --%>
      Name : <jsp:getProperty name="form" property="name"/> <br/>
      eMail: <jsp:getProperty name="form" property="email"/>
   </body>
</html>
```

3.5.5 Erweitertes Erzeugen von JavaBeans

Um JavaBeans in JSPs zu erzeugen und an einen Variablennamen zu binden, verwenden Sie das Tag `<jsp:useBean>`. Dieses überprüft zunächst, ob der entsprechende Name im angegebenen Kontext (scope) bereits existiert:

- Existiert bereits ein Objekt mit diesem Namen, wird es an die lokale Variable der JSP gebunden.
- Anderenfalls erzeugt der Webserver durch Aufrufen des parameterlosen Konstruktors ein neues Objekt und legt dieses im entsprechenden Kontext ab.

Die folgenden beiden Code-Fragmente sind dabei vollkommen gleichwertig:

Listing 3.47: Manuelles vs. automatisches Erzeugen und Ablegen

```
...
<%
   de.akdabas.jli.j2ee.beans.FormBean form;

   // Überprüfen, ob bereits eine JavaBean mit diesem Scope
   // existiert
   if (session.getAttribute("form") == null) {

      // Erzeugen eines neuen JavaBean-Objekts
      form = new de.akdabas.jli.j2ee.beans.FormBean();
      session.setAttribute("form", form);
   } else {

      // Binden und Casten des vorhandenen JavaBean-Objekts
      Object o = session.getAttribute("form");
      form = (de.akdabas.jli.j2ee.beans.FormBean) o
   }
%>
...
   <%-- Vollkommen gleichwertig zu obigem Code-Fragment --%>
   <jsp:useBean id="form"
                class="de.akdabas.jli.j2ee.beans.FormBean"
                scope="session" />
...
```


Auf eines müssen Sie jedoch in beiden Fällen achten: Hat die bereits vorhandene Variable einen anderen Typ als die neue (Attribut `class`), so wird eine Typumwandlung (*Cast*) vorgenommen.

Bei zueinander inkompatiblen Typen kommt es dabei zu einer `java.lang.ClassCastException`.

Setzen von Initialisierungsparametern

JavaBeans werden, wie Sie bereits wissen, über den parameterlosen Konstruktor erzeugt. Dies macht es jedoch sehr umständlich, Initialisierungsparameter zu übergeben. Auch hier unterstützt Sie die Tag-Schreibweise mit einer besonderen Form:

Sollten Sie mit den Bezeichnungen »geschlossene Form« oder »Rumpf« eines Tags nicht vertraut sein, finden Sie in Kapitel 10, »eXtensible Markup Language (XML)«, die benötigten Antworten.

Bisher verwendeten Sie das Tag `<jsp:useBean/>` immer in seiner geschlossenen Form, ohne Rumpf. Dieser hat jedoch die nützliche Eigenschaft, dass er nur dann ausgewertet wird, wenn die JavaBean auch tatsächlich erzeugt wird (*if-Zweig* in Listing 3.47). Um also einen Initialisierungsparameter zu setzen, platzieren Sie einfach das entsprechende `<jsp:setAttribute>` -Tag im Rumpf.

Listing 3.48: Setzen eines Initialisierungsparameters

```
<%-- Initialisieren einer JavaBean --%>
<jsp:useBean id="NameDerJavaBean" class="KlasseDerJavaBean">
   <jsp:setProperty id="NameDerJavaBean"
                    property="NameDesProperties"
                    value="InitialwertDesProperties"/>
</jsp:useBean>
```


Dieses Verhalten des Tags `<jsp:useBean>` kann Ihnen natürlich auch beim Debuggen einer JSP helfen. Um zu testen, wann ein Objekt neu initialisiert und wann lediglich neu gebunden wird, fügen Sie einfach eine entsprechende Ausgabe in den Rumpf des Tags ein. Diese wird nur nach dem Erzeugen des Objekts ausgegeben werden.

3.5.6 Vorteile von JavaBeans

Prinzipiell können Sie alles bisher Gelernte auch ohne JavaBeans, allein durch die Verwendung von lokalen Variablen realisieren und diese anschließend durch Bindung an die vordefinierten Variablen »zu Fuß« mit dem benötigten Gültigkeitsbereich versehen. Doch gerade bei größeren Applikationen bietet die Verwendung von JavaBeans entscheidende Vorteile gegenüber der *Selfmade-Methode*:

- *Wiederverwendung von Software-Komponenten*

  Allein dieser Punkt würde die Verwendung von JavaBeans schon rechtfertigen. Durch die Kapselung der Logik zu einem sinnvollen Baustein können verschiedene Komponenten und JSPs auf diesen zugreifen, ohne eine Zeile seines Codes zu duplizieren.

- *Verständlichkeit der JSP*

  Durch die Verwendung von JavaBeans wird es auch Java-Unkundigen ermöglicht, den Inhalt einer JSP zu verstehen und diesen, eine entsprechende Schulung vorausgesetzt, sogar zu verwenden. Ihre JSPs werden übersichtlicher und sind klarer strukturiert.

  So könnte ein eingewiesener Web-Designer etwa die Darstellung des Internet-Auftritts umgestalten, ohne eine Zeile Java-Code schreiben zu müssen, indem er einfach die Ausgabe-Tags an den richtigen Stellen der JSP einfügt.

- *Kapselung*

  Durch die Verlagerung der Geschäftslogik, wie Berechnungen, Umformungen etc., in JavaBeans wird diese von der Darstellungsschicht (den JSPs) getrennt, so dass beide unabhängig voneinander gewartet werden können.

  Jede Schicht hat eine klare Aufgabe:

  – JSPs interagieren mit dem Benutzer, sammeln Eingabewerte und stellen Ergebnisse dar.

  – JavaBeans enthalten die Geschäftslogik. Sie berechnen Ergebnisse und speichern den aktuellen Zustand.

- *Konzentration zusammengehörender Daten*

  Durch JavaBeans werden zusammengehörende Daten, wie Name und E-Mail-Adresse eines Kontaktes, in einem gemeinsamen Container verwaltet. Die JavaBean bildet eine logische Einheit und kann in verschiedenen Kontexten wiederverwendet werden. Insbesondere können Sie Ihre einfachen JavaBeans zu immer komplexeren JavaBeans zusammensetzen.

3.6 Zusammenfassung

Nachdem Sie das vorangegangene Kapitel in die Grundlagen von JSPs und deren Syntax eingeführt hat, sollten Sie nun mit dem Umgang von JSPs vertraut und in der Lage sein, auch komplexere JSPs und JSP-basierte Applikationen zu entwerfen. Das nächste Kapitel zeigt Ihnen eine weitere Technik, auf HTTP-Anfragen zu reagieren und was sich hinter Ihren JSPs wirklich verbirgt, während Sie in Kapitel 5 lernen, wie Sie eigene Tag-Bibliotheken entwerfen, um oft verwendeten Code zu kapseln und wiederverwenden zu können.

3.6.1 Goldene Regeln für die Verwendung von JSPs

Mit Goldenen Regeln ist das so eine Sache: der eine bevorzugt dies, ein anderer schwört auf jenes. Nichtsdestotrotz wollen wir es wagen und Ihnen zum Abschluss *unsere* sechs goldenen Regeln für die Erstellung von JSPs mitgeben:

- Fassen Sie alle `page`-Direktiven im oberen Teil der JSP zusammen. Dies erhöht die Übersichtlichkeit.
- Lassen Sie die JSP-Deklarationen auf die `page`-Direktiven folgen.
- Im Anschluss an die JSP-Deklarationen folgt ein JSP-Scriptlet, in dem Sie alle benötigten Request-Parameter auslesen und Ergebnisse berechnen. Speichern Sie diese in lokalen Variablen oder JavaBeans und verwenden Sie zwischen dem eigentlichen Markup nur JSP-Ausdrücke bzw. Getter der JavaBeans.
- Vermeiden Sie die Vererbung in JavaBeans. Nicht alle Webserver unterstützen dieses Feature und die Anwendung wird hierdurch fehleranfällig.
- Lagern Sie häufig auftauchenden Code in andere JSPs oder besser in JavaBeans aus.
- Geben Sie JavaBeans den Vorzug vor »nackten« Werten in Form von Strings oder Integern. JavaBeans bilden leicht verständliche, logische Einheiten und können wiederverwendet werden. Dies macht den Mehraufwand beim Erstellen von JSP *und* Bean schnell wieder wett.

3.7 Übungen

Folgende Übungen sollen Ihnen helfen, das in diesem Kapitel erworbene Wissen zu festigen.

1. Erstellen Sie eine »JSP zum Debuggen«, in der Sie alle wichtigen Informationen aus Request und Response ausgeben.
2. Verwenden Sie die JSP aus der vorangehenden Übung, um Ihre JSP aus den Übungen zu Kapitel 2 zu debuggen. Überschreiben Sie dabei mittels `<jsp:param>` verschiedene Request-Parameter.
3. Schreiben Sie die ausgelesenen Werte in eine JavaBean und speichern Sie diese in der Benutzer-Session.
4. Leiten Sie den Benutzer nun von Ihrer JSP auf eine andere JSP weiter und geben Sie den Inhalt der JavaBean dort aus. Stellen Sie sicher, dass dies auch ohne Cookie-Unterstützung des Browsers funktioniert und verwenden Sie zur Ausgabe weder JSP-Scriptlets noch JSP-Ausdrücke.
5. Lassen Sie den Browser die resultierende JSP durch Setzen des Dokumenttyps nun als Textdatei interpretieren. Und definieren Sie eine JSP zum Abfangen von Fehlern (z.B. Bean nicht vorhanden etc.).

3.8 Fragen zur Selbstkontrolle

Alle Übungen gemeistert? Dann sollten die folgenden Fragen kein Problem darstellen.

1. Welche vordefinierten Variablen stehen Ihnen innerhalb von JSPs zur Verfügung?
2. Was unterscheidet die `include`-Direktive vom Tag `<jsp:include>`?
3. Wozu dienen die Attribute `errorPage` und `contentType`?
4. Welche Vorteile hat die Verwendung von JavaBeans gegenüber lokalen Variablen und wie kann man eine Bean über mehrere Anfragen (Requests) hinweg verwenden?