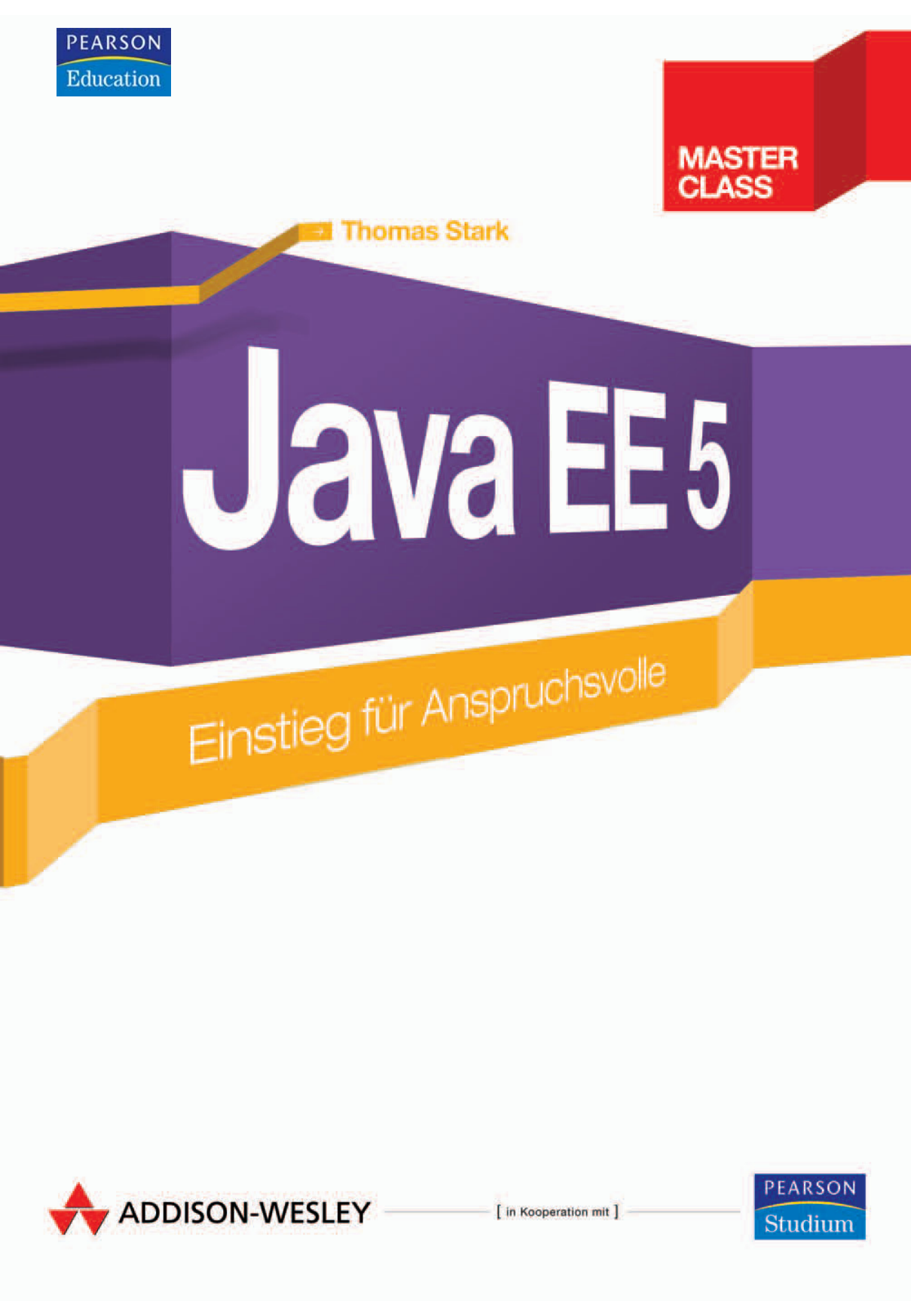

PEARSON
Education
MASTER
CLASS
Thomas Stark
Java EE 5
Einstieg für Anspruchsvolle
ADDISON-WESLEY
[in Kooperation mit]
PEARSON
Studium

Java EE 5

PEARSON Education
ADDISON-WESLEY

MASTER CLASS

Thomas Stark

Java EE 5

Einstieg für Anspruchsvolle

Auf CD: JBoss, Apache Tomcat und Java EE

ADDISON-WESLEY [in Kooperation mit] PEARSON Studium

Bibliografische Information Der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie;
detaillierte bibliografische Daten sind im Internet über *http://dnb.ddb.de* abrufbar.

10 9 8 7 6 5 4 3 2 1

09 08 07

ISBN-13: 978-3-8273-2362-0
ISBN-10: 3-8273-2362-2

ein Imprint der Pearson Education Deutschland GmbH,
Martin-Kollar-Straße 10–12, D-81829 München/Germany

www.pearson-studium.de
Lektorat: Brigitte Bauer-Schiewek, bbauer@pearson.de
Korrektorat: Petra Kienle
Einbandgestaltung: Marco Lindenbeck, webwo GmbH (mlindenbeck@webwo.de)
Grafische Gestaltung der Abbildungen: Ulrike Altmann (Berlin)
Fachlektorat: Ralph Steyer
Herstellung: Martha Kürzl-Harrison, mkuerzl@pearson.de
Satz: reemers publishing services gmbh, Krefeld (www.reemers.de)
Druck und Verarbeitung: Kösel, Krugzell (www.KoeselBuch.de)

Printed in Germany

Inhaltsverzeichnis

Vorwort

Sie möchten Java mit dem Internet verknüpfen und global verfügbare Dienste anbieten? Sie denken, dass Intranetanwendungen und Webportalen die Zukunft gehört? Dann kann Ihnen dieses Buch sicher weiterhelfen.

Mit der Java Enterprise Edition (Java EE) verlassen Sie den klassischen Java-Bereich und wenden sich serverseitigen Anwendungen zu. Dabei treffen Sie auf neue Herausforderungen, aber auch auf neue Möglichkeiten: Denn während Sie sich vorher auf die Belange eines einzelnen Benutzers konzentrieren konnten, müssen Sie nun auf die Wünsche vieler parallel arbeitender Anwender Rücksicht nehmen. Dafür ermöglichen Sie es diesen Anwendern auch, unabhängig von Standort, Betriebssystem oder Client zusammenzuarbeiten und Daten miteinander auszutauschen.

Dieses Buch soll Sie beim Perspektivenwechsel vom »Desktop Java« hin zu »Java Server Side Solutions« unterstützen und Sie in die Lage versetzen, Ihr eigenes Chat-Portal, Ihre onlinegestützte Nutzerverwaltung, Ihr Reportsystem oder was auch immer Sie sich unter einer verteilten Anwendung vorstellen umzusetzen.

Das Schreiben dieses Buchs hat mir eine Menge Freude bereitet und natürlich bin ich an Ihrer Meinung interessiert. Ich freue mich auf Ihre Anregungen und Kritiken unter *masterclass@akdabas.de* und wünsche Ihnen nun viel Spaß beim Lesen.

Danksagung

Auch dieses Buch verdankt seine Entstehung neben dem Autor vielen anderen Menschen, denen ich an dieser Stelle für ihre Unterstützung danken möchte.

An allererster Stelle natürlich meiner süßen Freundin Ulli: Ich liebe dich! Du machst mein Leben lebenswert; du gibst mir Kraft, Mut und Lebenssinn; du bist das Beste, was mir passieren konnte, und dir widme ich dieses Buch.

Meiner Familie und insbesondere Tini und Matze für eure Unterstützung – ich habe auch diesmal nicht vergessen, dass ihr namentlich genannt werden wollt. Ihr seid einfach meine Lieblingsgeschwister :-)

Tina und Philipp: danke, dass ihr euch meine unfertigen Kapitel zu Gemüte geführt und auch sonst immer ein offenes Ohr habt.

Ulli, dafür dass du mich während der Schreiberei an anderen Fronten vertrittst.

Dirk und Mirko, für eure Unterstützung, euren Rat und euer Vertrauen in Java und »angrenzende« Technologien.

Dem Verlag Addison-Wesley und meiner Lektorin Brigitte Bauer-Schiewek für die gute Betreuung und die Möglichkeit, dieses Buch zu schreiben.

Und meinen Freunden: Ich bin glücklich, mit jedem von euch ein Stück meines Lebens teilen zu dürfen.

Thomas Stark

1

Einleitung

Java Enterprise Edition 5 – Java EE 5 – was ist das eigentlich? Nun, in einem Satz könnte man sagen: »Java EE ist das am weitesten verbreitete Betriebssystem für Web- und Business-Anwendungen.«

Java ist eine einfach zu erlernende und auf fast allen Systemen verfügbare Programmiersprache und wie geschaffen für eine Welt, in der webbasierte Portale den Zugriff auf Serverapplikationen gestatten und Informationen global verfügbar sein müssen.

Java EE besteht dabei im Gegensatz zur Java Standard Edition (Java SE) nicht aus einem einzelnen Programmpaket, sondern umfasst vielmehr eine Fülle von Spezifikationen und Schnittstellen, die Neueinsteiger zunächst oft verwirren. Dieses Buch versucht nun, eine Brücke zwischen beiden Editionen zu schlagen und Ihnen einen Überblick über die Java-EE-Technologien zu geben, mit denen Sie beim serverseitigen Einsatz von Java am häufigsten konfrontiert werden.

Dieses Kapitel wird Sie mit drei wichtigen Grundkonzepten der serverseitigen Java-Programmierung vertraut machen und Ihnen den Bezug und die Installation der für dieses Buch immer wieder benötigten Softwarepakete zeigen. Außerdem erhalten Sie einen Überblick über die in den folgenden Kapiteln behandelten Technologien und deren Beziehungen untereinander. Dieses erste einführende Kapitel soll Ihnen damit als Leitfaden dienen, der sich anschließend durch das ganze Buch zieht.

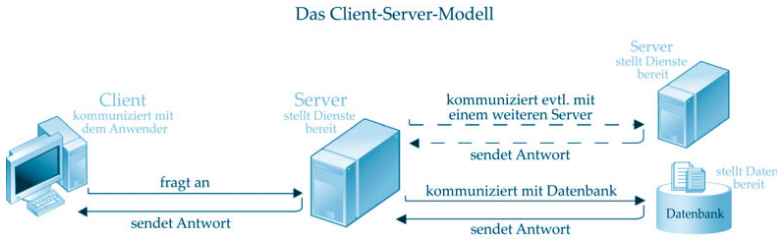
1.1 Zentrale Konzepte

Obwohl die Java Enterprise Edition inzwischen nahezu jeden Bereich der Programmierung erfasst hat und stetig erweitert wird, gibt es drei allgemeine Konzepte, die Ihnen immer wiederbegegnen werden. Diese Konzepte sind dabei nicht unbedingt neu und auch nicht auf die Programmiersprache Java beschränkt. Stattdessen haben sie sich im Laufe der Jahre etabliert und sich als praktikabel und effizient herausgestellt.

1.1.1 Das Client-Server-Modell

Wurde die Java Standard Edition für die Entwicklung und den Betrieb von Java-Programmen auf Einzelplatzrechnern entwickelt, so ist die Enterprise Edition vor allem für den Einsatz auf zentralen Servern gedacht, wo sie verschiedene Dienste für Clients erbringt, die über Protokolle wie TCP/IP oder HTTP kommunizieren.

Abbildung 1.1
Aufbau eines klassischen Client-Server-Modells



Dies bringt einige Veränderungen hinsichtlich der Programmiertechniken mit sich. Denn während Ihnen in der klassischen Java-Programmierung viele Optionen der Interaktion mit dem Benutzer und dessen Umgebung zur Verfügung stehen, werden Sie im Enterprise-Bereich durch die begrenzten Möglichkeiten des jeweiligen Kommunikationsmodells (z.B. HTTP) beschränkt. Andererseits ist die gesamte Anwendung nun nicht mehr auf eine einzelne Java Virtual Machine (VM) beschränkt, sondern sie kann sich bei Bedarf auch auf mehrere Rechner verteilen.

Hierdurch lassen sich abgeschlossene Aufgabenbereiche, wie etwa die Verwaltung und Speicherung von Daten in einer Datenbank, zentral implementieren, während anschließend mit weiteren abgeschlossenen Teilanwendungen darauf zugegriffen werden kann. Auf diesen service-orientierten Ansatz werden Sie noch häufiger stoßen.

Info

In verteilten Anwendungen werden Teilaufgaben häufig als zentrale Dienste (engl. Service) realisiert.

1.1.2 Mehrschichtanwendungen

Auch wenn Ihre gesamte Anwendung in Zukunft von einem einzelnen Server erbracht werden soll, ist es sinnvoll, sie in verschiedene Schichten einzuteilen, die anschließend klar getrennte Aufgabenbereiche übernehmen.

Diese Architektur wird als *Mehrschichtenanwendung* (engl. Multi-Tier Application) bezeichnet und ermöglicht es einzelne Aspekte der Anwendung weiterzuentwickeln bzw. an neue Umgebungen anzupassen, ohne unschöne Nebeneffekte auf die übrige Applikation befürchten zu müssen. Schließlich möchten Sie nicht die gesamte Anwendung neu programmieren, um lediglich eine andere Datenbank zu verwenden.

Tipp

Mehrschichtanwendungen erleichtern den Austausch einzelner Implementierungsdetails.

Am häufigsten trifft man auf die klassische Drei-Schichten-Anwendungen bestehend aus Darstellung, Anwendungslogik und Datenhaltung.

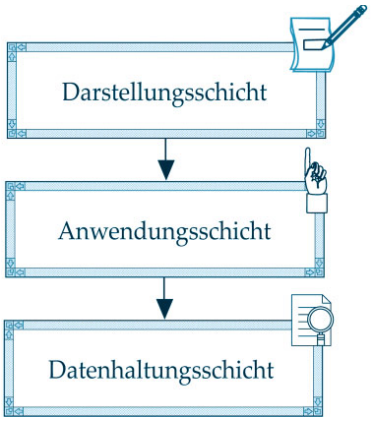


Abbildung 1.2
Klassisches Drei-Schichten-Modell

Die unterste und damit am weitesten vom Anwender abgeschottete Schicht ist die Datenhaltung. Sie ist für die permanente Speicherung der Daten verantwortlich und greift dabei beispielsweise auf Tabellen einer relationalen Datenbank zu. Den darüber liegenden Schichten stellt sie diese Daten in Form von Java-Objekten zur Verfügung, so dass diese nicht erkennen können, ob die Ablage im Augenblick über eine Datenbank erfolgt oder auf einer Datei basiert.

Info

Das Verbergen der tatsächlichen Implementierung hinter einer Schicht, die den darauf aufsetzenden Schichten lediglich eine Schnittstelle zur Verfügung stellt, bezeichnet man auch als Kapselung.

Die auf der Datenhaltungsschicht aufsetzende Anwendungsschicht enthält die Geschäftslogik und bildet damit den Kern des Programms.

Der Anwender arbeitet schließlich mit einer für seine Umgebung optimierten Darstellungsschicht, die die Daten der Anwendungsschicht grafisch aufbereitet. Dabei kann es sich im Endeffekt von der Swing-Oberfläche bis zum Webportal um nahezu alles handeln. Die Trennung von Darstellung und Anwendung ermöglicht es Ihnen sogar, verschiedene Oberflächen für unterschiedliche Umgebungen parallel anzubieten.

1.1.3 Ein API und seine Implementierung

Ein in fast allen Java-EE-Bereichen immer wiederkehrendes Konzept ist die Trennung von API und dessen Implementierung. Dabei stützt man sich bei der Programmierung auf eine Menge von abstrakten Klassen und Interfaces, die dann im Betrieb durch eine Service-Implementierung ausgefüllt werden.

Abbildung 1.3
API und Implementierung

Diese Architektur kennen Sie vielleicht von der Java Database Connectivity (JDBC), bei der der endgültige Datenbanktreiber erst zur Laufzeit bestimmt werden kann. Der große Vorteil dieses Konzepts ist die hierdurch erreichte Modularität Ihrer Anwendungen: Auf diese Weise können Sie Ihre Datenbank über eine standardisierte Verbindung (*Connection*) abfragen, ohne die spezifischen Eigenarten dieser kennen zu müssen.

Info

Der Einsatz allgemeiner APIs erleichtert den späteren Austausch der verwendeten Implementierung.

1.2 Aufbau des Buchs

Die gesamte Java-EE-Spezifikation umfasst eine solche Fülle an Möglichkeiten, dass eine umfassende Beschreibung leicht mehrere Bücher füllen würde. Es gibt allerdings auch kaum einen Programmierer, der sich mit allen Technologien gleichermaßen beschäftigen möchte.

Dieses Buch wird sich deshalb am oben beschriebenen Drei-Schichten-Modell orientieren, Ihnen die wichtigsten Technologien der jeweiligen Schicht vorstellen und Sie in die Lage versetzen, diese erfolgreich einzusetzen.

Dieses Buch ist dabei nicht als umfassende Referenz, sondern als Lehrbuch konzipiert, welches Sie in die verschiedenen Technologien einführen und Ihnen die grundlegenden Ideen dahinter vermitteln möchte. Obwohl die einzelnen Kapitel logisch aufeinander aufbauen, bilden sie dennoch in sich geschlossene Einheiten, die auch einzeln durchgearbeitet werden können, etwa wenn Sie bereits über ein entsprechendes Grundwissen in anderen Bereichen verfügen oder sich gezielt in ein bestimmtes Gebiet einarbeiten möchten.

Die folgenden Absätze stellen Ihnen die verschiedenen, in diesem Buch behandelten Technologien kurz vor. Abbildung 1.4 dient dabei als eine Art Landkarte, die Ihnen sicher die Orientierung erleichtert, während Sie auf neuen Pfaden wandeln. Hier können Sie ablesen, wie die einzelnen Technologien aufeinander aufbauen.

1.2.1 Die Darstellungsschicht

Die Konzepte der Darstellungsschicht sind am leichtesten zu erlernen, da sie auch ohne Logik und Datenhaltung sofort zu anschaulichen Resultaten führen. Die Darstellungsschicht bildet den Einstiegspunkt in eine Anwendung und kaum ein System kommt ohne sie aus. Sie ist die Schnittstelle zum Anwender, nimmt dessen Daten auf und übermittelt ihm anschließend die grafisch aufbereitete Antwort des Systems. Als Grundlage für den Client dienen dabei häufig Webbrowser wie Microsofts *Internet Explorer (IE)* oder *Firefox*. Diese sind inzwischen auf (fast) jeder Plattform verfügbar und machen die so genannten *webbasierten Applikationen* so populär.

Die Java Enterprise Edition enthält gleich eine ganze Reihe von Technologien, die Ihnen die Entwicklung webbasierter Anwendungen erleichtern.

Info

Als webbasierte Applikationen bezeichnet man Anwendungen, bei denen Server und Client über das Inter- oder Intranet miteinander kommunizieren. Die Kommunikationsbasis bildet hierbei häufig das im Internet verwendete *Hypertext Transfer Protocol (HTTP)*.

Kapitel 2 – JavaServer Pages (JSP)

Dieses Kapitel zeigt Ihnen, wie Sie statische HTML-Seiten mit Java-Anweisungen kombinieren, um dynamische Dokumente zu erzeugen, deren Aussehen und Inhalt erst zum Zeitpunkt der Anfrage bestimmt werden.

Kapitel 3 – Servlets

Auch Servlets gestatten es Ihnen, dynamische Dokumente zu erzeugen, womit JSPs und Servlets in gewisser Konkurrenz zueinander stehen. In diesem Kapitel lernen Sie, wo die Stärken und Schwächen der jeweiligen Technologie liegen und wie Sie beide Ansätze miteinander kombinieren können.

Kapitel 4 – Tag-Bibliotheken (Taglibs)

Tag-Bibliotheken versetzen Sie in die Lage, den beschränkten Vorrat an HTML-Elementen um eigene Tags mit spezifischer Funktionalität zu ergänzen. Sie runden damit die Basistechnologien für die Erzeugung von dynamischen Webseiten mittels Java ab und bilden zusammen mit JSPs und Servlets eine echte Alternative zu CGI und PHP.

Kapitel 5 – Struts

In diesem Kapitel wagen wir einen Blick über den Tellerrand hinaus und beschäftigen uns mit dem auf JSPs, Servlets und Tag-Bibliotheken aufbauenden OpenSource-Framework *Struts*. Dieses erleichtert Ihnen die Entwicklung leistungsfähiger und flexibler Webanwendungen, indem es vorgefertigte Funktionalitäten bereitstellt und Sie von lästigen Standardaufgaben befreit.

Info

Application Frameworks bilden einen Rahmen für die Anwendung. Genau wie Ihnen ein API vorgefertigte Methoden zur Verfügung stellt, unterstützen Application Frameworks Sie, indem sie Ihnen die Implementierung von Standardaufgaben abnehmen und es Ihnen so ermöglichen, sich auf das Wichtigste, die Geschäftlogik, zu konzentrieren.

Kapitel 6 – JavaServer Faces (JSF)

Nachdem das Struts-Framework veröffentlicht worden war, setzte ein Siegeszug der webbasierten Anwendungen ein. Die in Struts etablierten Konzepte erwiesen sich dabei als so leistungsstark, dass Sun viele Ideen aufnahm und in Form der *JavaServer Faces (JSF)* etablierte. Aufbauend auf den in Kapitel 5 vermittelten Technologien, führt Sie dieses Kapitel in Suns Pendant JSF ein und zeigt Ihnen, welche Gemeinsamkeiten und Unterschiede zwischen JavaServer Faces und Struts bestehen.

Kapitel 12 – XSL, XPath und FO

Basierend auf dem in Kapitel 11 erworbenen Wissen über die Verarbeitung von XML-Dokumenten, werden Sie in diesem Kapitel drei wichtige Technologien kennen lernen, die Ihnen ebenfalls die Erstellung dynamischer Dokumente von der Webseite bis zu komplexen Formaten wie PDF gestatten.

1.2.2 Geschäftslogik und Kommunikation

Die Logik- und Kommunikationsschicht bleibt dem Anwender in der Regel verborgen, doch für Sie als Programmierer ist sie häufig die wichtigste aller Schichten. Die Geschäftslogik bildet schließlich den entscheidenden Faktor, der Ihre Applikation von allen anderen unterscheidet und die internen Berechnungen sowie die eigentlichen Geschäftsgeheimnisse enthält.

So könnten Sie beispielsweise ohne Problem ein Webportal mit dem Look&Feel von *Google* erzeugen und auch die Speicherung von Daten in Datenbanken ist heute kein großes Geheimnis mehr, doch erst das Wissen um die Erstellung guter Indices würde es Ihnen ermöglichen, einem der 100 größten Unternehmen der Welt Konkurrenz zu machen. Mit anderen Worten, Sie können ohne Probleme etwas bauen, das so aussieht wie

Google, doch ohne die dahinter stehende Geschäftslogik können Sie einfach nicht das Gleiche leisten.

Info

Google ist eine sehr populäre Suchmaschine, die Ihnen das gezielte Suchen nach Dokumenten mit bestimmten Inhalten im Internet ermöglicht.

Kapitel 7 – Java Naming and Directory Interface (JNDI)

Nachdem Sie die ersten Kapitel mit der Erstellung von Webportalen vertraut gemacht haben, widmet sich Kapitel 7 dem Java Naming and Directory Interface (JNDI). Diese Technologie ist *Das Fräulein vom Amt* für verteilte Java-Anwendungen und ermöglicht den Zugriff auf Namens- und Verzeichnisdienste, über die die verschiedenen Dienste zusammengeführt werden.

Info

In einem Namens- und Verzeichnisdienst können Applikationen sich und ihre *Dienste* registrieren sowie Objekte ablegen, um diese anderen Anwendungen verfügbar zu machen.

Kapitel 8 – Enterprise JavaBeans (EJB)

An diese Komponente denken Java-Entwickler bei der Nennung von Java EE häufig zuerst und in der Tat bildet sie einen Grundpfeiler für das gesamte Framework. *Enterprise JavaBeans (EJB)* sind dazu gedacht, die Anwendungslogik aufzunehmen und anderen Programmteilen in Form von Containern zur Verfügung zu stellen.

Kapitel 8 stellt die einzelnen Komponenten und ihre Funktionen vor und führt Sie anhand einfacher Beispiele in die EJB-Architektur ein.

Kapitel 9 – Java Message Service (JMS)

Wenn JNDI das »Fräulein vom Amt ist«, dann ist der Java Message Service (JMS) der Postbote, da er die asynchrone Kommunikation zwischen verschiedenen Programmen ermöglicht und nicht allein auf Java beschränkt ist.

Kapitel 9 befasst sich mit dieser Technologie und zeigt, wie Ihre Programme Daten und Objekte austauschen können.

Kapitel 13 – Webservices

Hinter Webservices verbirgt sich eine der jüngsten und gleichzeitig aktivsten Java-EE-Spezifikationen. Sie ermöglicht es Systemen unterschiedlichster Plattformen, auf einfache Art und Weise, Dienste und Daten miteinander zu teilen und stellt somit einen Höhepunkt plattformübergreifender Kommunikation dar.

1.2.3 Datenhaltung und Datenrepräsentation

Daten darstellen und verarbeiten zu können, ist gut und schön. Doch ohne die Möglichkeit, diese auch dauerhaft zu speichern, wären viele Anwendungen nutzlos. Vielleicht haben Sie bei der Arbeit mit Java Applets Ihre Daten in Dateien gespeichert oder über die Java Database Connectivity (JDBC) bereits auf eine Datenbank zugegriffen. Dieses Buch stellt Ihnen in den Kapiteln 10 und 11 nun zwei weitere Möglichkeiten vor.

Kapitel 10 – Java Persistenz API

Die Abbildung von Objekten auf relationale Datenbanken und zurück – das so genannte O/R-Mapping - gehört zu den aufwändigsten Fließbandaufgaben im Leben eines Java-Programmierers. Denn obwohl es sich dabei um Standardverfahren wie das Lesen und Schreiben von Werten handelt, werden diese häufig noch von Hand programmiert. Kapitel 10 führt Sie in das vollkommen neu geschaffene Java Persistenz API ein, das Sie – wenn Sie es möchten – von der Last, SQL-Statements schreiben zu müssen, befreit.

Info

Als Persistenz bezeichnet man in der Informatik die Fähigkeit, Objekte in nichtflüchtigen Speichern wie Datenbanken oder Dateien abzulegen. Eine Persistenz-Schicht oder ein Persistenz-Framework übernimmt dann die Aufgabe der Speicherung und Reinstanziierung dieser Objekte.

Kapitel 11 – eXtensible Markup Language (XML)

Neben der Definition von abstrakten Dokumenten, die anschließend in verschiedene Formate transformiert werden können, eignet sich XML auch zur Ablage und zum Austausch von Daten. Kapitel 11 zeigt Ihnen, wie Sie XML erfolgreich erzeugen und verarbeiten können.

1.2.4 ... und ist da sonst gar nichts mehr?

Natürlich beinhaltet die Java-EE-Spezifikation weit mehr als die hier vorgestellten Technologien, wobei diese nach Meinung des Autors zu den am häufigsten eingesetzten gehören. Dieses Buch ist nicht als Kompendium, sondern vielmehr als Lehrbuch gedacht, das dem Leser einen Überblick über die wichtigsten Techniken und die Ideen dahinter geben soll.

Die in diesem Buch vermittelten Inhalte stellen eine breite Basis dar, von der aus Sie weitere Streifzüge durch das Java-EE-Framework unternehmen können.

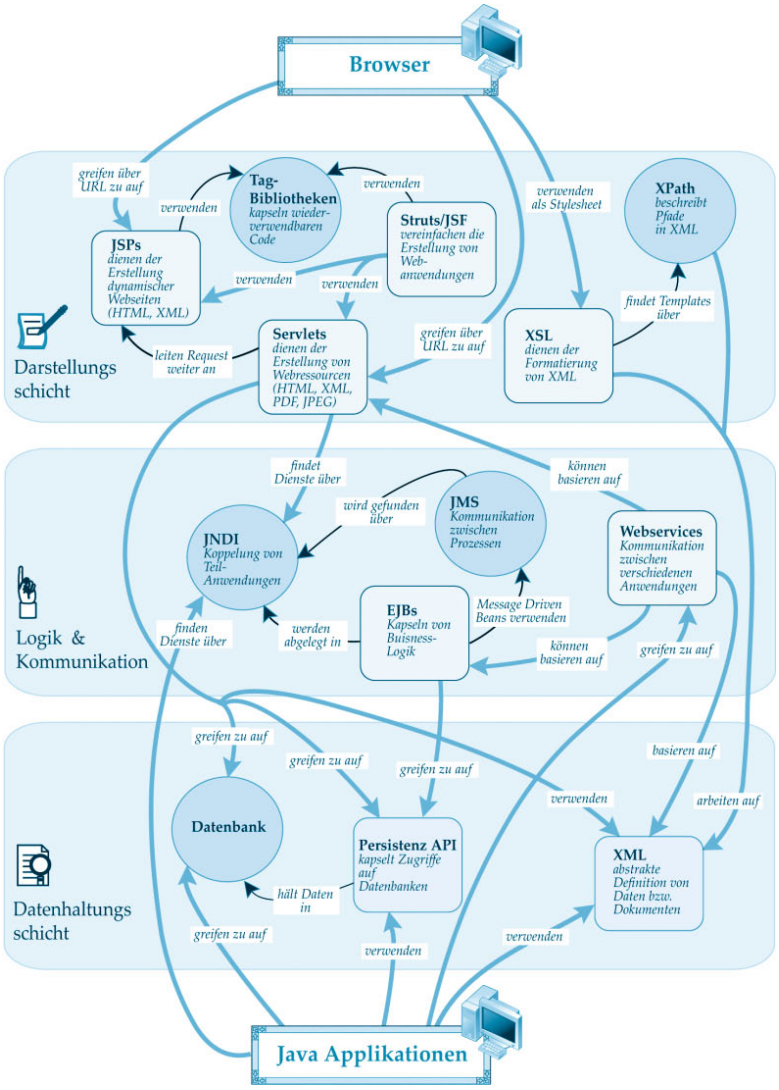


Abbildung 1.4
Verknüpfungen zwischen den vorgestellten Java-EE-Technologien

Es gibt wie überall auch in der Informatik kein Allheilmittel, mit dem sich alle Probleme lösen lassen. Vielmehr erreichen Sie die besten Resultate erst durch die Kombination verschiedener Ansätze.

Wir möchten Sie mitnehmen auf eine interessante Entdeckungsreise zu weiterführenden Java-Technologien und Ihnen durch die Vorstellung der zugrunde liegenden Konzepte die Vertiefung in die eine oder andere Richtung ermöglichen.

1.3 An wen richtet sich dieses Buch?

Dieses Buch richtet sich an ambitionierte Java-Programmierer, die zum Beispiel in Form von Applets oder kleinen Programmen bereits erste Erfahrungen mit der Programmiersprache Java gesammelt haben und diese nun für mittlere und große Anwendungen einsetzen möchten.

Info

Einige Beispiele greifen der Anschaulichkeit halber auf eine Datenbank zu. Wir verwenden diese, da die JDBC mittlerweile in nahezu jedem guten Java-Buch erklärt wird und quasi zum Standard gehört. Die Listings sind jedoch immer so ausgelegt, dass Sie deren Funktion auch ohne JDBC-Kenntnisse leicht nachvollziehen können.

Sie sollten mit den Elementen und der Syntax von Java-Programmen vertraut sein, die Grundlagen objektorientierter Programmierung verstanden haben und grundlegende HTML-Kenntnisse mitbringen. Keine Angst, Sie müssen nicht jeden Kniff kennen oder wissen, wie Sie eine komplexe Anwendung mit Swing oder AWT erzeugen, aber Sie sollten Schleifen implementieren, Ausgaben erzeugen und Variablen deklarieren können. Damit steht Ihrer erfolgreichen Java-EE-Karriere nichts mehr im Wege.

1.3.1 Konventionen

Um das Lesen dieses Buchs und der darin enthaltenen Listings so übersichtlich wie möglich zu gestalten, existieren eine Reihe von Konventionen, die Ihnen dieser Absatz vorstellt. Klassennamen und Bezeichnungen, die sich auf Listings beziehen, werden auch im Fließtext mit `Courier New` dargestellt. Methodennamen werden zusätzlich mit einem Klammerpaar versehen: `methode()`.

Listing 1.1 Konventionen innerhalb von Listings

```
/**
 * Innerhalb von Listings sind Kommentare kursiv
 */
public class Masterclass extends Java {

   // Reservierte Schlüsselworte in Java werden Fett gedruckt ...
   private int variable;

   // ... und wichtige Anweisungen farbig hervorgehoben
   private String important;

}
```

Achtung

In diesen hervorgehobenen Kästen finden Sie weiterführende Informationen, Verweise auf externe Ressourcen und Zusammenfassungen. Diese Informationen sollen Ihr Augenmerk beim ersten Lesen eines Absatzes nicht ablenken, Ihnen jedoch anschließend Anregungen geben und das schnelle Rekapitulieren eines Absatzes ermöglichen.
Innerhalb der Achtungskästen werde ich versuchen, Sie auf typische Stolpersteine hinzuweisen. Diese Informationen sollten Sie auf jeden Fall beachten.

1.3.2 Disclaimer

Ich werde Sie hin und wieder auf externe Quellen, beispielsweise das Internet, hinweisen, mit denen Sie Ihr Wissen erweitern und vertiefen können. Nun ist der Verweis von einer statischen Quelle wie diesem Buch auf ein sehr dynamisches Medium wie dem Internet immer auch mit gewissen Risiken verbunden. Somit versichere ich Ihnen, dass alle hier verlinkten Quellen zum Zeitpunkt des Schreibens existierten und den beschriebenen Inhalt zur Verfügung stellten. Da sich deren Inhalte jedoch in Zukunft verändern können, übernehme ich keinerlei Verantwortung für diese.

Und noch ein Punkt soll kurz erwähnt werden (um auch in Zukunft ruhig schlafen zu können): Dies ist kein Lehrbuch für Juristen, sondern eine praktische Einführung für Entwickler und da ich den Einsatz von *Open-Source-Technologien* stark befürworte und an einigen Stellen verschiedene OpenSource-Lizenzen beschreibe und vergleiche, möchte ich darauf hinweisen, dass diese Ausführungen und Vergleiche keinen Anspruch auf Richtigkeit oder Vollständigkeit erheben, sondern lediglich den Dschungel aus Abkürzungen und Begriffen etwas lichten sollen. In jedem Fall gilt die der jeweiligen Software beiliegende Lizenz und wie heißt es doch so schön: Fehler und Irrtümer sind nicht ausgeschlossen.

Tipp

Wenn Sie sich für die verschiedenen *OpenSource-Lizenzen* und ihre Bestimmungen interessieren, empfehle ich Ihnen die Webseite *http://www.opensource.org/licenses/* der Open Source Initiative (OSI).

1.4 Bezug und Installation der Software

Bevor Sie sich jedoch ins Programmiervergnügen stürzen können, benötigen Sie neben der Java Enterprise Edition eine Reihe von zusätzlicher Software, die Sie auch auf der beiliegenden CD finden. Um die vorgestellten Beispiele selbst erfolgreich anwenden zu können, empfehlen wir Ihnen die folgenden Software-Pakete.

1.4.1 Java Platform Enterprise Edition 5

Dieses Paket bildet natürlich den Grundstein für (fast) alles. Sie können dieses Produkt von Suns Homepage *http://java.sun.com/javaee/* für viele Plattformen herunterladen und in der Regel problemlos installieren.

1.4.2 Die Wahl der »richtigen« Entwicklungsumgebung

Obwohl Sie zur Erstellung von Java-Programmen eigentlich nichts außer einem Editor benötigen, lassen sich viele Programmierer von einer integrierten Entwicklungsumgebung (engl. *Integrated Development Environment, IDE*) unterstützen. Die Empfehlung einer solchen kommt für einen Autor dem Lauf über ein Minenfeld gleich, denn die Wahl der »richtigen« IDE rangiert auf der Rangliste von Programmierern geführter *Flame Wars* gleich hinter »dem einzig richtigen Betriebssystem« oder »der einzig richtigen Programmiersprache«.

Info

Als *Flame War* (engl. to flame, aufflammen) bezeichnet man eine kontroverse Diskussion, bei der die Teilnehmer ihre Meinungen und Einstellungen zu bestimmten Dingen oft auf eine unsachliche oder beleidigende Weise als gegebene »Tatsachen« darstellen. Ort für diese sind häufig einschlägige *Chatforen* und *Newsgroups*. Inzwischen gibt es sogar Newsgroups, die sich ausschließlich dieser »Kommunikationsform« widmen. Wer also Lust auf einen sinnlosen Streit über Computer und andere Dinge hat, wird unter *de.alt.flame* fündig.

Meine Devise zu diesem Thema lautet: »Jedem das Seine«. Und wenn Sie bereits Ihre ultimative Programmierumgebung gefunden haben, dann meinen allerherzlichsten Glückwunsch dazu. Sollte hingegen der unwahrscheinliche Fall eintreten, dass Sie noch keinen Favoriten haben, kommen hier zwei Tipps des Autors.

Eclipse

Diese freie IDE ist aus einem IBM-Projekt hervorgegangen und kann kostenlos von der Webseite *http://www.eclipse.org* heruntergeladen werden. Eclipse kann durch verschiedene Plug-ins erweitert werden und lässt sich für nahezu jede Aufgabe anpassen. Die leider noch spärliche Dokumentation, die vielen Konfigurationseinstellungen und einige unsauber programmierte Plug-ins verlangen dem Programmierer allerdings hin und wieder etwas Geduld und Fleiß beim Googeln ab. Auf der anderen Seite bekommen Sie mit Eclipse einen Allrounder für fast alle Lebenslagen.

IntelliJ IDEA

Aus dem Lager der kommerziellen Anbieter kommt *JetBrains IntelliJ IDEA*. Eine intuitiv bedienbare Java-IDE, die sich vor allem durch geringen Ressourcenbedarf und Unterstützung für viele Anwendungsszenarien auszeichnet.

IntelliJ IDEA kann von der Homepage des Unternehmens *http://www.jetbrains.com* heruntergeladen werden und eignet sich vor allem für Unternehmen, die auf eine hohe Produktivität ihrer Mitarbeiter angewiesen sind.

Eine (Test-)Version beider IDEs für Linux und Windows finden Sie auf der beiliegenden CD.

1.4.3 Das Kompilierungstool Apache Ant

Ein weiteres interessantes und empfehlenswertes Projekt ist *Apaches Ant* (*http://ant.apache.org*). Hinter Ant (engl. Ameise) verbirgt sich ein nahezu allmächtiges Übersetzungswerkzeug, welches Sie zum komfortablen Kompilieren von Java-Klassen verwenden können. So finden Sie zu jedem Beispiel der beiliegenden CD ein vorkonfiguriertes Ant-Skript (*build.xml*), welches die korrekte Übersetzung der Java-Dateien unabhängig vor der gewählten IDE garantiert. Nach erfolgreicher Installation des Programms genügt ein Wechsel in das Beispielverzeichnis mit dem anschließenden Aufruf von

```
ant
```

Listing 1.2
Aufruf von Apache Ant

Info

Durch den Aufruf `ant` im jeweiligen Projektverzeichnis können Sie die Beispiele der CD auf einfache Weise kompilieren.

1.4.4 Apache Tomcat Server

Auch bei dem in den ersten Kapiteln verwendeten Servlet-Container empfehlen wir Ihnen das Produkt eines Apache-Projekts, wenngleich die hier vorgestellten Beispiele natürlich auch auf anderen Servern mit Java-Laufzeitumgebung ausgeführt werden können.

Der Apache Tomcat (*http://jakarta.apache.org/tomcat*) ist einer der populärsten Servlet Container und bildet die Laufzeitumgebung für die webbasierten Anwendungen in Kapitel 2 bis 6. Die Installation gestaltet sich java-typisch einfach und systemunabhängig: Nachdem Sie das Archiv *jakarta-tomcat.zip* heruntergeladen oder von der CD kopiert haben, entpacken Sie es in ein Verzeichnis Ihrer Wahl – fertig.

Konfiguration des Apache Tomcat

Konfiguriert wird der Apache Tomcat über die Datei *server.xml* im Verzeichnis */conf*, welche natürlich in der jeweiligen Distribution enthalten ist. Listing 1.3 zeigt die Minimalversion der Datei für die Version 5 des Apache Tomcat, in die Sie die in den folgenden Kapiteln gezeigten Konfigurationseinträge einfügen müssen.

Listing 1.3 Konfigurationsdatei des Apache Tomcat (server.xml)

```
<!-- Minimalkonfiguration des Apache Tomcat Servers -->
<Server port="8005" shutdown="SHUTDOWN">

   <!-- Symbolischer Name des Services -->
   <Service name="Catalina">

      <!--
          Verwendeter Port unter dem der Server und damit auch
          die Webanwendungen angesprochen werden können
      -->
      <Connector port="8080"/>

      <!-- Definition einer Engine -->
      <Engine name="Catalina" defaultHost="localhost">

         <!--
             Definition des virtuellen Host ('localhost');
             Die Webanwendungen können später über folgende
             URLs angeprochen werden:
                http://127.0.0.1:8080/NameDerWebanwendung oder
                http://localhost:8080/NameDerWebanwendung

             Durch Aufruf der folgenden URL können Sie den
             erfolgreichen Start des Apache Tomcat Servers
             testen: http://127.0.0.1:8080
         -->
         <Host name="localhost" appBase="webapps"
               unpackWARs="true" autoDeploy="true">
            <!--
               Hier stehen später die Konfigurationseinträge
               aus den einzelnen Kapiteln.
            -->
         </Host>
      </Engine>
   </Service>
</Server>
```

Info

Eine umfangreiche Dokumentation (engl.) zu den Konfigurationsmöglichkeiten des Apache Tomcat finden Sie unter: *http://tomcat.apache.org/tomcat-5.5-doc/*.

Starten des Apache Tomcat

Nach erfolgreicher Installation des Tomcat können Sie ihn über die Skripte *startup.bat* bzw. *startup.sh* im Verzeichnis */bin* hochfahren. Sollte keine Fehlermeldung erscheinen, können Sie den korrekten Start über den URL *http://localhost:8080* testen.

Info

HTTPs steht für Hypertext Transfer Protocol *Secure*. Dabei handelt es sich um eine Variante des HTTP-Protokolls, bei der die Daten verschlüsselt übertragen werden.

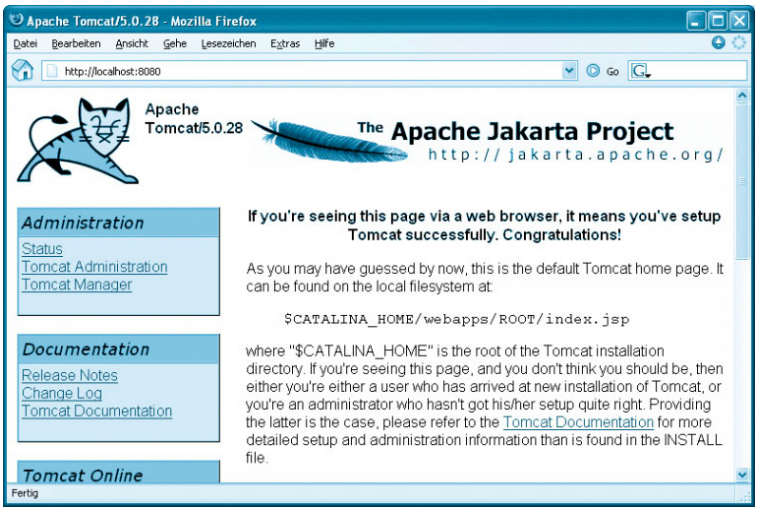


Abbildung 1.5
Test des Apache Tomcat

Konfiguration einer verschlüsselten Übertragung

Das in den späteren Beispielen eingesetzte Hypertext Transfer Protocol (HTTP) stammt aus einer Zeit, als das Internet vornehmlich zum Informationsaustausch für wissenschaftliche Zwecke genutzt wurde. Es basiert dabei auf zwei bereits seit den 70er Jahren existierenden Protokollschichten: TCP/IP. Damals war das Internet ein sehr übersichtlicher Ort und es gab quasi keine geschützten Bereiche.

Diese Basis ist jedoch denkbar ungeeignet für die Übertragung verschlüsselter Informationen und so ist es bereits mit geringem technologischem Aufwand möglich, Informationen Dritter aus Formularen bei deren Übertragung auszulesen. Abhilfe kann hier der Einsatz des *Hypertext Transfer Protocol secure* (HTTPS) schaffen. Obwohl alle folgenden Beispiele auch ohne verschlüsselte Übertragung funktionieren, zeigt Ihnen der folgende Absatz, wie Sie diese konfigurieren können.

1. Erzeugen eines Zertifikats mit dem »keytool«

Seit dem JDK 1.4 liefert Ihnen Sun ein einfaches kommandozeilenbasiertes Programm namens `keytool` mit, welches sich für die Erstellung von HTTPS-Zertifikaten eignet. Führen Sie zum Starten unter Windows folgenden Befehl auf der Kommandozeile aus:

```
%JAVA_HOME%\bin\keytool -genkey -alias tomcat -keyalg RSA
```

Listing 1.4
Erzeugen eines Zertifikats unter Windows

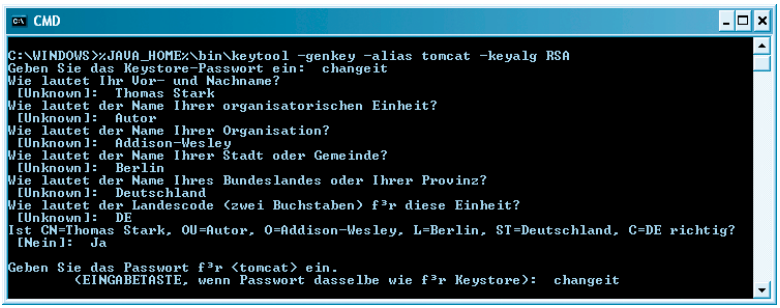
Unter Linux lautet der entsprechende Befehl:

Listing 1.5
Erzeugen eines Zertifikats unter Linux/Unix

```
$JAVA_HOME/bin/keytool -genkey -alias tomcat -keyalg RSA
```

Anschließend erfragt das Skript ein Passwort sowie verschiedene Angaben zu Ihrer Person, die später an den Client gesendet werden.

Abbildung 1.6
Erfassen der Daten für ein HTTPS-Zertifikat



2. Konfiguration des Apache Tomcat für HTTPS

Nachdem Sie das Skript ausgeführt haben, befindet sich in Ihrem Home-Verzeichnis die verschlüsselte Datei *.keystore*, die das Zertifikat enthält. Nun müssen Sie lediglich den folgenden Eintrag in Ihre Tomcat-Konfigurationsdatei *server.xml* (siehe Listing 1.3) einfügen und schon können Sie Ihre Dateien auch verschlüsselt übertragen.

Listing 1.6
HTTPS-Konfiguration des Apache Tomcat (server.xml)

```
<!-- Konfiguration des Apache Tomcat Servers mit HTTPs -->
<Server port="8005" shutdown="SHUTDOWN">

   <!-- Symbolischer Name des Services -->
   <Service name="Catalina">

      <!--
          Verwendeter Port unter dem der Server und damit auch
          die Webanwendungen angesprochen werden können
      -->
      <Connector port="8080"/>

      <!--
          Konfiguration des HTTPs-Zertifikates;
          dieser Connector lauscht auf Port '8443' und
          ermöglicht eine verschlüsselte Kommunikation
      -->
      <Connector port="8443"
                 keystoreFile="Pfad/zur/keystore/Datei"
                 keystorePass="Passwort/der/keystore/Datei"
                 acceptCount="100" scheme="https" secure="true"
                 clientAuth="false" sslProtocol="TLS"/>

      <!-- Definition einer Engine -->
      <Engine name="Catalina" defaultHost="localhost">

         <!--
             Definition des virtuellen Host ('localhost');
             die Webanwendungen können später über folgende
             URLs angeprochen werden:
                http://127.0.0.1:8080/NameDerWebanwendung oder
                http://localhost:8080/NameDerWebanwendung
```

Listing 1.6 (Forts.)
HTTPS-Konfiguration des Apache Tomcat (server.xml)

```
            Um die Kommunikation zu verschlüsseln, verwenden
            Sie analog die folgenden URLs:
               https://127.0.0.1:8443/NameDerWebanwendung oder
               https://localhost:8443/NameDerWebanwendung
         -->
         <Host name="localhost" appBase="webapps"
               unpackWARs="true" autoDeploy="true">
            <!--
               Hier stehen später die Konfigurationseinträge
               aus den einzelnen Kapiteln.
            -->
         </Host>
      </Engine>
   </Service>
</Server>
```

1.4.5 JBoss Application Server

Während für die ersten Kapitel der oben beschriebene Apache Tomcat vollkommen ausreicht, benötigen die Beispiele der Kapitel 7 bis 13 statt eines Webservers einen vollwertigen Java EE Application Server. Unsere Wahl fiel dabei auf den ebenfalls frei verfügbaren *JBoss*, den Sie samt Dokumentation von der Homepage *http://www.jboss.org* herunterladen können und natürlich ebenfalls auf der CD finden.

Info

Während ein Web Application Server, wie der Apache Tomcat, für die Bereitstellung von Webanwendungen optimiert ist, ermöglicht ein Application Server, wie etwa der JBoss, auch die Bereitstellung anderer Dienste, wie beispielsweise Enterprise JavaBeans. Der JBoss erweitert damit die Möglichkeiten des Apache Tomcat.

Der JBoss Application Server besteht aus einer Vielzahl von einzelnen Modulen, die analog zum Apache Tomcat über XML-Dateien im Verzeichnis *server/default/conf* konfiguriert werden. Da eine vollständige Beschreibung der Konfiguration aller Services ebenfalls ein ganzes Buch füllen würde, beschränken wir uns an dieser Stelle auf die Installation und zeigen an den entsprechenden Stellen, welche Einträge Sie zum Testen der Beispiele vornehmen müssen.

Installation und Test

Nachdem Sie das Programmarchiv heruntergeladen oder von der CD kopiert haben, genügt es, dieses in ein beliebiges Verzeichnis zu entpacken. Auch der JBoss ist bereits vorkonfiguriert und lässt sich anschließend sofort über die Skripte *run.sh*, bzw. *run.bat* im Verzeichnis *bin* starten.

Abbildung 1.7
Test des JBoss

Achtung

Das JBoss-Programmpaket enthält neben zahlreichen anderen Modulen auch einen integrierten Apache Tomcat, um beispielsweise seine eigenen dynamischen Webseiten für das *JBoss Management* auszuliefern. Da aber stets nur ein Server auf Port 8080 (Listing 1.2) arbeiten darf, können Sie nur einen der beiden Server (Tomcat oder JBoss) starten oder müssen die Konfiguration der verwendeten Ports entsprechend abändern.

1.4.6 Achtung, fertig, los!

Nachdem Sie sich mit den drei Grundkonzepten der Java-EE-Programmierung vertraut gemacht, für eine passende Entwicklungsumgebung entschieden und die benötigte Software installiert haben, können Sie sich nun direkt und ohne Umwege in das Programmiervergnügen stürzen – viel Spaß dabei!

2

JavaServer Pages

Dieses Kapitel wird Sie in die Lage versetzen, Ihre statischen HTML-Seiten mit dynamischem Java-Code zu verknüpfen, um diese wiederverwendbar zu machen oder zu personalisieren. Sie werden lernen, wie Sie Java-Fragmente mit HTML-Elementen verknüpfen, um ansprechende und leistungsfähige Java-Anwendungen zu schreiben. Ihre Anwender können auf diese dann mit einem gewöhnlichen Browser, wie dem Internet Explorer oder Firefox zugreifen, ohne selbst auf eine Java-Laufzeitumgebung angewiesen zu sein.

JavaServer Pages oder kurz JSP sind eine von Sun Microsystems im Rahmen der Enterprise Edition entwickelte Technologie, um Java-Code in HTML-Seiten einbetten und beim Abruf der Seite auf dem Server ausführen zu können. Sie bilden damit eine Alternative zum *Common Gateway Interface (CGI)*, *PHP* oder den *Active Server Pages (ASP)* von Microsoft, mit denen Sie vielleicht schon Erfahrungen sammeln konnten.

Um den in einer JSP enthalten Java-Code ausführen zu können, benötigen Sie natürlich analog zu Ihren Java-Programmen eine Laufzeitumgebung oder Virtual Machine, die den Code interpretiert und ausführt. Diese Laufzeitumgebung besteht bei JavaServer Pages aus einem *Servlet-Container*, wie dem Apache Tomcat, dessen Installation und Funktionsweise Ihnen das erste Kapitel beschreibt.

Info

Ein solcher Servlet-Container kann dabei auch in einen Java Enterprise Application Server, wie beispielsweise den in diesem Buch verwendeten JBoss Application Server (*www.jboss.org*), integriert sein. Eine Einführung und einen Überblick über die verschiedenen Bestandteile des JBoss finden Sie in Kapitel 1.

Achtung

Da der Apache Tomcat neben JSPs auch statische Ressourcen wie HTML-Dokumente und Bilder ausliefern kann, wird er in verschiedenen Quellen auch als *Webserver* bezeichnet. Um hier keine Verwirrung zwischen der Nomenklatur zu schaffen, können Sie sich den Servlet-Container in diesem und den folgenden Kapiteln als eine in einen Webserver integrierte Java-Laufzeitumgebung für JSPs und Servlets vorstellen.

Bevor Sie jedoch beginnen, eigene Webanwendungen zu entwickeln, soll Ihnen der nächste Abschnitt zunächst einen Einstieg in das Zusammenspiel von Browser, Internet und Servlet-Container vermitteln.

2.1 Webanwendungen gestern und heute

Was ist eigentlich das Internet? Was wir unter dem Begriff Internet in der Regel verstehen, ist der weltweit größte Zusammenschluss von Computern, welche all die von uns inzwischen für unverzichtbar gehaltenen Dienste wie das *World Wide Web (WWW)* oder *E-Mail* anbieten. Jeder ans Internet angeschlossene Computer ist dabei im gesamten Internet unter einer eindeutigen Nummer bekannt und kommuniziert mit den anderen Rechnern über ein festgelegtes Protokoll, wie etwa das *Hypertext Transfer Protocol (HTTP)*.

Info

Natürlich handelt es sich bei einem Server genau genommen um ein Programm, welches auf einem Rechner läuft und den Service anbietet. So kann ein Rechner auch mehrere Server(-Programme) gleichzeitig hosten und über unterschiedliche Ports bereitstellen. Um das Bild aber nicht zu verkomplizieren, werden wir hier mit einer vereinfachten Darstellung arbeiten und den gesamten Rechner als einen Server betrachten.
Um viele der Beispiele in diesem Buch auch an einem Rechner nachvollziehen zu können, wird Ihr Arbeitsplatzrechner dann als Server und als Client fungieren und in diesem Fall mit sich selbst kommunizieren.

Man unterscheidet die angeschlossenen Rechner dabei nach

- Servern, die Daten oder Dienste bereitstellen, und
- Clients, die diese Dienste abrufen.

Achtung

Keine Panik, wenn Sie beim ersten Lesen noch nicht alle Einzelheiten dieses Absatzes nachvollziehen können. Sie sollen zunächst einen Überblick über die verschiedenen, an Webanwendungen beteiligten Technologien und deren Zusammenspiel erhalten.
In späteren Absätzen und Kapiteln werden Sie immer wieder Verweise auf diese Einführung finden, z.B. wenn es darum geht, bestimmte Einschränkungen oder Vorgehensweisen zu verstehen. Sie können dann hierher zurückblättern und sich ein zunehmend genaueres Bild machen.

Ein typischer Client ist zum Beispiel Ihr Notebook, mit dem Sie Ihre E-Mails überprüfen oder auch Ihr PDA, auf den Sie sich die aktuellen Nachrichten Ihres Newstickers herunterladen. In verteilten Umgebungen, wie Sie Ihnen in späteren Kapiteln begegnen werden, kann ein Rechner dabei sowohl Client als auch Server zugleich sein, doch zunächst genügt es, wenn Sie sich unter einem Client Ihren Arbeitsplatzrechner und unter einem Server einen Rechner vorstellen, welcher Dokumente und Dateien bereithält.

Info

Über die genaue Geburtsstunde des Internets gibt es verschiedene Auffassungen: Einerseits könnte man sie mit der Verwendung des *TCP/IP-Protokolls* gleichsetzen. Es wurde 1973 erstmalig eingesetzt und basierte auf dem Ende der 60er beim amerikanischen Militär verwendeten *ARPA-Net*.
Richtig populär wurde das Internet jedoch erst mit der Einführung des *Hypertext Transfer Protocol (HTTP)*, durch das jeder leicht Zugang ins Netz der Netze findet, weshalb oft auch das damit verbundene Jahr 1989 als Geburtsstunde des Internets genannt wird.

2.1.1 Das Hypertext Transfer Protocol (HTTP)

1989 entwickelte Tim Berners-Lee am CERN, dem europäischen Kernforschungslabor in Genf, ein Kommunikationsprotokoll, das den dort arbeitenden Wissenschaftlern den Zugang zu Informationen erleichtern sollte. Dieses Hypertext Transfer Protocol (HTTP) arbeitete auf *Anwendungsebene,* war in gewissem Maße objektorientiert und funktionierte nach einem simplen Frage-Antwort-Schema. Es definierte lediglich, wie Daten von einem Rechner, dem Server, auf einen anderen Rechner, den Client zu übertragen sind.

Dazu wurden diese Daten in einzelne, kleine Teilpakete gepackt, wie bei der Post mit einer Adresse versehen und auf die Reise geschickt. Der Empfänger (Client) musste die empfangenen Pakete dann nur noch in der richtigen Reihenfolge aneinander fügen und schon hatte er die Information.

Info

Was ist ein *Protokoll*? Nun stellen Sie sich einmal vor, Sie wären beim norwegischen Kronprinzen Haakon Magnus und seiner Frau Mette-Marit zum Dinner eingeladen. Dann würde das höfische Protokoll vorschreiben, dass der Kronprinz Sie zuerst begrüßt, Sie dann zurückgrüßen, sich verneigen können usw. Ein Protokoll bestimmt demnach also den Ablauf der Kommunikation und nicht ihren Inhalt.

Aufgrund dieses einfachen Prinzips erfreute sich HTTP schnell wachsender Beliebtheit, doch da wohl keiner der beteiligten Wissenschaftler damals etwas von dessen Einsatz für die Übertragung sekundengenauer Börsendaten, dem authentifizierten Online-Banking, Chat-Foren und unternehmensweiten Intranetapplikationen ahnte, müssen wir einige seiner Beschränkungen heute mit diversen Konventionen umgehen.

Dabei ist HTTP bei weitem nicht das einzige Protokoll, über das Computer miteinander kommunizieren. Viele *Kommunikationsprotokolle*, wie beispielsweise das bekannte File Transfer Protocol (FTP), setzen dabei allerdings eine *feste Verbindung* zwischen den Kommunikationspartnern voraus. Dadurch kann der Server zu jedem Zeitpunkt feststellen, an wen er im Augenblick Daten übermittelt und ob der Aufbau einer weiteren Verbindung möglich ist. Diese Protokolle werden deshalb auch als *Zustandsprotokolle* (engl. »stateful protocols«) bezeichnet.

Im Gegensatz dazu verpflichtet HTTP als *zustandsloses Protokoll* (engl. »stateless protocol«), den Server dazu, auf jede eingehende Anfrage (Request) mit einer Antwort (Response) zu reagieren, auch wenn die Antwort lediglich aus einem Abweisen des Clients besteht. Zustandlose Protokolle benötigen dabei wesentlich weniger Ressourcen als Zustandsprotokolle, da der Server viel weniger Daten zu verwalten hat. Dafür »vergisst« der Server jede Übertragung, und damit auch den beteiligten Rechner sofort nachdem er sie abgeschlossen hat. Doch schauen wir uns nun eine solche Übertragung im Einzelnen an.

2.1.2 Der Request – eine Seite anfordern

Haben Sie heute schon die Nachrichten im Newsticker Ihrer Wahl gelesen oder sich die Java-Dokumentation auf der Webseite von Sun angeschaut? Die Informationsgesellschaft benötigt stets aktuelle Informationen und das rund um die Uhr, weltweit.

Doch was passiert eigentlich, wenn Sie in Ihren Browser eine URL, wie z.B. *http://www.dwd.de* (*Deutscher Wetterdienst*) eingeben und somit ein Dokument aus dem Internet anfordern? Nun, zunächst wird der Browser über das so genannte *Domain Name System (DNS)* die IP-Nummer des Servers ermitteln, der sich hinter dem von uns angegebenen Namen (z.B.

dwd.de) verbirgt. Wir können uns das DNS damit als so etwas wie das Telefonbuch des Internets vorstellen und brauchen uns darüber keine Gedanken mehr zu machen.

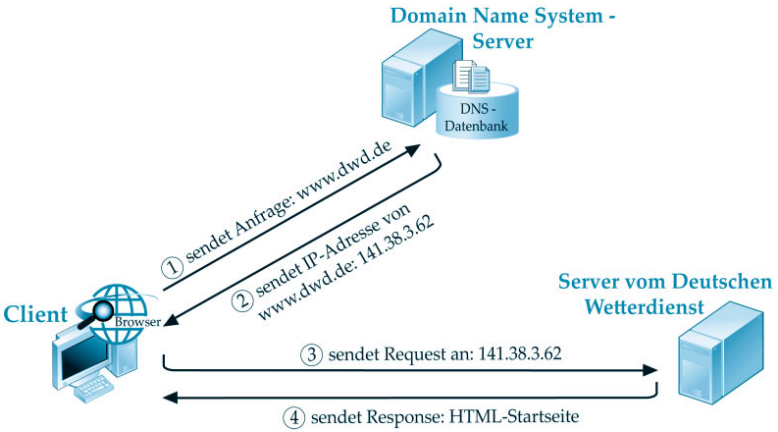


Abbildung 2.1
Zuordnung von Domains auf IP-Adressen durch das Domain Name System (DNS)

Das *Domain Name System (DNS)* ist einer der meistgenutzten Dienste des Internets überhaupt. Er dient dazu, symbolischen Servernamen wie *www.dwd.de* den tatsächlichen IP-Adressen der Server (141.38.3.62) zuzuordnen.

Nachdem Ihr Browser die IP-Adresse des Servers ermittelt hat, sendet er Ihre Anfrage, die im Folgenden *Request* genannt wird, an diesen Server und wartet auf dessen Antwort. Abbildung 2.2 zeigt den schematischen Aufbau eines solchen Request.

Info

Das Kapitel Java Naming and Directory Interface (JNDI) beschäftigt sich mit Namensdiensten und zeigt Ihnen, wie Sie mit Java EE auf diese zugreifen können.

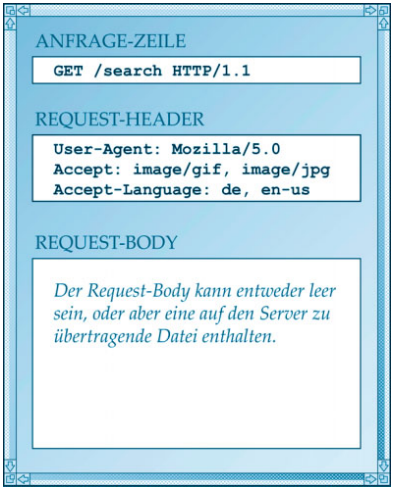


Abbildung 2.2
Schematischer Aufbau eines HTTP-Request

Ein HTTP-Request beginnt stets mit einer einzelnen Anfragezeile, die drei wichtige Informationen enthält, deren Bedeutung sich Ihnen in den nächsten Kapiteln erschließen wird:

1. Die Anfrage-Methode (`GET`)
2. Die angeforderte Ressource (`/search`)
3. Die vom Client unterstützte Version des HTTP-Protokolls (`HTTP/1.1`)

Mithilfe dieser drei Informationen teilt Ihr Browser dem Server mit, welche Ressource Sie benötigen, und versetzt diesen in die Lage, Ihnen zu antworten. Manchmal ist es jedoch ratsam, dem Server zunächst auch etwas mehr von sich preiszugeben, damit dieser seine Antwort optimal auf Ihre Bedürfnisse zuschneiden kann. Informationen dieser Art könnten beispielsweise der von Ihnen verwendete Browser (Mozilla) oder auch die bevorzugten Ausgabesprachen (Deutsch und Englisch) sein.

Info

Die Request-Header sind optional und müssen nicht vom Browser übertragen werden. Einige Browser gestatten es auch, die übertragenen Informationen einzuschränken.

Für diese Metainformationen sieht das HTTP-Protokoll die so genannten *Request-Header* vor, die immer unmittelbar nach der einleitenden Anfrage-Zeile übermittelt werden und folgende Form haben.

Listing 2.1 Schematischer Aufbau von Request-Headern

```
HeaderName: Wert_A, Wert_B ....
```

Die Request-Header sind übrigens nicht zu verwechseln mit den so genannten *Request-Parametern*. Diese sind Bestandteil der angeforderten Ressource und werden verwendet, um den Request zu personalisieren und mit eindeutigen Merkmalen zu versehen. So enthält *Google* beispielsweise nicht für jeden Suchbegriff eine eigene Webseite, sondern überträgt die eingegebenen Begriffe der Suche als Request-Parameter `q` – für Query (engl. Frage) – der Seite `search`.

Wenn Sie also nach Seiten mit dem Schlüsselbegriff *Java* googlen, lautet die vom Browser angeforderte URL

Listing 2.2 URL beim Googeln nach Java

```
http://www.google.de/search?q=Java
```

Wenn Sie jedoch speziell nach der *Java Enterprise Edition* suchen, steht in der Browser-Zeile anschließend:

Listing 2.3 URL beim Googeln nach der Java Enterprise Edition

```
http://www.google.de/search?q=Java+Enterprise+Edition
```

Request-Parameter bilden damit das Pendant zu den Variablen Ihrer Java-Programme und werden dazu eingesetzt, Eingaben der aktuellen Seite zu übertragen, während Request-Header client-spezifische Meta-Informationen übermitteln. In diesem und dem nächsten Kapitel werden Sie lernen, wie Sie beide auslesen und weiterverarbeiten können.

Info

Request-Parameter sind vergleichbar mit seitenspezifischen Variablen und Bestandteil der URL. Request-Header werden in der Regel vom Browser gesetzt und für den Anwender unsichtbar übermittelt.
Bei einigen Anfragemethoden werden auch die Request-Parameter, ähnlich wie die Request-Header, »unsichtbar« übermittelt. Näheres hierzu erfahren Sie in Kapitel 3.

2.1.3 Die Response – ein Server antwortet

Jeden empfangenen *Request* quittiert der Server mit einer *Response* (engl. Antwort), welche ebenfalls aus drei Teilen besteht.

Abbildung 2.3
Schematischer Aufbau einer http-Response

Auch die vom Server übermittelte HTTP-Response beginnt mit einer Statuszeile, welche drei für den Browser essenzielle Informationen enthält:

1. Die vom Server verwendete Version des HTTP-Protokolls (`HTTP/1.1`)
2. Einen numerischen Status-Code (`200`), wobei die Kombination `404` für `Not found` wohl die bekannteste sein dürfte
3. Eine kurze Beschreibung des Status in Textform (`OK`)

Ebenso wie der Request kann auch die Response Meta-Informationen in Form von Response-Headern enthalten. Hierzu zählen beispielsweise das Datum der Auslieferung (`Date`) oder die Art des Dokuments (`Content-Type`).

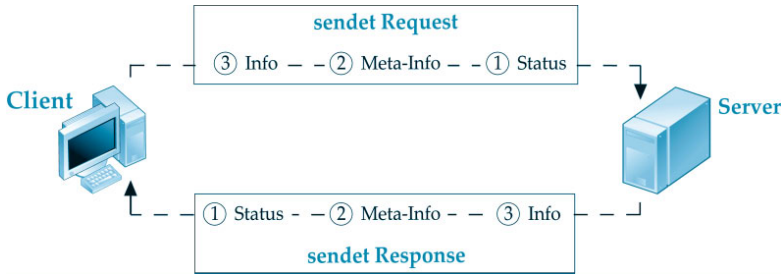
Den Abschluss der Response bildet der Response-Body, welcher die angeforderte Webseite (gewöhnlich im HTML-Format) enthält, die anschließend durch Ihren Browser dargestellt wird.

```
<html>
   <body>Hello World</body>
</html>
```

Listing 2.4
Eine einfache HTML-Seite (vgl. Abbildung 2.3)

Die Kommunikation über HTTP basiert also auf einem Frage-Antwort-Spiel, bei dem Client und Server *Informationen*, z.B. eine HTML-Seite, und *Meta-Informationen* miteinander austauschen.

Abbildung 2.4
Kommunikation zwischen Browser und Server



2.1.4 Hypertext Markup Language (HTML)

Nachdem sich Client und Server mithilfe des Domain Name System (DNS) gefunden und die erforderlichen Informationen untereinander ausgetauscht haben, ist es Aufgabe des Browsers, diese Informationen nun in ansprechender Form darzustellen.

Info

Der Name *Hypertext* (griech. »hyper«, »über«) leitet sich übrigens aus der Eigenschaft her, dass der Text auf ein anderes Dokument verweisende Textpassagen (Links) enthalten kann. Diese »dahinter liegenden« Seiten geben dem Text damit in gewisser Weise eine Dimension der Tiefe und ermöglichen erst das eigentliche Surfen durch das Internet, bei dem wir von einer Seite zur nächsten wechseln.

Natürlich sind Informationen in »Rohform« ohne die Möglichkeit, diese zu strukturieren, in den seltensten Fällen wirklich von Nutzen: Wer möchte schon lange Aufsätze lesen, wenn er nur auf der Suche nach einer bestimmten Information ist?

Egal, ob Adobe Acrobat, Microsoft Word oder QuarkXPress; nahezu jedes Textsatzprogramm speichert Informationen über die Anordnung und Formatierung von Bildern und Texten, und so existieren eine Unzahl verschiedener Formate, um diese Daten abzulegen. Die meisten dieser Formate sind jedoch dafür gedacht, von Maschinen »geschrieben« und interpretiert zu werden, und damit ohne ein entsprechendes Werkzeug nicht zu erstellen.

Deshalb ersann die Gruppe um Tim Berners-Lee ein einfaches, aber effektives Format, welches rudimentäre Formatierungen ermöglichte und von einem entsprechend geschulten Menschen mit einem einfachen Texteditor erstellt werden konnte: die *Hypertext Markup Language* (HTML).

Obgleich sie inzwischen mehrmals überarbeitet und erweitert wurde, bildet diese Auszeichnungssprache (*Markup Language*) auch heute noch die Grundlage für über 90% der im Internet zur Verfügung gestellten Dokumente.

Als Markup Language bezeichnet man eine Sprache, welche neben der reinen Textinformation auch Informationen über die Darstellung des Inhalts enthält. Bei HTML geschieht dies in Form so genannter Tags, die in den eigentlichen Text eingefügt werden.

```
Das ist ein <font color="red"> formatierter </font> Text.
```

Listing 2.5
Mit Tags formatierter Markup-Text

Die Markup-Informationen definieren letztendlich die Anordnung von Text und Bild und werden in der Regel nicht vom Browser dargestellt. Über die Browser-Funktion SEITENQUELLTEXT ANZEIGEN können Sie jedoch auch als Anwender einen Blick auf das vollständige Dokument werfen.

Tipp

Mehr Informationen über Auszeichnungssprachen und ihre Verarbeitung finden Sie im Kapitel über die eXtensible Markup Language (XML).

Das Resultat

Die DNS-Abfrage, die Auswertung des HTTP-Request und die Darstellung der übermittelten HTML-Seite durch den Browser: Das alles geschieht für uns meist völlig unbemerkt, wenn wir uns z.B. bei unserer morgendlichen Tasse Kaffee nebenbei über das Wetter informieren.

Abbildung 2.5
Was für ein Wetter ...

2.1.5 HTTP vs. HTML

HTTP und HTML, die Sie in den vorangegangenen Abschnitten kennen gelernt haben, sind zwei ähnlich klingende Begriffe für zwei grundverschiedene Dinge:

- HTTP ist ein *Protokoll*, welches beschreibt, wie und in welcher Reihenfolge Daten über das Netz ausgetauscht werden. Es legt z.B. fest, dass die Header vor dem Body gesendet werden müssen oder die Kommunikation immer beim Client beginnt.
- HTML ist eine *Beschreibungssprache*, welche Ihr Dokument strukturiert und dem Browser Informationen über die gewünschte Darstellung gibt.

Tipp

Dieses Buch widmet sich der Einführung in die Java Enterprise Edition (Java EE) und kann aus Platzmangel leider keine umfassende Einführung in HTML geben. Wir setzen daher ein grundsätzliches Wissen voraus, wobei wir den resultierenden HTML-Code möglichst einfach halten werden.
Außerdem möchten wir Sie auf das SelfHtml-Projekt unter *http://de.selfhtml.org/* hinweisen, das ihnen bei Bedarf eine hervorragende Referenz bereitstellt.

So können über das Internet mittels HTTP beispielsweise auch PDF-Dokumente übertragen werden, während HTML-Seiten nicht zwingend von einem Server gesendet werden müssen. So legen viele Software-Anbieter ihre Dokumentationen in Form von HTML-Dateien auf einer CD bei.

Wenn Sie also zukünftig Webanwendungen entwickeln, deren Resultate im Browser dargestellt werden sollen, dann werden Sie sich vor allem mit der Beschreibungssprache (X)HTML beschäftigen, die Sie einsetzen können, um beinahe jede beliebige Webseite zu erzeugen. Das Übertragungsprotokoll HTTP und die damit verknüpften Beschränkungen müssen Sie hingegen mehr oder weniger als gegeben hinnehmen. Es bietet Ihnen auf der anderen Seite einen etablierten Kommunikationsstandard, der von nahezu allem Clients verstanden wird.

2.2 Ihre erste dynamische JSP

Nachdem Ihnen der vorangegangene Absatz einen ersten Überblick über die verschiedenen, an Webanwendungen beteiligten Technologien gegeben hat, können Sie nun beginnen, Ihre Webseiten in dynamische JavaServer Pages zu verwandeln. Ausgangspunkt soll dabei das folgende (statische) HTML-Dokument sein:

```
<html>
   <body style="text-align:center; color:green">
      Dies ist eine statische HTML-Seite.
   </body>
</html>
```

Listing 2.6
Eine statische HTML-Seite (static.html)

Dieses Dokument enthält nur statischen Markup in Form der beiden Tags `<html>` und `<body>` und kann von einem beliebigen Browser dargestellt werden. Abbildung 2.6 zeigt das vom Firefox interpretierte Resultat.

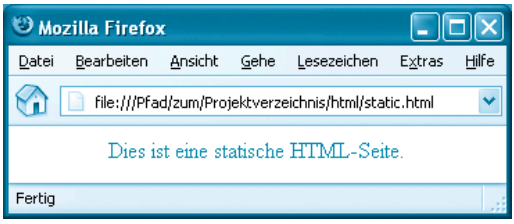


Abbildung 2.6
Eine statische HTML-Seite

2.2.1 Einfügen von Java-Anweisungen

Das HTML-Dokument aus Listing 2.6 besteht nur aus klassischen HTML-Tags, welche den darzustellenden Text umschließen und vom Browser interpretiert werden. Um nun Java-Code zu integrieren, verwenden Sie nichts anderes als eine neue Form von *Tags*.

```
<html>
   <body style="text-align:center; color:green">
      Dies ist eine dynamische JSP. <br/>
      Es ist jetzt genau: <%= new java.util.Date() %>
   </body>
</html>
```

Listing 2.7
Eine einfache JSP (simple.jsp)

Das ist es also?! Man fügt die Java-Anweisung einfach an der gewünschten Stelle in die Seite ein und umschließt sie mit den Tags `<%=` und `%>`? Nicht ganz, denn wenn Sie das Dokument nun wie zuvor über die Funktion DATEI->ÖFFNEN Ihres Browser aufrufen, erhalten Sie das in Abbildung 2.7 gezeigte, wohl eher unerwartete Ergebnis:

Abbildung 2.7
Öffnen der ersten JSP mit dem Browser

JSPs benötigen eine Java-Laufzeitumgebung und werden nicht korrekt interpretiert, wenn Sie diese über die DATEI->ÖFFNEN-Funktion Ihres Browser öffnen. Die JSPs müssen deshalb innerhalb der Java Virtual Machine eines Servlet-Container – wie Apache Tomcat – zur Ausführung gebracht werden.

Was ist schiefgelaufen? Erinnern Sie sich an die einführenden Sätze zu Beginn dieses Kapitels. Demnach bedarf es einer Java-Laufzeitumgebung auf Seiten des Servers, um eingebetteten Java-Code entsprechend ausführen zu können. Der Browser kann nämlich weder etwas mit den neuen Tags noch mit den Java-Klassen anfangen und interpretiert diese deshalb als Textinformation und stellt sie pflichtbewusst dar.

2.2.2 Einrichten der Webanwendung

Achtung

In diesem Abschnitt werden Sie lernen, wie Sie den Apache Tomcat – standalone oder integriert in den JBoss – für Ihre Webanwendung konfigurieren. Die dabei beschriebenen Dateien und Pfade gelten dabei ausschließlich für diesen Servlet-Container! Sollten Sie sich für ein anderes Produkt entschieden haben (mit dem die Beispiele natürlich ebenso funktionieren), entnehmen Sie die entsprechenden Informationen bitte dessen Dokumentation

Dieser Abschnitt zeigt Ihnen, welche Schritte notwendig sind, um Ihre Webanwendung für die JSP-Laufzeitumgebung zu konfigurieren. Als Servlet-Container kommt hierbei der Apache Tomcat zum Einsatz, den Sie entweder als eigenständiges Produkt oder als in den JBoss Application Server integriertes Modul betreiben können.

Nachdem Sie den Servlet-Container – wie in Kapitel 1 beschrieben – heruntergeladen und installiert oder von der beiliegenden CD kopiert haben, gliedert sich die Konfiguration in drei Schritte.

Schritt 1: Anlegen eines Projektverzeichnisses

Als Erstes benötigen Sie ein Arbeitsverzeichnis, welches alle zur Webanwendung gehörenden Dateien enthält. Dazu zählen natürlich zunächst die JSPs, aber auch eventuell eingebettete Bilder, Stylesheets etc. und nicht zuletzt auch die Konfigurationsdatei *web.xml*. Abbildung 2.8 zeigt die entsprechende Verzeichnisstruktur.

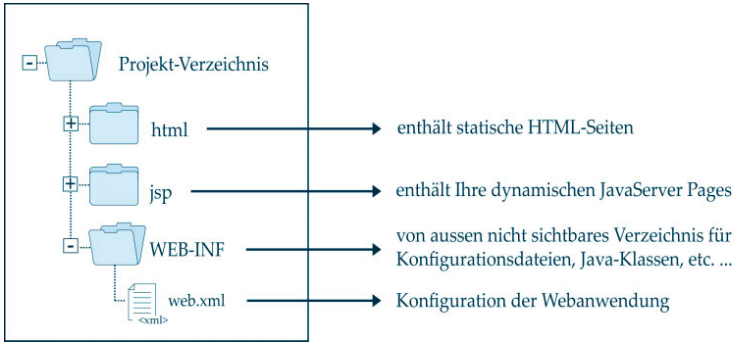


Abbildung 2.8
Projektverzeichnis für die Webanwendung

Die hier beschriebene Verzeichnisstruktur samt Konfigurationsdatei (*web.xml*) finden Sie auch auf der beiliegenden CD.

Achtung

Sie können die Verzeichnisstruktur Ihrer Webanwendung flexibel gestalten und die Verzeichnisse *images*, *html*, *jsp* etc. nach Belieben benennen oder verschachteln.

Das Verzeichnis *WEB-INF* ist jedoch ein wichtiges Verzeichnis für den Servlet-Container, dessen Bedeutung und Verwendung Sie in den nächsten Kapiteln kennen lernen werden. Der Ordner *WEB-INF* muss in genau dieser Schreibweise im Wurzelverzeichnis Ihrer Webanwendung existieren und die Konfigurationsdatei *web.xml* enthalten. HTML-Dateien, Bilder oder JSPs, die Sie unterhalb von *WEB-INF* ablegen, können nicht über den Browser abgerufen werden.

Die einzelnen Ordner enthalten dabei folgende Dokumente:

- In *images* und *html* werden alle statischen Dokumente und Bilder abgelegt.
- Der Ordner *jsp* beinhaltet alle JavaServer Pages (JSP).
- Das zwingend benötigte Verzeichnis *WEB-INF* enthält die Konfigurationsdatei (*web.xml*) der Webanwendung und wird vom Apache Tomcat in exakt dieser Schreibweise erwartet.

Während das Verzeichnis `WEB-INF` zwingend existieren muss, dient die Trennung zwischen HTML-Seiten und JSPs nur der Übersicht. Neben diesen Ordnern können auch noch weitere existieren – etwa für Cascading Stylesheets (CSS) oder JavaScript-Dateien.

Tipp

Seit der Servlet-Spezifikation 2.4 können Sie bei Bedarf auch auf den Web Deployment Descriptor (*web.xml*) verzichten. Hierfür muss der Servlet-Container aber auch diese Spezifikation unterstützen. Sicherer ist es, ihn einfach anzulegen, zumal größere Webanwendungen ihn für die Servlet-Konfiguration benötigen.

Schritt 2: Konfigurieren der Anwendung über den Web Deployment Descriptor

Damit der Servlet-Container Ihre Applikation überhaupt bereitstellen kann, muss er diese zunächst selbst kennen und gegebenenfalls benötigte Initialisierungen vornehmen. Für diese und andere Setup-Informationen dient ihm hierbei der so genannte *Web Deployment Descriptor* (engl. »Deployment«, dt. »Bereitstellung, Konfiguration«) in Form der Datei *web.xml*.

Info

Bereitstellungsdateien oder Deployment Descriptors werden Ihnen an vielen Stellen wiederbegegnen. Es handelt sich dabei häufig um XML-Dateien, die Konfigurationsinformationen für Ihre Anwendung enthalten.

Diese erwartet der Apache Tomcat im Unterverzeichnis *WEB-INF* und hat für das erste Beispiel zunächst folgende Form:

Listing 2.8 Web Deployment Descriptor (web.xml)

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE web-app PUBLIC
   "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
   "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
   <display-name>Eine erste Webanwendung</display-name>
</web-app>
```

Der Web Deployment Descriptor (*web.xml*) besteht, wie es die Endung schon aufzeigt, aus einer XML-Datei. Deren Verarbeitung mit Java werden Sie in Kapitel 11 über die eXtensible Markup Language kennen lernen. Sollten Sie mit XML und dessen Syntax noch nicht vertraut sein, achten Sie bitte darauf, das Listing getreu zu übernehmen, oder verwenden Sie die Beispieldatei auf der beiliegenden CD.

Achtung

Auch wenn der Web Deployment Descriptor (*web.xml*) noch keinerlei wichtige Informationen für den Servlet-Container enthält, muss er dennoch auf jeden Fall verfügbar sein. Fehlt die Datei *WEB-INF/web.xml* gehen verschiedene Servlet-Container von einer fehlerhaften Konfiguration aus und werden die Anwendung unter Umständen nicht bereitstellen.

Wie Sie in Listing 2.8 sehen, besteht die einzige Nutzinformation der Anwendung zunächst aus einem symbolischen Namen (`display-name`). Im nachfolgenden Kapitel über Servlets werden Sie den Web Deployment Descriptor jedoch verstärkt nutzen, um Ihre Anwendung unter verschiedenen Gesichtspunkten zu konfigurieren.

Schritt 3: Konfiguration des Apache Tomcat

Info

Der Apache Tomcat enthält bereits eine Webanwendung, über die Sie den Servlet-Container auch konfigurieren können. Eine Dokumentation über den Tomcat Manager finden Sie z.B. unter *http://tomcat.apache.org/tomcat-5.5-doc/manager-howto.html*.

Nachdem Sie den Servlet-Container installiert und die Verzeichnisstruktur sowie die Konfigurationsdatei *web.xml* erstellt oder von der CD kopiert haben, müssen Sie dem Apache Tomcat natürlich noch mitteilen, dass er die Webanwendung bereitstellen soll. Hierfür gibt es drei verschiedene Möglichkeiten:

1. Sie erstellen das Projektverzeichnis aus Schritt 1 in einem bestimmten Ordner, den der Apache Tomcat automatisch nach bereitzustellenden Webanwendungen durchsucht. Standardmäßig sind das die Ordner *$TOMCAT_HOME/webapps*, wenn Sie den Apache Tomcat standalone einsetzen, bzw. *$JBOSS_HOME/server/default/deploy*, wenn Sie den integrierten Tomcat verwenden möchten.
2. Statt das komplette Projektverzeichnis in Ordner des Apache Tomcat zu kopieren, können Sie über das auf der CD befindliche Ant-Script auch ein Webarchiv (WAR) erzeugen und dieses kopieren. Eine kurze Einführung in die Verwendung von Ant-Skripten finden Sie ebenfalls in Kapitel 1.

Info

Ein Webarchiv (WAR) ist nichts anderes als ein mit ZIP komprimierter Ordner, der das Unterverzeichnis *WEB-INF* samt der Datei *web.xml* enthält und die Endung *.war* hat. Wenn ein Servlet-Container ein solches Archiv findet, wird er es standardmäßig versuchen zu entpacken und die enthaltene Webanwendung bereitzustellen.

3. Sie können das Projektverzeichnis auch an beliebiger Stelle belassen und stattdessen den Apache Tomcat über die Konfigurationsdatei *server.xml* so konfigurieren, dass er dieses Verzeichnis mit einbezieht.

   Fügen Sie hierfür einfach ein `<Context>`-Element in die Konfigurationsdatei (*server.xml*) des Apache Tomcat ein.

Listing 2.9
Einbinden der Webanwendung über ein Context-Element der Konfigurationsdatei (server.xml)

```
<Server port="8005" shutdown="SHUTDOWN">
...
   <Service name="Tomcat-Standalone">
...
      <Engine name="Standalone" defaultHost="localhost">
...
         <Host name="localhost" appBase="webapps">
...
            <!-- Einbinden der Webanwendungen unter der URL
                 'http://localhost:8080/kap02/' -->
            <Context path="/kap02"
                     docBase="/Pfad/zum/Projektverzeichnis"/>
         </Host>
      </Engine>
   </Service>
</Server>
```

Achtung

Wenn Sie sich für die dritte Variante entscheiden, müssen Sie den Wert des Attributs `docBase` Ihrem tatsächlichen Projektverzeichnis anpassen, also beispielsweise *c:\Beispiele* bzw. */home/user/examples*.

Die drei genannten Möglichkeiten sind im Übrigen vollkommen gleichwertig, so dass Sie sich nur für eine entscheiden müssen. Die einzige offene Frage ist nun: Wie öffnen Sie Ihre JSP über den Servlet-Container?

Dazu müssen Sie den Apache Tomcat bzw. den JBoss nun wie in Kapitel 1 beschrieben starten und anschließend einfach die entsprechende URL eintippen. Wenn Sie sich für Variante 3 – die Konfiguration des Apache Tomcat via *server.xml* – entschieden haben, lautet die URL:

Listing 2.10
URL, um Ihre erste JSP über den Servlet-Container zu öffnen

```
http://localhost:8080/kap02/jsp/simple.jsp
```

Bei den Varianten 1 und 2 ersetzen Sie einfach die Zeichenkette `kap02` durch den Namen der *war*-Datei oder des Verzeichnisses.

Achtung

Achten Sie bei der Eingabe des Pfads auf korrekte Groß- und Kleinschreibung, da diese bei den Pfaden unterschieden wird.

Die URL aus Listing 2.10 setzt sich dabei aus folgenden Elementen zusammen:

- `http://` – definiert das zu verwendende Protokoll (HTTP).
- `localhost` – ist der symbolische Name Ihres Rechners, auf dem der Servlet-Container läuft.

  Der Servername `localhost` steht stellvertretend für die IP-Adresse `127.0.0.1`, welche für Ihren lokalen Rechner reserviert ist. Wenn Sie im Browser eine Ressource dieses Servers öffnen, rufen Sie stets Dienste auf Ihrem lokalen Rechner, und damit den dort laufenden JBoss bzw. Apache Tomcat auf.

  Ihr Rechner ist in diesem Fall Server und Client zugleich (siehe Abschnitt 2.1).

Tipp

Der symbolische Servername `localhost` hat die gleiche Funktionalität wie die IP-Adresse `127.0.0.1` und verweist auf Ihren lokalen Rechner.

- `8080` – gibt den Port an, auf dem der Servlet-Container zu erreichen ist.

  Da der Standardport `80` für HTTP auf den meisten Systemen bereits belegt oder nur mit besonderen Administratorrechten zugänglich ist, verwenden die meisten Servlet-Container standardmäßig `8080`.
- `kap02` – ist die Wurzelressource, unter der Ihre Webanwendung verfügbar gemacht wird und die über das Attribut `path` (Listing 2.9) definiert wird. Sollten Sie sich für die Konfigurationsoptionen 1 oder 2 entschieden haben, setzen Sie hier bitte den Namen des Verzeichnisses oder des Webarchivs ohne die Endung *.war* ein.
- `/jsp/` – lokaler Pfad zur Datei `simple.jsp` innerhalb der Webanwendung.

Abbildung 2.9 zeigt Ihnen das Resultat, wenn Ihre erste JSP (Listing 2.7) über den Servlet-Container ausgeliefert wird.

Abbildung 2.9
Ihre erste JSP wird über den Servlet-Container ausgeliefert.

An der Stelle, wo vorher Ihr JSP-Tag `<%= new java.util.Date() %>` stand, erscheinen nun – wenn auch noch unformatiert – das aktuelle Datum und die Uhrzeit, zu denen die Seite angefordert wurde, und ein Blick in den Seitenquelltext (SEITENQUELLTEXT ANZEIGEN) offenbart Folgendes:

```
<html>
   <body style="text-align:center; color:green">
      Dies ist eine dynamische JSP. <br/>
      Es ist jetzt genau: Wed May 17 10:28:15 CEST 2006
   </body>
</html>
```

Listing 2.11
Vom Browser interpretierter HTML-Code

Offensichtlich wurde also zusammen mit dem Servlet-Container ein Java-Interpreter gestartet, welcher die JSPs vor der Auslieferung auf JSP-Tags überprüft, den enthaltenen Java-Code zur Ausführung bringt und das Ergebnis in die HTML-Seite einfügt. Der Browser »sieht« danach ausschließlich reinen HTML-Code und kann Ihre dynamischen JSPs nicht von statischen Seiten unterscheiden!

Zusammenfassung: Einrichten der Webanwendung

Fassen wir die wichtigsten Punkte dieses Abschnitts noch einmal zusammen:

- Sie benötigen ein Projektverzeichnis, das den Ordner *WEB-INF* und darin den Web Deployment Descriptor (*web.xml*) enthält.
- Dieses Projektverzeichnis oder ein daraus erzeugtes Webarchiv (WAR) kann vom Servlet-Container bereitgestellt werden. Hierfür gibt es verschiedene Möglichkeiten der Konfiguration.
- Nachdem Sie den Servlet-Container – ob standalone oder eingebettet – gestartet haben, können Sie Ihre JSPs über eine URL abrufen. Diese setzt sich aus der Konfiguration, einem speziellen Port (8080), dem lokalen Pfad der Ressource innerhalb des Projektverzeichnisses zusammen.

  Bei der URL ist auf korrekte Groß- und Kleinschreibweise zu achten, weil der Servlet-Container hier differenziert.
- Der Browser interpretiert nach der Auslieferung der Seite durch den Servlet-Container reinen HTML-Code und kann nicht unterscheiden, ob das angeforderte Dokument statisch oder dynamisch ist.

Wenn Sie zum ersten Mal eine Webanwendung bereitstellen, dann sind dies zugegebenermaßen eine Menge Schritte. Die gute Nachricht ist, dass Sie diese Schritte nur ein einziges Mal durchführen müssen und Sie sich nun voll und ganz auf Ihre JSPs konzentrieren können.

Tipp

Trotz der Endung *.jsp* interpretiert der Browser eine über den Servlet-Container ausgelieferte JSP in der Regel als HTML-Dokument. Dies wird durch den Response-Parameter MIME-Type erreicht, den der Servlet-Container automatisch setzt. Mehr hierzu finden Sie in Abschnitt 2.6.1.

Wenn Sie Ihre Anwendung beim Durcharbeiten der nächsten Absätze um weitere JSPs, HTML-Seiten oder Bilder erweitern, müssen Sie – wenn Sie sich für die Konfigurationsoptionen 1 oder 2 entschieden haben – lediglich das Projektverzeichnis bzw. die WAR-Datei aktualisieren und anschließend den Apache Tomcat neu starten.

2.3 Bausteine für JavaServer Pages

Nun, da Sie wissen, wie Sie Ihre Webanwendungen über diesen bereitstellen, können wir direkt und unmittelbar in die Entwicklung weiterer JSPs einsteigen. Bisher ist eine JSP zunächst nichts anderes, als eine um *besondere* Tags erweiterte HTML-Seite.

Diese Tags unterteilen sich in vier Klassen:

- Ausdrücke der Form `<%=` *Java-Ausdruck* `%>`
- Scriptlets der Form `<%` *Java-Code* `%>`
- Deklarationen der Form `<%!` *Java-Code* `%>`
- Kommentare der Form `<%--` *Kommentartext* `--%>`

Die nächsten Teilabschnitte zeigen Ihnen, wie Sie diese vier unterschiedlichen Typen von JSP-Anweisungen einsetzen können, um leistungsfähige, dynamische Seiten zu erstellen.

2.3.1 JSP-Ausdrücke

JSP-Ausdrücke (engl. JSP-Expressions) haben Sie bereits im ersten Beispiel kennen gelernt. Diese Tags haben die Form:

Listing 2.12 Allgemeine Form eines JSP-Ausdrucks

```
...
   <%= Eine einzelne Java-Anweisung %>
...
```

Wie es das Gleichheitszeichen bereits andeutet, dienen JSP-Ausdrücke dazu, einzelne Java-Anweisungen in den HTML-Quelltext zu integrieren. Wird die betreffende HTML-Datei später von einem Browser angefordert, ersetzt der Server den JSP-Ausdruck durch das Ergebnis der Java-Anweisung.

Tipp

JSP-Ausdrücke werden in der resultierenden HTML-Seite durch das Ergebnis Ihrer Auswertung in der Java VM ersetzt.

Haben Sie beim Ausprobieren des `Date`-Beispiels (Listing 2.7) einmal versucht, das verwendete Datum etwa mit einem `java.text.SimpleDateFormat` zu formatieren?

Listing 2.13
Formatieren des Datums (fehlerhafte Version)

```
...
   Es ist jetzt genau:
   <%= java.text.SimpleDateFormat formatter =
                        new java.text.SimpleDateFormat();
       formatter.format(new java.util.Date()) %>
...
```

Dies führt beim Aufruf der Seite unweigerlich zu einem Fehler! JSP-Ausdrücke sind nämlich auf eine *einzelne (!)* Anweisung beschränkt. Das folgende Listing verschafft den gewünschten Erfolg:

Listing 2.14
Formatieren des Datums (korrekte Version)

```
...
   Es ist jetzt genau:
   <%= new java.text.SimpleDateFormat()
                        .format(new java.util.Date()) %>
...
```

Als Eselsbrücke können Sie sich dabei merken, dass sich jede Anweisung, die Sie über einen JSP-Ausdruck in ihre JSP integrieren, in einem vergleichbaren Java-Programm durch eine `System.out.print()`-Anweisung ausgeben lassen muss.

Listing 2.15
Ausgabe des formatierten Datums in einem Java-Programm

```
// Das führt zu einem Kompilierungsfehler (!)
System.out.print( java.text.SimpleDateFormat formatter =
                     new java.text.SimpleDateFormat();
                  formatter.format(new java.util.Date()));

// Diese Version funktioniert hingegen bestens
System.out.print( new java.text.SimpleDateFormat()
                                  .format(new java.util.Date()));
```

Aus dieser Analogie ergeben sich nun auch zwei weitere Konsequenzen für JSP-Ausdrücke:

1. Der Java-Code in JSP-Ausdrücken wird *nie* mit einem Semikolon (;) abgeschlossen, genau wie Ausdrücke innerhalb der Klammern von `System.out.print()` nicht abgeschlossen werden.
2. Sie können in JSP-Ausdrücken nur Anweisungen aufrufen, die eine Zeichenkette (`String`), einen primitiven Datentyp oder ein Objekt (z.B. `Date`) zurückgeben, wobei für Letzteres intern die Methode `toString()` gerufen wird (vgl. Listing 2.7).

Tipp

JSP-Ausdrücke `<%=...%>` sind vergleichbar mit einer `System.out.print()`-Anweisung für JSPs!

Die folgende Definition einer Variablen, welche zwar ebenfalls aus einer einzelnen Java-Anweisung besteht, ist also unzulässig, obwohl sie nur aus einer einzelnen Java-Anweisung besteht: weil eine Variablendeklaration eben kein Objekt zurückliefert.

Listing 2.16 Unzulässige Zuweisung zu einer Ganzzahl

```
...
   Anzahl der Seitenaufrufe: <%= int i = 3 %>
...
```

Wohingegen folgender Code zum Erfolg führt, weil Sie hierbei ein Objekt vom Typ `Integer` erzeugen, welches über eine Methode `toString()` verfügt.

Listing 2.17 Zulässige Ausgabe einer Zahl

```
...
   Anzahl der Seitenaufrufe: <%= new Integer("3") %>
...
```

Schließlich können Sie mit JSP-Ausdrücken sogar Text ausgeben, wofür im Augenblick allerdings noch wenig Motivation besteht.

Listing 2.18 Ausgabe von Strings via JSP-Expression

```
...
   Das ist ein <%= "dynamisch erzeugter" %> Text.
...
```

2.3.2 Mehrzeilige Java-Anweisungen durch Scriptlets

Nun gibt es zugegebenermaßen eine verhältnismäßig geringe Anzahl von Java-Ausdrücken, die für sich allein stehend ein sinnvolles Ergebnis liefern. Um sinnvoll in Java programmieren zu können, benötigen Sie stattdessen mindestens drei weitere Dinge:

1. Die Möglichkeit, lokale Variablen zu definieren, um Ergebnisse zwischenzuspeichern, und Hilfsobjekte wie `Formatter` (Listing 2.13) an einen symbolischen Namen zu binden.
2. Die Möglichkeit, durch `if-else` oder ähnliche Strukturen Entscheidungen zur Laufzeit zu treffen.
3. Schleifen, um über Objekte variabler Länge, wie Listen oder Arrays, iterieren zu können oder auch einfach Anweisungen zu wiederholen.

Hierfür definiert die Spezifikation die so genannten *JSP-Scriptlets*. Diese ähneln den JSP-Ausdrücken, es fehlt ihnen jedoch ein Gleichheitszeichen (=):

Listing 2.19 Allgemeine Form von JSP-Scriptlets

```
...
   <% ... Eine beliebige Folge von Java-Anweisungen ... %>
...
```

JSP-Scriptlets können überall dort in das HTML-Dokument eingefügt werden, wo auch ein JSP-Ausdruck stehen könnte. In JSP-Scriptlets können Sie so programmieren, wie Sie es von den `main()`-Methoden Ihrer *Java-Anwendungen* her gewohnt sind.

Tipp

Warum Sie in JSP-Scriptlets und -Ausdrücken auch auf Variablen anderer Scriptlets zugreifen können, zeigt Ihnen der nachfolgende Absatz 2.3.3.

Das folgende Listing demonstriert dies anhand einer JSP, deren Hintergrundfarbe durch die Kombination dreier Zufallszahlen für den Rot-, Grün- und Blauanteil bestimmt wird.

Listing 2.20
JSP mit dynamischer Hintergrundfarbe (background.jsp)

```
<%
   // Innerhalb dieses JSP-Scriptlets erzeugen Sie 3 Zufallszahlen
   // deren Kombinationen die Hintergrundfarbe ergeben
   java.util.Random generator = new java.util.Random();
   String red = Integer.toHexString(generator.nextInt(256));
   String green = Integer.toHexString(generator.nextInt(256));
   String blue = Integer.toHexString(generator.nextInt(256));
%>
<html>
   <body style="background-color:#<%= red + green + blue %>">
      Diese Seite hat eine dynamische Hintergrundfarbe.
      Testen Sie es, indem Sie mehrmals den Reload-Button betätigen.
   </body>
</html>
```

Innerhalb des JSP-Scriptlet erzeugen Sie zunächst einen Generator (`Random`) für Ihre Zufallszahlen und generieren mit diesem drei Zahlenwerte zwischen 0 und 255. Diese überführen Sie anschließend in ihre hexadezimale Darstellung und stehen nun vor einem Problem: Sie müssen die zufällig erzeugten Werte in die JSP integrieren, denn im Gegensatz zu JSP-Ausdrücken erzeugen JSP-Scriptlets nicht automatisch eine Ausgabe in die resultierende Seite.

Tipp

Innerhalb von JSP-Scriptlets muss jede Anweisung mit einem Semikolon (;) abgeschlossen werden.

Wenig von Erfolg gekrönt ist die folgende Anweisung, welche den resultierenden Zahlenwert statt in die HTML-Seite, in das Logfile Ihres Apache Tomcat einfügt:

Listing 2.21
Ausgabe in das Logfile des Tomcat

```
...
   <%
      // Gibt den Farbwert in die Logdatei des Apache Tomcat aus
      System.out.println(red + green + blue);
   %>
...
```

Ausgabe von Variablen

Die einfachste Möglichkeit, in *JSP-Scriptlets* definierte Variablen in die resultierende HTML-Seite zu schreiben, sind wieder *JSP-Ausdrücke*. Innerhalb dieser können Sie auf alle zuvor in JSP-Scriptlets definierten Variablen zugreifen. Schon bietet die Ausgabe von Strings in JSP-Ausdrücken ungeahnte Möglichkeiten.

Listing 2.22
Ausgabe von Variablen via JSP-Ausdruck

```
...
   <body style="background-color:#<%= red + green + blue %>">
      Diese Seite hat eine dynamische Hintergrundfarbe.
      Testen Sie es, indem Sie mehrmals den Reload-Button betätigen.
   </body>
...
```

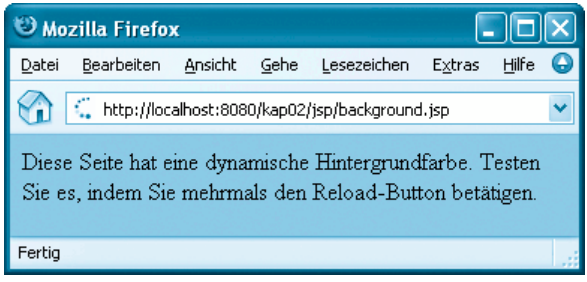
Wenn Ihnen die Verknüpfung der Farbwerte mit einem Pluszeichen (+) in diesem Listing etwas komisch vorkommt, machen Sie sich an dieser Stellt noch einmal die Analogie zu `System.out.println()` bewusst: Der Inhalt des JSP-Ausdrucks wird als String interpretiert und in die resultierende HTML-Seite integriert. Bei der Anweisung `red + green + blue` handelt es sich an dieser Stelle also um eine Verknüpfung im Sinne von Zeichenketten. Sie hätten stattdessen auch folgende Zeile definieren können:

Listing 2.23
Eine weitere Möglichkeit, die Zufallszahlen in die JSP zu integrieren

```
...
   <body style="<%= "background-color:#" + red + green + blue %>">
...
```

Egal, ob Sie sich nun für die Variante aus Listing 2.22 oder Listing 2.23 entscheiden – Sie können Ihre JSP nun bei jedem Aufruf mit einer neuen Hintergrundfarbe versehen, was durch mehrmaliges Betätigen des AKTUALISIEREN-Schalters ausgiebig getestet werden kann.

Abbildung 2.10
Eine der 16777216 verschiedenen Möglichkeiten



Natürlich existiert auch eine Möglichkeit, innerhalb großer JSP-Scriptlet-Blöcke eine HTML-Ausgabe zu erzeugen. Schließlich wäre es ziemlich umständlich, wenn Sie für jede Anweisung das aktuelle Scriptlet unterbrechen müssten, um einen einzelnen JSP-Ausdruck einzufügen. Statt des JSP-Ausdrucks in Listing 2.22 hätten Sie auch das folgende (einzeilige) JSP-Scriptlet verwenden können.

Listing 2.24
Ausgabe von Variablen via JSP-Scriptlet

```
...
   <body style="background-color:#<% out.print(red+green+blue); %>">
      Diese Seite hat eine dynamische Hintergrundfarbe.
      Testen Sie es, indem Sie mehrmals den Reload-Button betätigen.
   </body>
...
```

Das vordefinierte Writer-Objekt out

Listing 2.24 verwendet die in allen JSPs vordefinierte Variable `out`, welche – ähnlich `System.out` – für Ausgaben verwendet werden kann. Nur dass diese Ausgaben eben nicht auf der Kommandozeile oder in der Protokolldatei des Apache Tomcat laden, sondern wie gewünscht in der resultierenden Webseite.

Info

Die Ausgabe eines Objekts über einen JSP-Ausdruck und die `print()`-Methode der vordefinierten Variable out sind vollkommen gleichwertig. In diesem Buch werden wir aber die nach Ansicht des Autors intuitivere Schreibweise der JSP-Ausdrücke weiterverwenden.

Bei der Verwendung der Variable `out` vom Typ `javax.servlet.jsp.JspWriter` sind dabei zwei Dinge zu beachten:

1. `out` wird in JSP-Scriptlets und nicht etwa in JSP-Ausdrücken verwendet.
2. Wie in jedem JSP-Scriptlet muss eine Anweisung auch hier mit einem Semikolon (;) abgeschlossen werden, auch wenn es sich, wie im oben stehenden Listing um die einzige Anweisung des Scriptlet handelt.

Schließlich demonstriert Ihnen Listing 2.24, dass Sie nicht nur in JSP-Ausdrücken auf zuvor definierte Variablen zurückgreifen können, sondern Ihnen auch in nachfolgenden JSP-Scriptlets alle zuvor definierten Objekte zur Verfügung stehen.

Achtung

Die vorangegangenen Beispiele haben Ihnen gezeigt, dass Sie mit JSP-Ausdrücken und -Scriptlets nicht nur Texte wie das aktuelle Datum in Ihre HTML-Seiten einfügen, sondern auch auf Meta-Informationen wie Attributwerte (z.B. die Hintergrundfarbe) Einfluss nehmen und diese zur Laufzeit manipulieren können. In diesem Zusammenhang sei noch einmal deutlich darauf hingewiesen: Alle JSP-Anweisungen werden nur *vor* der Auslieferung Ihrer Seite vom Server interpretiert und ausgeführt. Alles, was der Browser des Clients »sieht« ist statisches HTML. Wie die Seite erzeugt wurde, kann im Nachhinein nicht mehr entschieden werden.

Auf diese Weise können Sie z.B.:

- HTML-Tags erzeugen,
- Texte und Formatierungen Ihrer HTML-Seiten manipulieren,
- Ganze Passagen ein- und ausblenden,
- Formulare und andere HTML-Elemente, wie Combo- oder Select-Boxen, Radio-Buttons oder Input-Felder vorbelegen.

2.3.3 Scriptlets, Ausdrücke und der rote Faden Ihrer JSP

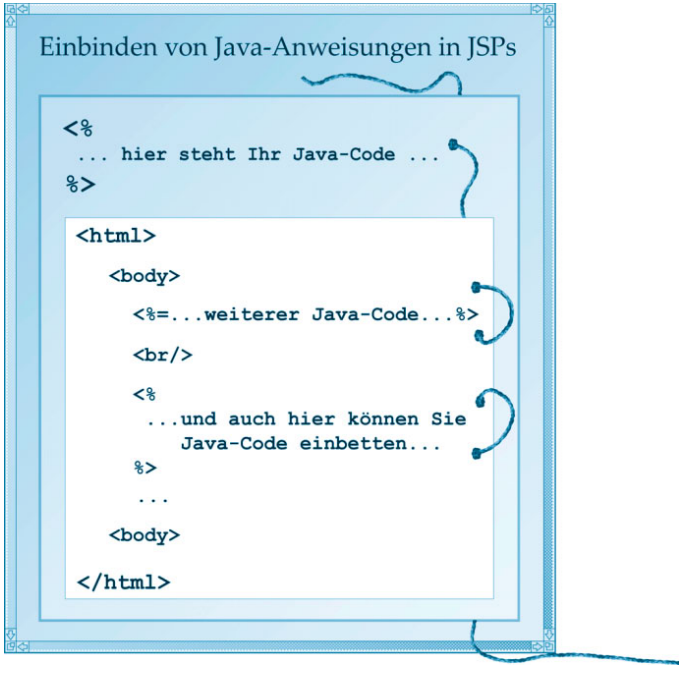
In den beiden vorangegangenen Abschnitten haben Sie zwei grundlegende Bausteine Ihrer zukünftigen JSPs kennen gelernt: JSP-Ausdrücke und JSP-Scriptlets. Während JSP-Ausdrücke dabei für Ausgaben in die resultierende HTML-Seite genutzt werden können, nehmen JSP-Scriptlets Java-Code auf und dienen unter anderem dazu, lokale Variablen zu deklarieren oder Zwischenergebnisse zu berechnen.

Info

JSP-Ausdrücke und -Scriptlets bilden das Analogon zur Methode `main()` Ihrer JSPs, die bei jedem Request ausgeführt und von oben nach unten abgearbeitet wird. Im Kapitel über Servlets können Sie außerdem nachlesen, wie nah dieser Vergleich der Wirklichkeit tatsächlich kommt.

Listing 2.20 hat Ihnen dabei gezeigt, dass sowohl JSP-Ausdrücke als auch JSP-Scriptlets miteinander verzahnt sind und z.B. auf zuvor deklarierte Variablen anderer Blöcke zugreifen können. JSP-Ausdrücke und -Scriptlets bilden gewissermaßen einen roten Faden, der sich wie ein Java-Programm durch Ihre ganze JSP zieht: Wenn ein Browser Ihre JSP anfordert, dann beginnt der Servlet-Container beim ersten Scriptlet oder Ausdruck, wertet diesen aus und setzt die Bearbeitung beim nächsten Scriptlet oder Ausdruck fort.

Abbildung 2.11
Roter Faden Ihrer JSPs



2.3.4 Definieren von Java-Methoden durch JSP-Deklarationen

JSP-Ausdrücke und -Scriptlets sind leistungsfähige Bausteine Ihrer JavaServer Pages und ermöglichen es, nahezu jede Funktionalität zu implementieren. Sie ziehen sich wie ein roter Faden durch die gesamte JSP und bilden damit deren `main()`-Methode, die von oben nach unten abgearbeitet wird.

Bis jetzt sind Ihre JSPs damit gewissermaßen Skripte, deren Befehle zufälligerweise in Java programmiert sind. Doch in diesem Abschnitt werden Sie lernen, wie Sie JSPs zu vollwertigen Objekten im Sinne objektorientierter Programmierung machen, indem Sie z.B. globale Konstanten oder Methoden definieren.

Motivation: ein einfacher Zugriffszähler

Haben Sie beim Herumexperimentieren mit Ihrem bisherigen Wissen über Java und JSPs vielleicht schon einmal versucht, einen Zugriffszähler für Ihre JSPs zu realisieren? Wenn ja, dann hatte der entsprechende Code vielleicht folgende Form:

Listing 2.25
Ein einfacher Zugriffszählers mit lokaler Variable (simpleCounter.jsp)

```
<%
   // Deklaration einer lokalen Zählvariable
   int counter = 1;
%>
<html>
   <body style="text-align:center; color:green">
      Sie sind der <%= counter++ %>. Besucher dieser JSP.
   </body>
</html>
```

Wenn Sie diesen Zugriffszähler testen, werden Sie vielleicht überrascht sein: Denn egal, wie oft Sie den AKTUALISIEREN-Schalter Ihres Browsers auch betätigen, das Ergebnis bleibt stets das Gleiche: Jeder Besucher Ihrer JSP ist automatisch der erste!

Globale vs. lokale Variablen

Um dieses Verhalten zu verstehen, müssen Sie sich vor Augen halten, dass es sich bei `i` nicht um eine globale, sondern um eine *lokale* Variable handelt – ähnlich den Hilfsvariablen, welche Sie innerhalb von Methoden zur Speicherung eines Zwischenergebnisses verwenden. Damit wird `i` natürlich auch bei jedem Aufruf der JSP neu erzeugt und mit `1` initialisiert und somit ist jeder Besucher der erste.

Info

JSP-Deklarationen können dazu verwendet werden, globale Variablen und Methoden zu definieren. Der in ihnen enthaltene Java-Code wird nicht automatisch bei jedem Request an diese JSP ausgewertet, kann jedoch von JSP-Scriptlets und -Ausdrücken aus aufgerufen werden.

Um das gewünschte Resultat zu erzeugen, benötigen Sie eine Variable, die außerhalb des roten Fadens definiert ist und ihren Wert auch über mehrere JSP-Aufrufe hinweg behält. Dies ermöglichen Ihnen schließlich *JSP-Deklarationen*, wie sie in diesem Abschnitt behandelt werden.

JSP-Deklarationen haben folgende allgemeine Form:

Listing 2.26
Allgemeine Form von JSP-Deklarationen

```
...
   <%! ... Deklaration von Methoden und globalen Variablen ... %>
...
```

Wie Sie sehen, unterscheiden sich JSP-Deklarationen nur durch ein einleitendes Ausrufezeichen (`!`) von JSP-Scriptlets, doch die Wirkungsweise ist eine gänzlich andere. Ändern Sie Listing 2.25 nur in diesem Sinne ab und schon »funktioniert« Ihr Zähler:

Listing 2.27
Ein einfacher Zugriffszähler mit globaler Variable (simpleCounter2.jsp)

```
<%!
   // Deklaration einer globalen Zählvariable
   int counter = 1;
%>
<html>
   <body style="text-align:center; color:green">
      Sie sind der <%= counter++ %>. Besucher dieser JSP.
   </body>
</html>
```

Durch das Ausrufezeichen (`!`) steht die Deklaration von `i` nun in einer JSP-Deklaration und damit außerhalb der gedachten Methode `main()`. Damit wird `i` zu einer klassenglobalen Variable, wie sie auch in einer Java-Klasse definiert werden könnte. Um dies zu unterstreichen, können Sie `i` nun auch mit einem *Modifizierer* (`private`) versehen, der hier zunächst natürlich nur zu Anschauungszwecken eingefügt wird und keinen Einfluss auf den tatsächlichen Programmfluss hat.

Wenn Sie das Servlet-Kapitel gelesen haben, werden Sie jedoch verstehen, warum Sie Ihre globalen Variablen und Methoden in JSPs mit dem Modifizierer `private` versehen sollten:

Listing 2.28
Globale Variable mit Modifizierer (private)

```
...
   <%!
      // Deklaration einer globalen Zählvariable mit Modifizierer
      private int i = 1;
   %>
...
```

Listing 2.27 zeigt Ihnen außerdem, dass der in JSP-Deklarationen enthaltene Code zwar nicht bei jedem Aufruf der Seite automatisch ausgeführt wird, Sie die hier definierten Variablen und Methoden jedoch problemlos in Ihren JSP-Ausdrücken und -Scriptlets verwenden können.

Info

Mehr über die dauerhafte Speicherung von Variablenwerten erfahren Sie in Kapitel 10 über Persistenz.

Achtung

In diesem Beispiel basiert Ihr Zugriffszähler auf einer globalen Variable, welche nicht in einer Datenbank oder einer Datei zwischengespeichert wird und nur so lange existiert, wie die Java Virtual Machine (JVM) im Speicher läuft. Bei einem Neustart des Servlet-Containers wird natürlich auch sie erneut initialisiert.

Ein erweiterter Zugriffszähler

Neben globalen Variablen können Sie in JSP-Deklarationen auch vollständige Methoden deklarieren, was in JSP-Ausdrücken und -Scriptlets vollkommen unmöglich ist. Um dies zu demonstrieren, erweitern wir den Zugriffszähler nun um die Methode `getStartDate()`, die eine formatierte Version des Datums zurückgibt, an dem der Server gestartet wurde.

Listing 2.29
Erweiterter Zugriffszähler (counter2.jsp)

```
<%!
   /* Globale Variablen innerhalb der JSP */
   private static java.util.Date date = new java.util.Date();
   private static java.text.SimpleDateFormat formatter;
   private int counter = 1;
%>

<%!
   /*
    * Innerhalb von JSPs können Sie sogar Klassenmethoden (static)
    * deklarieren.
    */
   private static String getStartDate() {

      // Wurde der Formatter schon initialisiert?!
      if (formatter == null) {
         formatter = new java.text.SimpleDateFormat();
      }
      return formatter.format(date);
   }
%>
<html>
   <body style="text-align:center; color:green">
      Sie sind der <%= counter++ %>. Besucher dieser JSP. <br/>
      Start des Webservers am: <%= getStartDate() %>
   </body>
</html>
```

Anhand dieses Listings können Sie viele Eigenschaften von JSP-Deklarationen nachvollziehen:

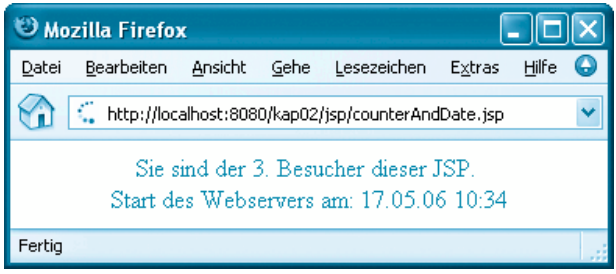
- Es sind mehrere Deklarationsblöcke (`<!% ... %>`) pro JSP zugelassen.
- JSP-Deklarationen gestatten es Ihnen, neben der Deklaration von Instanzvariablen (Member-Variablen oder nicht statischen Variablen) auch die Definition von Klassenvariablen und Klassenmethoden (`static`).
- Es ist möglich, in einer JSP-Deklaration mehrere Variablen und Methoden gemeinsam zu deklarieren.
- Es ist möglich, in JSP-Deklarationen auf Variablen und Methoden zuzugreifen, welche in anderen JSP-Deklarationen definiert werden.

- Innerhalb von JSP-Ausdrücken und -Scriptlets können Sie deklarierte Methoden wie in Java aufrufen und auf globale Variablen zugreifen. Dabei ist es im Gegensatz zu JSP-Ausdrücken/Scriptlets egal, ob die entsprechenden JSP-Deklarationen vor oder nach dem JSP-Ausdruck stehen.

Folgende Dinge können Sie in JSP-Deklarationen allerdings *nicht*:

- Während Anweisungsfolgen in JSP-Scriptlets über mehrere Blöcke verteilt sein können (Listing 2.25), müssen JSP-Deklarationen immer abgeschlossen sein. Das heißt, dass Methoden- und Variablendeklarationen immer vollständig in *einer* Deklaration enthalten sein müssen und nicht in einer anderen fortgeführt werden können.
- Während Sie in JSP-Ausdrücken und -Scriptlets auf Variablen und Methoden in JSP-Deklarationen zugreifen können, ist dies umgekehrt nicht möglich. In JSP-Deklarationen stehen Ihnen *nur* die Variablen und Methoden zur Verfügung, welche in JSP-Deklarationen definiert oder als Methodenparameter übergeben werden.

Abbildung 2.12
Zugriffszähler mit formatierter Ausgabe des Datums (counterAndDate.jsp)



2.3.5 Vergleich: Ausdrücke, Scriptlets und Deklarationen

In diesem Abschnitt werden die bisher behandelten Bestandteile von JSPs noch einmal miteinander verglichen:

Tabelle 2.1
Vergleich zwischen Ausdrücken, Scriptlets und Deklarationen

	JSP-Ausdruck	JSP-Scriptlet	JSP-Deklaration
Deklaration	<%= ... %>	<% ... %>	<%! ... %>
Platzierung in der JSP	Beliebig zwischen HTML-Code platzierbar.	Beliebig zwischen HTML-Code platzierbar.	Beliebig zwischen HTML-Code platzierbar.
Ausgabe in die JSP	Resultat der Anweisung wird direkt in die Seite integriert.	Ausgaben in die resultierende Seite erfolgen über die Variable out.	Ausgaben in die resultierende Seite sind *nicht* möglich.

	JSP-Ausdruck	JSP-Scriptlet	JSP-Deklaration
Mögliche Anweisungen	Auf eine Ausgabeanweisung beschränkt.	Anweisungsfolgen möglich.	Vollständige Deklaration von Methoden möglich.
Abschluss einer Anweisung	Anweisung wird nicht abgeschlossen.	Alle Anweisungen werden von einem Semikolon (;) abgeschlossen.	Alle Anweisungen werden von einem Semikolon (;) abgeschlossen. Methoden müssen vollständig sein.
Definition von Variablen	Ist nicht möglich.	Lokale Variablen möglich.	Globale Variablen möglich. Statische wie Member-Variablen.
Definition von Methoden	Ist nicht möglich.	Ist nicht möglich.	Ist möglich.
Zugriff auf Methoden und Variablen anderer Blöcke	Ist möglich.	Ist möglich.	Auf Methoden/Variablen anderer JSP-Deklarationen beschränkt.
Variablen anderer Blöcke	Müssen vorher deklariert sein, wenn diese in JSP-Scriptlets stehen.	Müssen vorher deklariert sein, wenn diese in JSP-Scriptlets stehen.	Können an beliebiger Stelle deklariert sein.

Tabelle 2.1 (Forts.) Vergleich zwischen Ausdrücken, Scriptlets und Deklarationen

2.3.6 Kommentare

Kommentare sind Gedächtnisstützen für Programmierer und sollen den sie umgebenden Quellcode beschreiben. Dabei muss jeder Programmierer für sich selbst entscheiden, wie viel Kommentar er – und möglicherweise auch ein anderer an diesem Projekt beteiligter Entwickler – benötigt, um den von ihm geschriebenen Code auch zwei Monate später noch zu verstehen. Nur ganz auf Kommentare verzichten sollten Sie nicht.

Auch Markup-Sprachen wie HTML unterstützen Kommentare, die hier folgende Form haben:

```
<!-- ... Dies ist ein HTML-Kommentarkasten ... -->
```

Listing 2.30 Kommentare in HTML

Wenn Sie diese Kommentarkästen in Ihren JSPs verwenden, werden die Kommentare zwar nicht unmittelbar vom Browser angezeigt, sie sind aber dennoch Bestandteil der übertragenen HTML-Seite. Das heißt:

- Sie benötigen Netzressourcen, weil sie mit übertragen werden müssen.
- Sie lassen sich über die Quelltextfunktion Ihres Browsers (SEITENQUELLTEXT ANZEIGEN) sichtbar machen.

Da besonders der zweite Punkt Rückschlüsse auf die interne Funktionsweise Ihrer Webanwendung zulässt und damit unter Sicherheitsaspekten bedenklich ist, sollten Sie auf diese Form der Kommentare verzichten und stattdessen lieber JSP-Kommentare verwenden.

Listing 2.31
Ein JSP-Kommentar

```
<%-- ... Dies ist ein JSP-Kommentarkasten ... --%>
```

Achtung

Innerhalb der Tags <% und %> gelten jedoch die Regeln der Java-Welt. Hier haben Kommentare weiterhin die Form (siehe auch Listing 2.29):

Listing 2.32:
Kommentare innerhalb von JSP-Scriptlets und -Deklarationen

```
// Einzeiliger Javakommentar innerhalb von Scriptlets/Deklarationen

/*
 * Mehrzeiliger Javakommentar innerhalb von JSP-Scriptlets oder
 * JSP-Deklarationen
 */
```

Diese Kommentarkästen werden vor der Auslieferung der Seite vom Servlet-Container entfernt und nicht mit übertragen. Sie sind somit kein Bestandteil des resultierenden Dokuments und dies reduziert die Netzlast.

Tipp

Alle spezifischen Bausteine Ihrer JSP werden von den Symbolen <% und %> umschlossen. Sie dienen dazu, den integrierten Java-Quelltext vom restlichen Markup zu unterscheiden, und werden in jedem Fall vom Servlet-Container interpretiert.
Sollen die reservierten Symbole <% und %> jedoch auf Ihren HTML-Seiten erscheinen – etwa um eine Dokumentation über JSPs zu schreiben –, so verwenden Sie stattdessen die Schreibweisen <\% und \%>. Dieses Vorgehen bezeichnet man auch als Maskieren.

2.3.7 Entscheidungen und Schleifen

Egal, ob Sie nun Applets, Webserver oder JSPs implementieren; es gibt klassische Strukturen objektorientierter Sprachen, die Sie immer wieder benötigen: *Entscheidungen* und *Schleifen*. Dieser Abschnitt zeigt Ihnen an einfachen Beispielen, wie Sie beide in Ihre JSPs integrieren und ist gleichzeitig ein guter Selbsttest für das bisher vermittelte Wissen über JSP-Ausdrücke –Scriptlets und Deklarationen.

Die if-else-Anweisung

Ohne die Möglichkeit, den Programmablauf durch Bedingungen zur Laufzeit zu beeinflussen, beschränken sich Ihre Programme auf das bloße Abarbeiten von Scripts. Die einfachste Möglichkeit, Entscheidungen zu implementieren, sind *if-else-Anweisungen*:

```
<%!
   /* Diese Methode gibt sonntags 'true' zurück  */
   private static boolean isSunday() {
      java.util.Calendar day = java.util.Calendar.getInstance();
      if (day.get(day.DAY_OF_WEEK) == day.SUNDAY) {
         return true;
      }
      return false;
   }
%>
<html>
   <body style="text-align:center; color:green">
      <%--
           Abhängig vom Wochentag ändert diese JSP ihr Verhalten
      --%>
      <% if (isSunday()) { %>
         Es ist Sonntag, dass heißt Brötchen und Cappuccino :-)
      <% } else { %>
         Kein Sonntag, der Cappuccino muss noch warten :-(
      <% } %>
   </body>
</html>
```

Listing 2.33
Beispiel für eine if-else-Anweisung (isSunday.jsp)

Zunächst deklarieren Sie die Methode `isSunday()`, die ermittelt, ob zufälligerweise gerade Sonntag ist. Anschließend definieren Sie Ihre HTML-Seite und integrieren abhängig vom Rückgabewert der Methode die eine oder die andere Information.

if-else-Entscheidungen in JSPs haben folgende Form, wobei der else-Zweig bei Bedarf natürlich auch entfallen darf:

```
<% if (Bedingung) { %>
   ... HTML oder JSP-Anweisungen wenn Bedingung erfüllt ist.
<% } else { %>
   ... HTML oder JSP-Anweisungen wenn Bedingung nicht erfüllt ist.
<% } %>
```

Listing 2.34
Allgemeine Form von if-else-Entscheidungen

Achtung

Achten Sie darauf, dass alle öffnenden geschweiften Klammern ({) stets wieder korrekt geschlossen werden (}). Sonst erhalten Sie beim Aufruf der JSP eine Fehlermeldung.

Dabei kann sowohl der `if`- als auch der `else`-Zweig neben HTML-Code natürlich auch weitere JSP-Anweisungen (Scriptlets, Ausdrücke) enthalten. Diese werden allerdings nur ausgeführt, wenn dabei der entsprechende Zweig durchlaufen wird. Sie können – genau wie in Java-Programmen – auch mehrere `if-else`-Entscheidungen ineinander schachteln, sofern alle öffnenden Zweige ({) anschließend wieder geschlossen werden (}).

Die switch-Verzweigung

Wenn abhängig von den verschiedenen Werten einer Variable (oder eines Ausdrucks) unterschiedliche Anweisungen ausgeführt werden sollen, ist eine `switch`-Verzweigung eleganter, als mehrere `if-else`-Entscheidungen

ineinander zu schachteln. switch-Verzweigungen haben in JSPs folgende Form, wobei jeder Zweig auch hier neben HTML-Code weitere JSP-Anweisungen beinhalten kann:

Listing 2.35 Allgemeine Form einer Switch-Verzweigung in JSPs

```
<% switch (Argument) {
     case 0 : %> ...HTML-Markup oder weitere JSP-Anweisungen... <%
     case 1 : %> ...HTML-Markup oder weitere JSP-Anweisungen... <%
       break;
     case 2 : %> ...HTML-Markup oder weitere JSP-Anweisungen... <%
     default: %> ...HTML-Markup oder weitere JSP-Anweisungen... <%
} %>
```

Um den konkreten Wochentag auszugeben, können Sie Listing 2.33 beispielsweise so anpassen:

Listing 2.36 Beispiel einer Switch-Verzweigung (dayOfWeek.jsp)

```
<%
   // Dieses JSP-Scriptlet ermittelt bei jedem Request den
   // aktuellen Wochentag.
   java.util.Calendar today = java.util.Calendar.getInstance();
   int dayOfWeek = today.get(today.DAY_OF_WEEK);
%>
<html>
   <body style="text-align:center; color:green">
      <%-- Ausgabe des Wochentags über Switch-Case-Verteiler --%>
      Heute ist <% switch (dayOfWeek) {
                     case  1 : %> Sonntag. <%
                        break;
                     case  2 : %> Montag. <%
                        break;
                     case  3 : %> Dienstag. <%
                        break;
                     case  4 : %> Mittwoch. <%
                        break;
                     case  5 : %> Donnerstag. <%
                        break;
                     case  6 : %> Freitag. <%
                        break;
                     case  7 : %> Samstag. <%
                  } %>
   </body>
</html>
```

Bei der switch-case-Struktur handelt es sich um eine so genannte Fall-Through-Anweisung. Dass heißt, nachdem der Prozess in den dem Variablenwert entsprechenden case-Zweig gesprungen ist, arbeitet er alle Anweisungen dieses und der nächsten Zweige ab, bis er auf eine break-Anweisung trifft.

Die Verarbeitung fällt also von einem Startpunkt aus gewissermaßen durch alle folgenden case-Zweige bis zum nächsten break durch oder bis die switch-Anweisung beendet ist.

Diesmal verwendet das Beispiel keine JSP-Deklaration, sondern arbeitet – zur Übung – mit einer lokalen Variable (weekday). Je nach Wert der Variablen springen Sie dabei an einen anderen Punkt Ihrer Switch-Verzweigung.

Schleifen

Ein weiteres wichtiges Element zur Verarbeitung von Listen, Mengen und Feldern (Arrays) sind Schleifen. Sie dienen außerdem dazu, Code für eine feste Anzahl von Durchgängen zu wiederholen. Bestimmte Dinge kann man schließlich gar nicht oft genug sagen:

Listing 2.37 Beispiel für eine For-Schleife (carpeDiem.jsp)

```
<html>
   <body style="text-align:center; color:green">
      <%-- Definition einer for-Schleife in JSPs --%>
      <% for (int i = 0; i < 3; i++) { %>
         Carpe Diem: Genieße den Tag ! <br/>
      <% } %>
   </body>
</html>
```

Abbildung 2.13
Resultat der For-Schleife

Die allgemeine Form für `for`-Schleifen ist:

```
<%-- Allgemeine Form einer for-Schleife in JSPs --%>
<% for (Initialisierung; Abbruchbedingung; Inkrement) { %>
   ...HTML und weitere JSP-Anweisungen...
<% } %>
```

Listing 2.38
Allgemeine Form für For-Schleifen

Und da ich Sie nicht mit Analogien langweilen möchte, werden die `while`- und die `do-while`-Schleifen nur schematisch dargestellt.

```
<%-- Eine While-Schleife ... --%>
<% while (Abbruchbedingung) { %>
   ...HTML und weitere JSP-Anweisungen...
<% } %>

<%-- ... und ihr DoWhile-Pendant --%>
<% do { %>
   ...HTML und weitere JSP-Anweisungen...
<% } while (Abbruchbedingung) %>
```

Listing 2.39
Allgemeine Form für While- und Do-While-Schleifen

2.4 Die vordefinierten Variablen einer JSP

Die ersten Absätze dieses Kapitels haben Ihnen die grundlegenden Bausteine von JSPs näher gebracht und Sie mit der entsprechenden Syntax vertraut gemacht. Bis auf einige Spielereien mit dem Datum und dem Abarbeiten fest definierter Skripte, wie etwa das Ermitteln des Wochentags, waren dabei jedoch nur wenige sinnvolle Anwendungen möglich, weil Ihnen bisher noch ein wichtiges Element fehlt: die Kommunikation mit der Umwelt.

Dieser und die nächsten Absätze machen Sie mit den dafür notwendigen vordefinierten Variablen vertraut und zeigen Ihnen, wie Sie diese verwenden können, um das Verhalten Ihrer JSPs auf den jeweiligen Request anzupassen.

Eigentlich haben Sie schon mit vordefinierten Variablen gearbeitet! Denn in einigen der vorangehenden Beispiele haben Sie bereits auf die Variable `out` zugegriffen, um Ausgaben in JSP-Scriptlets zu erzeugen (Listing 2.24).

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 vordefinierte Variablen und ihr Verhalten zu verstehen, erweitern wir zunächst unser Wissen um das Internet, HTTP und einige damit verbundene Konventionen.

2.4.1 Die verschiedenen Kontexte des Servlet-Containers

Wie Sie bereits wissen, ist HTTP ein *zustandsloses Protokoll (Stateless Protocol)*. Das bedeutet, dass der Server auf einen eingehenden Request mit einer Response reagiert, um diese und damit auch den Client dann aber sofort wieder zu »vergessen«. 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.

Info

HTTP ist ein zustandsloses Protokoll, bei dem ein Server nur den aktuellen Client erkennt und anschließend sofort wieder vergisst. Es existieren jedoch verschiedene Techniken, die eine Identifikation über mehrere Requests hinweg ermöglichen.

Info

Jede in Java definierte Variable wird mit einem Gültigkeitsbereich (*Scope*) ausgestattet. Dieser definiert den Rahmen, innerhalb dessen die Information der Variablen gelten soll. So gilt eine lokale Variable so lange, wie Sie sich innerhalb der Methode, in der sie deklariert wurde, befinden. Verlassen Sie den Gültigkeitsbereich einer Variablen, kann der darin enthaltene Wert nicht mehr ausgelesen werden.

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 wiederzuerkennen. Hierfür können Sie sich Ihren Servlet-Container als eine Art Empfangsdame einer großen Firma vorstellen. Diese nimmt alle eingehenden Anfragen von Kunden entgegen, überprüft, ob ein 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 gewissen 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.

Achtung

Alle Informationen in jedem der drei Kontexte werden unter einem symbolischen Namen abgespeichert, unter dem sie auch wieder ausgelesen werden können. Speichern Sie ein neues Objekt unter einem bereits vergebenen Namen, wird das ursprüngliche Objekt überschrieben und kann dementsprechend nicht mehr ausgelesen werden.

Der Application-Scope

Auf dem ersten der Notizblöcke steht Applikation (`application`). 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 Webapplikation) oder etwa die Buchhaltung, womit natürlich in diesem Fall beispielsweise die Datenbank gemeint sein kann.

Info

Viele Server bieten auch die Funktionalität des Re-Initialisierens einer Webanwendung ohne Neustart, was insbesondere dann von Vorteil ist, wenn der Server mehrere Anwendungen parallel beherbergt. Natürlich wird der Application-Scope dieser Anwendung auch in diesem Fall erneuert.

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 sie ihn, ob er bereits eine Kundennummer besitzt. Verneint der Kunde dies, 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.

Info

Als Session-Timeout wird die Zeitspanne bezeichnet, innerhalb der ein Client den nächsten Request senden muss. Wird diese Zeitspanne überschritten, werden die gespeicherten Informationen verworfen. Das tatsächliche Session-Timeout wird vom Administrator konfiguriert und ist von Server zu Server unterschiedlich.

Achtung

Die Session-ID wird in der Regel über ein Cookie weitergegeben. Sie ist damit an den Browser gebunden und 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, bleibt Ihre Session-ID erhalten.
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.

Besitzt ein Client bereits eine eindeutige Kundennummer und ist das zugehörige Blatt im Notizblock noch 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 zugewiesen.

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: Auch Sie als JSP-Entwickler können ebenfalls in Ihren JSPs Informationen, die Sie benötigen, auf jedem der drei Blöcke hinterlegen und so beispielsweise den Inhalt eines Warenkorbs auf dem Session-Block festhalten. Beim nächsten Request können Sie diese Information dann auslesen und weiterverwenden.

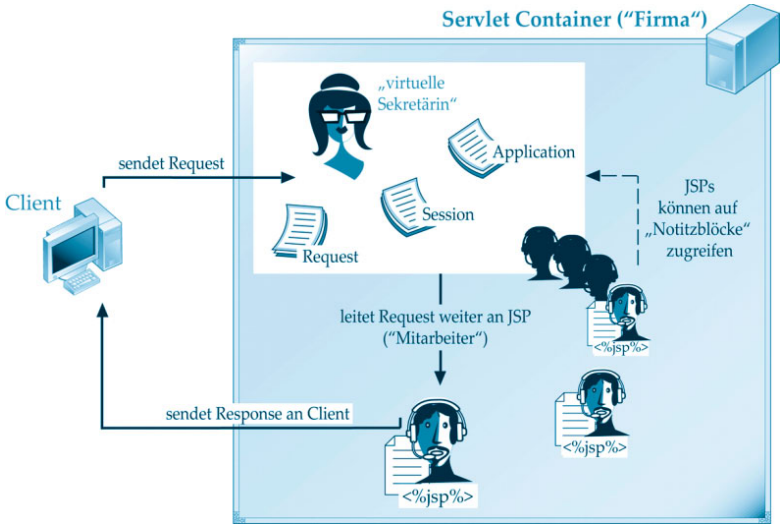


Abbildung 2.14
Die verschiedenen Kontexte eines Servlet-Containers

2.4.2 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 aufgrund des beschränkten Umfangs dieses Buchs in den folgenden Abschnitten nur eine Einführung in die Verwendung vordefinierter Variablen geben und nicht alle Möglichkeiten beleuchten, die sie bieten.

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

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

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

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

- `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`)

  Diese Variable entspricht dem Sitzungsblock der virtuellen Sekretärin. 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.

Achtung

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 Timeout automatisch aus dem Speicher des Servers gelöscht.

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

- `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 Request 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 Schlüsselworts `this` in Java und verweist auf Ihr JSP-Objekt.

2.5 Das Auslesen des HTTP-Request

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

2.5.1 Request-Parameter

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

Info

Request-Parameter können auch direkt in einen Link eingebunden werden (siehe Listing 2.2).

Listing 2.40
Ein dynamisches HTML-Formular (readUserParam.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="readUserParam.jsp">

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

         <%--
              Eingabe eines neuen Wertes für den Request-Parameter;
              Dieser wird beim nächsten Request ausgelesen.
         --%>
         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.

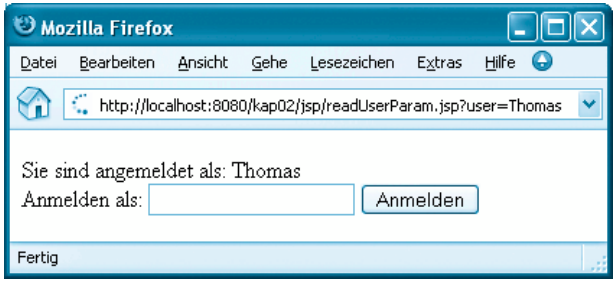
Tipp

Mehr zum Thema HTML und Formulare finden Sie unter *http://de.selfhtml.org/html/formulare/*.

```
...
   <form action="readUserParam.jsp">
...
```

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

Abbildung 2.15
Ein dynamisches HTML-Formular



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

Listing 2.41
Vorbelegen von Input-Feldern

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

Hierbei verwenden Sie den Wert der zuvor deklarierten Variable als Attributparameter für `value`.

2.5.2 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 werden, welchen Wert Sie auslesen möchten, beschränken sich die Anwendungen in den meisten Fällen auf das Sammeln von Debug-Informationen, wie es das folgende Listing demonstriert:

Listing 2.42
Ausgabe aller Request-Parameter (readAllParams.jsp)

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

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

Info

Eine Einführung in Request-Header und Request-Parameter finden Sie in Absatz 2.1.2.

Achtung

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

2.5.3 Auslesen von Request-Headern

Die vordefinierte Variable `request` vom Typ `HttpServletRequest` stellt eine Reihe von interessanten Methoden bereit, die Ihnen folgende Dinge ermöglichen:

- Die URL und den Port des Webservers oder Clients identifizieren.
- Die Version des HTTP-Protokolls auslesen.
- Überprüfen, ob der aktuelle Request über eine sichere Verbindung erfolgt.

Doch um den geneigten Leser nicht zu langweilen, werden wir nicht alle Methoden bis ins kleinste Detail ausloten. Bei Interesse empfehle ich Ihnen die Lektüre der dem SDK beiliegenden JavaDocs und den Mut, diese einfach selbst zu testen. Hierfür müssen Sie nichts weiter tun, als die entsprechende Methode der Variablen `request` in einer JSP aufzurufen und das Ergebnis über einen JSP-Ausdruck auszugeben.

Bevor sich der folgende Abschnitt mit den JSP-Direktiven beschäftigt, zeigen Ihnen die nächsten Beispiele, wie Sie analog zu den Request-Parametern auch die Request-Header auslesen können.

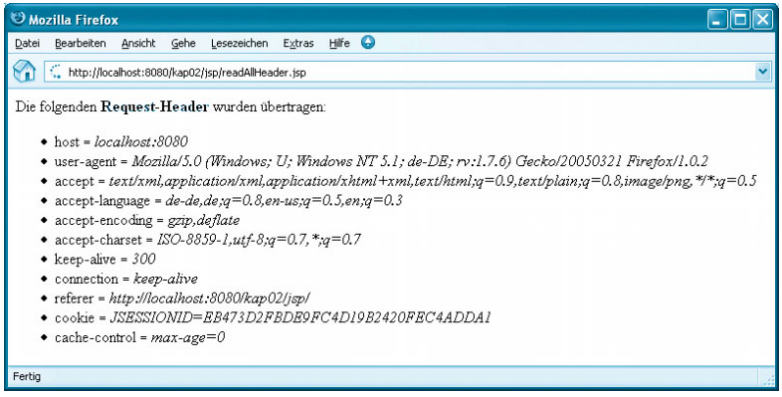
Listing 2.43
Auslesen aller Request-Header (readAllHeader.jsp)

```
<%
   // Auslesen aller übermittelten Request-Header
   java.util.Enumeration headerNames = request.getHeaderNames();
%>
<html>
   <body>

      <%-- Ausgabe aller Request-Header zwecks Debugging --%>
      Die folgenden Request-Header wurden übertragen:
      <ul>
        <%
          while (headerNames.hasMoreElements()) {
            String headerName = (String) headerNames.nextElement();
        %>
```

Listing 2.43 (Forts.) Auslesen aller Request-Header (readAllHeader.jsp)

```
            <li>
              <%= headerName %> = <%= request.getHeader(headerName) %>
            </li>
          <% } %>
        </ul>
    </body>
</html>
```

Abbildung 2.16 Wie geschwätzig ist Ihr Browser? Ausgabe der HTTP-Header



Keine Panik, für den erfolgreichen Einsatz von JSPs müssen Sie nicht alle HTTP-Header kennen, wenngleich sie Ihnen sicherlich interessante Optimierungsansätze bieten. Sie werden später immer mal 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.

Info

Der Asterisk (*) ist im HTTP-Protokoll ein Platzhalter, der jeden beliebigen Wert annehmen kann. Der optionale Parameter q einiger Request-Header gibt hingegen die akzeptierte Qualität der jeweiligen Ressource zwischen 0 (sehr niedrig) bis 1 (sehr hoch) an.

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

Listing 2.44 Ermitteln aller übermittelten Request-Header

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

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 allerdings mehr als einen Wert, wie etwa `Accept` oder `Accept-Language` in Abbildung 2.16. Damit Sie auch in diesem Fall alle gesetzten Werte auslesen können, hält das `HttpServletRequest`-Objekt die Methode `getHeaders()` bereit, welche ebenfalls eine `Enumeration` zurückgibt – nur diesmal eben mit den Werten.

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

Listing 2.45
Auslesen von Headern mit mehreren Werten

Mehr zur vordefinierten Variable `request` vom Typ `javax.servlet.http.HttpServletRequest` finden Sie in den JavaDocs von Sun.

Achtung

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

2.5.4 Besonders häufig verwendete Request-Header

Abschließend sei schließlich noch erwähnt, dass einige Request-Header so häufig verwendet werden, dass sie über eigene Methoden ausgelesen werden können. Die Verwendung dieser Methoden anstelle von `getHeader()` macht Ihren Java-Code lesbarer, weil die Bedeutung der Methoden aus ihrem Namen ersichtlich wird und die Intention des Programmierers leichter nachvollzogen werden kann. Zu diesen Request-Headern gehören:

- `getContentType()`
  Über den Header, den diese Methode auswertet, 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.

Tipp

Dem Content Type oder auch MIME-Type und den angesprochenen Cookies werden Sie in den nächsten Abschnitten und Kapiteln noch begegnen.

- `getCookies()`
  Cookies (dt. Kekse) sind Name-Wert-Paare, über die Sie praktisch eigene Header erstellen können. Diese werden in der Regel initial 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 zu ihren Benutzern oder in Chat-Räumen verwendet.
- `getLocale()` und `getLocales()`

  Viele Anwender bevorzugen Dokumente in ihrer jeweiligen 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 auswählen.

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

2.6 Direktiven – Eigenschaften einer JSP

In diesem Abschnitt geht es um die Eigenschaften Ihrer JSPs. Sie werden lernen, wie Sie Java Packages importieren, den Typ Ihres Dokuments festlegen und auf Fehler reagieren. Die Direktiven 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 Imports, Vererbung oder die Pufferung im Speicher. Die Seitendirektive `page` hat dabei übrigens nichts mit der gleichnamigen vordefinierten Variablen zu tun. Im Unterschied zu dieser werden Sie die Direktive bald zu schätzen wissen.
- `include`
  Diese Direktive ermöglicht es Ihnen, die JSP als zusammengesetztes Dokument zu beschreiben, welches modular aus eigenständigen Teildokumenten aufgebaut sein kann. So können Sie beispielsweise gemeinsam genutzte Komponenten, wie Banner oder Menüleisten in eigenständige JSPs auslagern.
- `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 (Tag Bibliotheken) gewidmet wird.

Alle drei Direktiven haben stets die gleiche Form:

Listing 2.46 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. Zum Beispiel:

Listing 2.47 Beispiele für JSP-Direktiven

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

2.6.1 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 somit prinzipiell überall in der JSP platziert werden. Da einige Attribute 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.

Info

Direktiven werden unabhängig vom Javacode in JSP-Ausdrücken, -Scriptlets oder -Deklarationen interpretiert.

Mögliche Attribute sind dabei `import`, `extends`, `isThreadSafe`, `session`, `buffer`, `autoflush`, `contentType`, `pageEncoding`, `isELIgnored`, `info`, `errorPage`, `isErrorPage` und `language`. Die Wichtigste ihrer Funktionen werden Sie auf den nächsten Seiten kennen lernen.

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.

Info

Natürlich gibt es in Java keine echten Imports von Klassen, wie beispielsweise bei einem Include. Stattdessen gestattet es diese Anweisung für Klassen, nur den verkürzten (z.B. `Date`) und nicht den voll qualifizierenden Namen (z.B. `java.util.Date`) anzugeben. Diese missverständliche Nomenklatur hat Sun allerdings selbst eingebaut.

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

Listing 2.48
Import java.util.Date

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

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

Listing 2.49
Beispiele für den Klassen-Import

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

Benutzersitzungen vermeiden

Das Session-Objekt, das Sie im nachfolgenden Abschnitt genauer kennen lernen werden, ist dafür gedacht, benutzerspezifische Daten von einem Request bis zum nächsten zu speichern. Doch wenn Sie diese Daten gar nicht brauchen, können Sie sich die dafür benötigten Ressourcen des Servlet-Containers auch sparen:

Listing 2.50
Vermeiden von Sitzungen

```
<%-- Direktive um die Unterstützung von Sessions zu vermeiden --%>
<%@ page session="false" %>
```

Achtung

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

Tipp

Prinzipiell können page-Direktiven an jeder beliebigen Stelle innerhalb der JSP platziert werden. Um die Lesbarkeit Ihrer JSPs zu erhöhen, empfehle ich Ihnen jedoch, diese am Anfang der JSP zusammenzufassen.

Hierdurch teilen Sie dem Servlet-Container mit, dass *diese JSP* nicht an Sitzungen teilnimmt und keinen Zugriff auf die vordefinierte Variable `session` benötigt. Der Servlet-Container 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.

Verschiedene Dokumenttypen erzeugen

Bisher haben Sie mit Ihren JSPs reine 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-Header übermittelt und kann von Ihnen z.B. auch über die vordefinierte Variable `response` gesetzt werden.

Info

Unter einem Plug-in (dt.: einstöpseln) versteht man ein kleines Hilfsprogramm, das zur Laufzeit in eine bestehende Applikation eingefügt wird, um deren Funktionen zu erweitern.

Listing 2.51
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 (z.B. HTML), den *MIME-Type*, mit. Anhand dieser Information entscheidet der Browser, wie er das Dokument rendert und ob

zur Darstellung etwa ein Plug-in wie der Acrobat Reader benötigt wird. Das folgende Listing zeigt einige häufig verwendete MIME-Typen. Eine (nahezu) vollständige Liste finden Sie z.B. unter *http://de.selfhtml.org/diverses/mimetypen.htm*.

Tipp

Wenn Sie Lust haben, entwerfen Sie doch einmal eine HTML-Tabelle und staffieren Sie diese mit dem Excel-MIME-Type (Listing 2.52) aus. Seien Sie gespannt!

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

Listing 2.52
Einige häufig verwendete MIME-Typen

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

```
<%-- 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>
```

Listing 2.53
Dynamisches XML per JavaServer Page (dynamicXML.jsp)

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

Achtung

Abbildung 2.17 belegt auch, dass der Browser die Art der Darstellung nur über den Mime-Type bestimmt, denn obwohl Ihr Dokument ebenfalls die Endung *.jsp* hat, wird es ganz anders dargestellt.

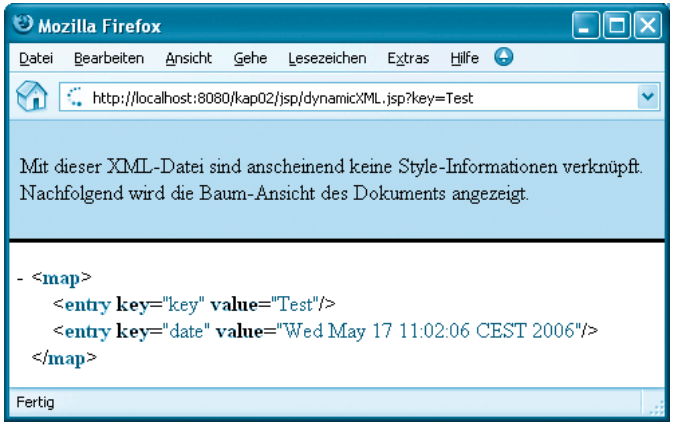


Abbildung 2.17
Ein dynamisch erzeugtes XML-Dokument

Der Response-Header `contentType` darf natürlich nur ein *einziges Mal* gesetzt und *muss* (wie alle anderen Header) vor dem Inhalt des Dokuments übermittelt werden. Mehr zum Aufbau und zur Übertragung einer HTTP-Response finden Sie in Abschnitt 2.1.3.

Einstellen des zu verwendenden Zeichensatzes

Neben dem Dokumenttyp legt die Page-Direktive `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« (ê) gerendert. Mit dem Zeichensatz für kyrillische Buchstaben (ISO-8859-5) hingegen würde die gleiche Folge als »kleines Härtezeichen« (ъ) 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 2.54 Definition des zu verwendenden Zeichensatzes (schematisch)

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

oder konkret:

Listing 2.55 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. So entstand das *Unicode-System* (Abkürzung UTF-8), das auch von Java intern verwendet wird. In Ihren JSPs sollten Sie also nach Möglichkeit UTF-8 als Zeichensatz verwenden.

Listing 2.56 Zeichensatz für JSP-Entwickler

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

Info

Der Standardzeichensatz, wenn Sie keine Content-Type-Direktive bestimmen, ist übrigens `text/html` mit dem Zeichensatz `ISO-8859-1` (Latin 1).

Festlegen der Fehlerseite

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

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 User) in der Regel nichts mit der daraus resultierenden Fehlermeldung anfangen

kann, sollten Sie ihn über die Page-Direktive `errorPage` auf eine von Ihnen bereitgestellte Fehlerseite weiterleiten.

Listing 2.57
Festlegen einer Fehlerseite

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

Die Definition der Direktive `errorPage` bringt den Servlet-Container dazu, im Fall der Fälle statt der ursprünglichen, fehlerhaften Seite auf die unter dem relativen URL `errorPage` angegeben Seite weiterzuleiten (*forward*).

Tipp

Die zusätzliche vordefinierte Variable `exception` ist vom Typ `java.lang.Throwable`.

Diese sollte dann wiederum die Page-Direktive:

Listing 2.58
Deklarieren einer JSP als Fehlerseite

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

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

Listing 2.59
Seite mit möglichem Fehler (produceError.jsp)

```
<%-- 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>
```

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.

Abbildung 2.18
Seite mit potenziellem Fehler

Listing 2.60
Fehlerbehandlung (handleError.jsp)

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

<%-- Zur einfacheren 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/>

      <%-- Zur Ausgabe des Stacktrace in die JSP verwenden Sie
           auch hier die vordefinierte Variable 'out'. --%>
      Der Stacktrace lautet:
      <% exception.printStackTrace(new PrintWriter(out)); %>

   </body>
</html>
```

Info

In echten Webanwendungen sollte die Verwendung der `errorPage`-Direktive natürlich nur eine letzte Fallback-Lösung sein, um Ihre Anwender nicht mit unvorhergesehenen Ausnahmen zu konfrontieren. Wenn Sie auf potenziell fehleranfälligen Code stoßen, sollten Sie diesen in einem `try-catch`-Statement kapseln und den Request im Fall der Fälle selbstständig auf eine Fehlerseite weiterleiten. Dies macht Ihre JSP flexibel und ist sauberer programmiert.

Achtung

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

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.

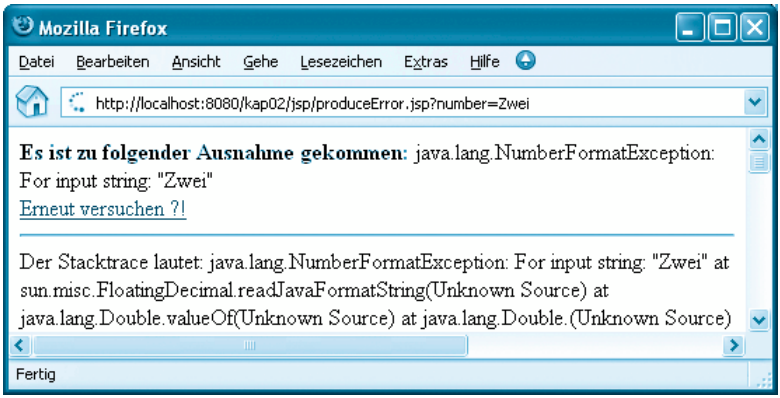


Abbildung 2.19
Ein abgefangener Fehler

Abbildung 2.19 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.

2.6.2 Einbinden von Dateien, die Direktive include

Dieser Abschnitt zeigt Ihnen verschiedene Techniken, mit denen Sie externe Ressourcen in Ihre JSPs integrieren oder auf diese weiterleiten können. Bei diesen externen Ressourcen kann es sich um HTML-Dokumente oder sogar um andere JSPs handeln.

Include per Direktive

Wie Sie wissen, wird der Java-Code Ihrer JSPs vor der Ausführung vom Servlet-Container 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 Servlet-Container anzufordern? Genau dazu dient die *Include-Direktive*. Das folgende Listing demonstriert Ihnen dies, indem es die Debug-Seite aus Listing 2.42 integriert.

Listing 2.61
Dauerhaftes Einbinden der Debug-Seiten (includeDirektive.jsp)

```
<html>
   <body style="text-align:center; color:green">
      Diese Seite ist um Debug-Informationen ergänzt.
      Der Code der Debugseite wird dabei fest eingebunden. <hr />

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

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

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

Tipp

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: Mit der aktuellen Konfiguration erkennt der Servlet-Container automatisch, wenn sich Ihre JSPs geändert haben, und kompiliert diese bei Bedarf neu. Dadurch können Sie Ihre JSPs editieren und sich die Änderungen durch Drücken des AKTUALISIEREN-Schalters Ihres Browsers betrachten. In obigem Beispiel wird der Code der eingebundenen Seite jedoch direkt als Kopie eingefügt. Wenn Sie nun das Original (*readAllParams.jsp*) anpassen, müssen Sie den Servlet-Container dazu veranlassen, auch die Seite *includeDirective.jsp* (Listing 2.61) neu zu kompilieren, um die Änderungen zu übernehmen.

Tipp

Um den Servlet-Container dazu zu veranlassen, eine bestimmte JSP erneut zu übersetzen, können Sie diese einfach öffnen und erneut speichern. Unter Linux steht Ihnen außerdem das elegante touch-Kommando zur Verfügung.

Diese Technik eignet sich daher eher für statische Seiten wie Menüs, 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 und so hält die JSP-Spezifikation eine alternative Vorgehensweise bereit.

Tipp

Eine per *jsp:inlcude-Anweisung* eingefügte Seite muss zwar wie HTML aussehen, sie kann aber ebenfalls dynamisch zur Laufzeit erstellt werden. Das heißt, eine JSP, welche für sich genommen im Browser dargestellt werden kann, kann auch zur Laufzeit in andere JSPs eingebunden werden, sie hat eben nur keinen Zugriff auf den sie umgebenden Code. In Kapitel 4 über Tag-Bibliotheken erfahren Sie mehr darüber, wie Sie eigene Tags analog zu denen in Listing 2.62 erstellen können.

Statt die einzufügende Seite direkt hineinzukompilieren, können Sie auch einfach deren Resultat in Ihre Seite integrieren. Dabei schickt sich der Servlet-Container 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 Servlet-Container diese über einen »ganz normalen« Request aufruft:

- Die einzufügende Seite muss »in sich geschlossen« sein. Das heißt, 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 weiteres spezielles Tag:

Listing 2.62
Include per Tag (includeAktion.jsp)

```
<html>
   <body style="text-align:center; color:green">
      Diese Seite ist um Debug-Informationen ergänzt.
      Hierbei wird die Debugseite vom Webserver aufgerufen und
      ihr Resultat dynamisch zur Laufzeit eingefügt. <hr />

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

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

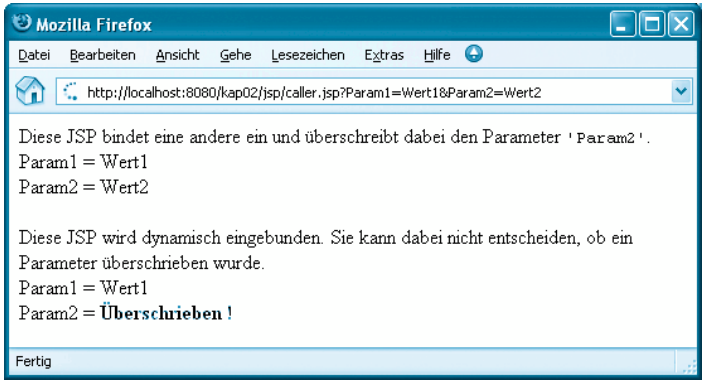
Listing 2.63
Bindet eine JSP dynamisch ein und überschreibt den Parameter Param2 (caller.jsp)

```
<html>
   <body>
      Diese JSP bindet eine andere ein und überschreibt dabei den
      Parameter 'Param2'. <br/>
      Param1 = <%= request.getParameter("Param1") %> <br/>
      Param2 = <%= request.getParameter("Param2") %> <br/>

      <%-- Beim Aufruf der eingebundenen JSP wird der Parameter
           'Param2' überschrieben. --%>
      <jsp:include page="callee.jsp">
         <jsp:param name="Param2" value="Überschrieben !" />
      </jsp:include>

    </body>
</html>
```

Listing 2.64
Wird eingebunden und gibt den Wert der Parameter aus (callee.jsp)

```
<html>
   <body>
      Diese JSP wird dynamisch eingebunden. Sie kann dabei nicht
      entscheiden, ob ein Parameter überschrieben wurde. <br/>
      Param1 = <%= request.getParameter("Param1") %> <br/>
      Param2 = <%= request.getParameter("Param2") %> <br/>
    </body>
</html>
```

Abbildung 2.20
Überschreiben von Request-Parametern



Forward zur Laufzeit

Mit der in diesem Abschnitt vorgestellten Technik können Sie nicht nur fremde Ressourcen einbinden. Es ist auch möglich, den Request auf andere JSPs weiterzuleiten (engl. forward) und das Erzeugen einer Response an diese zu delegieren.

Tipp

Ein weiteres Beispiel für das dynamische Weiterleiten (Forwarding) eines Request zur Laufzeit finden Sie in Listing 2.75.

Auch in diesem Fall können Sie die Parameter des ursprünglichen Request überschreiben.

Listing 2.65
Alternative Definition eines Forward (forward.jsp)

```
<html>
   <body>
      Auf dieser Seite werden Sie weitergeleitet !
      Dieser Text erscheint nicht.

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

2.7 Verwendung der Benutzer-Session

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

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

2.7.1 Was ist die Benutzer-Session?

Die Session (dt. »Sitzung«) ist, wie Sie bereits wissen, ein dem Benutzer zugeordnetes Objekt, in welchem Sie Informationen zu diesem Benutzer ablegen können. Die Session wird Ihnen in Form einer vordefinierten Variablen (`session`) zur Verfügung gestellt.

Info

Eine Einführung in die Benutzer-Session und die Kontexte des Servlet-Containers erhalten Sie in Abschnitt 2.4.1 – Stichwort »virtuelle Sekretärin«.

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. 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.

Info

Cookies werden Ihnen im Servlet-Kapitel (Kapitel 3) wiederbegegnen.

2.7.2 Ein kleines Zahlenspiel

Um die Arbeit mit einer *Session* näher kennen lernen zu können, soll ein kleines Ratespiel implementiert werden, 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 2.66
Ein Ratespiel (numberGuess.jsp)

Listing 2.66 enthält fast alles bisher Gelernte und ist damit eine gute Zusammenfassung und zugleich ein Test für Sie.

```
<%!
   /* 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 des Request-Parameters 'guess', der die letzte
```

Listing 2.66 (Forts.)
Ein Ratespiel (numberGuess.jsp)

```
    // Eingabe des Benutzers - so vorhanden - enthält
    String param = request.getParameter("guess");
    if (param != null) {
       guess = new Integer(param);
    }
%>
<html>
   <body style="color:green">

      <%-- Codieren eines URLs (Siehe 2.7.4) --%>
      <form action='<%=response.encodeURL("numberGuess.jsp")%>'>

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

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

               // Initialisieren der Session-Attribute
               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 des Session-Attributes
                     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>
```

Achtung

Da nicht alle Servlet-Container in der Standardkonfiguration das Autoboxing – also das automatische Umwandeln von primitiven Typen in die Objekt-Pendants (z.B. `int` nach `Integer`) und umgekehrt – unterstützen, haben wir in diesem Beispiel bewusst auf dieses Java 5 Feature verzichtet.

Hilfsmethoden über JSP-Deklarationen

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

Listing 2.67
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);
   }
%>
```

Achtung

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

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

Zugriff auf Request-Parameter

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

Listing 2.68
Auslesen von Request-Parametern

```
...
   // Auslesen des Request-Parameters 'guess', der die letzte
   // Eingabe des Benutzers - so vorhanden - enthält
   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.

2.7.3 Arbeiten mit der Session

Neben Altbekanntem, zeigt Ihnen Listing 2.66 auch, wie Sie Objekte in Form von *Attributen* zu einer Session hinzufügen, aus dieser auslesen und schließlich wieder entfernen:

Listing 2.69
Arbeiten mit dem Session-Kontext

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

// Auslesen eines Session - Attributes
session.getAttribute("SymbolischerNameDesObjektes");

// Entfernen eines Session - Attributes
session.removeAttribute("SymbolischerNameDesObjektes");
```

Auf diese Weise können Sie Benutzerdaten, wie die Objekte eines virtuellen Warenkorbs oder einen 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 2.70 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-Attributes
                     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 Objekte in der Session abgelegt werden. Elementare Typen (`int`, `long`, `double` ...) werden nicht unterstützt. Deshalb arbeitet Listing 2.66 auch mit `Integern`, anstatt mit dem an sich intuitiveren `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 2.71).

Listing 2.71 Typecast nach Auslesen aus der Session

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

Ebenso wie in der Session können Sie auch im oben beschriebenen Request-Objekt 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 Request und verfallen danach.

2.7.4 Kodieren von URLs

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

Die einzelnen Session-Objekte des Servlet-Containers müssen dabei eindeutig zugeordnet werden. Hierfür generiert der Servlet-Container 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 und finden, wenn Sie Abbildung 2.16 nochmals betrachten, dort die folgende Zeile:

Listing 2.72
Beispiel für ein Session-Cookie

```
...
   // Ausgabe des Beispielcookies in Abbildung 2.16
   cookie = JSESSIONID=AC693913193F486B4AA3729AB90CCF3A
...
```

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 Variable `response`.

Listing 2.73
Kodieren eines URLs

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

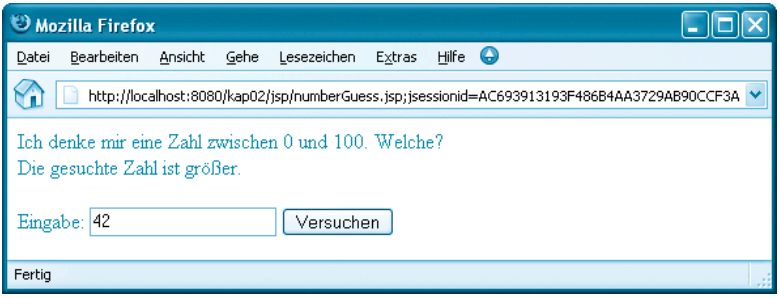
Diese Methode übernimmt als Parameter den zu kodierenden URL und überprüft anschließend, ob die Session-ID über ein Cookie übertragen werden kann oder als Parameter zu übertragen ist.

Info

Die Methode `encodeURL()` stellt sicher, dass die Session-ID in der URL codiert wird, wenn sie nicht über ein Cookie übertragen werden kann.

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 2.21 die Unterstützung für Cookies deaktiviert: Nun wird die Session-ID in Form des Parameters `jsessionid` übertragen.

Abbildung 2.21
Ein kleines Zahlenratespiel



2.8 JSPs und JavaBeans

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

JavaBeans sind laut Sun konfigurierbare Container, welche Variablen speichern (persistent machen) und darauf basierende Geschäftslogik kapseln. Je besser diese Trennung vollzogen wird – JSPs für die Darstellung von Inhalten und JavaBeans für die Speicherung –, desto leichter 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.

Tipp

Im Kapitel über *Persistenz* werden Sie lernen, wie Sie den Zustand Ihrer JavaBeans und damit auch den Zustand Ihrer Anwendung dauerhaft, z.B. in einer Datenbank, speichern.

2.8.1 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 nichtöffentlichen (`private`) Instanzvariable 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.

Info

Um eine Klasse mit einem leeren Konstruktor zu versehen, können wir diesen entweder explizit definieren oder einfach alle Konstruktoren weglassen. Findet Java innerhalb einer Klasse keinen Konstruktor, fügt es automatisch den leeren Konstruktor hinzu.

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

2.8.2 Eine einfache JavaBean

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

Listing 2.74
Eine JavaBean (FormBean.java)

```
package de.akdabas.javaee.beans;
import java.util.List;
import java.util.LinkedList;

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

   /** Der eingegebene Name */
   private String name;
   public String getName() {
      return name;
   }
   public void setName(String aName) {
      name = aName;
   }

   /** Die eingegebene E-Mail-Adresse */
   private String eMail;
   public String getEmail() {
      return eMail;
   }
   public void setEmail(String aAddress) {
      eMail = aAddress;
   }

   /** Möglicherweise aufgetretene Eingabe-Fehler */
   private List<String> errors = new LinkedList<String>();
   public List<String> getErrors() {
      return errors;
   }
```

Listing 2.74 (Forts.) Eine JavaBean (FormBean.java)

```
    /** Signalisiert, ob alle Angaben korrekt sind */
    public boolean isValid() {
       return (errors == null) || (errors.size() == 0);
    }

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

       // Initialisieren der Fehlerliste
       this.errors = new LinkedList<String>();

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

       // Überprüfen der 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.");
       }
    }
}
```

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.

Info

Eine JavaBean ist ein Container und kapselt sowohl Daten als auch Logik. Sie kann ihrerseits wiederum aus anderen JavaBeans aufgebaut sein.

Die Methode `validate()` enthält dabei die so genannte *Geschäftslogik* der JavaBean. In der Geschäftslogik werden die Werte der Properties `name` und `email` überprüft und anhand dieser werden die beiden anderen Attribute, `valid` und `errors`, entsprechend 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. Da der Javacompiler des Servlet-Containers allerdings nur mit JSPs umgehen kann, können JavaBeans nicht mit diesem übersetzt werden, sondern müssen manuell kompiliert werden.

Info

JavaBeans und andere Java-Klassen müssen vor dem Start des Servlet-Containers übersetzt werden.

Hierfür erweitern Sie Ihr Arbeitsverzeichnis zunächst ein weiteres Mal, um die folgenden Ordner.

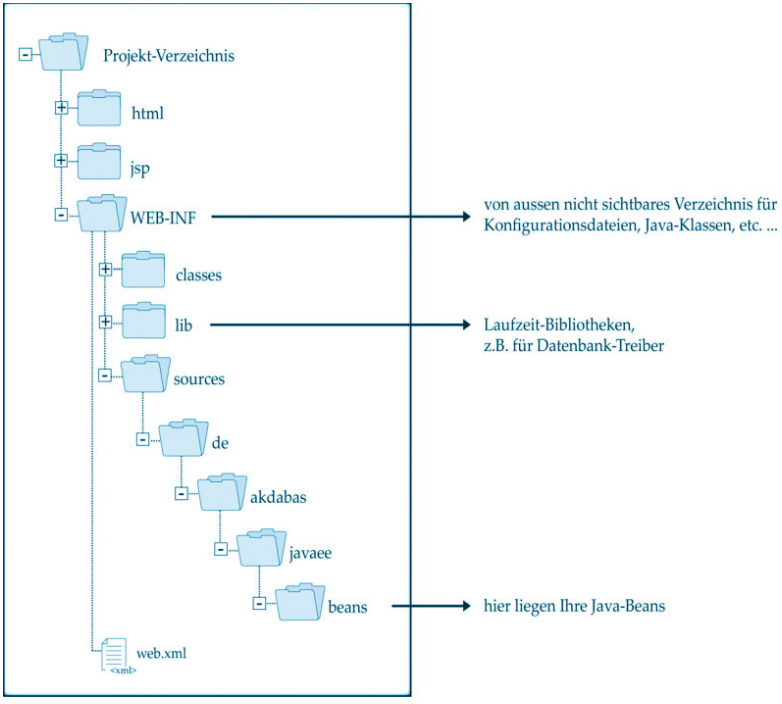


Abbildung 2.22
Erweitern des Arbeitsverzeichnisses für JavaBeans

Achtung

Es ist relativ egal, an welcher Stelle im Verzeichnisbaum Sie die Source-Dateien Ihres Java-Quellcodes ablegen. Damit Ihre Java-Klassen im Nachhinein vom Servlet-Container verwendet werden können, müssen sich die *class*-Dateien jedoch zwingend im Ordner *WEB-INF/classes* befinden.

Nachdem Sie die JavaBean ihrem Package entsprechend im Ordner *beans* abgelegt haben, übersetzen Sie diese ins Verzeichnis *WEB-INF/classes*. Für diesen Zweck verwenden wir in diesem Buch das freie OpenSource-Werkzeug *Ant* (*http://ant.apache.org*), welches Sie auf der beiliegenden CD finden. Dazu kopieren Sie einfach das ebenfalls beiliegende Ant-Script in Form der Datei *build.xml* in das Arbeitsverzeichnis und führen anschlie-

ßend dort den Befehl `ant` aus. Sie können die Datei natürlich ebenso mit jedem anderen Java-Compiler oder auch direkt aus Ihrer Entwicklungsumgebung (IDE) heraus übersetzen.

2.8.3 Eingabe der Daten in die JavaBean

Als Nächstes benötigen Sie eine JSP zur Eingabe und Validierung der Daten. Diese enthält neben JSP-Ausdrücken und -Scriptlets weitere JSP-Tags.

Listing 2.75 JSP zur Eingabe (beanInput.jsp)

Auch in diesem Beispiel wurde auf die neue `for`-Schleife von Java 5 verzichtet, um Kompatibilitätsprobleme mit verschiedenen Servlet-Containern zu vermeiden.

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

<%-- Binden der JavaBean an den symbolischen Name 'form' --%>
<jsp:useBean id="form"
             class="de.akdabas.javaee.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, um die eingegebenen Daten zu prüfen
   form.validate();

   // Weiterleiten des Requestes, wenn die Daten gültig sind
   if (form.isValid()) {
      response.sendRedirect("beanOutput.jsp");
   }
%>
<html>
   <body>

      <%-- Kodieren des URLs (Siehe 2.7.4) --%>
      <form action='<%= response.encodeURL("beanInput.jsp") %>'>
         Bitte füllen Sie das folgende Formular aus !

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

         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 2.76
Einbinden einer JavaBean in die JSP

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

Hierdurch wird ein Objekt vom Typ `de.akdabas.javaee.beans.FormBean` erzeugt und an den symbolischen Namen `form` gebunden. Dieses Listing ist äquivalent zu folgender Variablendeklaration, kommt aber ohne ein JSP-Scriptlet aus:

Listing 2.77
Äquivalentes Einbinden einer JavaBean

```
<% de.akdabas.javaee.beans.FormBean form =
                    new de.akdabas.javaee.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 Webdesigner effizient zusammenarbeiten: Der Java-Entwickler entwickelt und wartet die JavaBeans, während der Webdesigner das Layout verändern kann, ohne den Java-Code zu beeinflussen.

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

- `id`

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

- `class`

  Mit diesem Attribut geben Sie den voll qualifizierenden Namen der JavaBean-Klasse an.

- `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), `request`, `session` und `application`.

- `type` (optional)

  Dieses Attribut verwenden Sie, wenn Sie Ihre JavaBean über ein Interface oder über eine von ihr implementierte Superklasse 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 2.4.1). 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 2.4 zu den vordefinierte Variablen) gebunden und sind ebenso lange gültig wie die Sitzung, d.h. bis etwa 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 Servlet-Containers oder dessen expliziter Löschung verwendet werden
- request
  Diese Variablen werden für die Dauer eines Request gebunden. Der Unterschied zum Scope page (Default) besteht darin, dass Aktionen wie <jsp:include> und <jsp:forward> als *ein* Request behandelt werden. Diese Variablen stehen daher allen beteiligten JSPs – der ursprünglichen wie der weitergeleiteten – zur Verfügung.

Setzen von Variablenwerten

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

Listing 2.78 Manuelles Setzen von Properties

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

Auch für das *Setzen* von JavaBean-Attributen existiert in der JSP-Spezifikation ein entsprechendes Tag:

Listing 2.79 Setzen von Properties via <jsp:setProperty>

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

   <%-- Setzt den Wert des Attributes '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`
  Enthält den Variablennamen der JavaBean. Die Bean muss zuvor über das Tag `<jsp:useBean>` *oder* ein äquivalentes JSP-Scriptlet erzeugt worden sein.
- `property`
  Enthält den Namen des JavaBean-Attributs – *nicht* den Namen des Setters (kein `set`)! Der übergebene Wert `email` wird vom Tag selbstständig in `setEmail()` umgewandelt.
- `value`
  Enthält den Wert, den das Property annehmen soll. Dabei kann es sich z.B. um eine konstante Zeichenkette oder auch um einen per JSP-Ausdruck eingefügten Request-Parameter handeln.

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

Listing 2.80
Schematische Form des <jsp:setAttribute>-Tag

Setzen aller vorhandenen Request-Parameter

Im obenstehenden 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 zugehörige JavaBean-Property gleichnamig, so können Sie diese durch folgendes Tag synchronisieren:

```
<%-- Setzen aller Requestparameter mit gleichnamigem Attribut --%>
<jsp:setProperty name="SymbolischerNameDerJSP" property="*" />
```

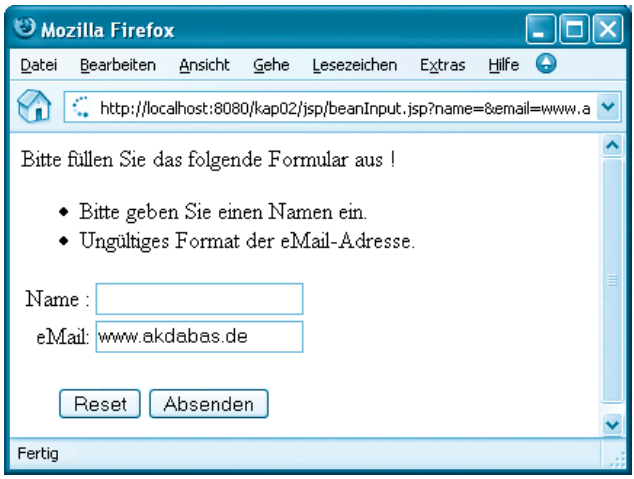
Listing 2.81
Setzen aller vorhandenen Request-Parameter

Das Ergebnis

Die restlichen Elemente aus Listing 2.75 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 im gleichnamigen Kapitel (Kapitel 4) kennen lernen werden.

Nachdem Name und E-Mail-Adresse der JavaBean gesetzt sind, überprüft Listing 2.75 diese über die Methode `isValid()`. Sind alle Werte korrekt, wird der Request an die Seite *beanOutput.jsp* weitergeleitet. Anderenfalls werden die in der JavaBean erzeugten Fehlermeldungen ausgegeben und ermöglichen es so dem Benutzer, diese zu korrigieren.

Abbildung 2.23
Eingabe und Validierung von Parametern



2.8.4 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 natürlich ebenfalls durch ein Tag ermöglicht. Nachdem Sie die JavaBean über ein Tag oder Scriptlet erzeugt und eingebunden haben, können Sie natürlich auch auf die dort vorhandenen Getter zugreifen und diese beispielsweise über einen JSP-Ausdruck ausgeben:

Listing 2.82
Ausgabe von Properties via JSP-Scriptlet

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

Eleganter realisieren Sie dies allerdings mit folgenden Tags:

Listing 2.83
Ausgabe von Properties via JSP-Tag

```
...
   <%-- Ausgabe von JavaBean-Attributes 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 2.75) leiten Sie den Request nach einer erfolgreichen Validierung der Eingaben auf die JSP *beanOutput.jsp*:

Listing 2.84
Ausgabe der Daten einer JavaBean (beanOutput.jsp)

```
<%-- Binden der JavaBean an den symbolischen Name 'form' --%>
<jsp:useBean id="form"
             class="de.akdabas.javaee.beans.FormBean"
             scope="session" />
<html>
   <body>
      Ihre Daten wurden erfolgreich eingegeben !
```

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

Listing 2.84 (Forts.)
Ausgabe der Daten einer JavaBean (beanOutput.jsp)

2.8.5 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 vergeben ist:

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

Die folgenden beiden Code-Fragmente sind dabei vollkommen gleichwertig:

```
...
<%
   de.akdabas.javaee.beans.FormBean form;

   // Überprüfen, ob bereits eine JavaBean in diesem
   // Gültigkeitsbereich existiert
   if (session.getAttribute("form") == null) {
      // Erzeugen eines neuen JavaBean-Objektes
      form = new de.akdabas.javaee.beans.FormBean();
      session.setAttribute("form", form);
   } else {
      // Binden und Casten des vorhandenen JavaBean-Objektes
      Object obj = session.getAttribute("form");
      form = (de.akdabas.javaee.beans.FormBean) obj
   }
%>
...
   <%-- Vollkommen gleichwertig zu obigem Code-Fragment --%>
   <jsp:useBean id="form"
                class="de.akdabas.javaee.beans.FormBean"
                scope="session" />
...
```

Listing 2.85
Manuelles vs. automatisches Erzeugen und Ablegen

Achtung

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. Dieser Mechanismus macht es jedoch sehr umständlich, Initialisierungsparameter zu übergeben. Auch hier unterstützt Sie die Tag-Schreibweise mit einer besonderen Form:

Bisher verwendeten Sie das Tag `<jsp:useBean/>` immer in seiner geschlossenen Form, als leeres Element ohne Rumpf. Der Rumpf hat jedoch die nützliche Eigenschaft, dass er nur dann ausgewertet wird, wenn die JavaBean auch tatsächlich erzeugt wird (`if`-Zweig in Listing 2.85). Um also einen Initialisierungsparameter zu setzen, platzieren Sie einfach das entsprechende `<jsp:setAttribute>` -Tag im Rumpf.

Info

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

Listing 2.86 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 Tag `<jsp:useBean>` kann Ihnen natürlich auch beim Debuggen einer JSP helfen. Um zu testen, wann ein Objekt neu initialisiert und wann es lediglich neu gebunden wird, fügen Sie einfach eine entsprechende Ausgabe in den Rumpf des Tag ein. Diese wird nur beim Erzeugen des Objekts ausgegeben werden.

2.8.6 Vorteile von JavaBeans

Prinzipiell könnten Sie alles bisher Gelernte auch ohne JavaBeans, durch die Verwendung von lokalen Variablen realisieren, die Sie 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 gemeinsam auf diesen Baustein 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 Webdesigner etwa die Darstellung des Internetauftritts 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 dabei 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 Kontakts, in einem gemeinsamen Container verwaltet. Die JavaBean bildet eine logische Einheit und kann in verschiedenen Kontexten wiederverwendet werden. Insbesondere können Sie aus Ihren einfachen JavaBeans immer komplexere zusammensetzen.

2.9 Zusammenfassung

Rekapitulieren wir noch einmal: Dieses Kapitel hat Sie mit der Syntax von JavaServer Pages (JSPs) vertraut gemacht und Ihnen gezeigt, wie Sie Ihre statischen HTML-Seiten durch die Verwendung von JSP-Ausdrücken, -Scriptlets und -Deklarationen in nahezu vollwertige Java-Objekte verwandeln können, aus denen hinterher HTML-Dokumente erzeugt werden.

Hierbei werden Sie vom Servlet-Container mit einer Reihe von vordefinierten Variablen unterstützt, die Ihnen z.B. den Zugriff auf die aktuelle Anfrage (Request), die Benutzersitzung (Session) oder auch die vom Browser verarbeitete Antwort (Response) gestatten.

Sie haben gelernt, den MIME-Type Ihres Dokuments und den zu verwendenden Zeichensatz anzupassen. Mit einfachen Bordmitteln konnten Sie außerdem ein leistungsfähiges Fehlermanagement realisieren.

Den Abschluss bildete schließlich die Vereinigung von JSPs und JavaBeans. Diese Kombination gestattet es Ihnen, Java-Logik in wiederverwendbaren Klassen zu kapseln und sich in den Dokumenten auf die Darstellung zu konzentrieren. Hierdurch werden Ihre JSPs übersichtlicher und Sie erreichen eine saubere Trennung zwischen Darstellung und Geschäftslogik.

2.9.1 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 möchte ich es wagen und

Ihnen zum Abschluss *meine* fünf 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 den Page-Direktiven folgen.
- Im Anschluss an die JSP-Deklarationen sollte ein JSP-Scriptlet folgen, 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 HTML-Markup nur JSP-Ausdrücke bzw. Getter der JavaBeans.
- 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 erforderlichen Mehraufwand beim Erstellen von JSP *und* Bean schnell wieder wett.

3

Servlets

In diesem Kapitel geht es darum, HTTP-Requests mit vollwertigen Java-Klassen – so genannten *Servlets* – zu beantworten. Sie werden lernen, wie Sie Servlets erstellen, übersetzen und in eine Webapplikation einbinden, wo die Unterschiede zwischen JSPs und Servlets liegen und wie Sie dynamische Grafiken erzeugen. Den Abschluss dieses Kapitels bilden dann *Servlet-Filter* und *Session-Listener*, hinter denen sich nützliche Technologien verbergen, die Ihnen das Leben als Programmierer von Webanwendungen drastisch vereinfachen können.

Obwohl Servlets die ältere Technologie darstellen und es sie schon lange vor den JSPs gab, werden Servlets in diesem Buch erst nach den JavaServer Pages behandelt. Hintergrund ist, dass JSPs nach Meinung des Autors leichter zu erlernen sind und einen guten Einstieg in die Erzeugung von dynamischen Webseiten mittels Java bieten. Während es im letzten Kapitel darum ging, HTML-Code mit Java zu verknüpfen, zeigt Ihnen dieses Kapitel nun, wie Sie aufwändige Geschäftslogik aus Ihren JSPs verlagern können, um flexible und erweiterbare Webanwendungen zu erstellen.

3.1 Ein einfaches Servlet

Im vorangegangen Kapitel wurde deutlich, dass der Servlet-Container den Java-Code Ihrer JSPs vor der Auslieferung der Seite zur Ausführung bringt und die dynamischen Inhalte z.B. durch HTML-Elemente ersetzt. Der Client »sieht« daraufhin nichts weiter als das erzeugte Dokument und kann nicht unterscheiden, ob es sich bei der dargestellten Seite um eine statische Ressource handelt oder ob diese dynamisch erzeugt worden ist. Aber was genau passiert eigentlich bei dieser Ausführung des Java-Codes einer JSP?

Tipp

In Kapitel 2 haben Sie bereits gesehen, dass Sie mit JSPs neben HTML selbstverständlich auch andere textbasierte Formate wie XML erzeugen können.

3.1.1 JSPs und Javacode

Wenn Sie schon mit Java gearbeitet haben und beispielsweise mit der Entwicklung von eigenständigen Java-Applikationen vertraut sind, wissen Sie, dass Ihre Java-Anweisungen dabei zunächst in einen Zwischencode übersetzt werden müssen, der anschließend von der Virtual Machine (VM) interpretiert wird. Zur Übersetzung bedienen Sie sich des Kommandozeilen-Tools `javac` oder nutzen gleich die eingebaute Funktionalität Ihrer IDE.

Natürlich muss auch der in Ihren JSPs enthaltene Javacode vor der Ausführung in Bytecode übersetzt werden, doch um die Sache nicht zu verkomplizieren, haben wir uns bei der Beschreibung von JSPs vorerst keine Gedanken darüber gemacht, *wie* der Java-Code übersetzt und zur Ausführung gebracht wird. Das werden wir nun nachholen:

Wenn eine JSP – nachdem sie erstellt oder verändert wurde – zum ersten Mal aufgerufen wird, übersetzt ein Hintergrundprozess des Apache Tomcat diese JSP in eine korrespondierende Java-Datei und kompiliert sie in Java-Bytecode. Bei jener im Hintergrund automatisch erzeugten Java-Klasse handelt es sich um ein Servlet und ein weiterer Prozess des Servlet-Containers – die *Servlet Engine* – bringt sie schließlich zur Ausführung.

Info

Der Servlet-Container wandelt JSPs vor dem ersten Aufruf in Java-Klassen – so genannte Servlets – um und bringt diese anschließend zur Ausführung.

Wenn Sie die Beispiele der JSPs im letzten Kapitel selbst ausprobiert haben, können Sie sich die erzeugten Java-Klassen auch ansehen. Die Verzeichnisse, in denen sich diese befinden, lauten:

Listing 3.1
Verzeichnisse, unter denen die JSP-Klassen erzeugt werden

```
// Arbeitsverzeichnis des JBoss Application Servers
$JBOSS_HOME/server/default/work/jboss.web/localhost/kap02

// Arbeitsverzeichnis des Apache Tomcat Webservers
$TOMCAT_HOME/work/catalina/localhost/kap02
```

Tipp

Je nach Konfiguration des Servers existieren die Arbeitsverzeichnisse nur zur Laufzeit und werden beim Herunterfahren wieder entfernt.

Nur keine Scheu, nachdem Sie die Java-Klassen Ihres Servlet-Containers lokalisiert haben, können Sie diese auch näher in Augenschein nehmen. Die zu Ihrer ersten JSP gehörende Klasse hat beispielsweise den Namen *simple_jsp.java* und sollte etwa folgende Form haben (die Kommentare wurden natürlich erst nachträglich eingefügt, um Sie auf die wichtigsten Stellen hinzuweisen):

Listing 3.2
Ihre erste JSP nach der Übersetzung

```
package org.apache.jsp.jsp;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

public final class simple_jsp extends HttpJspBase {
...
   public void _jspService(HttpServletRequest request,
                           HttpServletResponse response)
      throws java.io.IOException, ServletException {

      JspFactory _jspxFactory = null;
      PageContext pageContext = null;
      HttpSession session = null;
      ServletContext application = null;
      ServletConfig config = null;
      JspWriter out = null;
      Object page = this;

      try {
         _jspxFactory = JspFactory.getDefaultFactory();
         response.setContentType("text/html");
         pageContext = _jspxFactory.getPageContext(...);
         application = pageContext.getServletContext();
         config = pageContext.getServletConfig();
         session = pageContext.getSession();
         out = pageContext.getOut();

         // Hier beginnt der wirklich interessante Teil :-)

         out.write("<html>\r\n");
         out.write("   <body style=\"text-align:center;\">\r\n");
         out.write("    Dies ist eine dynamische JavaServer Page.");
         out.write("    Es ist jetzt genau: ");
         out.print(new java.util.Date());
         out.write("   </body>\r\n");
         out.write("</html>\r\n");

         // Und ?! Haben Sie es wiedererkannt ?

      } catch (Throwable t) {
         if (!(t instanceof SkipPageException)) {
            if (pageContext != null)
               pageContext.handlePageException(t);
         }
      }
   }
}
```

Wenn Sie die etwas ungewöhnliche Klasse `simple_jsp.java` genauer betrachten, entdecken Sie im oberen Teil die Ihnen bereits sehr vertrauten vordefinierten Variablen von JSPs. Im mittleren Teil – dort, wo die vielen `out.write()`-Anweisungen stehen – findet sich dann der vollständige HTML-Code der *simple.jsp*, welcher ausschließlich über die Variable `out` an den Client gesandt wird.

Info

Die in Listing 3.2 dargestellte Methode `service()` ist dabei natürlich nichts weiter als das JSP-Analogon zur `main()`-Methode bei einer eigenständigen Applikation, das Sie in den vorangegangenen Kapiteln kennen gelernt haben.

Und der JSP-Ausdruck, der den Zeitpunkt der Auslieferung dynamisch eingefügt hat? Zur Erinnerung sei hier noch einmal die entsprechende Zeile der JSP angegeben, wie sie in Kapitel 2 definiert wurde:

Listing 3.3
Ihr erster JSP-Ausdruck (simple.jsp)

```
...
      Es ist jetzt genau: <%= new java.util.Date() %>
...
```

Diese Mischung aus HTML-Markup, Text und Java-Anweisung hat in Listing 3.2 folgendes Pendant:

Listing 3.4
Übersetztes Pendant

```
...
        out.write("    Es ist jetzt genau: ");
        out.print(new java.util.Date());
        out.write("   </body>\r\n");
...
```

Info

Der Unterschied zwischen den Methoden `print()` und `write()` eines `java.io.PrintWriter` besteht darin, dass über `print()`-Anweisungen ausgegebene Objekte zunächst durch das serverspezifische Character Encoding – in der Regel UTF8 – umgewandelt und erst dann an die `write()`-Methode übergeben werden. Feste Zeichenketten (Strings) werden bereits bei ihrer Erzeugung im Speicher mit dem eingestellten Character Encoding erstellt und müssen deshalb nicht nochmals kodiert werden, Objekte, die Sie in Zeichenketten umwandeln wollen, eventuell schon. Wenn Sie sich nicht sicher sind, welche Methode Sie verwenden soll, probieren Sie zuerst `print()`.

Wie Sie sehen, enthält das im Hintergrund aus Ihrer JSP erzeugte Servlet alle statischen HTML-Anweisungen in Form von Strings und sendet diese über die Methode `write()` der vordefinierten Variablen `out` an den Client. Der Java-Code ist einfach zwischen diese statischen Anweisungen eingestreut, wobei für JSP-Ausdrücke die Methode `print()` verwendet wird.

3.1.2 In grauer Vorzeit ...

Die Servlet-Spezifikation gab es bereits vor den JSPs! Es zeigte sich jedoch, dass die Oberflächengestaltung oftmals Designern ohne fundiertes Java-Know-how übertragen wurde, was die Entwicklung leistungsfähiger Webanwendungen stark einschränkte.

Um diese Prozesse zu vereinfachen, wurden JSPs als Hilfsmechanismus eingeführt. Dieser verlagert die Erstellung der Java-Klassen in den Hintergrund und gestattet es dem Designer, den Fokus auf die grafischen Elemente und HTML zu legen.

Für anspruchsvollen Java-Code, dessen Blickwinkel mehr auf Seiten der Geschäftslogik liegt, kommen jedoch weiterhin Servlets zum Einsatz. Je nachdem, ob nun also die Darstellung des Inhalts oder die Flexibilität des Java-Codes im Vordergrund stehen soll, wird man eine Komponente der Webanwendung entweder als JSP oder als Servlet realisieren. Listing 3.5 zeigt Ihnen das Servlet-Pendant zu Ihrer ersten JSP:

Listing 3.5
Ein erstes Servlet (SimpleServlet.java)

```
package de.akdabas.jee.servlets;

import java.util.Date;
import java.io.PrintWriter;
import java.io.IOException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletException;

/**
 * Ein einfaches Servlet mit der gleichen Funktionalität wie Ihre
 * erste JSP (simple.jsp).
 */
public class SimpleServlet extends HttpServlet {

   /**
    * Diese Methode ist das Pendant zur Methode _jspService und
    * wird zur Verarbeitung der HTTP-Anfrage gerufen
    */
   public void service(HttpServletRequest request,
                       HttpServletResponse response)
      throws IOException, ServletException {

      // Setzen des zu verwendenden MIME-Types
      response.setContentType("text/html;charset=ISO-8859-1");

      // Binden der (in JSPs vordefinierten) Variable out
      PrintWriter out = response.getWriter();

      // Erzeugen von HTML-Code ...
      out.println("<html><body style=\"text-align:center\">");
      out.println("   Dies ist das Resultat einen Servlets.<br/>");

      // ... mit dynamischen Elementen
      out.println("   Es ist jetzt genau: " + new Date());
      out.println("</body></html>");
   }
}
```

Unterschiede zwischen JSP und Servlet

Im Unterschied zu einem generierten Servlet (Listing 3.2), welches das Interface `HttpJspBase` implementiert, ist dieses von Hand erstellte Exemplar vom Typ `javax.servlet.http.HttpServlet`. Dies bedeutet im Klartext allerdings nichts anderes, als dass die für die Bearbeitung verwendete Methode jetzt `service()` statt `_jspService()` heißt.

Zu Beginn spezifizieren Sie den zu verwendenden MIME-Type und teilen dem Browser den zu verwendenden Zeichensatz mit, denn analog zu JSPs müssen auch bei Servlets alle HTTP-Header vor dem Body an den Client übertragen werden.

Listing 3.6
Setzen des Content Type

```
...
      // Setzen des zu verwendenden MIME-Types
      response.setContentType("text/html;charset=ISO-8859-1");
...
```

Diese Anweisung haben wir bereits in unseren JSPs verwendet, um den Typ des Dokuments erst zur Laufzeit festzulegen.

Nun binden Sie die in JSPs bereits vordefinierte Variable `out` aus dem via Parameter zur Verfügung gestellten `HttpServletResponse`-Objekt. Denn im Gegensatz zu JavaServer Pages müssen Sie in Servlets alle Variablen bis auf den Request und den Response selbst binden.

Listing 3.7
Binden der Variablen out

```
...
      // Binden der (in JSPs vordefinierten) Variable out
      PrintWriter out = response.getWriter();
...
```

Nun können Sie beginnen, Markup zu erzeugen und an den Client zu senden. Dabei sehen Sie an Listing 3.5 noch einmal deutlich, wie aufwändig die Erzeugung von HTML-4.x-konformen Codes unter Java mitunter ist: Denn da Anführungszeichen in Java Beginn und Ende einer Zeichenkette signalisieren, müssen Sie jedes Anführungszeichen Ihrer resultierenden Seite mit einem Backslash maskieren.

Listing 3.8
Maskieren von Anführungszeichen der resultierenden HTML-Seite

```
...
      out.println("<html><body style=\"text-align:center\">");
...
```

Dies kann die Geduld eines Programmierers vor allem bei attributreichen HTML-Seiten auf eine harte Probe stellen.

3.1.3 Einbinden eines Servlet

Um sich das Resultat Ihrer `simple.jsp` anzeigen zu lassen, haben Sie den Servlet-Container konfiguriert und die JSP im Ordner *jsp* abgelegt. Auch Servlets müssen über einen Servlet-Container ausgeliefert werden. Entsprechend können Sie für die folgenden Beispiele entweder das Projekt aus dem vorangegangen Kapitel erweitern oder eine neue Webanwendung konfigurieren.

Tipp

Auch die Servlets aus diesem Kapitel benötigen eine Java-Laufzeitumgebung. Da die verwendeten Verzeichnisstrukturen kompatibel sind, können Sie hierfür entweder den Apache Tomcat erneut konfigurieren oder das Projekt aus Kapitel 2 weiterverwenden.

Wenn Sie Ihr Servlet jedoch analog zur *simple.jsp* in einem Ordner *servlets* ablegen und anschließend über den Servlet-Container abrufen, erhalten Sie das folgende Ergebnis (Abbildung 3.1)

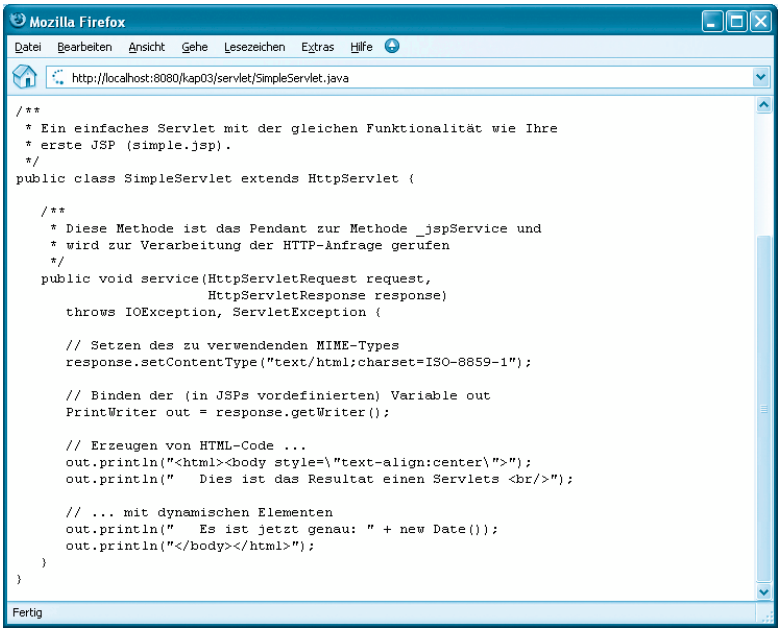


Abbildung 3.1
Direkter Aufruf des Servlet-Quelltexts über einen URL

Offenbar genügt es bei Servlets nicht, diese in einem Unterverzeichnis abzulegen und anschließend aufzurufen. Aufschluss gibt ein kurzer Blick auf die Arbeitsweise des Servlet-Containers beim Aufruf einer JSP:

1. Zunächst überprüft der Servlet-Container, ob sich in seinem Arbeitsverzeichnis (Listing 3.1) eine aus einer JSP erzeugte Java-Klasse befindet, und führt diese gegebenenfalls aus.
2. Existiert keine solche Datei, löst der Servlet-Container über die Einträge der Datei *server.xml* den Pfad der angeforderten Ressource auf und überprüft, ob diese Datei Tags der Form *<% JSP-Code %>* enthält. In diesem Fall übersetzt er die Datei in eine Java-Klasse und führt diese anschließend aus.
3. Enthält die angeforderte Datei hingegen keinen JSP-Markup, liefert der Servlet-Container diese (wie eine HTML-Seite) an den Client aus.

Da Ihr Servlet keine JSP-Tags enthält, »denkt« der Servlet-Container, es handle sich um eine Textdatei, und liefert diese ohne weitere Bearbeitung aus. Auch das Einfügen eines entsprechenden Tag bringt Sie nicht weiter, da Ihre `service()`-Methode nun in die neue Methode der JSP integriert werden würde und `import`-Anweisungen hier beispielsweise nicht erlaubt sind.

Info

Über verschiedene Werkzeuge, wie beispielsweise *Ant*, ist es auch möglich, die JSPs bereits vor der Bereitstellung durch den Servlet-Container manuell zu übersetzen. Dies spart Ressourcen des Webservers und kann Ihnen Kompilierungsfehler bereits vor dem ersten Aufruf der JSP aufzeigen.

Das korrekte Einbinden von Servlets erfordert eine Reihe von Schritten, die auf den nun folgenden Seiten im Einzelnen beschrieben werden, beginnend mit der manuellen Übersetzung eines Servlet in Bytecode.

Übersetzen des Servlet

Zunächst müssen Sie Ihr Servlet, wie jede andere Java-Klasse, von der Textform in interpretierbaren Bytecode übersetzen. Dazu verwenden Sie den Befehl `javac` von der Kommandozeile, Ihre Entwicklungsumgebung (IDE) oder das in diesem Buch benutzte Werkzeug *Ant*. Ein entsprechend konfiguriertes Script finden Sie im Beispielverzeichnis zu diesem Kapitel auf der CD.

Anschließend legen Sie ein Arbeitsverzeichnis für Ihre Webanwendung an oder erweitern das in Kapitel 2 verwendete zu folgender Struktur:

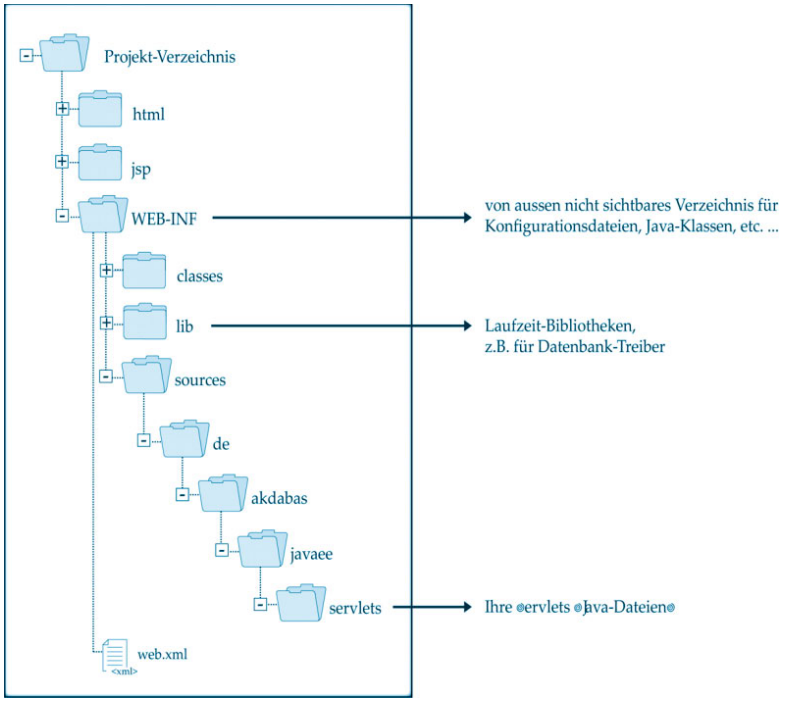


Abbildung 3.2
Arbeitsverzeichnis für Servlet-Anwendungen

Die Quelldateien der Servlet-Klassen legen Sie im *sources*-Ordner unterhalb von *WEB-INF* ab und übersetzen sie anschließend in das Verzeichnis *WEB-INF/classes*. Die einzelnen Verzeichnisse haben folgende Funktion:

- *WEB-INF*

  Dieses Verzeichnis enthält den Web Deployment Descriptor (*web.xml*), den Sie in Kapitel 2 bereits kennen gelernt haben und dem Sie auch in diesem Kapitel an späterer Stelle wieder begegnen werden.

- *WEB-INF/sources*

  In diesem Ordner legen Sie alle Quelldateien (Java-Dateien) der Webanwendung einschließlich ihrer Pfade ab.

- *WEB-INF/classes*

  In dieses Verzeichnis kompilieren Sie die Quelldateien. Es spielt für den Webbrowser eine entscheidende Rolle und wird an genau dieser Stelle erwartet.

- *WEB-INF/lib*

  Wenn Sie für den Betrieb Ihrer Webanwendung zusätzliche Java-Bibliotheken in Form von *jar*-Dateien benötigen, etwa um PDF-Dokumente zu rendern oder GIF-Bilder zu erzeugen, so kopieren Sie diese bitte in das Verzeichnis *WEB-INF/lib*. Jede Webanwendung erhält beim Start des Servlet-Containers ihren eigenen Classpath, in den die Dateien dieses Ordners eingebunden werden.

Tipp

Alle JAR-Dateien im Ordner *WEB-INF/lib* werden beim Start des Servlet-Containers automatisch in den Classpath für diese Webapplikation eingebunden. Für andere Applikationen sind diese Dateien nicht sichtbar. Dieses Verhalten ist sehr nützlich, wenn Bibliotheken zueinander inkompatibel sind. Legen Sie deshalb alle applikationsspezifischen Bibliotheken in diesem Ordner ab.

- *build_lib*

  Dieses Verzeichnis enthält Bibliotheken, die Sie zwar zum Übersetzen der Anwendung, *jedoch nicht zur Laufzeit der Anwendung benötigen*. Dazu gehört zum Beispiel die Basisklasse für Servlets `javax.servlet.http.HttpServlet`.

  Da der Servlet-Container (Apache Tomcat) eine eigene Version dieser Klasse mitbringt und diese zueinander inkompatibel sein können, darf die Klasse nicht im Classpath der Webanwendung enthalten sein. Das auf der CD befindliche Ant-Script ist bereits entsprechend vorkonfiguriert.

Neben diesen Verzeichnissen dürfen außerhalb des *WEB-INF*-Ordners natürlich beliebige weitere Verzeichnisse existieren, um beispielsweise Bilder, Stylesheets oder JSPs aufzunehmen.

Achtung

Gerade in Produktionsumgebungen ist es sehr gefährlich, die Quelldateien über den Servlet-Container verfügbar zu machen, da ein potenzieller Angreifer bei genauer Kenntnis des Codes natürlich gezielt nach Schwachstellen im System suchen und Firmengeheimnisse erspähen kann.
Zwar wird der Ordner *WEB-INF* durch den Servlet-Container für Zugriffe von außen gesperrt, da jedoch auch andere Zugänge zum Dateisystem des Servers existieren können, gehören Ihre Quelldateien nicht auf das Produktivsystem. Niemals.

Konfiguration des Web Deployment Descriptor

Mit dem erfolgreichen Übersetzen Ihres Servlet in das *classes*-Verzeichnis sind Sie zwar einen großen Schritt weiter, aber immer noch nicht am Ziel. Denn nun müssen Sie Ihrem Servlet-Container noch mitteilen, unter welchem URL Ihr Servlet erreichbar sein soll. Hierzu dient wieder der Web Deployment Descriptor (*web.xml*), den Sie nun folgendermaßen erweitern:

Listing 3.9 Web Deployment Descriptor (web.xml)

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app PUBLIC
   "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
   "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
   <display-name>Eine erste Webanwendung</display-name>

   <!-- Definition eines symbolischen Servlet-Namens -->
   <servlet>
      <servlet-name>SimpleServlet</servlet-name>
      <servlet-class>
            de.akdabas.jee.servlets.SimpleServlet
      </servlet-class>
   </servlet>

   <!-- Binden des symbolischen Namens an einen URL -->
   <servlet-mapping>
      <servlet-name>SimpleServlet</servlet-name>
      <url-pattern>/servlets/HelloWorld</url-pattern>
   </servlet-mapping>

</web-app>
```

Das Binden (engl. *Mapping*) eines Servlet an einen frei wählbaren URL besteht immer aus *zwei* Einträgen in der Datei *web.xml*. Zunächst versehen Sie Ihre Klasse über ein `<servlet>`-Tag mit einem symbolischen Namen. Dieser muss innerhalb der Datei eindeutig sein.

```
<!-- Definition eines eindeutigen, symbolischen Servlet-Namens -->
<servlet>
   <servlet-name>EindeutigerSymbolischerServletName</servlet-name>
   <servlet-class>VollständigerPfadDerServletKlasse</servlet-class>
</servlet>
```

Listing 3.10
Vergabe eines symbolischen Namens

Nachdem Sie Ihr Servlet mit einem symbolischen Namen ausgestattet haben, binden Sie diesen über `<servlet-mapping>`-Elemente an die gewünschten URLs:

```
<!-- Binden des symbolischen Servlet-Namens an ein URL-Pattern -->
<servlet-mapping>
   <servlet-name>EindeutigerSymbolischerServletName</servlet-name>
   <url-pattern>MusterDerZuBindendenURL</url-pattern>
</servlet-mapping>
```

Listing 3.11
Binden des symbolischen Namens an ein URL-Muster

Dabei kann ein Servlet über verschiedene `<servlet-mapping>`-Einträge auch an mehrere URLs gebunden werden. Außerdem kann der angegebene URL auch aus einem regulären Ausdruck bestehen und das Servlet so an eine ganze Klasse von URLs binden:

```
<servlet-mapping>
   <servlet-name>SimpleServlet</servlet-name>
   <url-pattern>/myServlets/*.servlet</url-pattern>
</servlet-mapping>
```

Listing 3.12
Mapping auf eine Klasse von URLs

In diesem Fall antwortet das `SimpleServlet` beispielsweise auf alle Requests, deren lokaler Pfad mit */myServlets/* beginnt und mit der Endung *.servlet* endet.

Aufrufen des Servlet

Nachdem Sie Ihr Servlet nun erfolgreich übersetzt und an einen URL gebunden haben, genügt ein Neustart des Servlet-Containers und schon können Sie Ihr Werk bewundern.

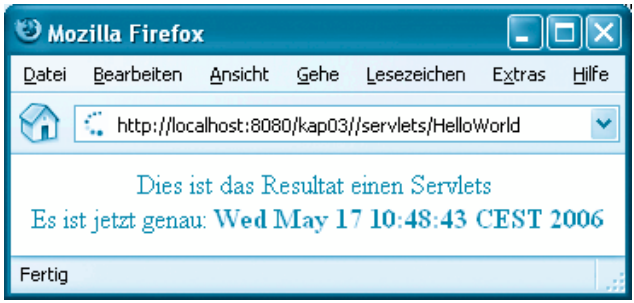


Abbildung 3.3
Ein einfaches Servlet

Die Konfiguration des Servlet-Containers unterscheidet sich für JSPs und Servlets in der Regel nicht. Eine Beispielkonfiguration für den Apache Tomcat finden Sie auf der beiliegenden CD und im vorhergehenden Kapitel.

3.1.4 HelloWorld-Servlet vs. HelloWorld-JSP

Das Servlet aus Listing 3.5 leistet also dasselbe wie Ihre erste JSP. Wenn Sie jedoch den Aufwand für beide Lösungen vergleichen, werden Sie sicher ebenfalls zu dem Schluss kommen, dass JSPs deutlich leichter zu handhaben waren. Warum sollten Sie also Servlets einsetzen?

Nun, zunächst einmal sollten Sie wissen, dass Servlets, auch wenn wir sie in diesem Buch erst nach den JSPs behandeln, die ältere Technologie darstellen. Noch 1999 standen Java-Programmierern, um HTTP-Requests komfortabel zu beantworten, ausschließlich Servlets zur Verfügung. Da jedoch 99% der Anfragen dabei eine Antwort im HTML-Format erwarten und dies, wie Listing 3.5 aufzeigt, eine recht mühevolle Aufgabe sein kann, ergänzte Sun die Servlet-Spezifikation um *JavaServer Pages (JSP)*. Jetzt verstehen Sie auch, warum die Java-Packages für JSPs unter dem Namensraum `javax.servlet.*` untergebracht sind. JSPs werden dabei zur Laufzeit in Servlets umgewandelt, kompiliert und verhalten sich anschließend wie Java-Klassen.

Info

Wie Sie später sehen werden, gilt die Regel: Alles, was mit einer JSP realisierbar ist, kann auch von einem Servlet geleistet werden. Darüber hinaus bieten Servlets aber einige Vorzüge, die den Mehraufwand mehr als wettmachen. Im Augenblick unterscheiden sich Servlets und JSPs für uns vor allem in einem Punkt: Die in JSPs vordefinierte Variable `out` muss bei Servlets explizit über das Response-Objekt angefordert und an eine Variable gebunden werden. Zunächst stehen Ihnen augenscheinlich nur die Parameter `request` und `response` der Methode `service()` zur Verfügung.

3.2 Der Lebenszyklus eines Servlet

Ein Servlet durchläuft im Servlet-Container einen fest vorgeschriebenen Lebenszyklus (Lifecycle), in den unterschiedlichen Stadien diese Lebenszyklus kann ein Servlet Ressourcen belegen, damit auf Requests reagieren und die Ressource abschließend wieder freigeben. Typische Beispiele hierfür sind das Speichern von Informationen der aktuellen Sitzungen oder Verbindungen zu einer Datenbank. Um eine solche Datenbankverbindung soll es im folgenden Beispiel gehen.

Achtung

Um das Gewicht dieses Buchs im Rahmen zu halten, kann an dieser Stelle keine Einführung in die *Java Database Connectivity (JDBC)* folgen. Allerdings findet sich inzwischen in jedem guten Java-Buch ein Kapitel zu diesem Thema.

Alle in diesem Abschnitt beschriebenen Methoden werden dabei bereits von der Basisklasse `HttpServlet` (siehe Listing 3.5) definiert und müssen nur bei Bedarf überschrieben werden.

3.2.1 Die init()-Methode

Der Lebenszyklus eines jeden Servlet beginnt mit der Methode `init()`. Sie wird vom Servlet-Container unmittelbar nach der Erzeugung des Servlet aufgerufen und dient dem Setup des Servlet. Dieser Aufruf findet je nach Konfiguration *unmittelbar nach dem Start* des Servlet-Containers oder bei *erstmaligem Aufruf* des Servlet statt.

Achtung

Servlets werden über einen parameterlosen Konstruktor erzeugt. Verlagern Sie den Code, den Sie eigentlich im Konstruktor platzieren würden, in die Methode `init()`, zumal Sie in dieser, wie Sie gleich sehen werden, Zugriff auf Initialisierungsparameter haben.

Die `init()`-Methode wird zwar nur ein einziges Mal aufgerufen, es existieren aber dennoch gleich zwei verschiedene Versionen von ihr:

Listing 3.13
Signaturen der Init-Methode

```
/** Dient der Initialisierung des Servlets */
public void init() throws ServletException {
   // ...
   // Hier steht Ihr Java-Code zum Initialisieren des Servlets
   // ...
}

/**
 * Dient ebenfalls der Initialisierung des Servlets und gestattet
 * zusätzlich den Zugriff auf Konfigurations-Parameter
 */
public void init(javax.servlet.ServletConfig config)
   throws ServletException {

   // Bei dieser Methode muss super.init() zwingend gerufen werden !
   super.init(config);

   // ...
   // Hier steht Ihr Java-Code zum Initialisieren des Servlets bei
   // dem Sie über die Variable config Zugriff auf die
   // Initialisierungsparameter haben.
   // ...
}
```

Um Ihr Servlet zu initialisieren, müssen Sie lediglich *eine* der beiden Methoden überschreiben und den gewünschten Code einfügen. Wenn Sie keine Initialisierungsparameter benötigen, verwenden Sie die obere Signatur und müssen sich keine weiteren Gedanken mehr machen. Die zweite Signatur ermöglicht es Ihnen, Ihr Servlet mit Initialisierungsparametern auszustatten. Kommt Ihr Servlet hingegen ohne explizite Initialisierung aus, können Sie auch ganz auf die Implementierung von `init()` verzichten, woraufhin der Servlet-Container die Methode der Superklasse `HttpServlet` aufruft.

Achtung

Wenn Sie die zweite Methode mit Initialisierungsparametern verwenden möchten, ist jedoch Folgendes zu beachten:

Sie überschreiben hierbei die `init()`-Methode der Servlet-Basisklasse (`javax.servlet.HttpServlet`), die der Servlet-Container unter anderem zur Registrierung des Servlets im Kontext benötigt. Deshalb muss der erste Aufruf dieser Methode die überschriebene `init()`-Methode der Superklasse sein.

```
...
   public void init(javax.servlet.ServletConfig config)
                    throws ServletException {
       super.init(config);
       ...
       config.getInitParameter("password");
       ...
    }
...
```

Sonst ist Ihr Servlet möglicherweise auch nach der Initialisierung nicht unter dem im Web Deployment Descriptor angegebenen URL erreichbar.

Initialisieren einer Datenbankverbindung über Parameter

Wie bereits weiter oben angesprochen, können Sie den Web Deployment Descriptor (*web.xml*) auch dazu verwenden, um Ihre Servlets mit Parametern zu konfigurieren. Das hat den Vorteil, dass Sie Ihre Servlets universell schreiben und anschließend an die jeweiligen Verhältnisse anpassen können. In diesem Beispiel etwa ist es sinnvoll, die Verbindungsdaten für die Datenbank variabel zu halten, da diese Daten sich leicht ändern können und Sie sicher nicht jedes Mal Ihr Servlet neu übersetzen möchten.

Listing 3.14
Web Deployment Descriptor mit Initialisierungsparametern

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app PUBLIC
   "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
   "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
   <display-name>Eine Datenbank-Webanwendung</display-name>

   <!--
       Beispiel-Konfiguration des Datenbank-Servlets mit
       Initialisierungsparametern für die DB-Verbindung
   -->
   <servlet>
      <servlet-name>SymbolischerNameDesServlets</servlet-name>
      <servlet-class>ServletKlasse</servlet-class>
      <init-param>
         <param-name>jdbcClass</param-name>
         <param-value>TreiberklasseDerDatenbank</param-value>
      </init-param>
      <init-param>
         <param-name>dbURL</param-name>
```

Listing 3.14 (Forts.)
Web Deployment Descriptor mit Initialisierungsparametern

```
            <param-value>URLderDatenbank</param-value>
        </init-param>
        <init-param>
            <param-name>username</param-name>
            <param-value>ZuVerwendenderBenutzername</param-value>
        </init-param>
        <init-param>
            <param-name>password</param-name>
            <param-value>ZuVerwendendesPasswort</param-value>
        </init-param>
    </servlet>
...
</web-app>
```

Das folgende Listing-Fragment Ihrer Servlet-Klasse zeigt Ihnen nun, wie Sie die oben konfigurierten Daten wieder auslesen können:

Listing 3.15
init()-Methode zum Aufbau einer Datenbankverbindung (aus DatabaseServlet.java)

```
...
    /**
     * Initialisieren der Datenbank-Verbindung mit Parametern aus
     * dem Web Deployment Descriptor WEB-INF/web.xml
     */
    public void init(javax.servlet.ServletConfig config)
        throws ServletException {

        // Überschriebene init()-Methode der Superklasse aufrufen !
        super.init(config);

        // Parameter aus der Datei WEB-INF/web.xml auslesen
        String driver = config.getInitParameter("jdbcClass");
        String dbURL =  config.getInitParameter("dbURL");
        String dbUser = config.getInitParameter("username");
        String dbPass = config.getInitParameter("password");

        // Initialisieren der Datenbank-Verbindung
        try {
            // Laden des Datenbank-Treibers über den ClassLoader
            Class.forName(driver);

            // Aufbau der Datenbank-Verbindung
            connection =
                DriverManager.getConnection(dbURL, dbUser, dbPass);
        } catch(Exception exc) {
            throw new ServletException("SQL-Exception in init()", exc);
        }
    }
...
```

Info

Der Servlet-Container sperrt das Verzeichnis *WEB-INF*, in dem sich auch der Web Deployment Descriptor befindet, für Zugriffe, so dass in diesem Verzeichnis abgelegte Dateien nicht über den Servlet-Container ausgelesen werden können. Sofern Ihr Server also nicht anderweitig »von draußen« erreichbar ist, sind die im Web Deployment Descriptor abgelegten Daten und Passwörter vor den Augen Dritter, sofern diese keinen anderen Zugang zum Server besitzen, sicher.

3.2.2 Die Service-Methoden

Die Service-Methoden stellen das Herz eines jeden Servlet dar. Der hier implementierte Code wird *bei jedem Aufruf* eines mit diesem Servlet verknüpften URL aufs Neue ausgeführt.

In Ihrem ersten Servlet implementierten Sie die Methode `service()`, um die gewünschte Funktionalität aufzunehmen. Überschreiben Sie diese, wenn Ihr Servlet alle Request-Arten auf die gleiche Weise behandeln soll.

Das HTTP-Protokoll unterscheidet jedoch zwischen verschiedenen Request-Anfragen, von denen Tabelle 3.1 die wichtigsten aufzählt. Und so existieren verschiedene weitere Service-Methoden, die jeweils zur Beantwortung eines spezifischen Request-Typs vorgesehen sind.

Tabelle 3.1 Verschiedene HTTP-Request-Typen

Request-Typ	Verwendung	Service-Methode
GET	Mit diesem Request teilen Sie dem Server mit, dass der Client eine Ressource (z.B. eine Webseite) empfangen möchte.	`doGet()`
POST	Ein Post-Request zeigt an, dass der Client dem Server einige Informationen zur Verfügung stellt, wie es üblicherweise bei Formularen der Fall ist.	`doPost()`
HEAD	Über den Head-Request signalisiert der Client, dass er lediglich die Response-Header, nicht jedoch das resultierende Dokument empfangen möchte.	`doHead()`
PUT	Dieser Request-Typ ermöglicht es dem Client, Dokumente auf dem Server abzulegen.	`doPut()`
DELETE	Dieser Typ ist das Gegenstück zu PUT und wird dazu verwendet, Dokumente vom Server zu entfernen.	`doDelete()`
TRACE	Dieser Request-Typ ermöglicht das Aufspüren von Proxies zwischen Client und Server und wird vor allem zum Protokoll-Debugging verwendet.	`doTrace()`
OPTIONS	Mit diesem Request fragt der Client die zulässigen Request-Typen, für dieses Dokument ab.	`doOptions()`

Alle Service-Methoden besitzen die gleiche Signatur. Sie bekommen ein *Request*- und ein *Response*-Objekt und werfen im Fehlerfall eine `ServletException` bzw. `IOException`.

Info

Neben den in Tabelle 3.1 angegebenen Request-Typen spezifiziert HTTP 1.1 noch einige weitere, wie `LINK` und `UNLINK`. Für diese sieht die aktuelle Servlet-Spezifikation jedoch keine gesonderten Methoden vor.

Listing 3.16
Signatur der Service-Methoden

```
/** Grundsätzlicher Aufbau aller Servlet-Servicemethoden */
public void doRequestTyp(HttpServletRequest request,
                         HttpServletResponse response)
   throws ServletException, IOException {

   // ...
   // Hier steht Ihr Java-Code zur Erzeugung des Dokumentes
   // ...
}
```

Da Sie es in aller Regel mit POST- oder GET-Requests zu tun haben werden, ist es häufig sinnvoll, nur deren Service-Methoden zu überschreiben und in allen anderen Fällen die Standardfunktionalität der Basisklasse (`HttpServlet`) zu verwenden.

Implementieren einzelner Service-Methoden

Statt wie im ersten Servlet die Methode `service()` zu überschreiben, welche vom Servlet-Container verwendet wird, um das Servlet zur Antwort zu veranlassen, können Sie den Code auch in eine der Service-Methoden integrieren. Da Sie hierbei eine HTML-Seite zur Verfügung stellen, ist dies üblicherweise `doGet()`. Am einfachsten realisieren Sie dies, indem Sie die Methode `service()` umbenennen. Das nächste Listing zeigt Ihnen, dem Datenbankbeispiel folgend, eine `doGet()`-Methode, welche dazu verwendet werden kann, Datensätze aus einer Datenbank auszugeben.

Info

Dieses Beispiel geht von einer existierenden Datenbank samt der Tabelle `Adress` und den Spalten `Name` und `First_Name` aus. Um dieses Beispiel zu testen, passen Sie die Angaben aus Listing 3.17 gegebenenfalls an.

Listing 3.17
Ausgabe von Datensätzen einer Datenbank (aus DatenbankServlet.java)

```
...
   /** Service-Methode zur Verarbeitung von GET-Requests */
   public void doGet(HttpServletRequest request,
                     HttpServletResponse response)
                     throws IOException, ServletException {

      // Binden der (in JSPs vordefinierten) Variable out
      PrintWriter out = response.getWriter();

      // Erzeugen von HTML-Code ...
      out.println("<html><body><center>Datensätze</center>");
      out.println("<table><tr><td>Name</td><td>Vorname</td></tr>");

      // ... und Einbinden von dynamischem Inhalt
      try {
         ResultSet rs = null;
```

Listing 3.17 (Forts.) Ausgabe von Datensätzen einer Datenbank (aus DatenbankServlet.java)

```
            try {
               // Ausführen eines SQL-Statements via JDBC
               Statement stmt = connection.createStatement();
               rs = stmt.executeQuery
                       ("Select NAME, FIRST_NAME from ADRESS");

               // Iterieren über die Ergebnismenge des SQL-Statements
               while(! rs.isLast()) {
                  out.println("<tr><td>" + rs.getString(1) + "</td>");
                  out.println("<td>" + rs.getString(2) + "</td></tr>");
               }

            } finally {
               rs.close();
            }

         } catch(SQLException exc) {
            throw new ServletException("SQL-Exception", exc);
         }

         out.println("</table></body></html>");
      }
...
```

Doch wie erreichen Sie es jetzt, dass der Servlet-Container GET-Request an die Methode `doGet()` weiterleitet? Das erledigt die `service()`-Implementierung der Basisklasse `HTTPServlet` für Sie. Diese enthält leere Methodenrümpfe für alle `doXxx()`-Methoden und ruft diese dem Request-Typ entsprechend auf. Wenn Sie nun eine der Methoden überschreiben, wird stattdessen Ihr Code ausgeführt.

GET und POST gemeinsam behandeln

Wie bereits erwähnt, sind die häufigsten Request-Typen GET und POST und wenn Sie mit der klassischen HTML-Programmierung vertraut sind, wissen Sie, dass Sie beispielsweise für Formulare (`<form>`) über das Attribut `method` festlegen können, ob diese via POST oder GET übermittelt werden sollen.

Info

Das HTTP-Protokoll verfügt über unterschiedliche Anfrage-Methoden, die ursprünglich für verschiedene Einsatzzwecke, wie das Senden von Daten (POST) oder das Empfangen von Dokumenten (GET), vorgesehen wurden. In der Praxis behandeln viele Webserver jedoch POST und GET identisch.

Wie stellen Sie jetzt sicher, dass das Servlet die Daten des Formulars unabhängig vom gewählten Wert des Formulars empfängt? Sie könnten natürlich zunächst die Methode `doGet()` entwickeln, anschließend den Code kopieren und in die Methode `doPost()` einfügen. Eleganter ist es jedoch, wenn Sie innerhalb der Methode `doPost()` einfach die Methode `doGet()` aufrufen.

```
...
   /** Bearbeiten des POST-Requests analog zu GET */
   public void doPost(HttpServletRequest request,
                      HttpServletResponse response)
                      throws IOException, ServletException {

      // Weiterleiten des Request an die Methode doGet()
      doGet(request, response);
   }
...
```

Listing 3.18
Weiterleiten von POST-Requests an GET (aus DatenbankServlet.java)

Da alle Service-Methoden eine einheitliche Signatur besitzen, können Sie auf diese Weise die Behandlung beliebiger Request-Typen zusammenfassen.

Vorteile bei der Verwendung einzelner Service-Methoden

Wie Sie bereits gesehen haben, können Sie, insbesondere wenn Sie unterschiedliche Request-Typen gemeinsam behandeln möchten, entweder die vom Servlet-Container gerufene Methode `service()` direkt überschreiben oder eine der gewünschten `doXxx()`-Methoden implementieren und diese von den anderen Servicemethoden aus aufrufen. Im zweiten Fall überschreiben Sie nur die nötigsten Methoden und lassen den Servlet-Container sonst mit der Default-Implementierung der Basisklasse `HttpServlet` reagieren. Diese Lösung bringt folgende Vorteile mit sich:

- Es ist auch im Nachhinein möglich, etwa mit einer abgeleiteten Klasse, spezielle Methoden für PUT- oder DELETE-Requests zu integrieren.
- Sie erhalten eine automatische Unterstützung für HEAD-Requests.

  Registriert die Service-Methode der Basisklasse einen HEAD-Request, so sendet sie einfach einen GET-Request, samt Request-Headern und Statuscodes, jedoch ohne HTML-Markup zurück. HEAD-Anfragen werden unter anderem von Link-Validierern eingesetzt, um z.B. tote Links zu ermitteln, da die hier erzeugte Netzlast deutlich geringer ist.
- Sie erhalten Unterstützung für `If-Modified`-Requests. Diese werden von Proxies verwendet, um zu ermitteln, ob sich die zwischengespeicherte Seite inzwischen geändert hat und neu übertragen werden muss. Anderenfalls sendet der Server den Status-Code `304 Not-Modified` und die Seite wird aus dem Proxy-Cache geladen.
- Automatische Unterstützung von TRACE-Requests. Dieser Request-Typ wird von Programmierern verwendet, um die Route eines TCP/IP-Pakets zu bestimmen. TRACE-Requests beantwortet die `service()`-Methode eigenständig durch Zurücksenden der erforderlichen Request-Header.
- Automatische Unterstützung von OPTIONS-Requests. Hier überprüft die `service()`-Methode das Vorhandensein einer `doGet()`-Methode. Existiert eine solche, so antwortet der Server mit einem *Allow-Header* für GET, HEAD, OPTIONS und TRACE.

Wie Sie sehen, müssen Sie in der Regel nur den spezifischen Code für POST und GET selbst implementieren, für alles andere sorgt die Basisimplementierung. Sollten Sie dennoch eine spezielle Antwort benötigen, überschreiben Sie einfach zusätzlich die entsprechende Methode.

3.2.3 Die Destroy-Methode

Hin und wieder entscheidet der Servlet-Container, dass es an der Zeit ist, ein zuvor instanziiertes Servlet wieder aus dem Speicher zu entfernen, etwa weil es zu lange nicht aufgerufen wurde und der Server wertvolle Ressourcen freigeben möchte oder weil der Server heruntergefahren wird. In diesem Fall ruft er zuvor die Methode `destroy()`auf, die das Pendant zu `init()` bildet.

Listing 3.19
Die Destroy-Methode

```
...
   /** Wird verwendet um belegte Ressourcen freizugeben */
   public void destroy() {
      try {
         // Freigeben von Datenbank-Ressourcen
         connection.close();
      } catch(SQLException exc) {
         log("SQL-Exception in destroy()", exc);
      }
   }
...
```

Im Datenbankbeispiel dient die Methode `destroy()` dazu, die zuvor geöffnete Datenbankverbindung zu schließen. Weitere denkbare Beispiele sind das Speichern eines Zugriffszählers oder des Zustands eines Chat-Systems, um diesen gegebenenfalls wiederherstellen zu können.

3.2.4 Das vollständige Datenbank-Servlet

Nachdem Sie nun die am häufigsten verwendeten Methoden `init()`, `doPost()`, `doGet()` und `destroy()` am Beispiel eines Datenbank-Servlet einzeln kennen gelernt haben, folgt hier noch einmal das vollständige Listing:

Listing 3.20
Vollständiges Listing des Datenbank-Servlet

```
package de.akdabas.javaee.servlets;

import java.io.PrintWriter;
import java.io.IOException;
import java.sql.*;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletException;

/** Dieses Servlet erzeugt dynamischen Inhalt aus Datenbank-Daten */
public class DatabaseServlet extends HttpServlet {

   /** Globale Variable: Die geöffnete Datenbank-Verbindung */
   private Connection connection;

   /**
    * Initialisieren der Datenbank-Verbindung mit Parametern aus
    * dem Web Deployment Descriptor WEB-INF/web.xml
```

Listing 3.20 (Forts.)
Vollständiges Listing des Datenbank-Servlet

```
 */
public void init(javax.servlet.ServletConfig config)
   throws ServletException {

   // Überschriebene init()-Methode der Superklasse aufrufen !
   super.init(config);

   // Parameter aus der Datein WEB-INF/web.xml auslesen
   String driver = config.getInitParameter("jdbcClass");
   String dbURL =  config.getInitParameter("dbURL");
   String dbUser = config.getInitParameter("username");
   String dbPass = config.getInitParameter("password");

   // Initialisieren der Datenbank-Verbindung
   try {
      // Laden des Datenbank-Treibers über den ClassLoader
      Class.forName(driver);

      // Aufbau der Datenbank-Verbindung
      connection =
         DriverManager.getConnection(dbURL, dbUser, dbPass);
   } catch(Exception exc) {
      throw new ServletException("SQL-Exception in init()", exc);
   }
}

/** Service-Methode zur Verarbeitung von GET-Requests */
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
                  throws IOException, ServletException {

   // Binden der (in JSPs vordefinierten) Variable out
   PrintWriter out = response.getWriter();

   // Erzeugen von HTML-Code ...
   out.println("<html><body><center>Datensätze</center>");
   out.println("<table><tr><td>Name</td><td>Vorname</td></tr>");

   // ... und Einbinden von dynamischem Inhalt
   try {
      ResultSet rs = null;
      try {
         // Ausführen eines SQL-Statements via JDBC
         Statement stmt = connection.createStatement();
         rs = stmt.executeQuery
                 ("Select NAME, FIRST_NAME from ADRESS");

         // Iterieren über die Ergebnismenge des SQL-Statements
         while(! rs.isLast()) {
            out.println("<tr><td>" + rs.getString(1) + "</td>");
            out.println("<td>" + rs.getString(2) + "</td></tr>");
         }

      } finally {
         rs.close();
      }

   } catch(SQLException exc) {
      throw new ServletException("SQL-Exception", exc);
   }

   out.println("</table></body></html>");
}

/** Bearbeiten von POST-Requests analog zu GET */
public void doPost(HttpServletRequest request,
```

Listing 3.20 (Forts.) Vollständiges Listing des Datenbank-Servlet

```
                                    HttpServletResponse response)
                                    throws IOException, ServletException {

      // Weiterleiten des Requests an die Methode doGet()
      doGet(request, response);
   }

   /** Wird verwendet um belegte Ressourcen freizugeben */
   public void destroy() {
      try {
         // Freigeben von Datenbank-Ressourcen
         connection.close();
      } catch(SQLException exc) {
         log("SQL-Exception in destroy()", exc);
      }
   }
}
```

Um dieses Servlet zu testen, müssen Sie die folgenden Einträge in die Datei *web.xml* einfügen:

Listing 3.21 Erweiterung des Web Deployment Descriptor

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app PUBLIC
   "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
   "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
...
   <!--
       Beispiel-Konfiguration des Datenbank-Servlets mit
       Initialisierungsparametern für die DB-Verbindung
   -->
   <servlet>
      <servlet-name>DatabaseServlet</servlet-name>
      <servlet-class>
         de.akdabas.javaee.servlets.DatabaseServlet
      </servlet-class>
      <init-param>
         <param-name>jdbcClass</param-name>
         <param-value>com.mysql.jdbc.Driver</param-value>
      </init-param>
      <init-param>
         <param-name>dbURL</param-name>
         <param-value>jdbc:mysql://localhost/db</param-value>
      </init-param>
      <init-param>
         <param-name>username</param-name>
         <param-value>scott</param-value>
      </init-param>
      <init-param>
         <param-name>password</param-name>
         <param-value>tiger</param-value>
      </init-param>
   </servlet>
...
   <!-- Binden des Datenbank-Servlets an einen URL -->
   <servlet-mapping>
      <servlet-name>DatabaseServlet</servlet-name>
      <url-pattern>/servlets/DatabaseTest</url-pattern>
   </servlet-mapping>

</web-app>
```

Wenn Sie über eine passende Datenbank verfügen, können Sie Ihr Servlet nun durch Aufruf des URL *http://localhost:8080/kap04/servlets/DatabaseTest* ausgiebig testen.

Tipp

Den aktuellen MySQL-Datenbanktreiber können Sie von der Homepage *http://dev.mysql.com/downloads/connector/j/3.1.html* herunterladen. Um diesen in Ihrer Webanwendung verfügbar zu machen, kopieren Sie einfach die enthaltenen Java-Archive (jar) ins Verzeichnis *WEB-INF/lib*.

3.2.5 Vergleich zwischen GET und POST

Prinzipiell macht es zwar keinen Unterschied, ob Sie Ihre HTML-Seiten oder JSP-Formulare via GET oder POST übermitteln, da Sie (entsprechende Service-Methoden vorausgesetzt) im Servlet in der Regel die gleichen Methoden verwenden. Dennoch unterscheiden sich beide Methoden historisch bedingt voneinander. Die folgende Tabelle listet die wichtigsten Unterschiede auf:

Tabelle 3.2 Vergleich von GET- und POST-Requests

	GET	POST
Service-Methode	`doGet()`	`doPost()`
Verwendungszweck	Ressourcen vom Server empfangen	Daten an den Server übermitteln und Antwort abwarten
Formulardaten	Werden in Form von Name/Value-Paaren an den URL angefügt	Werden im Request-Body übermittelt und sind kein Bestandteil des URL
Länge des Request	Ist limitiert auf etwa 1 Kbyte	Ist nicht limitiert
Formulardaten – Benutzer	Benutzer »sieht« Formulardaten in der Adresszeile des Browsers	Werden vor dem Benutzer verborgen
Bookmarks	Können einfach gesetzt werden, alle Daten im URL kodiert	Können weniger gut gesetzt werden, da Daten aus Formularfeldern nicht mit abgespeichert werden

3.2.6 Zusammenfassung des Servlet-Lebenszyklus

Die erste, unmittelbar nach dem Konstruktor aufgerufene Methode ist `init()`, wobei diese mit einem `ServletConfig`-Objekt parametrisiert wird, über welches Sie Zugriff auf die im Web Deployment Descriptor hinterleg-

ten Initialisierungsparameter erhalten. Die parameterbehaftete `init()`-Methode ruft dann wiederum die parameterlose Version von `init()` auf, die Sie implementieren können, wenn Sie keinerlei Initialisierungsparameter benötigen. Sollten Sie die parametrisierte `init()`-Methode überschreiben, muss der erste Aufruf `super.init()` sein, da das Servlet sonst nicht erreichbar ist.

Info

In diesem Buch verwenden wir ausschließlich den Apache Tomcat, welcher Webapplikationen über den Web Deployment Descriptor (*web.xml*) konfiguriert. Auch wenn der Zugriff auf Initialisierungsparameter und `ServletContext` über das API vereinheitlicht sind, unterscheiden sich die einzelnen Servlet-Container hinsichtlich ihrer Konfiguration.
So verwendet SUNs *JavaServer Web Development Kit (JSWDK)* eine Datei namens *servlet.properties*. Bei BEAs *WebLogic-Application-Server* heißt das Pendant dieser Datei wiederum *weblogic.properties*.

Nachdem Ihr Servlet initialisiert ist, wird für jeden Request an einen URL, der mit dem Servlet verbunden ist, die Methode `service()` aufgerufen. Die Basisimplementierung der Klasse `HTTPServlet` überprüft dabei die Art des Request und ruft anschließend die dafür vorgesehene Servicemethode `doXxx()` auf, die Sie bei Bedarf überschreiben können. Die einzelnen Servicemethoden können auch aufeinander verweisen, um beispielsweise den Behandlungscode für GET und POST nur in einer Methode implementieren zu müssen. Dabei darf es nicht zu zyklischen Abhängigkeiten kommen.

Bevor das Servlet aus dem Speicher entfernt wird, ruft der Servlet-Container die Methode `destroy()` auf, in der das Servlet zuvor belegte Ressourcen freigeben und seinen inneren Zustand bei Bedarf auf ein persistentes Medium speichern kann.

3.3 Servlets vs. JavaServer Pages

Können Sie sich nun auch nicht mehr entscheiden, ob Sie für Ihre nächste dynamische Webressource auf JavaServer Pages zurückgreifen oder doch lieber ein Servlet verwenden?!

Dieser Absatz soll Ihnen die Gemeinsamkeiten und Unterschiede von Servlets und JSPs näher bringen und Sie gegebenenfalls endgültig von der Daseinsberechtigung für Servlets überzeugen. Denn obwohl die Unterschiede bisher eher marginal erscheinen und 99% der von Ihnen zu behandelnden Requests POST oder GET sein werden, bieten Servlets Ihnen an entscheidenden Stellen Vorteile, die JSPs einfach nicht leisten können.

3.3.1 Was Sie mit JSPs nicht machen können

Zunächst folgt eine Zusammenfassung der bisher besprochenen Eigenschaften, die Sie nur mit Servlets realisieren können. Zwar gibt es für jede nur erdenkliche JSP ein entsprechendes Servlet (auch wenn dieses noch so komplex und unübersichtlich sein mag), aber andersherum finden Sie nicht für alle Servlets eine gleichwertige JSP.

Binäre Formate senden

Bisher waren alle dynamisch erzeugten Dokumente textbasiert und konnten problemlos über einen `PrintWriter` ausgegeben werden. Im Alltag haben Sie es jedoch auch oft mit binär kodierten Dokumenten wie Bildern oder PDF-Dateien zu tun. Java bietet Ihnen zahlreiche Möglichkeiten, diese ebenfalls zur Laufzeit zu erstellen und grafische Auswertungen (Charts) somit erst zum Zeitpunkt des Request zu berechnen. Binärformate sind jedoch byteorientiert und können damit nicht über zeichenorientierte Ausgabeströme, zu denen alle `Writer` gehören, ausgegeben werden.

Info

Zu dem Punkt »*Binäre Formate senden*« erhielt ich einige Feedbacks, in denen mir Leser mitteilten, dass es dennoch möglich sei, binäre Dateien über JSPs zu versenden. Deshalb an dieser Stelle eine kurze Anmerkung dazu:
Bei dem in JSPs verwendeten `JspWriter` handelt es sich tatsächlich um einen `PrintWriter`, der sowohl zeichen- als auch byte-orientierte Daten, z.B. Bilder, versenden kann. Dies ist jedoch nur ein historisch bedingter Umstand und ich empfehle mit Nachdruck die Verwendung von Servlets, zumal die JSP in diesem Fall keinen Markup enthalten dürfte.

Binden mehrerer URLs

Während eine JSP immer über ihren tatsächlichen Pfad innerhalb der Webapplikation aufgerufen wird, werden Servlets über eine separate Konfigurationsdatei (*web.xml*) an einen oder mehrere URLs gebunden. Durch URL-Patterns können Sie ein Servlet auch an eine ganze Klasse von URLs binden.

Achtung

Da Servlets nicht über ihren tatsächlichen Speicherort (im Ordner *WEB-INF/classes*), sondern über einen daran gebundenen URL aufgerufen werden, müssen auch Pfade innerhalb des erzeugten Dokuments entweder global oder relativ zum jeweils gemappten URL sein, da der Client nicht zwischen HTML-Dokument, JSP und Servlet unterscheiden kann. Dies gilt insbesondere für eingebundene Bilder oder Cascading Stylesheets (CSS).

Verschiedene Service-Methoden für unterschiedliche Request-Typen

Auch wenn man nur selten zwischen unterschiedlichen Request-Typen und den jeweiligen Behandlungsmethoden unterscheiden muss, führt der einzige Weg, dies in Java zu tun, über Servlets. JSPs verfügen nur über eine innere Methode, welche immer aufgerufen wird.

3.3.2 Direkter Vergleich zwischen Servlets und JSPs

Tabelle 3.3 Vergleich zwischen Servlet und JSP

	JavaServer Pages	Servlets
Übersetzungszeitpunkt	Wird zur Laufzeit von der JSP-Engine erzeugt; kann mit entsprechenden Tools vorkompiliert werden	Müssen vor dem Einsatz manuell übersetzt werden
Konfiguration	Müssen nicht explizit konfiguriert werden	Müssen vor der Verwendung konfiguriert werden
Bindung an URL	Unter dem tatsächlichen Pfad, innerhalb der Webapplikation erreichbar	Werden über den Web Deployment Descriptor oder andere Konfigurationsdateien an einen oder mehrere symbolische URLs gebunden
Initialisierungsparameter	Nur allgemeine Kontextparameter möglich	Individuelle Initialisierungsparameter über Servlet-Konfiguration möglich
Vordefinierte Variablen	Stehen von Anfang an zur Verfügung	Müssen über Methoden aus dem Request- und dem Response-Objekt gebunden werden
HTML-Code	Kann direkt zwischen JSP-Anweisungen eingefügt werden	Muss über `print` oder `write` ausgegeben werden, dabei muss z.B. " durch \" ersetzt werden
Dokumententypen	Beschränkt auf textbasierte Dokumente	Unterstützen sowohl Text- als auch Binärformate
Services	Beschränkt auf eine einzige Service-Methode	Eigene Service-Methoden für jeden Request-Typ (PUT, GET, POST, ...) möglich
Umwandlung	Können durch Servlets ersetzt werden	Können nur bedingt durch JSPs ersetzt werden

3.3.3 Fazit

Beim direkten Vergleich der *simple.jsp*-Seite mit dem `SimpleServlet` stellen Sie fest, dass Sie das, was Sie im vorhergehenden Kapitel mit 6 Zeilen erreicht haben, auch mit 30 Zeilen realisieren können, wobei der Code im letzten Fall unübersichtlicher und weniger verständlich wird. Daraus folgt: Servlets sind nicht für jedes dynamisch erstellte Dokument gleichermaßen gut geeignet. Umgekehrt ist auch eine JSP, die nur aus Java-Code besteht, in der Regel nur schwer zu warten. Scheinbar ist eine Kombination beider Techniken die sinnvollste Variante.

Nach Veröffentlichung der JSP-Spezifikation und ersten Referenzimplementierungen setzte ein unglaublicher Run auf JSP-basierte Webapplikationen ein. Endlich konnten Java-Entwickler und Webdesigner zusammenarbeiten, ohne zu viel vom jeweils anderen Thema verstehen zu müssen. Die Folge waren vor Code überquellende JSP-Seiten, die auch der Autor bald nicht mehr verstand und die nur sehr mühsam zu warten waren. Der Einsatz von JavaBeans und Tag-Bibliotheken (denen sich das nächste Kapitel widmet) besserte die Lage zwar, behandelte jedoch nur die Symptome. Webanwendungen, welche ausschließlich aus JSPs, Tag-Bibliotheken und JavaBeans bestehen, werden unter dem Synonym *Model I* zusammengefasst. Dieses eignet sich vor allem für kleinere und mittlere Anwendungen, bei denen die zuständigen Entwickler sowohl über das nötige Java- als auch HTML-Know-how verfügen. Für den Einsatz etwa von *Enterprise JavaBeans (EJB)* ist dieses Modell jedoch nicht geeignet, da eine echte Trennung zwischen Darstellung, Geschäftslogik und Controller nur schwer erreicht werden kann.

Info

Webanwendungen, die ausschließlich aus JSPs, Tag-Bibliotheken und JavaBeans bestehen, werden auch als *Model-I*-Webanwendungen bezeichnet.

Nachdem der erste Hype der JSPs vorüber war, begannen Java-Entwickler, JSPs und Servlets miteinander zu kombinieren. Die Java-Logik wurde in Servlets und JavaBeans gekapselt, die ihrerseits mit EJB-Containern und Datenbanken kommunizierten und den Request anschließend zur Darstellung an eine JSP weiterleiteten. Die Steuerung übernahmen ein oder mehrere ausgezeichnete Servlets. Diese Anwendungen fasst Sun unter dem Begriff *Model II* zusammen, hinter dem sich nichts anderes als das bereits in den achtziger Jahren unter Smalltalk entwickelte *Model View Controller*-Konzept (MVC) verbirgt. Dieses Konzept wird in Kapitel 5 über Struts eingehend behandelt.

Info

Webanwendungen, die JSPs, Tag-Bibliotheken und JavaBeans zusammen mit Servlets einsetzen, werden als *Model-II*-Webanwendungen bezeichnet.

3.4 Cookies

Dieser Abschnitt befasst sich mit Cookies; ein weiterer Hype früherer Webapplikationen, von denen heute kaum noch jemand direkt spricht. Cookies sind kleine Informationsbrocken in Form von Name/Wert-Paaren, die vom Server erzeugt und mit einer definierten Lebensdauer versehen werden. Der Server fügt diese Cookies als Header an die Response an, wo sie vom Client empfangen werden. Unterstützt der Client die Verwendung von Cookies, speichert er diese an einer bestimmten Stelle auf seiner Festplatte und fügt die Informationen für die gesamte Lebensdauer des Cookie an alle Requests an, für die der Cookie bestimmt ist.

Info

Als Richtwert kann man davon ausgehen, dass ein Browser maximal 30 Cookies von einer Webseite akzeptiert. Diese Zahl lässt sich in einigen Browsern auch einstellen.

Leider wurden Cookies in vielen Fällen dazu missbraucht, eine Unmenge von Daten über den Client zu sammeln und gezielt Anwenderprofile zu erstellen. Dies störte die vom Benutzer geschätzte Anonymität und führte dazu, dass viele Anwender Cookies heutzutage abschalten oder nur noch sehr selektiv zulassen.

Ein typisches Einsatzgebiet für Cookies haben Sie bereits kennen gelernt: den Session-Kontext. Viele Servlet-Container verwalten die Benutzer-Sessions über Cookies, indem sie die jeweilige Session-ID als Wert eines bestimmten Cookies übermitteln. Sendet der Client diese beim nächsten Request zurück, wird die ID vom Server erkannt und dem Request die entsprechende Session zugeordnet.

3.4.1 Cookies erstellen

Sie können in Ihren Servlets Cookies einfach als Java-Objekte über ihren Konstruktor erstellen und anschließend in die Response einfügen. Der Konstruktor erwartet dabei zwei Parameter:

Listing 3.22 Konstruktor für Cookie-Objekte

```
...
   // Erstellen eines neuen Cookies
   Cookie cookie = new Cookie("NameDesCookie", "WertDesCookie");
...
```

Beide Parameter müssen dabei vom Typ `String` sein und dürfen weder Leerzeichen noch eines der folgenden reservierten Sonderzeichen enthalten.

```
// 'Verbotene' Sonderzeichen für Cookies
( ) [ ] = " / ? @ , ; :
```

Listing 3.23
Reservierte Sonderzeichen für Cookies

Um die Cookies verschiedener Clients unterscheiden zu können, müssen natürlich auch die gewählten Werte eindeutig sein. An dieser Stelle ist Ihr Cookie bereits einsatzfähig und kann an das `Response`-Objekt angefügt werden. Es ist jedoch sinnvoll, die Cookies zuvor mit bestimmten Eigenschaften auszustatten.

Die Lebensdauer eines Cookie

Da Cookies in der Regel dazu dienen, vom Server verwaltete Informationen einem Client zuzuordnen, ist es sinnvoll, eine maximale Lebensdauer für Ihren Cookie zu definieren. Diese gibt die Zeitspanne an, innerhalb der ein Cookie vom Client zurückgesendet werden soll. Auf diese Weise verhindern Sie, dass Ihr Server immer mehr nutzlose Objekte verwalten muss, wenn sich etwa ein Benutzer nur auf Ihre Webseite verirrt hat und nicht mehr dorthin zurückkehrt. Denn ist die Lebensdauer des Cookie überschritten, wird es vom Browser gelöscht und nicht mehr an den Server zurückgesendet. Damit können Sie das mit dem Cookie verknüpfte Objekt ebenfalls löschen.

```
...
   // Setzen der maximalen Lebensdauer eines Cookies
   cookie.setMaxAge(int LebensdauerDesCookiesInSekunden);
...
```

Listing 3.24
Lebensdauer eines Cookie definieren

Der Parameter gibt dabei die Sekunden an, bis das Cookie vom Browser gelöscht wird. Beim Standardwert `-1` bleibt der Cookie so lange bestehen, bis der Browser geschlossen wird. Eine `0` hingegen bewirkt, dass der Browser diesen Cookie sofort löscht.

Der Pfad eines Cookie

Über den Pfad eines Cookie können Sie bestimmen, an welche URLs der Cookie zukünftig angehängt werden soll. Im Standardfall ist das genau die Ressource, von der der Browser den Cookie empfangen hat, also z.B.

```
// Ursprünglicher Pfad eines Cookie
localhost:8080/kap03/servlets/HelloWorld
```

Listing 3.25
Vollständiger Pfad eines Cookie

Das genügt zwar für die meisten Fälle, doch in größeren Webanwendungen ist es in der Regel erforderlich, dass *alle* Servlets der Domäne die Informationen empfangen. Über die Methode `setPath()` können Sie den Pfad verallgemeinern.

```
...
   // Verallgemeinerter Pfad eines Cookie
   cookie.setPath("/kap03/servlets/ ");
...
```

Listing 3.26
Verallgemeinerter Pfad eines Cookie

Damit wird das Cookie zukünftig an alle URLs auf unserem Server gesendet, die mit dem Pfad */04_Servlets/servlet/* beginnen.

Achtung

Es ist jedoch nur möglich, den URL des Cookie zu *verallgemeinern*. Die Definition des Pfads */kap03/jsp/* ist also nicht möglich, da das Verzeichnis */jsp* nicht von */kap03/servlets/HelloWorld* umfasst wird.
Um ein Cookie an alle Ressourcen einer Webanwendung zu senden, verwenden Sie den Pfad /.

Die Domäne eines Cookie

Was für die Verzeichnisse eines Servers gilt, gilt natürlich auch für den Server und seine Domäne selbst. So wird ein Cookie vom Server *cookies.IhreDomain.de* zunächst nur an den Server *cookies* in der Domäne *IhreDomain.de* zurückgesandt. Wenn Sie Ihre Cookies innerhalb der gesamten Domäne verwenden wollen, so können Sie diese ähnlich dem Pfad verallgemeinern.

```
...
   // Verallgemeinerung der gültigen Domäne für ein Cookie ...
   cookie.setDomain(".IhreDomain.de");  // nur für Ihre Domain

   // ... oder auch:
   cookie.setDomain("..de");       // für die gesamte '.de'-Domain
...
```

Um zu verhindern, dass Server Cookies für fremde Domains setzen, muss die Domäne mit einem Punkt »**.**« beginnen.

Cookie einfügen

Achtung

Da auch Cookies als Response-Header übertragen werden, müssen sie vor dem eigentlichen Inhalt der Seite (Response-Body) erzeugt und eingefügt werden, da sie sonst eventuell nicht vom Client verarbeitet werden können. Erzeugen Sie Ihre Cookies also möglichst zu Beginn.

Nachdem Sie Ihren Cookie konfiguriert haben, können Sie ihn auch als Header in die Response einfügen. Dies übernimmt die Methode `addCookie()`. Das folgende Listing zeigt Ihnen die Erstellung und Konfiguration des Cookies noch einmal vollständig.

Listing 3.27 Erzeugen und Konfigurieren eines Cookie

```
...
   /** Servlet-Beispiel: Setzen eines Cookies */
   public void doGet(HttpServletRequest request,
                     HttpServletResponse response)
      throws IOException, ServletException {

      // Erzeugen des Cookie
      Cookie cookie = new Cookie("TestCookie", "TestValue");
```

```
      // Lebensdauer in Sekunden setzen (hier 1 Tag)
      cookie.setMaxAge(60*60*24);

      // Pfad auf alle Servlets verallgemeinern
      cookie.setPath("/04_Servlets/servlets");

      // Cookie-Header einfügen (Cookie: TestCookie=TestValue)
      response.addCookie(cookie);

      // ...
      // weiterführender Code Ihrer Service-Methode
      // ...
   }
...
```

Listing 3.27 (Forts.)
Erzeugen und Konfigurieren eines Cookie

Nun können Sie beispielsweise ein zu verwaltendes Objekt mit dem frei gewählten (eindeutigen) Wert (`TestValue`) in einer `HashMap` ablegen und dem Client beim nächsten Request wieder zuordnen.

3.4.2 Cookies lesen

Nachdem Sie Ihr Cookie beim ersten Request erzeugt und in die Response eingefügt haben, möchten Sie diesen Cookie und seinen Wert natürlich bei einem folgenden Request wieder auslesen. Dazu dient die Methode `getCookies()` des `HttpServletRequest`-Objekts. Da diese jedoch ein Array aller übermittelten Cookies zurückgibt, in dem die Position Ihres Cookie stets variieren kann, ist es sinnvoll, eine Methode zum Auslesen von Cookies zu definieren und diese über Basisklassen oder JavaBeans bereitzustellen:

```
...
   /**
    * Hilfsmethode zum Auslesen von Cookies:
    *
    * Gibt den Wert des übergebenen Cookies, so vorhanden zurück und
    * liefert anderenfalls 'null'
    *
    * @param HttpServletRequest Request-Objekt
    * @param String Name des zu extrahierenden Cookies
    * @return Wert des Cookies oder 'null', wenn es nicht existiert
    */
   private String getCookieValue(HttpServletRequest request,
                                 String name) {

      // Auslesen des Arrays mit allen Cookies des Requests
      Cookie[] cookies = request.getCookies();

      if (cookies != null) {

         // Suche nach dem passenden Cookie
         for (int i = 0; i < cookies.length; i++) {
            if (cookies[i].getName().equals(name)) {
               return cookies[i].getValue();
            }
         }
      }
      return null;
   }
...
```

Listing 3.28
Methode zum Auslesen eines Cookie-Value

Diese universelle Methode kann zum Auslesen aller Cookies verwendet werden. Der übergebene Parameter `name` enthält dabei den zuvor bei der Erzeugung vergebenen Namen des Cookie (z.B. `TestCookie`).

Typische Anwendungsfälle für Cookies

Ein häufiger Anwendungsfall für Cookies ist das Authentifizieren von Benutzern. Diese erhalten dabei nach erfolgreicher Anmeldung von einem Login-Servlet ein neues Cookie mit einer bestimmten Lebensdauer. Bei nachfolgenden Requests muss anschließend lediglich überprüft werden, ob ein solches Cookie existiert. Den Wert des Cookies können Sie außerdem für die Zuordnung von Java-Objekten zum jeweiligen Benutzer (beispielsweise bei virtuellen Warenkörben) verwenden.

3.5 Binäre Daten senden

Die größte Daseinsberechtigung für Servlets resultiert sicherlich aus Ihrer Fähigkeit, auch Binärdaten(-ströme) an den Client zu senden. In diesem Absatz werden Sie lernen, wie Sie sich diese Eigenschaft zunutze machen können, um beispielsweise dynamisch erzeugte Grafiken zu erstellen.

Info

Im nachfolgenden Beispiel werden die Grafiken mit dem `JPEGImageEncoder` von Sun erstellt, der (wenn auch wenig dokumentiert) Bestandteil des Java SDK ist.
Daneben gibt es natürlich eine Vielzahl weiterer Techniken, die es gestatten, dynamische Grafiken zu erstellen. Dieser Absatz soll dabei weniger die Erstellung dieser Grafiken, als vielmehr deren Übertragung an den Client demonstrieren.

3.5.1 Writer vs. Outputstream

Während Sie in JSPs auf die vordefinierte Variable `out` zurückgreifen, um Ausgaben in das resultierende Dokument zu erzeugen, müssen Sie diese Variable bei Servlets zunächst selbstständig aus dem `HttpServletResponse`-Objekt extrahieren (Listing 3.5).

Da das `HttpServletResponse`-Objekt neben der zeichenorientierten auch eine byteorientierte Verbindung ermöglicht, können Sie sich diesen Umstand nun zunutze machen. Hierfür verwendet das folgende Beispiel statt der Methode `getWriter()` die Methode `getOutputStream()`, die einen `OutputStream()` zurückgibt.

Achtung

Da sowohl `getWriter()` als auch `getOutputStream()` einen »Kanal« zum Client öffnen und dieser nur eine Antwort erwartet, dürfen Sie immer nur *eine von beiden* Methoden rufen – alles andere führt zu einer entsprechenden Fehlermeldung des Servlet-Containers.

3.5.2 Dynamische Bilder mit Servlets erstellen

Das folgende Servlet demonstriert Ihnen, wie Sie ein JPEG-Bild zur Laufzeit erstellen und an den Client übermitteln. Dazu wird auch das Wissen über das Setzen des MIME-Type aus dem vorangegangenen Kapitel benötigt.

Listing 3.29
Servlet zur Erzeugung dynamischer Daten (ImageServlet.java)

```
package de.akdabas.javaee.servlets;

import java.awt.*;
import java.awt.image.BufferedImage;
import java.io.IOException;

// 'Codec' und 'Encoder' sind bereits im Java SDK enthalten
import com.sun.image.codec.jpeg.JPEGCodec;
import com.sun.image.codec.jpeg.JPEGImageEncoder ;

import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/** Erzeugt ein dynamisches Bild und übermittelt es an den Client */
public class ImageServlet extends HttpServlet {

   /** Service-Methode des Servlets */
   public void doGet(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {

      // Setzen des Content-Types (MIME), damit der Browser die
      // binären Informationen auch korrekt interpretieren kann
      response.setContentType("image/jpeg");

      // Erzeugen des Grundbildes: ...
      int width = 400;
      int height = 50;
      BufferedImage image =
       new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB);
      Graphics2D g = (Graphics2D) image.getGraphics();

      // ... Setzen der Hintergrundfarbe, ...
      g.setColor(Color.gray);
      g.fillRect(0, 0, width, height);

      // ... ein wenig Text ...
      g.setColor(Color.white);
      g.setFont(new Font("Dialog", Font.PLAIN, 24));
      g.drawString("Java Enterprise Edition 5", 68, 35);

      // ... und ein kleiner Rahmen.
      g.setColor(Color.black);
      g.drawRect(0, 0, width - 1, height - 1);
```

Listing 3.29 (Forts.) Servlet zur Erzeugung dynamischer Daten (ImageServlet.java)

```
        // Aufräumen im AWT
        g.dispose();

        // Öffnen des OutputStreams, danach darf 'getWriter()'
        // NICHT mehr gerufen werden!
        ServletOutputStream sos = response.getOutputStream();

        // Senden des erzeugten Bildes an den Client
        JPEGImageEncoder encoder = JPEGCodec.createJPEGEncoder(sos);
        encoder.encode(image);
    }
}
```

Da hier nicht die Erzeugung von Grafiken mittels SDK demonstriert werden soll, beschränken wir uns auf die wichtigen, servlet-spezifischen Anweisungen.

Setzen des MIME-Type

Damit der Client den Inhalt »versteht« und ihn korrekt darstellen kann, muss man ihm mitteilen, um welche Art von »Dokument« es sich bei dieser Ressource handelt. In Kapitel 2 haben Sie gelernt, dass Sie dies über das Setzen des ContentType-Header tun können:

Listing 3.30 Setzen des JPEG-MIME-Type

```
...
        // Setzen des Content-Type (MIME), damit der Browser die
        // binären Informationen auch korrekt interpretieren kann
        response.setContentType("image/jpeg");
...
```

Öffnen des OutputStream

Nachdem Sie das Bild erzeugt haben, öffnen Sie den `OutputStream`, um es anschließend zu übertragen:

Listing 3.31 Öffnen des ServletOutputStream

```
...
        // Öffnen des OutputStream, dannach darf 'getWriter()'
        // NICHT mehr gerufen werden!
        ServletOutputStream sos = response.getOutputStream();
...
```

3.5.3 Die Konfiguration des Servlet

Bevor Sie Ihr Werk allerdings betrachten können, ist eine Konfiguration Ihres Servlet notwendig. Das folgende Listing zeigt Ihnen den entsprechenden Eintrag Ihres Web Deployment Descriptor (*web.xml*):

Listing 3.32 Konfiguration des ImageServlet

```
...
   <!-- Konfiguration des ImageServlets -->
   <servlet>
      <servlet-name>ImageServlet</servlet-name>
      <servlet-class>
         de.akdabas.javaee.servlets.ImageServlet
      </servlet-class>
   </servlet>
...
   <!-- Binden des Image-Servlets an einen URL -->
   <servlet-mapping>
      <servlet-name>ImageServlet</servlet-name>
```

```
      <url-pattern>/servlets/DynamicImage</url-pattern>
   </servlet-mapping>
...
```

Listing 3.32 (Forts.)
Konfiguration des ImageServlet

3.5.4 Das Resultat

Nachdem Sie das Servlet nun kompiliert und den Tomcat respektive JBoss neu gestartet haben, können Sie Ihre dynamisch erzeugte Grafik auch begutachten:

Abbildung 3.4
Dynamisch erzeugte JPEG-Grafik

3.6 Weiterleiten und Einfügen

Im Kapitel über JSPs haben Sie verschiedene Techniken kennen gelernt, mit denen Sie externe Ressourcen einbinden (*Include*) oder den Request an diese weiterleiten (*Forward*) können.

In Servlets kommt dafür ein so genanntes `RequestDispatcher`-Objekt zum Einsatz, das Sie über folgende Anweisung erzeugen können:

```
...
   // Erzeugen eines neuen RequestDispatcher-Objekts, welches
   // geeignet ist den Request weiterzuleiten oder eine externe
   // Ressource (JSP, Servlet, Image, ...) einzubinden
   javax.servlet.RequestDispatcher dispatcher =
      request.getRequestDispatcher("PfadDerExtenernenRessource");
...
```

Listing 3.33
Erzeugen eines RequestDispatcher-Objekts

Für den Pfad des `RequestDispatcher`-Objekts gelten dabei die gleichen Konventionen wie für das `PageContext`-Objekt in JSPs:

- Beginnt der relative Pfad mit einem Schrägstrich (»/«), so wird die Seite relativ zum Basisverzeichnis der Webanwendung (hier */kap04/*) gesucht.
- Anderenfalls wird der Pfad relativ zur aktuellen Seite aufgelöst.

Info

Ein Dispatcher leitet eingehende Requests (abhängig von Parametern) an andere Ressourcen weiter oder bindet diese ein. Frameworks wie Cocoon, Struts oder JSF verwenden Dispatcher-Servlets als zentralen Zugriffspunkt.

3.6.1 Einfügen einer Ressource

Nachdem Sie ein `RequestDispatcher`-Objekt für die gewünschte, externe Ressource erzeugt haben, können Sie deren Inhalt an beliebiger Stelle über folgenden Aufruf einfügen:

Listing 3.34 Einfügen einer externen Ressource

```
...
   // Einfügen des Inhalts der externen Ressource, analog zu
   // '<jsp:include />'
   dispatcher.include(request, response);
...
```

Der `RequestDispatcher` führt dabei einen neuen Request an die angegebene Ressource aus und fügt deren Ausgaben der aktuellen Variablen `out` hinzu.

3.6.2 Weiterleiten eines Request

Ebenso leicht wie das Einbinden einer externen Ressource ist das Weiterleiten von Requests:

Listing 3.35 Weiterleiten des Request an eine externe Ressource

```
...
   // Weiterleiten des Request, analog zu <jsp:forward />
   dispatcher.forward(request, response);
...
```

3.7 Fortgeschrittenes Arbeiten mit der Session

Bereits in Kapitel 2 haben Sie die Bedeutung und Verwendung der Session als Zwischenspeicher für anwenderspezifische Informationen kennen gelernt. Sie haben gesehen, wie Sie in Ihren JSPs auf die Session zugreifen, Objekte darin ablegen und auch wieder extrahieren können. Da Ihre Java-Server Pages im Hintergrund in Servlets umgewandelt werden, versteht es sich fast von selbst, dass Ihnen diese Funktionalität natürlich auch in Servlets zur Verfügung steht. Mit dem Unterschied, dass Sie die Session zunächst über die Request-Methode `getSession()` an eine lokale Variable binden müssen.

Info

Die in diesem Absatz gezeigten Techniken funktionieren natürlich auch mit JSPs. Da Sie aber ein grundlegendes Verständnis für die Funktionsweise des Web Deployment Descriptor voraussetzen, behandeln wir diese erst hier.

Der folgende Absatz wird das in Kapitel 2 Gelernte also nicht noch einmal wiederholen, sondern fortschrittliche Techniken im Umgang mit der Benutzer-Session und den darin gebundenen Objekten demonstrieren.

3.7.1 Aktive Sessions überwachen

Eine typische Anforderung von Webanwendungen ist es, sicherzustellen, dass der Webserver erst heruntergefahren oder neu gestartet wird, nachdem alle Anwender ihre Arbeit beendet haben. Da HTTP aber ein zustandsloses Protokoll ist, können Sie in der Regel nicht entscheiden, ob ein Anwender noch aktiv ist und weitere Requests von ihm zu erwarten sind, oder ob er sich eventuell schon ausgeloggt hat und nach Hause gefahren ist.

Das Interface HttpSessionListener

In diesen Situationen kann Ihnen ein so genannter `HttpSessionListener` weiterhelfen. Dabei handelt es sich um eine Java-Klasse, die ein bestimmtes Interface implementiert und immer dann benachrichtigt wird, wenn eine Session erzeugt wird oder verfällt.

Listing 3.36
Formale Definition des Interface HttpSession Listener

```
package javax.servlet.http;
import javax.servlet.http.HttpSessionEvent;

/** Verfolgt das Erzeugen und Verfallen von HttpSession-Objekten */
public Interface HttpSessionListener {

   /* Wird gerufen, wenn eine neue BenutzerSession erzeugt wurde */
   public void sessionCreated(HttpSessionEvent event)

   /* Wird gerufen, wenn eine Benutzer-Session verfällt (TimeOut)*/
   public void sessionDestroyed(HttpSessionEvent event)
}
```

Die erste der beiden Methoden wird immer dann gerufen, nachdem eine neue Session erzeugt wurde, und die zweite unmittelbar bevor eine `HttpSession` verfällt. Das dabei übergebene `HttpSessionEvent` enthält eine Referenz auf die jeweilige Session, die mit der Methode `getSession()` ausgelesen werden kann.

Tipp

Diese Event-gesteuerte Benachrichtigung über Zustandsänderungen funktioniert übrigens genauso wie beispielsweise das Eventhandling in modernen Java-GUIs unter Swing oder AWT.

Implementieren eines einfachen Session-Listener

Das Interface `HttpSessionListener` kann dabei von beliebigen JavaBeans implementiert werden, die anschließend als Listener für diese Ereignisse fungieren. Listing 3.37 zeigt ein Beispiel für den oben angesprochenen Listener für aktive Sessions

Listing 3.37
Ein Listener für aktive Sessions

```
package de.akdabas.javaee.listeners;

import java.util.*;
import javax.servlet.http.HttpSession;
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;
```

Listing 3.37 (Forts.) Ein Listener für aktive Sessions

```
/** Speichert aktive (gültige) Sessions */
public class ActiveSessionsListener implements HttpSessionListener {

   /** Enthält die aktiven Sessions <Session ID, Session> */
   private static Map<String, HttpSession> activeSessions =
      new HashMap<String, HttpSession>();

   /** Wird gerufen, wenn eine Session erzeugt wurde */
   public void sessionCreated(HttpSessionEvent event) {
      // Ermitteln der Session ID
      String sessionId = event.getSession().getId();

      // Debug-Ausgabe auf der Kommandozeile
      System.out.println("Session: " + sessionId + " erzeugt.");

      // Speichern der Session in den aktiven Sessions
      activeSessions.put(sessionId, event.getSession());
   }

   /** Wird gerufen, wenn eine Session verfällt */
   public void sessionDestroyed(HttpSessionEvent event) {
      // Ermitteln der Session ID
      String sessionId = event.getSession().getId();

      // Debug-Ausgabe auf der Kommandozeile
      System.out.println("Session: " + sessionId + " verfällt.");

      // Entfernen der Session aus den aktiven Sessions
      activeSessions.remove(sessionId);
   }

   /** Gibt die Liste der aktiven Sessions zurück */
   public static Map<String, HttpSession> getActiveSessions() {
      return activeSessions;
   }
}
```

Konfiguration des Session-Listener

Nachdem Sie den Listener implementiert und übersetzt haben, müssen Sie – analog zu Ihren Servlets – diesen natürlich noch am Servlet-Container registrieren. Dies erledigen Sie ebenfalls über einen Eintrag im Web Deployment Descriptor (`web.xml`).

Listing 3.38 Registrieren eines Session-Listener (schematisch)

```
<web-app>
...
   <!-- Konfiguration eines HttpSessionListeners -->
   <listener>
      <listener-class>PfadDerListenerKlasse</listener-class>
   </listener>
...
<web-app>
```

Achtung

Die DTD bzw. das Schema des Web Deployment Descriptor schreibt vor, dass alle `<listener>`-Tags vor den `<servlet>`-Definitionen stehen müssen. Fügen Sie den Eintrag aus Listing 3.39 also vor den Servlet-Konfigurationen ein (siehe auch Listing 3.41).

Wie Sie sehen, ist ein Listener weder an ein Servlet gebunden noch sind Sie auf einen einzelnen Listener beschränkt. Sie können beliebig viele Listener implementieren und alle über einen solchen Eintrag registrieren. Sollte das erwartete Ereignis eintreten, werden dann automatisch alle Listener benachrichtigt.

Listing 3.39
Konkrete Konfiguration des Listener

```
<web-app>
...
   <!-- Einbinden des ActiveSessionListeners -->
   <listener>
      <listener-class>
         de.akdabas.javaee.listeners.ActiveSessionsListener
      </listener-class>
   </listener>
...
</web-app>
```

Ein Servlet zur Demonstration des Session-Listener

Obwohl Ihr Session-Listener aus Listing 3.37 nach einem Neustart funktioniert und Sie dessen Funktionsweise durch Aufruf beliebiger JSPs oder Servlets bereits auf der Kommandozeile nachvollziehen können, werden Sie in diesem Abschnitt ein Servlet schreiben, welches den Listener verwendet und Ihnen außerdem zeigt, wie Sie Sessions auch manuell für ungültig erklären können. Diese Funktionalität ist vor allem während eines Logout-Request sinnvoll, damit ein Angreifer nicht einfach über die ZURÜCK-Schaltfläche in die Anwendung des vorherigen Terminalbenutzers zurückkehren kann.

Tipp

Das in diesem Beispiel verwendete Servlet lässt sich in Verbindung mit dem Session-Listener auch für die Implementierung einer leistungsfähigen Chat-Funktionalität verwenden.

Listing 3.40
Servlet zur Demonstration des ActiveSessionsListener

```
package de.akdabas.javaee.servlets;
import java.io.*;
import java.util.*;
import javax.servlet.http.*;
import javax.servlet.ServletException;

import de.akdabas.javaee.listeners.ActiveSessionsListener;

/**
 * Dieses Servlet gibt die aktiven Sessions in Form einer HTML-
 * Seite aus. Sie könnten dies auch mit einer JSP realisieren.
 */
public class ActiveSessionsServlet extends HttpServlet {

   /** Gibt die aktiven Sessions in Form einer HTML-Seite aus */
   public void doGet(HttpServletRequest request,
                     HttpServletResponse response)
      throws IOException, ServletException {

      // Binden des PrintWriters 'out'
      PrintWriter out = response.getWriter();
      out.println("<html><body>");
      out.println("Derzeit sind folgende Sessions gültig: ");
      out.println("<ul>");
```

Listing 3.40 (Forts.) Servlet zur Demonstration des ActiveSessionsListener

```
        // Ausgabe der aktiven Session-IDs samt Zeitstempel
        Collection<HttpSession> activeSessions =
           ActiveSessionsListener.getActiveSessions().values();
        for (HttpSession aSession : activeSessions) {
           Date creationDate = new Date(aSession.getCreationTime());
           out.println("<li>" + aSession.getId());
           out.println(" erzeugt am " + creationDate + "</li>");
        }

        // Binden bzw. Erzeugen der Session-Variable
        HttpSession session = request.getSession();

        // Session-Timeout dieses Anwenders auf 10 Sekunden setzen, um
        // Funktionsweise beim Timeout einer Session zu demonstrieren
        session.setMaxInactiveInterval(10);

        out.println("</ul></body></html>");
    }
}
```

Nun müssen Sie lediglich dieses Servlet konfigurieren und den Webserver anschließend neu starten.

Listing 3.41 Konfiguration des Servlet samt Session-Listener

```
<web-app>
...
   <!-- Einbinden des ActiveSessionListener -->
   <listener>
      <listener-class>
         de.akdabas.javaee.listeners.ActiveSessionsListener
      </listener-class>
   </listener>
...
   <!-- Einbinden eines Servlets zum Test des Listeners -->
   <servlet>
      <servlet-name>activeSessionsServlet</servlet-name>
      <servlet-class>
         de.akdabas.javaee.servlets.ActiveSessionsServlet
      </servlet-class>
   </servlet>
...
   <servlet-mapping>
      <servlet-name>activeSessionsServlet</servlet-name>
      <url-pattern>/servlets/ActiveSessionsServlet</url-pattern>
   </servlet-mapping>
</web-app>
```

Das Resultat

Nach abermaligem Neustart des Webservers können Sie die Funktionsweise des `HttpServletListener` auch über den Browser betrachten (Abbildung 3.5) und auf der Kommandozeile können Sie sich nun auch von der Funktionsweise beim Überschreiten des Session-Timeout überzeugen.

Info

Die Überprüfung, ob Sessions mit abgelaufenem Session-Timeout existieren, geschieht normalerweise in einem gesonderten Thread der Java Virtual Machine, welcher nur von Zeit zu Zeit angestoßen wird. Deshalb erscheint die zweite Ausgabe in diesem Beispiel vielleicht etwas verzögert.

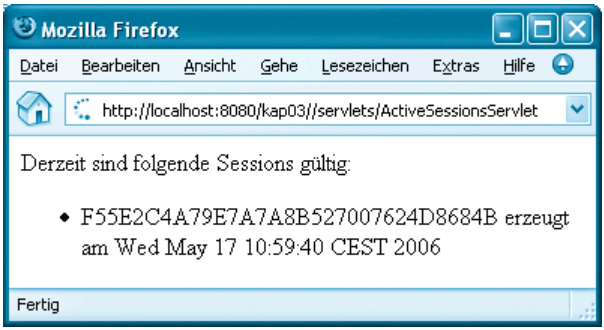


Abbildung 3.5
Ausgabe der aktiven Sessions

```
...
Session: F55E2C4A79E7A7A8B527007624D8684B erzeugt.
Session: F55E2C4A79E7A7A8B527007624D8684B verfällt.
...
```

Listing 3.42
Ausgabe auf der Kommandozeile

Achtung

In diesem Beispiel werden – der Einfachheit halber – die aktiven Sessions in einer statischen Klassenvariablen gespeichert, die von Instanz-Methoden gefüllt wird. In Produktionsumgebungen empfiehlt sich die Implementierung eines Singleton oder ähnlicher Zugriffsmethoden.

3.7.2 Weitere Session-Listener

Neben dem `HttpSessionListener` definiert die Spezifikation noch drei weitere Listener, deren Funktionsweise hier allerdings nur kurz angerissen werden soll.

Das Interface HttpSessionAttributeListener

Objekte mit diesem Interface funktionieren exakt wie der `HttpSessionListener` und werden auch wie dieser konfiguriert. Der Unterschied zwischen beiden besteht darin, dass das Interface `HttpSessionAttributeListener` Events über das Binden, Überschreiben und Löschen von Objekten in den Sessions registriert.

```
package javax.servlet.http;
import javax.servlet.http.HttpSessionBindingEvent;

/** Verfolgt Änderungen an Bindungen in HttpSession-Objekten */
public Interface HttpSessionAttributeListener {

   /* Gerufen, wenn ein neues Objekt an die Session gebunden wird */
   public void attributeAdded(HttpSessionBindingEvent event)

   /* Gerufen, wenn ein Objekt in der Session überschrieben wird */
   public void attributeReplaced(HttpSessionBindingEvent event)

   /* Gerufen, wenn ein Objekt aus der Session entfernt wird */
   public void attributeRemoved(HttpSessionBindingEvent event)
}
```

Listing 3.43
Signatur des Interface HttpSessionAttributeListener

Das Interface HttpSessionBindingListener

Für den Fall, dass ein gebundenes Objekt selbst darüber informiert werden soll, wann es an eine `HttpSession` gebunden ist und wann nicht, definiert die Servlet-Spezifikation das Interface `HttpSessionBindingListener` mit folgender Signatur:

Listing 3.44 Signatur des Interface HttpSessionBinding Listener

```
package javax.servlet.http;
import javax.servlet.http.HttpSessionBindingEvent;

/**
 * Für Objekte, die zugleich ihren eigenen Event-Handler darstellen
 * und selbst über Änderungen an Bindungen informiert werden sollen.
 */
public Interface HttpSessionBindingListener {

   /** Gerufen, wenn dieses Objekt an eine Session gebunden wird */
   public void valueBound(HttpSessionBindingEvent event)

   /* Gerufen, wenn dieses Objekt von einer Session gelöst wird */
   public void valueUnbound(HttpSessionBindingEvent event)
}
```

Tipp

Sie können übrigens eine JavaBean verwenden, um verschiedene Interfaces zu implementieren, diese fungiert dann als Listener für alle Event-Typen.

Wenn Sie ein Objekt an eine Session binden, überprüft der Servlet-Container, ob dieses das Interface `HttpSessionBindingListener` implementiert, und ruft gegebenenfalls die Methode `valueBound()`. Analog dazu werden Sie über die Methode `valueUnbound()` über das Lösen des Objekts von der Session informiert. Diese Callbacks laufen dabei unabhängig von den oben genannten Events des `HttpSessionAttributeListener` ab.

Da dieses Interface von den gebundenen Objekten selbst implementiert wird, müssen diese Listener nicht über den Web Deployment Descriptor (*web.xml*) eingebunden werden.

Das Interface HttpSessionActivationListener

Info

Unter Serialisieren versteht man in der Informatik das Überführen eines Objekts im Arbeitsspeicher in einen Datenstrom, der beispielsweise auch auf der Festplatte gespeichert werden könnte. Ein solches Objekt wird auch als persistentes Objekt bezeichnet. Neben der dauerhaften Speicherung spielt die Serialisierung beim Austausch von Objekten zwischen Servern eine große Rolle. Weitere Informationen hierüber finden Sie beispielsweise im Kapitel über EJBs.

Ein `HttpSessionActivationListener` kann schließlich dazu verwendet werden, das Serialisieren (Deaktivieren) und Deserialisieren (Aktivieren) von Session-Objekten zu verfolgen, wie es etwa beim Übertragen von Sessions in einem Cluster geschieht oder wenn Ihr Servlet-Container über Mechanismen zur persistenten Speicherung von Sessions verfügt.

Hierdurch haben Sie die Möglichkeit, lokal gültige Referenzen innerhalb einer Session zu schließen bzw. aufzubauen.

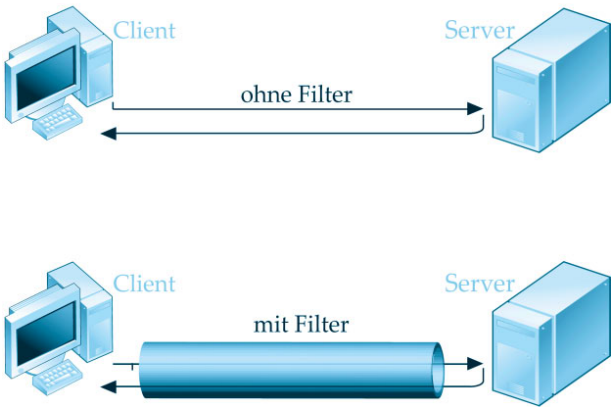
Listing 3.45
Signatur des Interface HttpSessionActivation Listener

```
package javax.servlet.http;
import javax.servlet.http.HttpSessionEvent;

public Interface HttpSessionActivationListener {

   /* Gerufen, wenn die Session dieses Objekts Re-Aktiviert wird */
   public void sessionDidActivate(HttpSessionEvent event)

   /* Gerufen, wenn die Session dieses Objekts serialisiert wird */
   public void sessionWillPassivate(HttpSessionEvent event)
}
```

3.8 Servlet-Filter – Servlets light

Ein wichtiges Merkmal für die Erstellung effizienter Webanwendungen stellt die Möglichkeit der Filterung dar. Sie können sich einen Filter dabei als eine Art Tunnel vorstellen, den sowohl Request als auch Response auf ihrem Weg zwischen Client und Servlet durchlaufen müssen. Abbildung 3.6 verdeutlicht dies.

Abbildung 3.6
Servlet-Filter – schematisch



Filter ermöglichen es, flexible Webanwendungen zu erstellen, und bieten ein breit gefächertes Einsatzfeld. Sie werden typischerweise verwendet um:

- Daten komprimiert zu übertragen,
- Request und Response zu loggen und Debug-Ausgaben zu erzeugen,
- Benutzereingaben aus Formularen zu validieren und Fehler abzufangen,

- Benutzer zu authentifizieren,
- Requests zu verteilen (RequestDispatching),
- XSL/T und Image-Conversion,
- ...

Info

Servlet-Filter lassen sich auch sehr gut zusammen mit Session-Listenern einsetzen, um flexible und wiederverwendbare Frameworks zu schaffen.

3.8.1 Das Interface javax.servlet.Filter

Alle Filter implementieren das Interface `javax.servlet.Filter`, welches die folgende Signatur besitzt:

Listing 3.46 Signatur des Filter-Interface

```
package javax.servlet;
import java.io.IOException;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.ServletException;

/* Filtert die Kommunikation zwischen Client und Servlet bzw. JSP */
public Interface Filter {

   /* Gerufen, zur Initialisierung des Filters */
   public void init(FilterConfig config);

   /* Service-Methode des Servlet-Filters */
   public void doFilter(ServletRequest request,
                        ServletResponse response,
                        FilterChain chain)
      throws IOException, ServletException;

   /* Gerufen, bevor der Filter aus dem Speicher entfernt wird */
   public void destroy();
}
```

Bei genauerem Hinsehen fällt die Analogie zu Servlets ins Auge:

- Filter durchlaufen den gleichen Lebenszyklus wie Servlets, wobei die einzelnen Methoden dieselben Aufgaben haben: Bereitstellen von Ressourcen, Service, Freigeben von belegten Ressourcen.
- Sie besitzen eine Service-Methode `doFilter()`, die mit den gleichen Objekten wie die Service-Methoden Ihrer Servlets parametrisiert wird und auch die gleiche `Exception` deklariert.
- Das `FilterConfig`-Objekt bietet nahezu dieselben Möglichkeiten wie ein `ServletConfig`-Objekt.

Info

Servlets und Servlet-Filter besitzen viele Analogien.

Im Unterschied zu Servlets enthält das API keine Basisimplementierung für Filter. Diese können Sie sich jedoch leicht selbst schaffen.

3.8.2 Ein einfacher Basisfilter

Das folgende Listing zeigt wohl den einfachsten Filter, den Sie gleichzeitig als Basisimplementierung verwenden können: Er tut buchstäblich nichts und leitet Request bzw. Response aus Abbildung 3.6 einfach nur hindurch:

Listing 3.47
Basisimplementierung eines Filters

```
package de.akdabas.javaee.filters;
import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.ServletException;

/** Basis-Implementierung eines Filters */
public class BaseFilter implements Filter {

   /** Die Konfiguration des Filters */
   protected FilterConfig filterConfig;

   /** Speichert die Konfiguration des Filters */
   public void init(FilterConfig aConfig) {
      filterConfig = aConfig;
   }

   /** Leitet Request und Response weiter */
   public void doFilter(ServletRequest request,
                        ServletResponse response,
                        FilterChain chain)
      throws IOException, ServletException {

      // Leitet den Request ohne Manipulationen weiter
      chain.doFilter(request, response);
   }

   /* Der Filter belegt prinzipiell keine Ressourcen */
   public void destroy() {
      filterConfig = null;
   }
}
```

Dieser Filter speichert lediglich die Filterkonfiguration in der Variablen `config` und leitet alle `ServletRequest`- und `ServletResponse`-Objekte einfach weiter. Hierzu rufen Sie die Methode `doFilter()` des übergebenen `FilterChain`-Objekts.

Info

Diese Basisimplementierung speichert die Filterkonfiguration in einer Variablen mit dem Scope `protected`, um Subklassen den Zugriff auf diese zu ermöglichen. Suns Design-Patterns folgend ist der Zugriff über Getter- und Setter-Methoden sicherer. Auf diese soll der Einfachheit halber hier jedoch verzichtet werden.

3.8.3 Konfiguration des Filters

Filter haben nicht nur einen ähnlichen Lebenszyklus wie Servlets, sie werden auch ganz analog zu diesen im Web Deployment Descriptor (*web.xml*) konfiguriert. Zuerst wählen Sie einen eindeutigen symbolischen Namen, verknüpfen diesen mit der Filter-Klasse und binden ihn anschließend an eine einzelne oder eine Klasse von URLs. Da Filter jedoch häufig zusammen mit Servlets eingesetzt werden und diese bereits an eine Klasse von URLs gebunden sind, können Sie Ihr Filter-Mapping statt mit einem URL-Muster auch mit einem symbolischen Servlet-Namen versehen. In diesem Fall kommt der Filter immer dann zum Einsatz, wenn der Request an dieses Servlet weitergeleitet wird.

Listing 3.48 Schematische Konfiguration vom Filtern

```
<web-app>
...
   <!-- Filterklasse an symbolischen Namen binden -->
   <filter>
      <filter-name>EindeutigerSymbolischerFilterName</filter-name>
      <filter-class>VollständigerPfadDerFilterKlasse</filter-class>
   </filter>
...
   <!-- Symbolischen Filter-Namen an URLs binden -->
   <filter-mapping>
      <filter-name>EindeutigerSymbolischerFilterName</filter-name>
      <url-pattern>PatternDerZuBindendeURLs</url-pattern>
   </filter-mapping>
...
   <!-- Symbolischen Filter-Namen mit einem Servlet verknüpfen -->
   <filter-mapping>
      <filter-name>EindeutigerSymbolischerFilterName</filter-name>
      <servlet-name>EindeutigerSymbolischerServleName</servlet-name>
   </filter-mapping>
...
</web-app>
```

Dabei kann die URL genau wie bei den Servlets entweder eindeutig sein oder eine ganze Klasse von URLs beschreiben, so wie in diesem Beispiel:

Listing 3.49 Konfiguration des Base-Filters

```
...
   <!-- Filterklasse an symbolischen Namen binden -->
   <filter>
      <filter-name>baseFilter</filter-name>
      <filter-class>
         de.akdabas.javaee.filters.BaseFilter
      </filter-class>
   </filter>
...
   <!-- Symbolischen Filter-Namen an URLs binden -->
   <filter-mapping>
      <filter-name>baseFilter</filter-name>
      <url-pattern>/*</url-pattern>
   </filter-mapping>
...
```

Das Pattern `/*` bewirkt dabei, dass der Filter bei allen eingehenden Requests zwischengeschaltet wird (vgl. Abbildung 3.6).

3.8.4 Manipulieren von Request und Response

Der große Vorteil von Servlet-Filtern ist, dass sie Request und Response manipulieren können, ohne dass die dahinter liegenden Servlets oder JSPs etwas davon merken. Dazu verwendet man in der Regel Wrapper-Klassen, die das gewünschte Objekt einhüllen und alle nicht veränderten Methodenaufrufe weiterleiten.

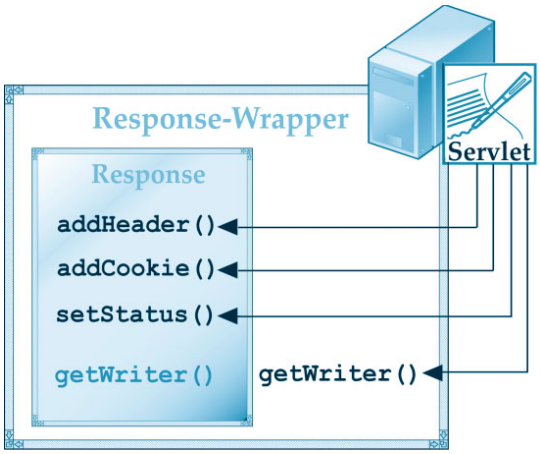


Abbildung 3.7
Funktionsweise eines Wrapper-Objekts

Nun müssen Sie Ihren Wrapper nicht von Grund auf neu schreiben, da Java bereits zwei Basisimplementierungen mitbringt. Diese implementieren alle von `HttpServletRequest` und `-Response` deklarierten Methoden und leiten diese an das eingehüllte Objekt weiter.

Info

Wenn Sie mit den *Design-Patterns* (Entwurfsmuster) vertraut sind, werden Sie bemerken, dass Wrapper das *Decorator-Pattern* (Dekorator-Muster) implementieren. Weitere Informationen bietet Ihnen das Buch »Entwurfsmuster« (ISBN 3827321999), erschienen im Juli 2004 bei Addison-Wesley.

Sie müssen lediglich von diesen Klassen erben und die gewünschten Methoden überschreiben. Auf diese Weise können Sie beispielsweise Filter schreiben, die bestimmte Request-Parameter oder das Character-Encoding setzen oder wie im nachfolgenden Beispiel die resultierende Seite formatieren.

Der ResponseFormatFilter

Als Nächstes können Sie einen Filter schreiben, der die Ausgabe der Servlets und JSPs abfängt, anschließend formatiert und weiterleitet.

Listing 3.50
Der ResponseFormatFilter

```
package de.akdabas.javaee.filters;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.StringTokenizer;
import de.akdabas.javaee.filters.BaseFilter;

/**
 * Diese Filter formatiert die resultierende HTML-Seite und
 * versieht den Begriff 'Masterclass' mit zusätzlichen Links
 */
public class ResponseFormatFilter extends BaseFilter
   implements Filter {

   /** Filtert die resultierende Seite */
   public void doFilter(ServletRequest request,
                        ServletResponse response,
                        FilterChain chain)
      throws IOException, ServletException {

      // Erzeugen des Wrapper-Objekts
      ResponseWrapper wrapper =
         new ResponseWrapper((HttpServletResponse) response);

      // Weiterleiten und Verarbeiten des Request
      chain.doFilter(request, wrapper);

      // Binden des PrintWriters 'out'
      PrintWriter out = response.getWriter();

      // Gespeichertes Dokument aus dem Wrapper extrahieren
      String content = wrapper.getContent();

      // StringTokenizer zur Zerlegung in Worte
      StringTokenizer st = new StringTokenizer(content,"\t\n\r,.;");
      while (st.hasMoreTokens()) {
         String word = st.nextToken();
         if (word.equals("Masterclass")) {
           word="<a href=\"http://javaee.akdabas.de\">"+word+"</a>";
         }
         out.write(" " + word + " ");
      }

      // Flushen und Schließen der Ausgabe
      out.close();
   }
}

/**
 * Hilfklasse: Wrapper zur Pufferung der Ausgabe.
 * Nur zu verwenden bei zeichenorientierten Seiten
 */
class ResponseWrapper extends HttpServletResponseWrapper
   implements HttpServletResponse {

   /* Pufferspeicher für die Daten */
   private CharArrayWriter buffer;

   /** Constructor */
   public ResponseWrapper(HttpServletResponse response) {

      // Superklasse initialisieren
      super(response);

      // Puffer initialisieren
      buffer = new CharArrayWriter();
```

Listing 3.50 (Forts.)
Der ResponseFormatFilter

```
    }

    /** Umgeleitete Methode des Reponse-Objekts */
    public PrintWriter getWriter() {
       return new PrintWriter(buffer);
    }

    /** Gibt die gepufferten Daten zurück */
    public String getContent() {
       return buffer.toString();
    }
}
```

Dieser Filter enthält eine innere Klasse, die von `HttpServletResponseWrapper` ableitet und sich nach außen hin wie ein `HttpServletResponse`-Objekt verhält.

Innerhalb des Filters erzeugt man ein Objekt dieser Klasse und leitet den Request anschließend mit diesem Wrapper-Objekt weiter.

Listing 3.51
Verwendung des Wrapper-Objekts statt der Original-Response

```
...
    // Erzeugen des Wrapper-Objekts
    ResponseWrapper wrapper =
       new ResponseWrapper((HttpServletResponse) response);

    // Weiterleiten und Verarbeiten des Requests
    chain.doFilter(request, wrapper);
...
```

Nun kann der Request von dem mit der URL verknüpften Objekt in gewohnter Weise behandelt werden. Mit einer einzigen Ausnahme: Wird die Methode `getWriter()` gerufen, so leitet der Wrapper diese nicht weiter, sondern gibt stattdessen einen `PrintWriter` zurück, dessen Ausgabe in ein `CharArrayWriter` umgeleitet wird. Der Client erhält also noch keine Daten.

Listing 3.52
Umhüllte Methode des Response-Objekts

```
...
    /** Umgeleitete Methode des Reponse-Objekts */
    public PrintWriter getWriter() {
       return new PrintWriter(buffer);
    }
...
```

Nachdem (!) der Request schließlich von den dahinter liegenden JSPs oder Servlets beantwortet wurde und die Methode `chain.doFilter()` zurückgekehrt ist, befindet sich das gesamte Dokument im Zwischenspeicher des `ResponseWrapper`. Nun haben Sie die Möglichkeit, den Inhalt des Dokuments nach dem Schlüsselwort `Masterclass` zu durchsuchen und dieses mit einem Link zu versehen.

Eine Testseite

Da JSPs ihre Daten in jedem Fall über das `Writer`-Objekt versenden und um zu demonstrieren, dass Filter auch auf diese wirken, können Sie die Wirkungsweise Ihres Filters an folgender Beispiel-JSP verfolgen.

Listing 3.53 Test-JSP für den ResponseFormatFilter

```
<html>
   <body style="text-align:center; color:green">
      Diese Seite enthält die gesuchte Zeichenkette Masterclass.
   </body>
</html>
```

Konfiguration des Filters

Dieser Filter soll zunächst nur für JSPs verwendet werden.

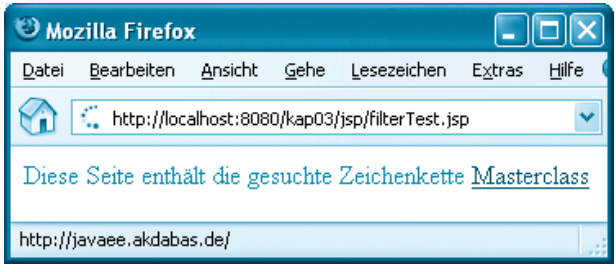
Listing 3.54 Konfiguration des ResponseFormatFilter

```
...
   <filter>
      <filter-name>responseFormatFilter</filter-name>
      <filter-class>
         de.akdabas.javaee.filters.ResponseFormatFilter
      </filter-class>
   </filter>
...
   <filter-mapping>
      <filter-name>responseFormatFilter</filter-name>
      <url-pattern>*.jsp</url-pattern>
   </filter-mapping>
...
```

Das Resultat

Wie üblich, sollten Sie Ihren Webserver neu starten oder die Anwendung über den Tomcat Manager reinitialisieren und können anschließend die Funktionsweise Ihres Filters prüfen:

Abbildung 3.8 Eine mit Servlet-Filtern formatierte HTML-Seite



3.8.5 Filterketten

Der große Vorteil von Servlet-Filtern gegenüber gewöhnlichen Servlets besteht darin, dass sich diese unbegrenzt ineinander verschachteln lassen. So können Sie beispielsweise innerhalb der Servicemethode `doFilter()` nicht entscheiden, ob es sich bei dem übergebenen Request um einen Wrapper handelt oder ob das Objekt, an das Sie den Request weiterleiten (`chain.doFilter()`), eine JSP, ein Servlet oder nur ein weiterer Servlet-Filter ist. Über diesen Mechanismus können Sie flexible und wiederverwendbare Komponenten definieren und diese nach Bedarf konfigurieren.

Info

Hintereinandergeschaltete Filter, die Request und Response nacheinander durchlaufen, werden als *Filterketten* bezeichnet.

Wer filtert wen?

Die Möglichkeit, verschiedene Filter zu verketten, wirft die interessante Frage auf, wer denn nun wen filtert? Alle an eine bestimmte URL gebundenen Filter werden beim Aufruf dieser aktiv und bilden eine *Filterkette* (engl. Filter-Chain), durch die das `Request`-Objekt der Reihe nach durchgereicht wird, bis es schließlich von der eigentlichen Ressource beantwortet wird. Anschließt durchläuft das `Response`-Objekt dieselbe Kette in ungekehrter Reihenfolge.

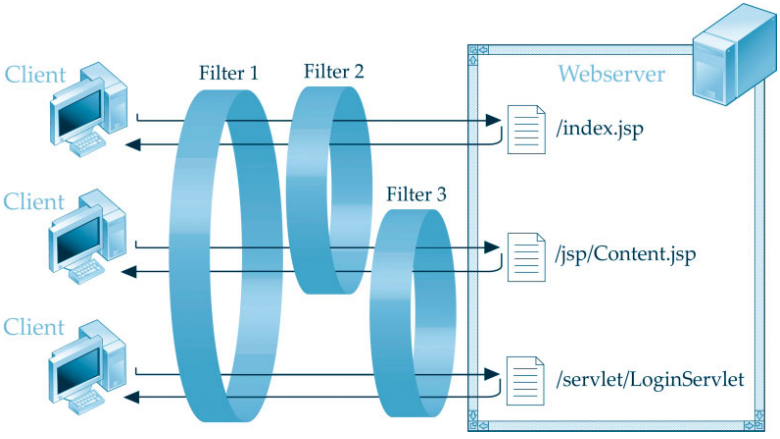


Abbildung 3.9
Schematischer Aufbau von Filterketten

Reihenfolge der Filter festlegen

Die Filter werden in der Reihenfolge geschaltet, in der sie im Web Deployment Descriptor definiert werden. Dabei ist zwischen den an URLs gebundenen Filtern und Filtern für benannte Servlets zu unterscheiden.

Listing 3.55
Gemischte Definition von Filtern

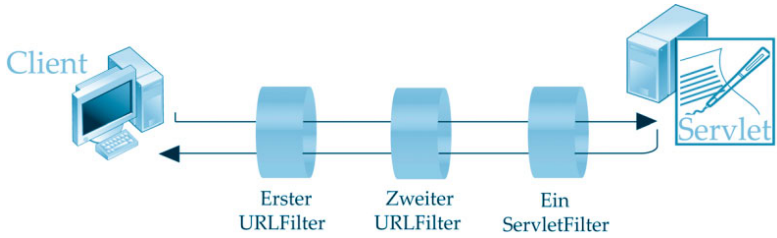
```
<web-app>
...
   <filter>
      <filter-name>ErsterAnURLgebundenerFilter</filter-name>
      <filter-class>VollständigerPfadDerFilterKlasse</filter-class>
   </filter>

   <filter>
      <filter-name>EinAnEinServletGebundenerFilter</filter-name>
      <filter-class>VollständigerPfadDerFilterKlasse</filter-class>
   </filter>

   <filter>
      <filter-name>ZweiterAnURLgebundenerFilter </filter-name>
      <filter-class>VollständigerPfadDerFilterKlasse</filter-class>
   </filter>
...
   <filter-mapping>
      <filter-name>ErsterAnURLgebundenerFilter</filter-name>
      <url-pattern>EinPattern</url-pattern>
   </filter-mapping>

   <filter-mapping>
      <filter-name>ZweiterAnURLgebundenerFilter</filter-name>
      <url-pattern>EinPattern</url-pattern>
   </filter-mapping>
```

Listing 3.55 (Forts.) Gemischte Definition von Filtern

```
    <filter-mapping>
       <filter-name>EinAnEinServletGebundenerFilter</filter-name>
       <servlet-name>EinSymbolischerServletName</servlet-name>
    </filter-mapping>
...
</web-app>
```

Würden Sie nun einen Request auslösen, der auf alle drei Filter passt, würde dieser zunächst `ErsterURLFilter` durchlaufen – er ist als Erstes definiert –, dann `ZweiterURLFilter` – er ist der zweite an eine URL gebundene Filter – und anschließend `EinServletFilter` – der erste Filter für benannte Servlets –, bevor er vom Servlet beantwortet oder weitergeleitet wird. Die Response nimmt den ganzen Weg dann in umgekehrter Reihenfolge:

Abbildung 3.10 Aufbau gestaffelter Filter



3.9 Zusammenfassung

Servlets oder JavaServer Pages – die besten Resultate erhalten Sie meist, wenn Sie beide Techniken zusammen einsetzen und ihre jeweiligen Stärken miteinander kombinieren. JSPs sind auf textbasierte Dokumente ausgerichtet, in denen sich die enthaltene Logik vor allem auf Entscheidungen zum Aussehen der resultierenden Seite beschränkt. Aufwändige Operationen, die auf Hilfsklassen und Frameworks wie Enterprise JavaBeans (EJB) angewiesen sind, machen JSPs hingegen häufig unübersichtlich und führen zu monolithischen Konstrukten.

Info

Manchmal ist es nicht leicht zu entscheiden, ob sich eine dynamische Seite leichter mit einem Servlet oder einer JSP realisieren lässt, aber vielleicht hilft Ihnen folgende Faustregel: Servlets eignen sich vor allem für Seiten mit umfangreicher Geschäftslogik, dafür ist die Erstellung von Markup in der Regel aufwändiger – bei JSPs ist es genau andersherum.

Servlets hingegen sind in der Regel etwas aufwändiger zu erstellen, bieten dem Entwickler aber auch mehr Raum zur Gestaltung und Möglichkeiten der Modularisierung. Dafür ist das Erzeugen von HTML-konformen Dokumenten mitunter eine Geduldsprobe für den Programmierer. Große Frameworks wie Cocoon, Struts oder JavaServer Faces verwenden deshalb zentrale (Servlet-)Controller, um den Request zu bearbeiten, kapseln die Daten

in JavaBeans und leiten den Request abschließend zur Darstellung an eine JSP weiter, die auf den befüllten JavaBeans operiert.

Neben JSPs und Servlets, enthält die Java-EE-Spezifikation viele weitere nützliche Elemente, welche flexible Lösungen für immer wiederkehrende Herausforderungen bereithalten. So haben Sie in diesem Kapitel z.B. auch Session-Listener, Servlet-Filter und Request- bzw. Response-Wrapper kennen gelernt und vielleicht Lust bekommen, etwas tiefer im `javax.servlet`-API und entsprechenden Tutorials zu stöbern. Das nächste Kapitel zeigt Ihnen, wie Sie oft benötigte Logikbausteine in eigene Tags kapseln, um diese leicht wiederverwendbar zu machen, bevor die Java-EE-basierten Frontends mit einem Ausblick auf ein paar populäre Frameworks abgeschlossen werden.

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()`

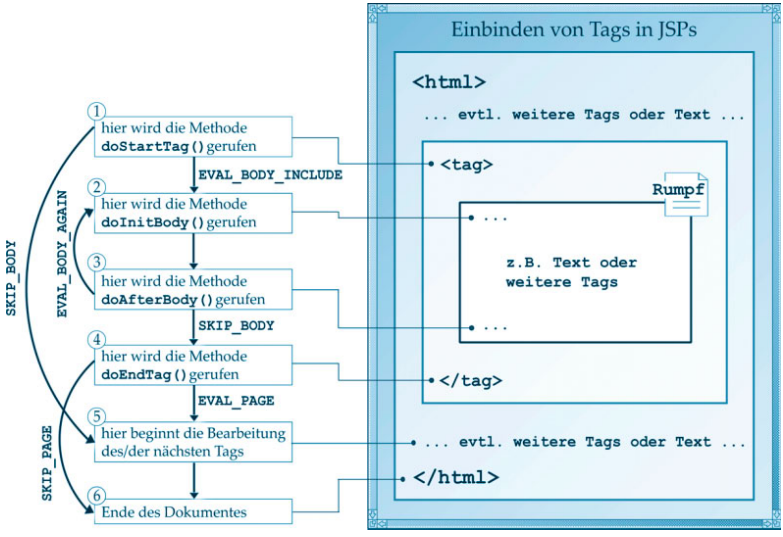
  Wenn Sie die Bearbeitung eines Tag nicht am Ende von `doStartTag()` abgebrochen haben, wird der Lebenszyklus des Tag durch die Methode `doEndTag()` abgeschlossen. Hier haben Sie die Möglichkeit, eventuell benötigte Ressourcen wieder freizugeben oder den Zustand des Tag in einer Datei zu speichern.

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.

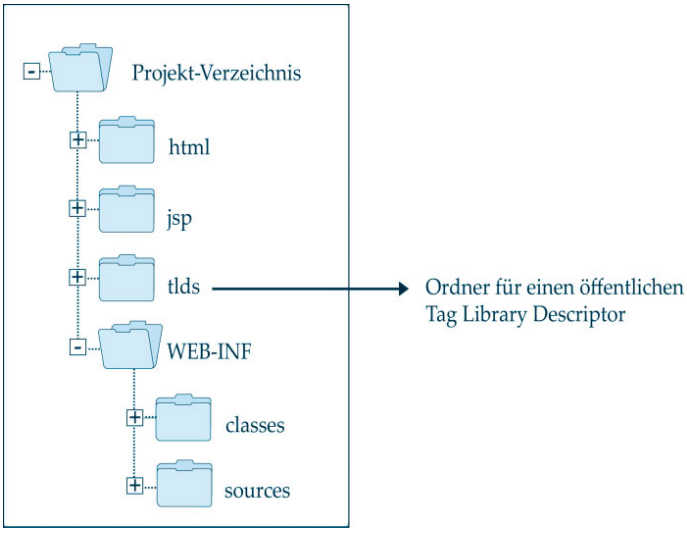


Abbildung 4.2
Veröffentlichen des Tag Library Descriptor

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:

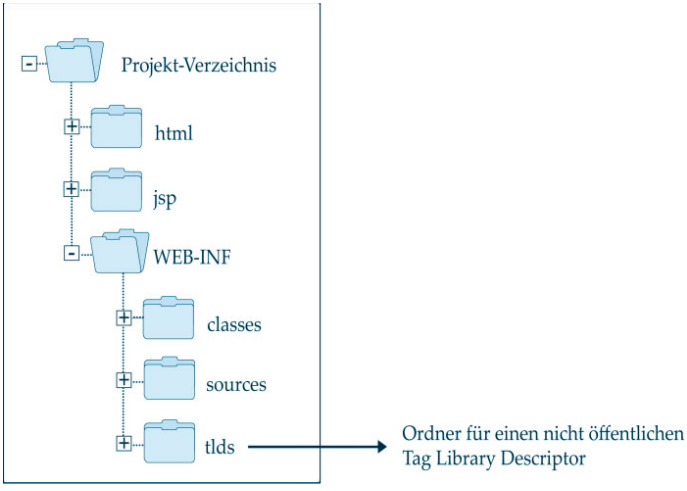


Abbildung 4.3
Indirektes Einbinden des Tag Library Descriptor

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.

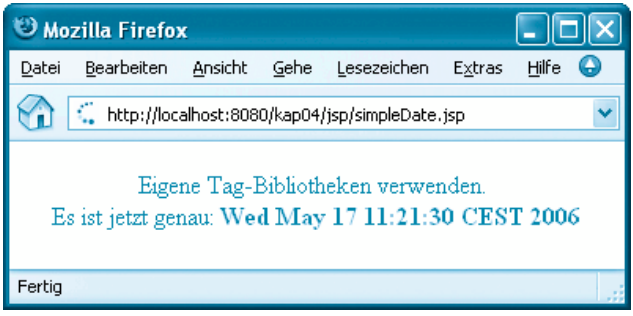
Jetzt können Sie das Tag `<date/>` beliebig innerhalb der JSP verwenden. Fast so, als wäre es ein Bestandteil der HTML-Spezifikation.

Listing 4.6
Verwenden des <date>-Tag (simpleDate.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:date />
   </body>
</html>
```

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.

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

```
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 (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.

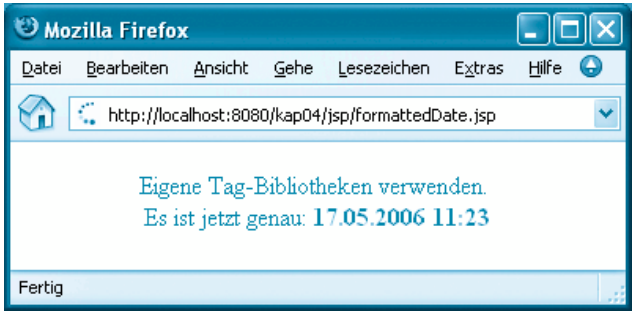
4.2.3 Verwendung innerhalb der JSP

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.

```
<%@ 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>
```

Listing 4.10
Verwendung von Runtime-Expressions für Tag-Attribute

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.

```
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
FilterSecureTag.java

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>
```

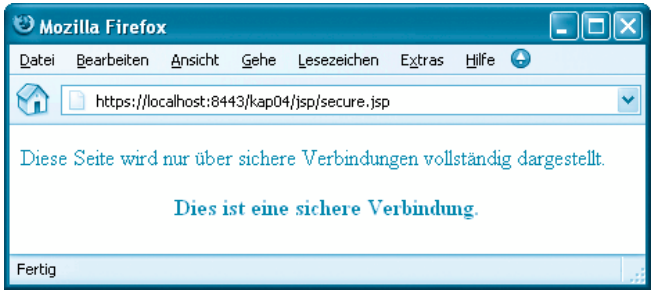


Abbildung 4.6
Aufruf über eine verschlüsselte Verbindung (HTTPS)

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:

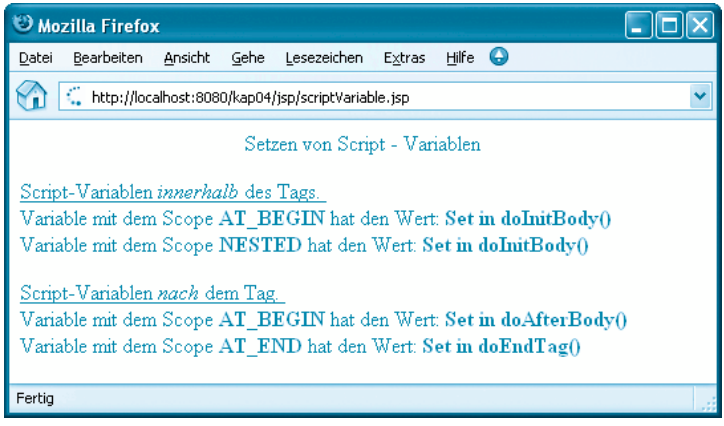
Listing 4.22 JSP zum Test der Script-Variablen

```
<%@ 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>
```

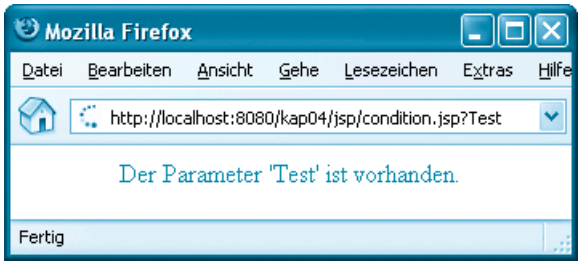


Abbildung 4.9
Tag-gesteuerte If-Else-Entscheidungen in JSPs

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.

5

Struts – die Diva unter den Frameworks

Dieses Kapitel behandelt ausnahmsweise keine Technologie aus Suns Java-EE-Spezifikation, sondern widmet sich einer Technik, die eng mit professioneller Software-Entwicklung verbunden ist: der Einsatz eines *Java Enterprise Framework*.

Info

Struts ist ein Framework, welches Ihnen bei der Entwicklung von java-basierten Webanwendungen als Rahmen dienen kann und Ihnen viele Standardaufgaben abnimmt.

Die Entscheidung über den Einsatz eines Framework als Basis für eine eigene Applikation hängt meist von vielen Faktoren ab: der Größe des Projekts, der Erfahrung der Programmierer und nicht zuletzt die ganz persönliche Präferenz des Programmierers. Ob Sie sich nun für oder gegen den Einsatz eines Framework entscheiden, sie sollten die Möglichkeiten und Konsequenzen Ihrer Wahl zumindest kennen.

Als Beispiel soll in diesem Kapitel *Apache Struts* (*http://struts.apache.org/*) dienen, eines der am weitesten verbreiteten Java-Frameworks überhaupt. Das Struts-Projekt wurde im Mai 2000 von Craig R. McClanahan ins Leben gerufen und liegt derzeit in der Version 1.2.9 vor, welche auch als Grundlage für dieses Kapitel dienen soll. Es unterstützt nicht nur die Entwicklung webbasierter Applikationen, sondern bietet auch viele Schnittstellen für die Interaktion mit Backend-Technologien wie *Enterprise JavaBeans (EJB)* und das *Java Naming and Directory Interface (JNDI)*, mit denen sich die nächsten Kapitel beschäftigen werden.

Ein komplexes Framework wie Struts sowie alle darin verborgenen Möglichkeiten zu beschreiben, sprengt den Rahmen eines solchen Kapitels natürlich bei weitem. Intention dieses Kapitels ist es, Ihnen einen guten

Überblick zu geben, wie die bisher kennen gelernten Technologien zusammenspielen und welche Idee hinter dem Einsatz eines Framework steckt.

Info

Craig R. McClanahan – Urvater von Struts – war auch federführend an der Entwicklung der JavaServer Faces beteiligt.

Als das Apache Struts Framework veröffentlicht wurde, leitete es in vielerlei Hinsicht ein Umdenken bei der Entwicklung webbasierter Frameworks ein. Viele der in diesem Framework vorgestellten Ideen wurden für andere Frameworks adaptiert und so ist es nicht verwunderlich, dass Ihnen im nächsten Kapitel über JavaServer Faces (JSF) ähnliche Konzepte wiederbegegnen werden.

5.1 Was sind Frameworks?

Eigentlich wissen Sie jetzt alles Nötige, um webbasierte Systeme zu entwickeln, die die Interaktion eines Benutzers mit dem Server ermöglichen. Mit *JSPs* und *Servlets* haben Sie Werkzeuge zur Hand, mit denen Sie statt statischer HTML-Seiten, dynamische Dokumente zur Laufzeit erzeugen und diese an den Client senden. *JavaBeans* unterstützen Sie dabei durch die Kapselung einzelner Eigenschaften eines Systems zu einem gemeinsamen Zustand und Tag-Bibliotheken ermöglichen es Ihnen, die Applikationslogik einmalig zu definieren und anschließend in allen JSPs gemeinsam zu nutzen.

Doch wenn Sie eine Weile mit den oben genannten Techniken arbeiten, fallen Ihnen eine ganze Reihe von Verbesserungsmöglichkeiten bei der Entwicklung von Webanwendungen ein.

- Sie können meist nur einen geringen Prozentsatz der Logik in Taglibs kapseln und auch die Wiederverwendbarkeit von Servlets ist gering.
- Innerhalb jeder JSP bzw. jedes Servlet müssen Sie ähnliche Schritte, zum Beispiel das Überprüfen von Parametern, immer wieder ausführen.
- Links auf andere Ressourcen, die Sie innerhalb der Webanwendung setzen, sind statisch, so dass eine Änderung des Workflow auch hier einen großen Aufwand bedeuten kann.
- Der Workflow der Webanwendung ist in der Applikationslogik (meist in den Servlets) versteckt und kann, aufgrund des umgebenden Codes, meist nur schwer nachvollzogen werden.

Info

Als Workflow bezeichnet man eine fest definierte Abfolge von Arbeitsschritten eines Programms. Ein typischer Workflow von Webanwendungen ist beispielsweise: Benutzerauthentifikation, Parameter auslesen, Daten verarbeiten, JSP rendern.

Keiner der oben genannten Nachteile wiegt, für sich allein genommen, wirklich schwer, doch in der Summe können Anpassungen innerhalb der Applikation manchmal in ein vollständiges Refactoring ausarten.

Info

Unter dem Begriff *Refactoring* versteht man die Überarbeitung von Quellcode, bei der die eigentliche Funktionalität erhalten bleibt. Das Refactoring soll vor allem die Lesbarkeit und die Struktur des Quellcodes verbessern, um spätere Erweiterungen oder Anpassungen zu erleichtern.

Der wesentlichste Aspekt Ihrer Arbeit besteht jedoch häufig in der Abbildung eines (Geschäfts-)Prozesses in einer Anwendung. Und damit Sie sich auf das konzentrieren können, unterstützen Sie *Frameworks*, indem sie Ihnen eine Reihe von immer wiederkehrenden Standardaufgaben abnehmen und einen einheitlichen Rahmen für Ihre Applikation schaffen.

Info

Integrierte Entwicklungsumgebungen (IDE) vereinfachen das Erstellen von Anwendungen.

Doch verwechseln Sie dieses Framework nicht mit Ihrer Entwicklungsumgebung (IDE), wie z.B. der Eclipse-Plattform oder IntelliJ IDEA. Diese haben auch die Aufgabe, Ihnen die Software-Entwicklung zu erleichtern und dabei auftretende Standardaufgaben wie die Erstellung einer *JavaBean* oder die Erzeugung der Dokumentation zu automatisieren. Doch die Arbeit einer IDE endet meist mit der Fertigstellung der Software oder kann auch nachträglich von einer anderen IDE übernommen werden. Im Extremfall können Sie sogar ganz auf den Einsatz einer IDE verzichten und Ihre Programme mit einem einfachen Editor erzeugen.

Frameworks hingegen bilden eine Plattform, auf die Ihre Applikation aufbauen kann. Die Arbeit eines Framework beginnt also erst zur Laufzeit des Programms selbst und da jedes Framework seine eigene Philosophie mitbringt, ist ein Wechsel im Nachhinein oft nur durch ein *Refactoring* der Applikation möglich.

Info

Frameworks bilden eine Plattform für Ihre Anwendungen und kommen erst zur Laufzeit des Programms selbst zum Einsatz.

Es gibt die unterschiedlichsten Typen von Programmierern: solche, die keiner Software vertrauen, die sie nicht selbst geschrieben haben; solche, die stets auf der Suche nach der neuesten, kaffeekochenden Entwicklungsumgebung, samt vorkonfigurierter Webapplikation sind; und jene, die die guten alten Zeiten vermissen, in denen Programmieren noch einen Touch von Erfindergeist hatte und nicht nur aus dem Konfigurieren vordefinierter Templates bestand. Und für sie alle gibt es Frameworks, die dem jeweiligen Programmierstil am ehesten entsprechen. So kann Ihnen dieses Kapitel lediglich zeigen, wie die bisher besprochenen Technologien zusammenspielen und wie ein Framework unter Umständen auch einzelne Technologien *modifiziert*: Obwohl es sich beispielsweise um eine Webanwendung handelt, werden Sie in diesem Kapitel beispielsweise keine eigenen Servlets schreiben ...

Nur Mut! Wenn Ihnen dieser Ansatz nicht gefällt, machen Sie sich auf die Suche nach »Ihrem« Framework und zur Not entwickeln Sie eben Ihr eigenes. Doch nun ist es Zeit für die Diva unter den Frameworks webbasierter Applikationen. Vorhang auf für Struts – und Action!

5.2 Struts – ein Webframework

Als Sun die Servlet-Spezifikation freigab, erkannten viele Programmierer die Vorteile, die dieser Ansatz brachte: Servlets waren leistungsstärker als vergleichbare CGI-Lösungen, sie waren portabel und nahezu unbegrenzt erweiterbar. Doch die vielen `println()`-Anweisungen und die umständliche Handhabung der Anführungszeichen stellte die Geduld vieler Programmierer auf eine harte Probe. Suns Antwort auf dieses Problem waren JavaServer Pages (JSP) und für einige Zeit schien die Welt wieder in Ordnung. Doch Java-Code innerhalb von JSPs erwies sich als schwer wartbar und wenig erweiterungsfähig.

Die Lösung bestand darin, beide gemeinsam einzusetzen: JSP waren für die Darstellung verantwortlich, während Servlets die Steuerung übernahmen und über JavaBeans miteinander kommunizierten. Diese Technik wurde unter dem Name »Model 2« bekannt, weil sie zeitlich nach den reinen JSP-Lösungen – die als »Model 1« bezeichnet werden – entwickelt wurde. Und da es in der Welt der Informatik nur selten etwas wirklich Neues gibt, zeigte sich bald, dass hinter dem »Model 2«-Konzept nichts anderes als das klassische Model-View-Controller-Pattern (MVC) der Programmiersprache Smalltalk steckte.

Model-View-Controller-(MVC-)Prinzip

Der MVC-Ansatz sieht eine Aufteilung der Applikation in drei Schichten vor:

- Die Datenrepräsentation und Geschäftslogik (Model)
- Die Darstellung und Interaktion mit dem Benutzer (View)
- Ein Vermittler zwischen diesen beiden Komponenten (Controller)

Bei webbasierten Systemen besteht der Controller natürlich aus einem Servlet, das zunächst alle eingehenden Requests entgegennimmt und deren Daten mit der im Hintergrund arbeitenden Geschäftlogik synchronisiert. Ist die Anfrage bearbeitet, leitet es den Request an eine JSP weiter, die die Ausgabe übernimmt.

Tipp

Diese Konfiguration wird in den Entwurfsmustern auch als Front-Controller bezeichnet.

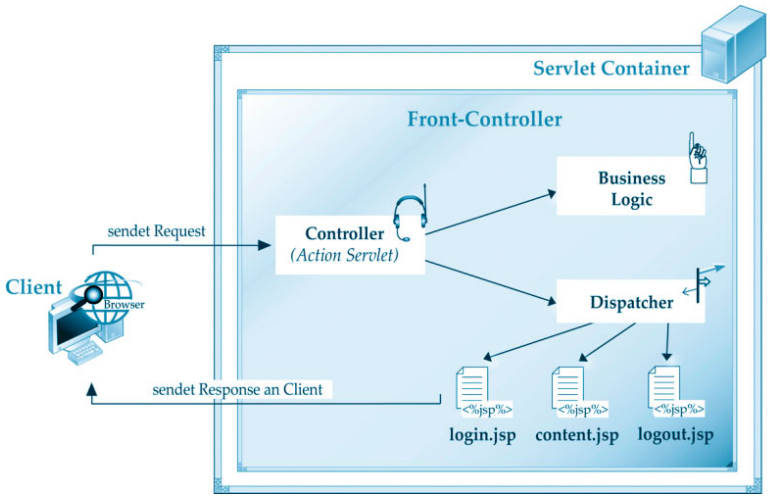


Abbildung 5.1
Das Struts-Servlet als Front-Controller

Die Vorteile dieses Ansatzes bestehen darin, dass sowohl Model- als auch View-Komponenten lediglich den Controller kennen müssen und nahezu beliebig ausgetauscht oder erweitert werden können. So kann eine JSP die Darstellung verschiedener Teile des Model übernehmen oder eine Model-Komponente die Anfragen unterschiedlicher JSPs bearbeiten.

5.2.1 Struts-Komponenten

Struts-basierte Applikationen werden im Wesentlichen über eine zentrale Konfigurationsdatei (*struts-config.xml*) ähnlich dem Web Deployment Descriptor (*web.xml*) konfiguriert. In dieser werden alle verwendeten Kom-

ponenten benannt und über symbolische Links miteinander verknüpft. Der Name Struts bedeutet dabei soviel wie Stütze oder Strebe, denn innerhalb des Framework kann eine neue Komponente über diese Konfigurationsdatei in die Applikation eingebunden und mit anderen verzahnt werden.

Info

Die Struts-Komponenten werden über die Konfigurationsdatei *struts-config..xml* miteinander verknüpft. Diese finden Sie unterhalb des Ordners *WEB-INF*.

JavaServer Pages (JSP) – realisieren die Darstellung

Diese Komponente sollten Sie nun schon zur Genüge kennen. Innerhalb von Struts-Applikationen dienen JSPs ausschließlich der Darstellung und sollten deshalb weder Scriptlets noch Deklarationen enthalten. JSPs kommunizieren über JavaBeans mit der Applikation und verwenden Tag-Bibliotheken, um ihren Ablauf zu steuern.

Tag-Bibliotheken – kapseln in den JSPs benötigte Logik

Struts wird zusammen mit vielen bereits vordefinierten Tag-Bibliotheken ausgeliefert, die Ihnen eine Vielzahl nützlicher Funktionen bereitstellen. Zusätzlich gibt es ein eigenes *Apache Jakarta Projekt*, welches sich ausschließlich mit der Entwicklung von Taglibs beschäftigt. Eine Übersicht bekommen Sie unter *http://jakarta.apache.org/taglibs/*. Daneben steht es Ihnen natürlich frei, eigene Bibliotheken einzubinden.

ActionForms – enthalten die HTTP-Parameter

Hinter ActionForms verbergen sich nichts anderes als spezielle *JavaBeans*, die die Aufgabe haben, Formulardaten, die der Benutzer in die JSP einträgt, zur weiteren Verwendung zwischenzuspeichern. Das Übertragen der Formulardaten in diese JavaBeans und zurück übernimmt das Struts-Framework für Sie.

ActionErrors – kapseln Fehler und Fehlermeldung

Egal, wie gut Sie Ihre Webanwendung gestalten: Hin und wieder wird es dazu kommen, dass ein Benutzer eine Fehleingabe tätigt oder schlicht und ergreifend eine Eingabe vergisst. In diesem Fall muss eine benutzerfreundliche Anwendung den Anwender auf den Fehler aufmerksam machen und dessen Korrektur ermöglichen. ActionErrors kapseln die aufgetretenen Fehler und ermöglichen eine verständliche Ausgabe der Exception.

ActionForwards – konfigurierbare Links zwischen den Komponenten

Um die lose Kopplung zwischen den JSPs und der Anwendungslogik zu ermöglichen, verwenden Sie statt fester Links vordefinierte ActionForwards, um von einer Komponente auf eine andere zu verweisen. Dadurch können wir einzelne Komponenten problemlos austauschen, indem wir einfach das Ziel unseres ActionForward umkonfigurieren.

Action – enthalten die Geschäftlogik

Sie sind die Hauptdarsteller in jeder Struts-Webapplikation und enthalten die (Geschäfts-)Logik. Vorher haben Servlets diese Aufgabe übernommen und Sie könnten den Controller auch dazu überreden, die Anfrage des Request statt an eine Action an ein anderes Servlet weiterzuleiten.

Actions operieren in der Regel auf den gesammelten Daten der ActionForm und implementieren einen bestimmten Workflow. Sie dienen außerdem dazu, mit etwaigen Backend-Komponenten wie beispielsweise Enterprise JavaBeans zu kommunizieren oder auf eine Datenbank zuzugreifen.

Konfiguration (struts-config.xml)

Dies ist das Herz jeder Struts-Applikation. In der XML-Datei *struts-config.xml* teilen wir unseren Komponenten Aufgaben zu und definieren den Workflow unserer Anwendung.

Ressourcen – enthalten den Text Ihrer Webseiten

Die JSPs einer Struts-Applikation unterscheiden sich noch in einem weiteren Punkt von herkömmlichen: Sie enthalten in der Regel keinen Text! Alle Zeichenketten werden über so genannte Ressource-Dateien verwaltet und erst zur Laufzeit integriert. Dies ermöglicht Ihnen:

- die Überarbeitung von Texten, ohne JSPs zu editieren
- die Gemeinsame Nutzung von Bezeichnungen über JSPs hinweg
- Unterstützung von Mehrsprachigkeit (Internationalization, i18n), indem Sie Ressource-Dateien in unterschiedlichen Sprachen erstellen

... und noch vieles mehr

Neben den hier genannten gibt es noch eine Vielzahl weiterer Komponenten, die die Entwicklung unserer Applikation vereinfachen. So können Sie beispielsweise dynamische Formular-Beans (*DynaBeans*) definieren, die zusammen mit Ihren JSPs wachsen, und Sie können über Struts Tiles JSP-Templates erstellen, die Ihre Layoutdefinitionen auch vererben.

Hierdurch können Sie Ihre JSPs effizient wiederverwenden und die gesamte Anwendung erhält ein einheitliches Look-and-Feel. Doch dies ist kein Buch über Struts und dieses Kapitel soll Ihnen lediglich Mut machen, sich bei der Erstellung Ihrer Applikationen von Frameworks unterstützen zu lassen.

5.2.2 Was Sie außerdem benötigen

Natürlich können Sie die gesamte Logik auch in den unterschiedlichen Struts-Komponenten unterbringen, genau wie Sie Ihre Webanwendung auch ausschließlich mit JSPs realisieren könnten. Doch Struts ist ein Framework für webbasierte Systeme und unterstützt Sie deshalb mit Komponenten, die auf die Bearbeitung von eingehenden Requests optimiert sind. Der Zustand einer Applikation wird hingegen normalerweise auch weiterhin durch eine Sammlung von JavaBeans repräsentiert, die beispielsweise als Value-Objekte zwischen EJBs ausgetauscht werden können.

Info

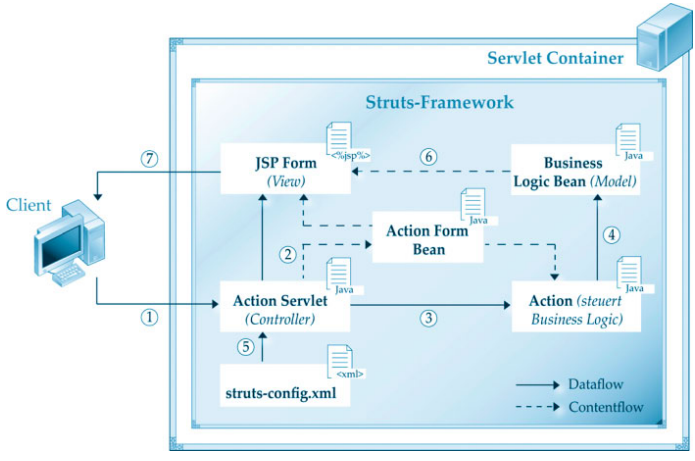
Streng genommen handelt es sich bei einer Ansammlung von JavaBeans zunächst natürlich lediglich um den Zustand einer Session. Der Zustand der Anwendung setzt sich dann aus den Zuständen aller aktiven Sessions zusammen.

In einem virtuellen Warenhaus wird es zum Beispiel eine JavaBean für den Warenkorb geben, die JavaBean-Artikel enthält. Eine weitere JavaBean kann dann das Benutzerprofil samt Kreditkarteninformationen enthalten und schließlich kann die gesamte Buchung in der Datenbank durch eine Transaktions-Bean erfolgen.

Dieser Teil der Applikation ist unabhängig vom Darstellungsmedium und sollte keinen Servlet- oder JSP-spezifischen Code enthalten. *ActionForms* oder *Servlets* übernehmen die Interaktion mit dem Client, während Ihre Datenrepräsentation unabhängig vom Übertragungsmedium oder der Darstellungsform realisiert werden sollte.

Abbildung 5.2 verdeutlicht das Zusammenspiel der einzelnen Komponenten noch einmal. Sollten Sie einmal den roten Faden des Beispiels verlieren, kommen Sie einfach an diesen Punkt zurück und vollziehen Sie die Abarbeitung eines HTTP-Request nach.

Abbildung 5.2
Zusammenspiel der einzelnen Struts-Komponenten



1. Der vom Client gesendete Request wird vom `ActionServlet` entgegengenommen.
2. Das `ActionServlet` überträgt die Daten des Request in die `ActionForm`, welche diese gegebenenfalls validiert.
3. Anschließend übergibt das `ActionServlet` das gefüllte `ActionForm` an die zugehörige `Action`.
4. Die `Action` kann die Daten der Form nun auch verwenden, um auf Geschäftslogik in Form von JavaBeans zuzugreifen.
5. Anschließend wird über die Konfigurationsdatei *struts-config.xml* bestimmt, welche JSP zur Darstellung des Ergebnisses verwendet werden soll.
6. Die JSP hat nun ihrerseits die Möglichkeit, auf die Daten der `ActionForm` und darin enthaltener Business-Logik Beans zuzugreifen, ...
7. ... um schließlich das Ergebnis darzustellen und die Kontrolle an den Client zurückzugeben, womit das Spiel von neuem beginnen kann.

5.3 Ein Adressbuch

In diesem Abschnitt entwickeln Sie ein webbasiertes Adressbuch, das auf dem Struts-Framework aufbaut und die wichtigsten Konzepte webbasierter Benutzerinteraktion zeigt. Im zweiten Teil werden Sie außerdem sehen, wie leicht sich eine auf Struts basierende Applikation erweitern lässt, indem Sie das Adressbuch beispielsweise mit einer Datenbank kommunizieren lassen werden. Doch nun Ärmel hoch und rein ins Vergnügen.

5.3.1 Bezug und Installation

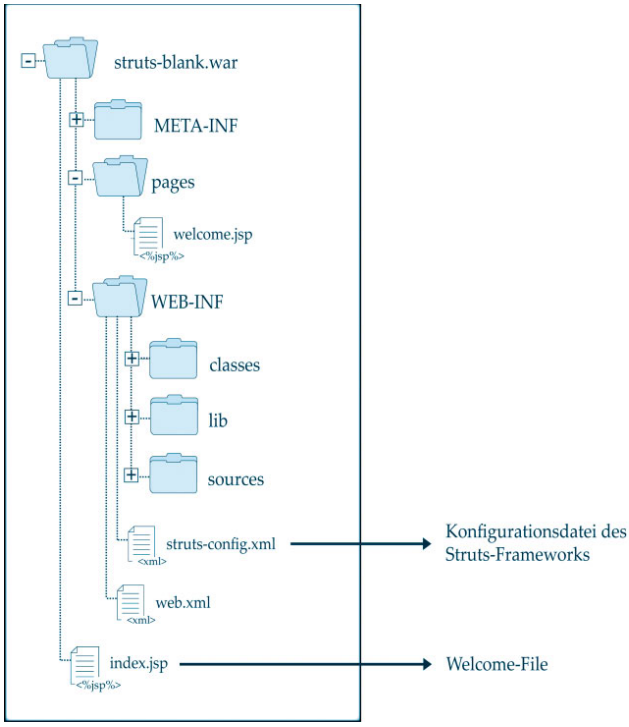
Zunächst müssen Sie das Struts Framework natürlich installieren. Dazu laden Sie sich einfach die aktuelle Struts-Distribution in Form einer ZIP-Datei unter *http://struts.apache.org* herunter und entpacken sie in ein Verzeichnis Ihrer Wahl. Die Distribution enthält verschiedene Webarchive (WAR), in denen sich nicht nur Beispielapplikationen und die Struts-Dokumentation befinden, sondern auch eine Art Starter-Kit namens *struts-blank.war*, das die vorkonfigurierte Struts-Plattform, inklusive aller zwingend benötigten Bibliotheken, enthält.

Tipp

Um die Struts-Dokumentation zu installieren, kopieren Sie einfach die entsprechenden WAR-Dateien ins Deployment-Verzeichnis Ihres Apache Tomcat oder JBoss und starten Sie diesen. Anschließend können Sie die Anwendung über folgende URL öffnen: *http://127.0.0.1:8080/struts-documentation/*.

Da Webarchive nichts anderes als ZIP-Archive mit einem Web Deployment Descriptor sind, können Sie die Datei *struts-blank.war* mit einem ZIP-Programm Ihrer Wahl entpacken und erhalten anschließend folgende Verzeichnisstruktur.

Abbildung 5.3
Entpackte Verzeichnisstruktur der Datei struts-blank.war



Sie können natürlich auch mit dieser Struktur weiterarbeiten. Um Sie jedoch in die Lage zu versetzen, eine Struts-Anwendung von Grund auf aufzubauen, wird Ihnen dieses Beispiel zeigen, wie Sie die zwingend benötigten Dateien in ein eigenes Projektverzeichnis überführen. Dieses finden Sie natürlich auch auf der beiliegenden CD.

Zwingend notwendig ist nur das Verzeichnis *WEB-INF/lib*, welches die benötigten Bibliotheken enthält, sowie alle Dateien, die direkt unterhalb des *WEB-INF*-Ordners liegen.

Die Tag Library Descriptors (TLDs) der Taglibs

In den vorangegangenen Beispielen haben Sie diese Dateien stets in einem eigenen Verzeichnis namens *WEB-INF/tlds* abgelegt, was die Übersichtlichkeit insbesondere dann erhöht, wenn Sie viele verschiedene Tag-Bibliotheken benötigen.

Auf jeden Fall müssen die Pfade zu den einzelnen TLSs innerhalb der Datei *web.xml* richtig angegeben sein. Näheres hierzu finden Sie in Kapitel 4.

Web Deployment Descriptor (web.xml)

Auch diese Datei ist schon vorkonfiguriert und muss, wenn Sie die TLDs nicht in ein anderes Verzeichnis verschieben, nicht editiert werden. Wir wollen trotzdem einen Blick darauf werfen, um die Prozesse hinter dem Framework besser verstehen zu können.

Listing 5.1
Der Web Deployment Descriptor (web.xml) – verkürzte Darstellung

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app>
   <display-name>Struts Addressbook Application</display-name>

   <!-- Konfiguration des Struts ActionServlet-Frontcontrollers -->
   <servlet>
      <servlet-name>action</servlet-name>
      <servlet-class>
         org.apache.struts.action.ActionServlet
      </servlet-class>

      <!-- Pfad der Struts Konfigurationsdatei -->
      <init-param>
         <param-name>config</param-name>
         <param-value>/WEB-INF/struts-config.xml</param-value>
      </init-param>
      <load-on-startup>2</load-on-startup>
   </servlet>

   <!-- Mapping des ActionServlets an alle URLs, die auf 'do'
        enden. -->
   <servlet-mapping>
      <servlet-name>action</servlet-name>
      <url-pattern>*.do</url-pattern>
   </servlet-mapping>

   <!-- Welche Datei soll aufgerufen werden, wenn die Anwendung ohne
        Angabe einer speziellen Datei aufgerufen wird. -->
   <welcome-file-list>
      <welcome-file>index.jsp</welcome-file>
   </welcome-file-list>

   <!-- Konfiguration der Struts Tag Library Descriptors -->
   <taglib>
      <taglib-uri>/tags/struts-bean</taglib-uri>
      <taglib-location>
         /WEB-INF/tlds/struts-bean.tld</taglib-location>
   </taglib>
   <taglib>
      <taglib-uri>/tags/struts-html</taglib-uri>
      <taglib-location>
         /WEB-INF/tlds/struts-html.tld</taglib-location>
   </taglib>
   <taglib>
      <taglib-uri>/tags/struts-logic</taglib-uri>
      <taglib-location>
         /WEB-INF/tlds/struts-logic.tld</taglib-location>
   </taglib>
</web-app>
```

Wie Sie sehen, verwendet das Framework nur ein einziges Servlet (`ActionServlet`) mit dem symbolischen Namen `action`, an das alle eingehenden Requests mit dem Präfix `.do` weitergeleitet werden. Die Konfiguration des Framework findet also weitestgehend nicht innerhalb der Datei `web.xml`, sondern über die Datei *struts-config.xml* statt, deren Speicherort Sie dem Action-Servlet über den Initialisierungsparameter `config` mitteilen.

Definition eines Welcome-File

Ein neues Element ist allerdings hinzugekommen: die Welcome-File-List. Mit diesem Element können Sie eine Liste von Dateien konfigurieren, die als Einstiegspunkt in die Anwendung geeignet sind. Wenn Sie beispielsweise im Internet nach einer bestimmten Adresse suchen und dazu den Suchdienst Google verwenden wollen, geben Sie typischerweise die URL *www.google.de* ein. Die Startseite, die Sie anschließend sehen, liegt allerdings unter der genauen URL *www.google.de/index.html*. Durch die Angabe der *Welcome-File-List* teilen Sie dem Server also mit, welche Ressource er ausliefern soll, falls keine Datei im Request spezifiziert ist. Praktischerweise verwenden Sie hier die Seite *index.jsp*, vorgeschrieben ist dies jedoch nicht.

Info

Welcome-Files geben an, welche Ressource der Webserver ausliefern soll, wenn keine spezielle Datei im Request angegeben ist.

Einbinden der Tag-Bibliotheken

Abschließend werden die benötigten Tag-Bibliotheken über die bereits vertrauten `<taglib>`-Elemente in die Applikation eingebunden und unter den symbolischen URLs */tags/struts-bean*, */tags/struts-html* und */tags/struts-logic* verfügbar gemacht.

Die Struts-Konfigurationsdatei struts-config.xml

Diese Datei hat (noch) keine spezifische Bezeichnung, wenngleich sie eine ungemein wichtige Rolle spielt. Sie befindet sich in der Regel zusammen mit den anderen Konfigurationsdateien im Ordner *WEB-INF* und hat folgenden schematischen Aufbau:

Listing 5.2
Schematischer Aufbau der Konfigurationsdatei struts-config.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE struts-config PUBLIC
   "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
   "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">

<struts-config>
   <!-- Dieses Element enthält später die Datenquelle
        (Siehe Absatz 5.4.1) -->
   <data-sources />

   <!-- Dieses Element enthält später ActionForms (Absatz 5.3.6) -->
   <form-beans />

   <!-- An dieser Stelle können ActionForwards definiert werden -->
   <global-forwards />

   <!-- Dieses Element enthält die Konfiguration der Actions
        (Absatz 5.3.7) -->
   <action-mappings />

   <!-- Hier werden die Ressourcen eingebunden (Absatz 5.3.5) -->
   <message-resources />
</struts-config>
```

Diese Datei wird vom ActionServlet während der Initialisierung geparst und verarbeitet. Sie werden nun nach und nach alle Komponenten der Anwendung in die jeweilige Sektion eintragen und auf diese Weise innerhalb der Anwendung registrieren. So werden beispielsweise alle ActionForms im Element `<form-beans>` unter einem symbolischen Namen abgelegt, unter dem sie im Element `<action-mappings>` ein oder mehreren Actions zugeordnet werden können.

Info

Struts Tiles ermöglichen es Entwicklern, komplexe JSPs aus verschiedenen wiederverwendbaren Modulen aufzubauen. Die Möglichkeiten von Tiles gehen dabei weit über das einfache Inkludieren von anderen JSPs hinaus.

Eine umfangreiche Struts-Applikation kann noch weitere Sektionen enthalten, in denen Sie beispielsweise applikationsweite Ausnahmen (`<global-exceptions>`) und ihre Behandlung oder spezielle Plug-ins (`<plug-in>`) wie zum Beispiel die Tiles-Template-Engine einbinden. Doch für die in diesem Kapitel entwickelte Basisapplikation benötigen Sie zunächst nur diese fünf Elemente.

Die finale Verzeichnisstruktur des Projekts

Um die Installation des Framework abzuschließen, erweitern Sie die Verzeichnisstruktur schließlich um einige Ordner wie in Abbildung 5.4 gezeigt.

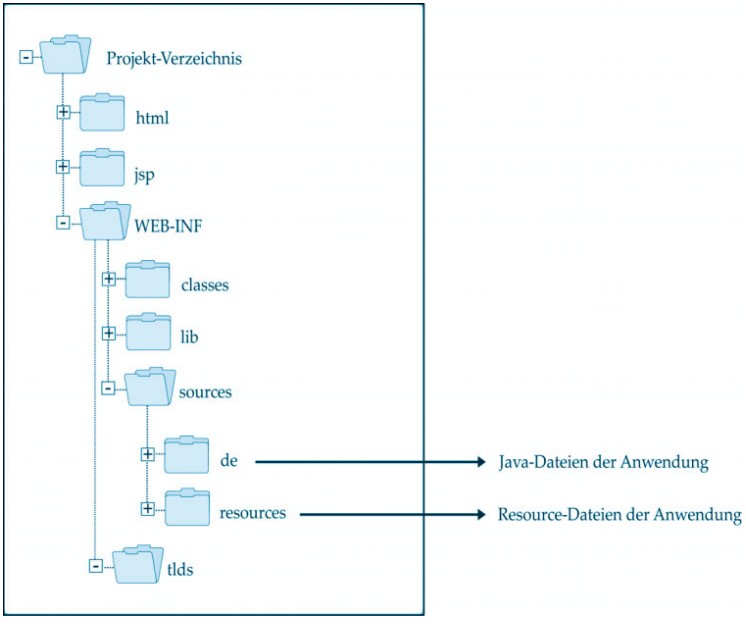


Abbildung 5.4
Finale Verzeichnisstruktur der Beispielanwendung

Damit sind die Vorbereitungen abgeschlossen und Sie können sich an die Umsetzung des Beispiels machen.

5.3.2 Unsere Business-Objekte

Software-Entwicklung sollte stets mit den Geschäftsobjekten beginnen, denn erst, wenn das Modell der Anwendung klar definiert ist, können Sie sich Gedanken über deren Darstellung und die Interaktion mit dem Benutzer machen.

In diesem Beispiel möchten Sie Adressobjekte über eine Webapplikation erstellen und verwalten und so besteht das Modell in unserem Fall zunächst aus einer einfachen `Adress`-Bean, die jeweils einen Eintrag repräsentiert. Dieses Objekt könnte beispielsweise auch von einer Enterprise JavaBean bereitgestellt werden.

Listing 5.3
Das Business-Objekt der Anwendung

```
package de.akdabas.struts.beans;

/**
 * Diese Bean stellt ein Value-Objekt dar und repräsentiert einen
 * Adress-Datensatz der Beispielanwendung.
 * Sie enthält keine Spezifika des Struts Frameworks
 */
public class Address {

   private String name;
   public void setName(String aLastName) {
      name = aLastName;
   }
   public String getName() {
      return name;
   }

   private String firstName;
   public void setFirstName(String aFirstName) {
      firstName = aFirstName;
   }
   public String getFirstName() {
      return firstName;
   }

   private String mail;
   public void setMail(String aMailAddress) {
      mail = aMailAddress;
   }
   public String getMail() {
      return mail;
   }

}
```

Später können Sie das Modell noch um JavaBeans erweitern, die diese Adressobjekte in einer Datenbank speichern oder den Versand von E-Mails ermöglichen, doch fürs Erste wollen wir lediglich Adressen zur Laufzeit erstellen können. Damit können wir uns an die Darstellung machen.

Tipp

Wenn Sie eine Klasse des Package `javax.servlet.*` in das Business-Objekt importieren, verzahnen Sie dadurch Ihr Business-Objekt mit der Darstellungsschicht und schieben der Wiederverwendung, etwa durch Enterprise JavaBeans (EJB), einen Riegel vor. Kapseln Sie Servlet-Logik in den Action-Klassen.

5.3.3 Die JavaServer Pages

Für die Basisapplikation benötigen Sie zunächst drei JSPs: eine zur Darstellung aller bereits vorhandenen Adressen, eine weitere zur Eingabe und die oben beschriebene Welcome-JSP (Listing 5.1), die den Benutzer in die Anwendung einführt.

Startseite – index.jsp

Jede Applikation benötigt einen definierten Einstiegspunkt, von dem aus die Benutzer die Anwendung betreten. In diesem Beispiel ist dies die Datei *index.jsp*, die Sie innerhalb des Web Deployment Descriptor auch schon entsprechend konfiguriert haben.

Listing 5.4
index.jsp

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

<!-- Weiterleiten des Requests durch Aufruf der Action mit dem
     symbolischen Namen 'initList' -->
<logic:redirect forward="initList"/>
```

Die JSP besteht effektiv nur aus zwei Zeilen und hat genau eine Aufgabe. Durch Verwendung des Struts-spezifischen Tag `<logic:redirect>` bringt sie den Browser dazu, einen ersten Request an das Struts-Servlet zu senden, und übergibt damit die Kontrolle über den Workflow an das Framework.

Das Struts-Servlet übernimmt die Weiterleitung des Folge-Request an die symbolische URL `initList` und steuert damit die Bearbeitung durch das dahinter liegende Action-Objekt. Der Benutzer bekommt von dieser Weiterleitung im besten Fall nichts mit.

Ausgabe der Adressliste – addressList.jsp

Die JSP *addressList.jsp* soll später alle diesem Nutzer zugeordneten Adressen ausgeben oder anzeigen, dass derzeit noch keine Adressen existieren.

Listing 5.5
addressList.jsp

```
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<%@ taglib uri="/tags/struts-logic" prefix="logic" %>

<!-- Struts Pendant zum klassischen HTML-Tag -->
<html:html locale="true">
   <head>
      <!-- Ausgabe einer Titel-Zeile -->
      <title><bean:message key="list.title"/></title>
```

Listing 5.5 (Forts.)
addressList.jsp

```
    <!-- Unterstützung relativer Pfade zu anderen Ressourcen -->
    <html:base/>
</head>
<body>

<!-- Ausgabe einer Seiten-Überschrift -->
<h3><bean:message key="list.heading"/></h3>

<table>
    <tr>
        <th><bean:message key="address.name"/></th>
        <th><bean:message key="address.firstName"/></th>
        <th><bean:message key="address.eMail"/></th>
    </tr>

    <!-- Test auf Vorhandensein eines Session-Attributs -->
    <logic:notPresent name="list" scope="session">
        <tr>
            <td colspan="3">
                <bean:message key="list.no.entries" />
            </td>
        </tr>
    </logic:notPresent>

    <!-- Iterieren über eine in der Session abgelegte Liste -->
    <logic:present name="list" scope="session" >
        <logic:iterate id="entry" name="list" scope="session">
          <tr>
           <td><bean:write name="entry" property="name" /></td>
           <td><bean:write name="entry" property="firstName"/></td>
           <td><bean:write name="entry" property="email" /></td>
          </tr>
        </logic:iterate>
    </logic:present>

    </table>
    <hr/>

    <!-- Pendant zum Anker-Tag in HTML -->
    <html:link forward="addressDetail">
        <bean:message key="list.new.entry" />
    </html:link>

</body>
</html:html>
```

In dieser JSP kommen eine Reihe weiterer Struts-spezifischer Tags zum Einsatz, die einige nützliche Features bieten:

- `<html:html>`

  Dieses Tag ersetzt das klassische `<html>`-Tag, welches das Dokument umschließen würde. Es bietet uns zusätzliche Unterstützung, um Standort und bevorzugte Sprache des Benutzers zu ermitteln.

- `<html:base>`

  Durch den Einsatz dieses Tag, das irgendwo innerhalb der Seite platziert werden kann, wird der Browser in die Lage versetzt, alle Pfade (Hyperlinks, etc.) relativ zur aktuellen Seite aufzulösen.

- `<html:link>`

  Schließlich ist dieses Tag das Pendant zum klassischen `<a>`-Tag in HTML. Es ermöglicht unter anderem die Weiterleitung an eine symbolische Struts-URL (`<global:forwards>`).

Neben den Pendants zu klassischen HTML-Tags existieren noch eine Reihe weiterer *Tag-Bibliotheken*, die es Ihnen z.B. ermöglichen,

- das Vorhandensein verschiedener Ressourcen zu überprüfen (`<logic:present>`, `<logic:notPresent>`) und entsprechend darauf zu reagieren,
- verschiedene Listenstrukturen zu durchlaufen (`<logic:iterate>`),
- verschiedene Ausgaben zu übernehmen (`<bean:message>`, `<bean:write>`). Abschnitt 5.3.4 stellt diese Tags genauer vor.

Abbildung 5.5 zeigt Ihnen die *addressList.jsp* beim ersten Start der Applikation. Hierbei existiert noch keine Adressliste in der Session.

Abbildung 5.5
addressList.jsp beim ersten Aufruf

Ein Formular, um neue Adressen einzugeben – addressDetail.jsp

Schließlich benötigen Sie noch eine JSP, über die Sie weitere Adressen in die Anwendung einfügen können. Diese Aufgabe übernimmt die *addressDetail.jsp*.

Listing 5.6
addressDetail.jsp

```
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<%@ taglib uri="/tags/struts-logic" prefix="logic" %>

<html:html locale="true">
   <head>
      <title><bean:message key="detail.title"/></title>
      <html:base/>
   </head>

   <body>
   <h3><bean:message key="detail.heading"/></h3>

   <!-- An dieser Stelle erfolgt ggf. die Ausgabe von
        Validierungs-Fehlern -->
   <html:errors/>
   <hr/>

   <table>
      <!-- Definition eines HTML-Formulars -->
      <html:form action="SaveAddress">
         <tr>
```

Listing 5.6 (Forts.)
addressDetail.jsp

```
            <td><bean:message key="address.name"/></td>
            <td><html:text property="name"/></td>
         </tr>
         <tr>
            <td><bean:message key="address.firstName"/></td>
            <td><html:text property="firstName"/></td>
         </tr>
         <tr>
            <td><bean:message key="address.eMail"/></td>
            <td><html:text property="email"/></td>
         </tr>
         <tr>
            <td>
               <!-- Hier wird ein Abbrechen-Link definiert ... -->
               <html:link forward="addressList">
                  <bean:message key="detail.cancel"/>
               </html:link>
            </td>
            <td>
               <!-- ... und hier ein Submit-Button -->
               <html:submit property="submit">
                  <bean:message key="detail.save"/>
               </html:submit>
            </td>
         </tr>
      </html:form>
   </table>
   </body>
</html:html>
```

In dieser JSP begegnen Ihnen schließlich drei weitere neue HTML-Tags.

- `<html:errors>`

  Sie haben bereits gelernt, dass Sie den Benutzer über so genannte *ActionErrors* auf etwaige Fehleingaben hinweisen können. Diese Hinweise werden durch das Tag `<html:errors>` in das Dokument integriert.

- `<html:form>`

  Dieses Tag ersetzt das klassische HTML-Tag `<form>`. Indem Sie dieses Tag verwenden, können Sie auch hier die symbolischen Links verwenden, um anzugeben, wohin die Daten dieses Formulars geschickt werden sollen.

- `<html:submit>`

  Über dieses Tag erzeugen Sie schließlich den Submit-Button, der die Form an die zugehörige Action übermittelt.

Sicherlich ist es am Anfang etwas lästig, statt der gewohnten HTML-Tags stets neue zu verwenden, doch werden Sie die zusätzliche Funktionalität bald nicht mehr missen wollen, zumal die neuen Tags durch Setzen von Standardwerten häufig kürzer sind als Ihre Originalversionen.

Listing 5.7
Vergleich zwischen klassischem <base>-Tag und Struts-Pendant

```
...
  <!-- In klassischem HTML würden Sie definieren: -->
  <base href="http://localhost:8080/kap05/jsp/addressList.jsp"/>
...
  <!-- Das Struts-Pendant verkürzt dies zu: -->
  <html:base />
...
```

Fazit

Ihre JSPs werden durch den exzessiven Einsatz der Tag-Bibliotheken wesentlich kürzer und damit überschaubarer. Durch die Verwendung von symbolischen Links zu anderen Komponenten können Sie den Workflow der Webanwendung auch im Nachhinein ändern, indem Sie einfach die Konfigurationsdatei *struts-config.xml* anpassen.

5.3.4 Struts Tag-Bibliotheken

Wir haben schon häufiger erwähnt, dass Struts bereits mit umfangreichen Tag-Bibliotheken ausgeliefert wird. Die wichtigsten sind `<logic>`, die verschiedene Ablaufstrukturen enthält, und `<bean>`, über die Sie die Aus- und Eingaben realisieren können.

Dieser Abschnitt stellt kurz die Funktionsweise der in diesem Beispiel verwendeten Tags vor. Ein vollständiges API zu diesen Bibliotheken finden Sie unter *http://struts.apache.org/struts-action/struts-taglib/*.

Die Logic-Bibliothek

Tabelle 5.1
Tags der Taglib <logic>

Tag	Beschreibung	Beispiel
`present` / `notPresent`	Stellt den Inhalt nur dar, wenn das angegebene Element existiert bzw. nicht existiert.	`<logic:present name="list" scope="session">`
`empty` / `notEmpty`	Überprüft, ob die angegebene `java.util.Collection` leer ist oder Elemente enthält.	`<logic:empty name="list">`
`iterate`	Wiederholt den Rumpf für jedes Element der übergebenen `java.lang.Collection`. Im Rumpf können Sie über die `id` auf das aktuelle Element zugreifen.	`<logic:iterate id="entry" name="list" scope="session">`

Die Bean-Biblitothek

Tabelle 5.2
Tags der Taglib <bean>

Tag	Beschreibung	Beispiel
`message`	Fügt eine internationalisierte Zeichenkette aus einer Ressource-Datei ein.	`<bean:message key="address.name"/>`
`write`	Über dieses Tag können Sie Attribute beliebiger JavaBeans analog `<jsp:getAttribute>` einfügen. Es ermöglicht Ihnen zusätzlich die Formatierung von Zahlen oder Datumswerten.	`<bean:write name="entry" property="name" />`
`include`	Dieses Tag ist das Pendant zu `<jsp:include>`. Dabei können wir auf symbolische Struts-Links (*ActionForward*) zugreifen.	`<bean:include forward="detail" >`

5.3.5 Ressource

Wie Ihnen sicher schon aufgefallen ist, enthalten die JSPs in diesem Beispiel keinerlei Text. Stattdessen fügen Sie das Tag `<bean:message>` als Platzhalter ein, welches später durch Texte in der jeweiligen Sprache ersetzt wird. Um die Platzhalter mit Leben zu füllen, erstellen Sie einfach folgende *application.properties*-Datei und kopieren sie ins Verzeichnis *WEB-INF/classes*.

Listing 5.8 application.properties

```
# -- Formatierende Tags für Validierungsfehler (<html:errors>)
errors.header=<UL>
errors.prefix=<LI>
errors.suffix=</LI>
errors.footer=</UL>

# Übergreifende Bezeichnungen verschiedener JSPs
address.name=Name
address.firstName=Vorname
address.eMail=eMail-Adresse

# Bezeichnungen der Adressliste (addressList.jsp)
list.title=Willkommen im Addressbuch
list.heading=Überblick
list.no.entries=Keine Adresse verfügbar
list.new.entry=Adresse hinzufügen

# Texte und Bezeichnungen des Adressformulars (addressDetail.jsp)
detail.error.name.required=Bitte geben Sie einen Namen ein !
detail.error.email.noat=eMail - Address enthält kein @ !

detail.title=Einen Benutzer hinzufügen
detail.heading=Bitte geben Sie die Benutzerdaten an
detail.cancel=Abbrechen
detail.save=Speichern
```

In dieser Datei geben Sie für jeden Platzhalter der JSP die Zeichenkette ein, durch die das `<bean:message>`-Tag ersetzt werden soll. Daneben existieren einige Standardschlüssel, durch die Sie z.B. die Ausgabe von Fehlermeldungen formatieren können.

Listing 5.9 Reservierte Keys zur Formatierung von ActionErrors

```
# -- Formatierende Tags für Validierungsfehler (<html:errors>)
errors.header=<UL>
errors.prefix=<LI>
errors.suffix=</LI>
errors.footer=</UL>
...
```

Wie Sie sehen, können Sie durch `MessageResources` beliebige Zeichenketten, also auch Markup, in die JSP einfügen. Dieser Markup wird jedoch erst vom Browser interpretiert und bleibt damit auf reine HTML-Tags beschränkt. Es ist nicht möglich, durch diese Technik Tags anderer Tag-Bibliotheken in das Dokument einzufügen und zur Ausführung zu bringen.

Internationalisierung

Um die Applikation nun mit einer weiteren Sprache – zum Beispiel Spanisch – auszustatten, reicht es aus, den Inhalt der gerade erzeugten Datei *application.properties* in die Datei *application_es_ES.properties* zu kopieren und diese ins Spanische zu übersetzen. Für Französisch erzeugen Sie die Datei *application_fr_FR.properties* und Engländer überraschen Sie schließlich mit der Datei *application_en_GB.properties*. Wie Sie sehen: Wenn Sie sich zunächst die kleine Mühe machen und Platzhalter statt Texten verwenden, ist die tatsächliche Internationalisierung ein Kinderspiel

Info

Bei einem Request versucht Struts zunächst, die vom Client bevorzugte Sprachdatei auszuwählen. Erst wenn eine bestimmte Sprache nicht unterstützt wird, fällt er auf die Standarddatei *application.properties* zurück. Das bevorzugte Sprachschema ermittelt das `ActionServlet` aus den Request-Headern (siehe Kapitel 2).

5.3.6 ActionForms

ActionForms sind JavaBeans, die mit einer oder mehreren Formularen verknüpft sind. Wird ein solches Formular an die dazugehörige Action abgeschickt (Submit), so überträgt Struts alle Daten des Formulars in die zugehörige ActionForm. Dazu müssen lediglich die Namen der JSP-Felder mit den Attributen (Properties) der Bean übereinstimmen.

Die Übertragung kann aber auch umgekehrt erfolgen: Sind in einem `ActionForm` bereits bestimmte Attribute mit einem Startwert initialisiert, so übernimmt das `<html:text>`-Tag diese ebenfalls in die Seite. Auf diese Weise ermöglichen es Ihnen `ActionForms`, Formulardaten auf einfache Art und Weise mit der Java-Logik zu synchronisieren.

Im Adressbuch dieses Beispiels besitzen Sie zunächst nur ein einziges Formular (Listing 5.6), für das Sie nun eine `ActionForm` erstellen können.

Listing 5.10
ActionForm zu unserem Formular

```
package de.akdabas.struts.forms;
import javax.servlet.http.*;

import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionMessage;

/**
 * Diese JavaBean speichert die Werte des Formulars
 * 'addressDetail.jsp'
 */
```

Listing 5.10 (Forts.) ActionForm zu unserem Formular

```
public class AddressForm extends ActionForm {

   /** Der ins Formular eingegebene Name */
   private String name;

   /** Der ins Formular eingegebene Vorname */
   private String firstName;

   /** Die ins Formular eingegebene E-Mail-Adresse */
   private String mail;

   /**
    * Diese Methode setzt die JavaBean zur Wiederverwendung zurück
    */
   public void reset(ActionMapping mapping,
                     HttpServletRequest request) {
      name = "";
      firstName = "";
      mail = "";
   }

   /**
    * Diese Methode wird zur Validierung der Parameter gerufen
    */
   public ActionErrors validate(ActionMapping mapping,
                                HttpServletRequest request) {
      ActionErrors errors = new ActionErrors();

      if ((name == null) || (name.equals(""))) {
         errors.add("name",
               new ActionMessage("detail.error.name.required"));
      }

      if (email.indexOf("@") == -1) {
         errors.add("email",
               new ActionMessage("detail.error.email.noat"));
      }
      return errors;
   }

   // Ab hier folgen die Getter und Setter für Felder des Formulars

   public void setName(String name) {
      this.name = name;
   }
   public String getName() {
      return name;
   }

   public void setFirstName(String firstName) {
      this.firstName = firstName;
   }
   public String getFirstName() {
      return firstName;
   }

   public void setMail(String mail) {
      this.mail = mail;
   }
   public String getMail() {
      return mail;
   }
}
```

Ihre FormBean (`ActionForm`) unterscheidet sich in drei Dingen vom Business-Objekt `Address` aus Listing 5.3:

- Die Methode `reset()`

  Durch diese Methode können Sie die FormBean initialisieren und Startwerte setzen, die die JSP etwa als Standardwerte ausgibt. Außerdem ermöglicht es Ihnen diese Methode, ein einmal erzeugtes Objekt wiederzuverwenden, um nicht jedes Mal eine neue Instanz erzeugen zu müssen.

- Die Methode `validate()`

  In dieser Methode können Sie Code unterbringen, der die Eingaben des Benutzers überprüft. Diese Methode wird ausgeführt, bevor das Action gerufen wird (Abbildung 5.2).

- Das FormBean ist von der Oberklasse `ActionForm` abgeleitet.

  Diese Oberklasse ist nötig, um die Methoden `reset()` und `validate()` auszuführen. Außerdem können Sie der Action-Klasse damit ein spezifisches Objekt übergeben.

Validierung

Nachdem Sie das Formular mit leeren Startwerten initialisiert haben, können Sie über die Methode `validate()` die Formulareingaben des Benutzers überprüfen. Eine korrekte Adresse soll in diesem Fall mindestens aus einem Namen und einer E-Mail-Adresse bestehen. Der Vorname eines Kontakts sei der Einfachheit halber in diesem Beispiel optional.

Info

Um Problemen mit Typkonvertierungen aus dem Weg zu gehen, implementiert die Basisklasse `ActionForm` gleich je zwei Methoden für `validate()` und `reset()`. Die eine erwartet dabei neben dem `ActionMapping` als Parameter stets einen einfachen `ServletRequest`, während die andere nur mit echten `HttpServletRequests` zusammenarbeitet. Sie müssen jedoch jeweils nur eine von beiden implementieren.

Dazu erzeugen Sie zunächst ein Wrapper-Objekt vom Typ `ActionErrors`, das die aufgetretenen Fehler aufnehmen soll. Dieses ist dabei vergleichbar mit einer speziellen `HashMap`, in der Fehler unter einem symbolischen Namen und mit einer Ressource abgelegt werden können.

```
ActionErrors errors = new ActionErrors ();
```

Anschließend überprüfen Sie das Vorhandensein des Namens und erzeugen gegebenenfalls ein entsprechendes Fehlerobjekt.

```
if ((name == null) || (name.equals(""))) {
 errors.add("name",new ActionMessage ("detail.error.name.required"));
}
```

Wie Sie sehen, verwenden Sie auch bei der Erzeugung eines Fehlers einen Schlüssel aus der Datei *application.properties*. Der Name, unter dem Sie den Fehler im `ActionErrors`-Objekt ablegen, kann dabei frei gewählt werden und ist zunächst nicht weiter von Belang.

Bei der E-Mail-Adresse können Sie die Validierung noch ein Stück weitertreiben und zumindest das Vorhandensein des @-Symbols überprüfen.

```
if ((mail == null) || (email.indexOf("@") == -1)) {
 errors.add ("email", new ActionMessage ("detail.error.email.noat"));
}
```

Sonst ist das Vorgehen identisch zum Namen. Wenn das zurückgegebene `ActionErrors`-Objekt am Ende der Methode nicht leer ist, wird das Formular nicht an die damit verknüpfte Action weitergeleitet. Stattdessen wird die JSP nochmals dargestellt, um dem Benutzer die Korrektur seiner Eingaben zu ermöglichen. Die aufgetretenen Fehler werden dabei über das Tag `<html:errors>` ausgegeben.

Abbildung 5.6
Bei Fehlern hat der Benutzer die Möglichkeit, seine Eingaben zu korrigieren

JSP-übergreifende Verwendung

ActionForms können auch mit mehreren, miteinander in Verbindung stehenden JSPs wie beispielsweise Assistenten – so genannten Wizards – verknüpft sein. Hierdurch können Felder einfach von einer JSP in eine andere verschoben werden, ohne dass die Applikation weiter angepasst werden müsste.

Info

Unter einem Wizard versteht man eine Folge von Menüs, durch die der Anwender nacheinander weitergeleitet wird und die in der Regel dazu dienen, eine komplexe Aufgabe oder Konfiguration in überschaubare Einzelschritte zu unterteilen, zwischen denen man in der Regel vor- und zurücknavigieren kann.

Dafür können ActionForms, analog zu den bisherigen Beans, mit einem Gültigkeitsbereich (Scope) ausgestattet werden und diesen legen Sie ebenfalls über die Konfigurationsdatei *struts-config.xml* fest.

FormBean kontra BusinessBean

Natürlich könnten Sie sich an dieser Stelle fragen, warum Sie die ganze Arbeit doppelt machen und sowohl Formbeans wie auch Business-Objekte erstellen sollen. Nun, ActionForms sind ein Bestandteil des Struts-Framework und eng mit dessen Ablauf verknüpft. Erweitern Sie ein Formular, so bekommt auch das zugehörige ActionForm ein neues Attribut und so fort.

Dabei können die Eingabefelder eines Formulars jedoch zu unterschiedlichen logischen Objekten des Applikationsmodells gehören. Wenn Sie beispielsweise per Online-Banking eine Überweisung von einem Konto auf ein anderes ausführen, agieren im Hintergrund möglicherweise zwei verschiedene Objekte miteinander. Der Betrag (möglicherweise ein weiteres Objekt) wird vom Belastungskonto abgebucht und dem anderen gutgeschrieben.

Natürlich könnten Sie versuchen, den Vorgang durch ein Überweisungsobjekt abzubilden, doch erinnern Sie sich: Gute Software-Entwicklung beginnt immer mit der Modellierung der Business-Logik und der Erschaffung kleinster logischer Einheiten, die dann möglichst unabhängig von der Darstellungsschicht wiederverwendet werden können.

5.3.7 Actions

Nun kommen wir endlich zu den Protagonisten des Stücks. Hinter jedem Request an die Struts-Webanwendung steht eine Action, die deren Bearbeitung übernimmt. Einige haben lediglich die Aufgabe, den Request an eine JSP weiterzuleiten (`ForwardAction`), andere enthalten Applikationslogik, indem sie die Werte aus einem Formular in Business-Objekte überführen oder in Datenbanken speichern. Kurz: Actions sind die Arbeitstiere der Applikation. Sie übernehmen die Aufgaben von Servlets und können direkt mit ihnen verglichen werden.

Die Action in diesem Beispiel soll die Werte des Formulars in ein `Address`-Businessobjekt überführen und dieses in einer Liste speichern. Die Liste von Adressobjekten wird dabei im Session-Scope des Benutzers abgelegt.

Listing 5.11
AddressAction

```
package de.akdabas.struts.actions;
import java.util.*;
import javax.servlet.http.*;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionForm;

import de.akdabas.struts.beans.Address;
import de.akdabas.struts.forms.AddressForm;
```

Listing 5.11 (Forts.) AddressAction

```
/** Diese Action speichert eine neu eingetragene Adresse */
public class AddressAction extends Action {

   /** Kapselt de nötige Geschäftslogik */
   public ActionForward execute(ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response)
         throws Exception {

      // Cast des allg. Form-Objekts zur AddressForm
      AddressForm addressForm = (AddressForm) form;

      // Erzeugen eines neuen Business-Objekts mit den bereits
      // validierten Daten der ActionForm
      Address newAddress = new Address();
      newAddress.setName(addressForm.getName());
      newAddress.setFirstName(addressForm.getFirstName());
      newAddress.setMail(addressForm.getMail());

      // Binden der HttpSession und Extrahieren der Adress-Liste
      HttpSession session = request.getSession();
      List<Address> addressList =
            (List<Address>) session.getAttribute("list");
      if (addressList == null) {
         addressList = new LinkedList<Address>();
      }

      // Hinzufügen der neu eingetragenen Addresse
      addressList.add(newAddress);

      // Ablegen der Liste in der Benutzer-Session
      session.setAttribute("list", addressList);

      // Nächste darzustellende JSP finden, welche ebenfalls in der
      // Konfigurations-Datei 'struts-config.xml' abgelegt wird.
      return mapping.findForward("continue");
   }
}
```

Eine Action unterscheidet sich in zwei Punkten von klassischen Servlets:

- Die Service-Methode `execute()`

  Diese Methode entspricht im Wesentlichen der `doPost()`- bzw. `doGet()`-Methode eines vergleichbaren Servlet. Sie enthält die Geschäftslogik dieser Operation und leitet den Request anschließend immer an ein anderes Objekt weiter (`ActionForward`), das dann die Darstellung der nächsten Seite übernimmt.

- Die Oberklasse `Action` (Typ `org.apache.struts.action.Action`)

  Ein Servlet ist in der Lage, alle Arten eines Request (nicht nur HTTP) zu behandeln und eine Antwort zu erzeugen. Außerdem muss ein Servlet in der Lage sein, mit dem umgebenden Servlet-Container zu kommunizieren. Eine Action hingegen ist darauf optimiert, Anwendungslogik auszuführen und den Request anschließend weiterzuleiten. Die Kommunikation mit dem Servlet-Container übernimmt das im Framework enthaltene ActionServlet für alle Actions gemeinsam.

Execute vs. doGet und doPost

Die Methode `execute()` hat gegenüber den klassischen Servlet-Methoden einige Vorteile: Zum einen haben Sie in ihr nicht nur Zugriff auf die Variablen `request` und `response` und zum anderen wird jede Art Ausnahme (`java.lang.Exception`) unterstützt. Im Gegensatz zu `doGet()` und `doPost()`, die lediglich `Servlet-` und `IOExceptions` zulassen.

Hinter der übergebenen `ActionForm` verbirgt sich nichts anderes als die zu diesem Formular gehörende `AddressForm`, die Sie allerdings zunächst zurückcasten müssen, um Zugriff auf die speziellen Attribute dieser Bean zu erhalten.

```
AddressForm addressForm = (AddressForm) form;
```

Als Nächstes erzeugen Sie eine neue *Business-Bean* und übertragen die bereits validierten Werte aus dem Formular in diese. Anschließend versuchen Sie, die `Address`-Liste aus der Session des Benutzers zu extrahieren, und erzeugen gegebenenfalls eine neue und legen die neue Adresse in dieser ab. Anschließend speichern Sie die Liste wieder in der Session.

An dieser Stelle sehen Sie eine weitere Besonderheit der Struts-Servicemethode: Statt die Ausgabe selbst zu erzeugen und anschließend einfach die Bearbeitung per `return` zu beenden, geben Sie am Ende immer ein `ActionForward`-Objekt zurück, welches den Request zur Darstellung an eine JSP oder auch zur Weiterverarbeitung an eine andere Action weiterleitet.

Ein `ActionForward` ist dabei wieder nichts anderes als die Java-Repräsentation eines symbolischen Struts-Links, dessen Ziel Sie in der Konfigurationsdatei (*struts-config.xml*) festlegen können.

Vordefinierte Actions

Natürlich müssen in einer großen Applikation auch viele Standardaufgaben wie das einfache Weiterleiten auf eine andere JSP oder das Einfügen einer solchen realisiert werden. Keine Angst, auch dafür hält das Struts-Framework praktische Helfer parat, die Sie vor der Aufgabe bewahren, für jeden symbolischen Link eine eigene Action schreiben zu müssen. So existieren eine ganze Reihe von Standard-Actions, etwa für die Weiterleitung (`ForwardAction`) und das Einbinden anderer Seiten (`IncludeAction`) oder als ganz besonderes Schmankerl: Actions, deren Verhalten vom Wert eines Parameters abhängt (`DispatchAction`).

5.3.8 Konfiguration der Anwendung

Nun haben Sie alle Stützen (*Struts*) der Beispielanwendung entwickelt und müssen sie nur noch zu einem sinnvollen Gebilde zusammenfügen. Dies realisieren Sie über die bereits oft zitierte Datei *struts-config.xml*, mit der Sie den Controller des Framework – das `ActionServlet` – konfigurieren.

Die Ressourcen

Beginnen wir mit den Ressourcen der Anwendung, denn schließlich müssen Sie dem Framework mitteilen, in welcher Datei es nach den Texten der Applikation suchen soll. Da Sie die Datei *application.properties* in diesem Beispiel im Verzeichnis *WEB-INF/classes/ressources* ablegen wollen, fügen Sie das folgende Element in die Konfigurationsdatei ein.

Listing 5.12 Konfiguration der MessageRessources

```
...
   <!-- Die Ressourcen; relativer Pfad zu 'WEB-INF/classes' -->
   <message-resources parameter="resources.application"/>
...
```

Die Dateiendung `properties` brauchen Sie in diesem Fall nicht mit anzugeben.

Die FormBeans

Um die `ActionForm` einzubinden, erzeugen Sie nun ein `<form-bean>`-Element, durch das Sie die Form unter einem symbolischen Namen (`addressForm`) verfügbar machen.

Listing 5.13 Konfiguration der ActionForm

```
...
   <form-beans>
      <!-- Einbinden der ActionForm mit dem symbolischen Namen
           'addressForm' -->
      <form-bean name="addressForm"
                 type="de.akdabas.struts.forms.AddressForm"/>
   </form-beans>
...
```

Die ActionForwards

Jetzt kommen die symbolischen Links (*ActionForwards*) der Anwendung an die Reihe. Dabei verknüpfen Sie jeden Link zunächst mit einer Action (`*.do`), die über das Struts-interne `ActionServlet` aufgerufen wird.

Listing 5.14 Konfiguration der ActionForwards

```
...
   <!-- Definition der symbolischen Links der Anwendung -->
   <global-forwards>
      <forward name="initList" path="/List.do"/>
      <forward name="addressList" path="/List.do"/>
      <forward name="addressDetail" path="/Detail.do" />
      <forward name="saveAddress" path="/SaveAddress.do"/>
   </global-forwards>
...
```

Durch dieses »doppelte Mapping« haben Sie auch später noch die Möglichkeit, einem symbolischen Link eine andere Action zuzuweisen bzw. eine Action unter verschiedenen Links erreichbar zu machen.

Die Actions

Schließlich fügen Sie alles zu einem gemeinsamen Bild zusammen, indem Sie die ActionForwards mit den JSPs und den zugehörigen Actions verbinden. Sowohl in der *AddressList.jsp* als auch in der *AddressDetail.jsp* verwenden Sie symbolische Links, um von einer Seite auf die jeweils andere zu verweisen.

Da Sie dabei keinen eigenen Code unterbringen, sondern lediglich von einer Action auf die nächste weiterleiten wollen, verwenden Sie in diesem Fall die allgemeine `ForwardAction`, die bereits im Framework enthalten ist.

Listing 5.15
Konfiguration der ForwardActions

```
...
   <action-mappings>
      <!-- Definition einfacher Forwards auf andere JSPs -->
      <action path="/List"
              type="org.apache.struts.actions.ForwardAction"
              parameter="/jsp/addressList.jsp"/>
      <action path="/Detail"
              type="org.apache.struts.actions.ForwardAction"
              parameter="/jsp/addressDetail.jsp"/>
   ...
   </action-mappings>
...
```

Nun geht es schließlich an die `AddressAction`! Die Attribute in Listing 5.16 haben dabei folgende Funktion:

- `path`
  Pfad eines symbolischen Links, wobei die Endung `do` in diesem Fall weggelassen werden kann
- `type`
  Voll qualifizierender Klassenname der Action
- `name`
  Symbolischer Name der zugehörigen ActionForm (Listing 5.13)
- `input`
  JSP, deren Formularfelder als Eingabe verwendet werden sollen
- `scope`
  Gültigkeitsbereich der ActionForm

Listing 5.16
Konfiguration der AddressAction

```
...
   <!-- Konfiguration der AddressAction -->
   <action path="/SaveAddress"
           type="de.akdabas.struts.actions.AddressAction"
           name="addressForm"
           input="/jsp/addressDetail.jsp"
           scope="request"
           validate="true">

      <!-- Definition eines lokalen Forwards (Listing 5.11) -->
      <forward name="continue" path="/List.do"/>
   </action>
...
```

Und es gibt noch ein weiteres Augenmerk bei dieser Konfiguration: den lokalen ActionForward! Wie Ihnen vielleicht schon vorher aufgefallen ist, haben Sie den symbolischen Link `continue` der `AddressAction` nicht zusammen mit den anderen ActionForwards definiert. Denn zusätzlich zu den globalen ActionForwards kann jede Action beliebig viele eigene besitzen. Dies ist besonders bei der Entwicklung von Assistenten oder der Implementierung eines bestimmten Workflow nützlich, da so auf einfache Art und Weise weitere Schritte in den Ablauf integriert werden können.

Das Resultat

Das war es! Die Beispielanwendung ist nun vollständig, durchkonfiguriert und einsatzbereit. Sie müssen sie nur noch übersetzen, sicherstellen, dass die Ressourcedatei sich im Ordner *WEB-INF/classes/resources* befindet und den Webserver starten. Listing 5.17 zeigt Ihnen noch einmal die gesamte Konfigurationsdatei *struts-config.xml* im Überblick.

Listing 5.17
Vollständige Konfiguration der Beispielanwendung

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC
   "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
   "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">

<struts-config>
   <data-sources />

   <form-beans>
      <!-- Einbinden der ActionForm mit dem symbolischen Namen
           'addressForm' -->
      <form-bean name="addressForm"
                 type="de.akdabas.struts.forms.AddressForm"/>
   </form-beans>

   <!-- Definition der symbolischen Links der Anwendung -->
   <global-forwards>
      <forward name="initList" path="/List.do"/>
      <forward name="addressList" path="/List.do"/>
      <forward name="addressDetail" path="/Detail.do" />
      <forward name="saveAddress" path="/SaveAddress.do"/>
   </global-forwards>

   <action-mappings>
      <!-- Definition einfacher Forwards auf andere JSPs -->
      <action path="/List"
              type="org.apache.struts.actions.ForwardAction"
              parameter="/jsp/addressList.jsp"/>
      <action path="/Detail"
              type="org.apache.struts.actions.ForwardAction"
              parameter="/jsp/addressDetail.jsp"/>

      <!-- Konfiguration der AddressAction -->
      <action path="/SaveAddress"
              type="de.akdabas.struts.actions.AddressAction"
              name="addressForm"
              input="/jsp/addressDetail.jsp"
              scope="request"
              validate="true">

         <!-- Definition eines lokalen Forwards (Listing 5.11) -->
         <forward name="continue" path="/List.do"/>
      </action>
   </action-mappings>

   <!-- Die Ressourcen; relativer Pfad zu 'WEB-INF/classes' -->
   <message-resources parameter="resources.application"/>

</struts-config>
```

Und so sieht die Anwendung aus, nachdem Sie Ihre erste Adresse eingefügt haben.

Abbildung 5.7
addressList.jsp nach erfolgreicher Eingabe

Wenn Sie beim Start der Applikation eine *ServletException* mit dem Fehlertext »Missing message« erhalten, überprüfen Sie den Standort und die Konfiguration der Property-Datei. Schon so manche Programmierstunde ging bei der Suche nach einem vermeintlichen Schreibfehler drauf, nur weil ich vergessen habe, die Datei (mittels Ant-Script) an die richtige Stelle zu kopieren.

5.3.9 Fazit

Wie Sie gesehen haben, erfordert es einigen Programmier- und Konfigurationsaufwand, die Grundstruktur der Anwendung gemäß dem Ansatz des Struts-Framework zu entwickeln. Für einfache, kleinere Demonstrationen einer Technologie lohnt sich der Aufwand also nur, wenn Sie mit dem Framework bereits so weit vertraut sind, dass Ihre Denkstruktur über den Aufbau einer entsprechenden Lösung schon automatisch dem logischen Konzept des Framework folgt. Anderenfalls können kleinere Lösungen auch mit den Basistechnologien allein, entweder nur mit *JSPs* oder in Kombination mit *Servlets*, realisiert werden.

Für größere Projekte, die in der Regel auch eine längere Planungsphase haben, lohnt sich der Aufwand, der durch ein Framework entsteht, allemal. Denn ist die Grundstruktur erst einmal implementiert, können spätere Erweiterungen leicht und ohne große Seiteneffekte in die bereits bestehende Lösung integriert werden.

Tipp

Auch das beste Framework kann Sie natürlich nicht vor Seiteneffekten, wie sie durch die intensive Benutzung des Session-Speichers gern auftreten, bewahren. Wenn Sie sich also z.B. nach dem Hinzufügen einer Suchoperation darüber wundern, warum bei unterschiedlicher Auswahl stets der zuerst gewählte Datensatz erscheint, überprüfen Sie doch einmal, ob diese *Hyperpersistenz* durch die Speicherung der Auswahl in der Session hervorgerufen wird. In diese gehören nur Daten, die wirklich für die gesamte Dauer einer Session gültig sind. Daten mit einem kleineren Gültigkeitsbereich, wie die aktuelle Auswahl, sollten entsprechend auch nur mit den dafür vorgesehen Scopes (Request und Page) versehen werden.

5.4 Erweiterungen

Unsere Lösung erlaubt es uns nun also, Adressen in der Session eines Benutzers zu speichern und weitere hinzuzufügen. Doch was nutzt das, wenn die registrierten Kontakte beim Schließen des Browser-Fensters im Nirwana verschwinden und stets aufs Neue eingegeben werden müssen? Um die Erweiterungsfähigkeit einer Framework-basierten Lösung zu demonstrieren, wollen wir in diesem Abschnitt zeigen, wie einfach es ist, eine bestehende Struts-Lösung um weitere Komponenten oder Funktionalitäten zu erweitern.

Dieser Absatz greift dabei einigen Technologien vor, die Sie in Zusammenhang mit Namensdiensten in Kapitel 7 nochmals genauer kennen lernen werden. Aus diesem Grund wird die Erklärung, was sich hinter einer Datenquelle verbirgt, in diesem Kapitel bewusst kurz gehalten.

5.4.1 Die Datenbank-Action

Erinnern Sie sich, dass Sie dem Anwender durch unsere *index.jsp* einen definierten Startpunkt in die Anwendung boten? Der Aufwand, an dieser Stelle eine *ForwardAction* zu konfigurieren und nicht etwa mit statischen Links zu arbeiten, hat sich gelohnt, da Sie diese nun einfach gegen eine eigene *Action* austauschen können. Doch zunächst einmal müssen Sie die Datenbankverbindung für die Anwendung definieren und in das Framework einbinden und wo anders sollte das geschehen als in der *struts-config.xml*?

Konfiguration der Datenquelle

Wir gehen davon aus, dass in unserer Datenbank bereits eine Tabelle mit den gespeicherten Adressen existiert und diese über eine spezifische URL erreichbar ist. Nun müssen wir in unserer Konfigurationsdatei lediglich den Platzhalter für Datenquellen

```
...
   <data-sources />
...
```

durch folgende Tags ersetzen

Listing 5.18 Definition einer Datenquelle

```
...
   <data-sources>
      <!-- Konfiguration einer Datenbank-Verbindung -->
      <data-source key="mysqlDataSource">
         <set-property property="driverClass"
                       value="com.mysql.jdbc.Driver"/>
         <set-property property="url"
                       value="jdbc:mysql://localhost:3306/mySQL"/>
         <set-property property="user"
                       value="Benutzername der Datenbank"/>
         <set-property property="password"
                       value="Password der Datenbank"/>
      </data-source>
   </data-sources>
...
```

InitAction

Nun erzeugen Sie die *InitAction*, die das Auslesen der Liste aus der Datenbank zu Beginn einer Session übernehmen wird.

Listing 5.19
InitAction zum Auslesen der Datenbank

```
package de.akdabas.struts.actions;
import java.sql.*;
import javax.servlet.http.*;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionForm;

import de.akdabas.struts.beans.Address;
import de.akdabas.struts.forms.AddressForm;

/** Initialisiert die Liste der Kontakte in der Benutzer-Session */
public class InitAction extends Action {

   /** Service-Methode der InitAction **/
   public ActionForward execute(ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response)
         throws Exception {

      javax.sql.DataSource dataSource;
      java.sql.Connection myConnection = null;
      java.util.List addressList = null;

      try {
         // Diese Methode gibt Ihnen die in der 'struts-config.xml'
         // konfigurierte Datenquelle zurück
         dataSource = getDataSource(request);
         myConnection = dataSource.getConnection();

         // An dieser Stelle steht dann zum Beispiel Ihr
         // SQL-Statement um die Liste auszulesen

         // Ablegen der Liste in der Benutzer-Session
         HttpSession session = request.getSession();
         session.setAttribute("list", addressList);

      } catch (SQLException sqle) {
         getServlet().log("Connection.process.", sqle);
      } finally {
         try { myConnection.close();
         } catch (SQLException exc) { exc.printStackTrace(); }
      }

      // Nächste darzustellende JSP finden, welche ebenfalls in der
      // Konfigurations-Datei 'struts-config.xml' abgelegt wird.
      return mapping.findForward("continue");
   }
}
```

Die entscheidenden vier Zeilen Code werden zwar durch die bei Datenbankoperationen üblichen `try-catch-finally`-Blöcke etwas aufgebläht, das einfache Prinzip wird dennoch deutlich. Nun müssen Sie die `InitAction` lediglich noch in die Anwendung einbinden, wozu Sie ein letztes Mal die Datei *struts-config.xml* bemühen.

Konfiguration der neuen Action

Listing 5.20 zeigt, wie Sie die *InitAction* einbinden können.

Listing 5.20
Einbinden der InitAction

```
...
   <global-forwards>
      <!-- Der in der index.jsp gerufene Forward lautet initList;
           Siehe auch Listing 5.4 -->
      <forward name="initList" path="/InitList.do"/>
...
   </global-forwards>

   <action-mappings>
...
      <!-- Die ForwardAction wird nun einfach gegen Ihre eigene
           ausgetauscht; Siehe auch Listing 5.17 -->
      <action path="/InitAddress"
              type="de.akdabas.struts.actions.InitAction">
         <forward name="continue" path="/List.do"/>
      </action>
   </action-mappings>
...
```

Fertig! Ein Neustart und schon werden die Daten zu Beginn aus der Datenbank ausgelesen. Analog können Sie die Daten natürlich beim Abmelden über eine *LogoutAction* auch wieder in die Datenbank zurückschreiben. Dazu müssen Sie lediglich an geeigneter Stelle einen entsprechenden Link setzen, eine Action schreiben und konfigurieren ...

5.5 Zusammenfassung

Dieses Kapitel sollte Ihnen eine Vorstellung davon geben, welche Möglichkeiten Sie durch den Einsatz von Frameworks erhalten und welche Konsequenzen die Entscheidung für ein bestimmtes Framework hat.

Für kleinere Applikationen sind der Einsatz eines Framework wie Struts und der damit verbundene Mehraufwand für Installation, Einarbeitung und Konfiguration sicher nicht gerechtfertigt. Doch schon bei mittleren und erst recht bei Applikationen, deren Weiterentwicklung noch nicht vollständig abzusehen ist, sollte über den Einsatz eines Framework nachgedacht werden.

Nun ist *Struts* sicherlich auch nicht jedermanns Sache und bei weitem komplexer als einfachere Frameworks wie beispielsweise Jakarta Turbine (*http://jakarta.apache.org/turbine/*). Aber Struts ist eben auch eines der leistungsfähigsten Frameworks, in dem viele Standardprozesse schon vorgebildet sind, und für die meisten Entwicklungsumgebungen (IDEs) existieren zahlreiche Plug-ins, die Sie schon bei der Entwicklung der Anwendung unterstützen.

Doch zwei Dinge sollten Sie über den Einsatz von Frameworks gelernt haben:

- Anpassung und Weiterentwicklung der Technologien und Standards

  Viele Frameworks nehmen leichte Anpassungen an den unterstützten *Java-EE-Technologien* vor und entwickeln diese weiter. So verwenden Sie in *Struts* beispielsweise *Actions* und *ActionForwards*, wo Sie sonst *Servlets* und *RequestDispatcher* eingesetzt hätten.

  Die Technik hinter diesen Standards bleibt stets die gleiche, nur steckt sie eben manchmal im anderen Gewand.

- Konfigurationsaufwand

  Viele Frameworks nehmen Standardaufgaben und bewahren Sie vor den berühmten *Copy-und-Paste*-Fehlern. Doch damit sie dies tun können, müssen sie zuerst verstehen, was wir von ihnen wollen. Deshalb nehmen Sie sich gerade vor größeren Projekten die Zeit, sich in das favorisierte Framework einzuarbeiten. Vielleicht indem Sie erst einen kleinen Teil umsetzen und anschließend nach Verbesserungen im *Workflow* suchen, bevor Sie sich an das Meisterstück machen.

Mit diesem Kapitel wollte ich Ihnen Mut machen, bei größeren Projekten ernsthaft über den Einsatz eines Framework nachzudenken. Wenn Ihnen Struts nicht zusagt: Probieren Sie andere Frameworks, die Ihren Software-Entwicklungsprozess besser unterstützen. Wie gesagt, *Struts* ist die Diva webbasierter Frameworks und Diven sind eben manchmal etwas eigenwillig.

6

JavaServer Faces (JSF)

Nachdem Sie nun sowohl JSPs als auch Servlets kennen gelernt haben und auch mit dem freien Web-Framework Struts vertraut sind, möchte Ihnen dieses Kapitel Suns Alternative in Form der JavaServer Faces (JSF) näher bringen. Warum ein weiteres Framework? Nun, Struts ist der De-facto-Standard, wenn es um große webbasierte Anwendungen geht und da Sie als zukünftige Java-EE-Entwickler mit großer Wahrscheinlichkeit auch mit diesem Framework in Berührung kommen werden, wollten wir Ihnen mit Kapitel 5 die Grundlagen der Arbeit mit Frameworks näher bringen.

Info

JSF können auch als Suns Antwort auf den Siegeszug des OpenSource-Framework Struts verstanden werden.

Nachdem Sun die Spezifikation für JSPs und Servlets herausgebracht hatte, begann der Siegeszug java-basierter Webapplikationen und nach dem großen Erfolg von Struts begannen die Sun-Entwickler, ein ähnliches Framework für die Java EE-Spezifikation zu entwickeln. Dabei haben Sie einen großen Vorteil: Sie müssen sich heute nicht mehr mit den Altlasten herumschlagen, mit denen die Struts-Entwickler zu kämpfen haben, sondern können aus bisher gesammelten Erfahrungen schöpfen. JavaServer Faces sind zwar zum Zeitpunkt des Schreibens dieser Zeilen noch nicht weit verbreitet, haben aber auch das Potenzial, Struts irgendwann vom Thron zu stürzen. Was die Zukunft bringen wird, wissen wir nicht. Es kann jedoch nur von Vorteil sein, mit beiden Technologien vertraut zu sein.

Um die Arbeit mit JSF zu verdeutlichen und auch einen Vergleich zu ermöglichen, werden Sie in diesem Kapitel eine ähnliche Beispielanwendung wie in Kapitel 5 realisieren. Da Sie nun allerdings mit gewissen Grundkonzepten bereits vertraut sein sollten, können und werden wir in diesem Kapitel auch etwas tiefer in die Technologie einsteigen. Wenn Sie

sich also während des Kapitels fragen, wieso Struts das nicht konnte, sollten Sie sich immer bewusst machen, dass beide Frameworks gleichermaßen leistungsfähig sind.

Wie die vorangegangenen Kapitel ist auch dieses nicht als vollständige Referenz für die JSF-Spezifikation zu verstehen, sondern soll vor allem die Arbeitsweise und Ideen des zugrunde liegenden Framework verdeutlichen.

6.1 Gemeinsamkeiten mit Struts

Um den Einstieg in das Framework möglichst einfach zu gestalten, beginnen wir damit, Gemeinsamkeiten beider Frameworks herauszuarbeiten. Dies ist vielleicht auch dann interessant für Sie, wenn Sie eine bereits bestehende Struts-Anwendung in eine JSF-Applikation überführen oder (und auch das funktioniert) beide Frameworks gemeinsam einsetzen möchten.

Tipp

Unter *http://struts.apache.org/struts-action/struts-faces/* finden Sie eine umfangreiche Bibliothek, die Ihnen die (stückweise) saubere Migration einer bestehenden Struts-Anwendung zu JSF erleichtert.

6.1.1 Ein Servlet als Front-Controller

Um das Model-View-Controller Pattern (MVC) gut implementieren zu können, arbeiten auch JavaServer Faces mit einem bereits im Framework enthaltenen Servlet als Frontcontroller. Dieser nimmt analog alle Requests auf der Endung `*.faces` entgegen und leitet sie dann an die entsprechenden Komponenten weiter. Listing 6.1 zeigt die hierfür erforderlichen Einträge im Web Deployment Descriptor (*web.xml*).

Tipp

Einen guten Einstieg in das Model-View-Controller Pattern (MVC) und die Arbeitsweise eines Front-Controller finden Sie in Kapitel 5.2.

Listing 6.1
Eintrag im Web Deployment Descriptor (web.xml)

```
...
   <!-- Bindet das Controller-Servlet des Frameworks ein -->
   <servlet>
     <servlet-name>FacesServlet</servlet-name>
     <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
     <load-on-startup>1</load-on-startup>
   </servlet>

   <!-- Leitet alle URLs auf '.faces' an das JSF Servlet -->
   <servlet-mapping>
      <servlet-name>FacesServlet</servlet-name>
      <url-pattern>*.faces</url-pattern>
   </servlet-mapping>
...
```

6.1.2 Konfiguration über eine zentrale XML-Datei

Ähnlich wie Struts, wird auch das Verhalten des JSF Framework über eine zentrale Konfigurationsdatei gesteuert, in der die einzelnen Komponenten über symbolische Namen zusammengebracht werden.

Das Herz einer JSF-Applikation hört auf den Namen *faces-config.xml* und sollte sich direkt unterhalb des Verzeichnisses *WEB-INF* befinden. Abschnitt 6.2.4 beschäftigt sich ausführlich mit dieser Datei.

6.1.3 Die Business-Komponente AddressBean

Erinnern Sie sich? Die Geschäftslogik und alle damit verbundenen Komponenten sollten stets unabhängig von der verwendeten Darstellungsschicht und damit wiederverwendbar implementiert werden. So haben Sie im vorangegangenen Kapitel schließlich auch zwischen der `AddressBean` und der `AddressForm` unterschieden, weil Letztere zwingend auf Struts-Komponenten aufbauen musste.

Um die Beispiele sauber voneinander zu trennen, haben wir sie in diesem Kapitel jedoch in das Package `de.akdabas.javaee.beans` verschoben.

Listing 6.2
Die AddressBean des Geschäftsmodells

```
package de.akdabas.javee.beans;

/**
 * Diese Bean stellt ein Value-Objekt dar und repräsentiert einen
 * Adress-Datensatz der Beispielanwendung.
 *
public class Address {

   private String name;
   public void setName(String aLastName) {
      name = aLastName;
   }
   public String getName() {
      return name;
   }

   private String firstName;
   public void setFirstName(String aFirstName) {
      firstName = aFirstName;
   }
   public String getFirstName() {
      return firstName;
   }

   private String mail;
   public void setMail(String aMailAddress) {
      mail = aMailAddress;
   }
   public String getMail() {
      return mail;
   }
}
```

Info

Der Begriff Plain Old Java Object (POJO) bedeutet im Deutschen so viel wie »ganz gewöhnliches Java-Objekt« und wird für JavaBeans verwendet, die keiner bestimmten Schnittstelle genügen müssen und keinen fest vorgeschriebenen Lebenszyklus durchlaufen.

Direktes Arbeiten auf den Business-Objekten

Da JSF mit so genannten Plain Old Java Objects (POJOs) arbeitet, können Sie in diesem Kapitel auch direkt auf den Business-Objekten arbeiten. Da diese keine direkte Abhängigkeit zum Framework besitzen, durchbrechen Sie damit zwar nicht das MVC Pattern, aber es gibt auch Entwicklermentalitäten, die dennoch eine strikte Trennung zwischen Logik und Darstellung auch auf Ebene der JavaBeans fordern.

Info

POJOs sind also einfache Java-Klassen.

Der Vorteil einer strikten Trennung der MVC-Komponenten ist, in großen Systemen – in denen Ihnen die Business-Objekte beispielsweise von Enterprise JavaBeans bereitgestellt werden – die interne Schnittstelle flexibel anpassen zu können, was Darstellung und Datenhaltung sauber trennt. In der Praxis ist dies jedoch nicht überall erforderlich.

6.1.4 Eine Welcome-Seite als Einstiegspunkt in die Applikation

Auch bei diesem Framework ist es ratsam, den Benutzer bei einem einfachen Request ohne Angabe einer speziellen Ressource an einen definierten Startpunkt der Anwendung weiterzuleiten und so definieren Sie im Web Deployment Descriptor auch hier wieder eine Begrüßungsseite:

Listing 6.3 Definition eines Welcome-File

```
...
   <welcome-file-list>
      <welcome-file>index.jsp</welcome-file>
   </welcome-file-list>
...
```

Diese hat keine andere Aufgabe, als den ersten `faces`-Request an das Controller-Servlet zu senden.

Listing 6.4 Ein Welcome-File (index.jsp)

```
<html>
   <body>
      <jsp:forward page="jsp/addressList.faces" />
   </body>
</html>
```

6.1.5 Internationalisierung

Und auch bei der Unterstützung von Ressource-Dateien, in denen die festen Texte der gesamten Anwendung hinterlegt werden, unterscheiden sich Struts und JSF nicht. So weicht die Datei *resources/application.properties* kaum von ihrem Struts-Pendant ab.

Listing 6.5
Application Properties statt fest einkodierter Zeichenketten

```
#Fehlermeldungen
errors_email="Bitte geben Sie eine gültige E-Mail-Adresse ein."

# Übergreifende Bezeichnungen verschiedener JSPs
address_name=Name
address_firstName=Vorname
address_eMail=eMail-Adresse

# Texte und Bezeichnungen des Adress-Formulars (addressDetail.jsp)
detail_error_name_required=Bitte geben Sie einen Namen ein !
detail_error_email_noat=eMail - Addresse enthält kein @ !
detail_heading=Bitte geben Sie die Benutzerdaten ein
detail_cancel=Abbrechen
detail_save=Speichern
```

6.2 Eine einfache JSF-Anwendung

Nachdem die Gemeinsamkeiten geklärt sind, können Sie sich nun in die JSP-spezifische Entwicklung eines Adressbuchs stürzen. Zunächst müssen Sie jedoch die spezifischen Klassen des Framework herunterladen und obwohl die JSF-Spezifikation noch sehr jung ist, existieren bereits zwei unabhängige Implementierungen:

- Referenzimplementierung von Sun (*http://java.sun.com/javaee/javaserverfaces/*)
- Die unabhängige MyFaces-Implementierung der Apache Group (*http://myfaces.apache.org/*)

Beide Implementierungen genügen dabei der Spezifikation und es ist prinzipiell egal, für welche Sie sich entscheiden. Wichtig ist nur, dass Sie die jeweils enthaltenen Java-Archive (JAR) in den Ordner *WEB-INF/lib* des Projektverzeichnisses kopieren.

Da die Apache-Software-Lizenz für die Verbreitung von Software in Form einer Buch-CD günstiger ist, werden wir in diesem Kapitel die MyFaces-Implementierung verwenden.

6.2.1 Darstellung: JavaServer Pages

Nachdem die Form der Business-Komponenten geklärt war, haben Sie im letzten Kapitel begonnen, die JSPs zu erstellen. Auch die in diesem Kapitel entwickelte Beispielanwendung besteht dabei aus einer Liste aller bereits existierenden Einträge und einem Formular, um neue Einträge hinzuzufügen.

addressList.jsp

Listing 6.6
Eine JSP zur Ausgabe der bereits erfassten Kontakte

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<html>
<body>

<%-- Diese Tag bildet den Rahmen für weitere JSF Tags --%>
<f:view>

   <%-- Pendant zum HTML-Tag <form>, welches mit einer JavaBean
        verknüpft ist --%>
   <h:form id="addressForm">

      <%-- Erzeugt eine Tabelle, deren Spalten aus den Elementen
           einer Collection erzeugt werden --%>
      <h:dataTable value="#{addressController.addressList}"
                   var="address">

         <%-- Definition einer Spalte mit Kopfzeile --%>
         <h:column>
            <f:facet name="header">
               <h:outputText value="Vorname"/>
            </f:facet>
            <h:outputText value="#{address.firstName}"/>
         </h:column>

         <h:column>
            <f:facet name="header">
               <h:outputText value="Nachname"/>
            </f:facet>
            <h:outputText value="#{address.name}"/>
         </h:column>

         <h:column>
            <f:facet name="header">
               <h:outputText value="eMail Adresse"/>
            </f:facet>
            <h:outputText value="#{address.mail}"/>
         </h:column>
      </h:dataTable>

      <hr/>

      <%-- Dieses Tag erzeugt eine Schaltfläche, deren Betätigung
           die Methode 'createAddress' der Action mit dem
        symbolischen Namen 'addressController' aufruft. --%>
      <h:commandButton action="#{addressController.createAddress}"
                       value="Addresse hinzufügen"/>
   </h:form>
</f:view>
</body>
</html>
```

Wie Sie sehen, verwendet diese JSP gleich zwei Tag-Bibliotheken: `f` wie Faces enthält dabei die Core-Tags des Framework, während `h` wie HTML Pendants der HTML Markup Language enthält.

- `<f:view>`: Dieses Tag bildet den Rahmen für alle weiteren JSF-Komponenten. Es ist damit mit dem `<html>`-Tag in JSPs vergleichbar.
- `<h:form>`: Dieses Tag bildet das Pendant zu einem `<form>`-Tag in HTML und gestattet es, Eingaben vorzunehmen.

- `<h:commandButton>` Mit diesem Tag erzeugen Sie eine Schaltfläche, die einen Request absendet und bei der das Framework die angegebene Methode (`action`) ruft.
- `<h:dataTable>` und `<h:column>` Rendern später eine Tabelle, deren Zeilen aus den Elementen einer Liste befüllt werden.
- `<h:outputText>` Dient dazu, Texte in die resultierende Seite auszugeben. Im Gegensatz zu Struts ist es hier auch möglich, zunächst feste Zeichenketten wie beispielsweise `"Nachname"` anzugeben und diese erst später durch Referenzen auf eine Properties-Datei zu ersetzen.

Abbildung 6.1 zeigt die spätere Form der JSP schon einmal.

Abbildung 6.1
Liste aller bereits gespeicherten Adressen

addressDetail.jsp

Bevor die Kontakte dargestellt werden können, müssen diese natürlich zunächst einmal eingegeben werden. Hierfür dient das folgende Formular:

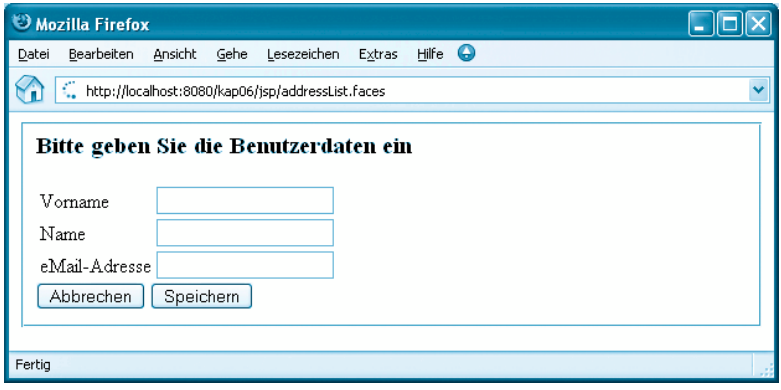
Listing 6.7
Formular zum Erfassen von Kontakten

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<html>
<body>
<f:view>
   <h:form>

      <%-- Einlesen des Resource-Bundles und binden an die lokale
           Variable msg --%>
      <f:loadBundle basename="resources.application" var="msg"/>

      <fieldset>
         <%-- Ausgabe eines Labels aus dem Ressource-Bundle --%>
         <h3><h:outputText value="#{msg.detail_heading}"/></h3>

         <%-- Deklaration eines Grids, welches nach jeweils 3
              Elementen eine neue Zeile beginnt.--%>
         <h:panelGrid columns="2">

            <%-- Ausgabe eines Labels aus den Ressourcen --%>
            <h:outputLabel for="firstName"
                           value="#{msg.address_firstName}"/>
            <h:inputText id="firstName"
```

Listing 6.7 (Forts.)
Formular zum Erfassen von Kontakten

```
                        value="#{addressController
                                .currentAddress.firstName}"/>

            <h:outputLabel for="name" value="#{msg.address_name}"/>
            <h:inputText id="name"
                        value="#{addressController
                                .currentAddress.name}"/>

            <h:outputLabel for="mail" value="#{msg.address_eMail}"/>
            <h:inputText id="mail"
                        value="#{addressController
                                .currentAddress.mail}">
            </h:inputText>

        </h:panelGrid>

        <h:commandButton action="#{addressController.cancel}"
                         value="#{msg.detail_cancel}"
                         immediate="true"/>
        <h:commandButton action="#{addressController.save}"
                         value="#{msg.detail_save}"/>
    </fieldset>
  </h:form>
</f:view>
</body>
</html>
```

Abbildung 6.2
Resultat des Listing 6.7



Die neuen Elemente des JSF Framework in Listing 6.7 sind:

- `<h:inputText>`: Dieses Tag rendert ein einfaches Eingabefeld (`<input>`). Die Werte werden bei einem Request an die Properties der angegebenen JavaBean übertragen.
- `<h:panelGrid>` Die Funktionsweise dieses Tag ist angelehnt an das `GridLayout` von Java-Swing. In diesem Beispiel wird dabei nach jeweils zwei eingefügten Elementen eine neue Zeile begonnen.

6.2.2 Datenbindungssyntax

Wie Sie in Listing 6.7 sehen, definiert das JSF-Framework eine eigene Syntax, um auf dynamische Daten zugreifen zu können. Die in der Literatur auch als Datenbindungssyntax bezeichnete Schreibweise besteht aus

einer Raute und zwei geschweiften Klammern, die den Pfad der einzubindenden bzw. aufzurufenden Komponente umschließen:

```
// Datenbindungssyntax in allgemeiner Form
#{ PfadDerKomponente }
```

Listing 6.8
Datenbindungssyntax

Bei der Komponente kann es sich um Properties einer FormBean, Aktionen oder eine Zeichenkette in einer Ressourcendatei handeln.

Der Einsatz dieser Datenbindungssyntax kann im Gegensatz zu eigenen Tags auch als Attributwert für andere Tags eingesetzt werden, was einen echten Vorteil gegenüber Struts darstellt, wo man sich häufig eines JSP-Ausdrucks bedient, um dieses Manko zu umgehen.

6.2.3 Controller

Controller sind das JSF-Gegenstück zu Struts Actions und werden gerufen, wenn auf der JSP eine Aktion, wie beispielsweise das Betätigen einer Schaltfläche, ausgelöst wird. Im Gegensatz zu Struts, wo `Action` und `ActionForm` immer in zwei unterschiedliche Klassen aufgeteilt sind, ermöglicht es das JSF-Framework, beide auch als eine JavaBean zu realisieren. In diesem Fall beinhaltet die JavaBean die »Action-Methoden« sowie die Liste aller Adressen und die aktuell bearbeitete Adresse gleichermaßen.

Listing 6.9
Der AddressController

```
package de.akdabas.javee.jsf;
import java.util.*;
import de.akdabas.javee.beans.Address;

/**
 * Controller für die Adressverwaltung.
 * Diese Klasse definiert Aktionen, die ausgeführt werden, wenn
 * z.B. CommandButtons betätigt werden und dient gleichzeitig
 * als JavaBean (FormBean) für die Variablen der JSPs
 */
public class AddressController {

   /* Konstanten, die in der Datei 'faces-config.xml' als outcomes
      für die Navigation dienen */
   private static final String OUTCOME_LIST = "listAddresses";
   private static final String OUTCOME_EDIT = "editAddress";

   /** Die aktuell zu editierende Adresse. */
   private Address currentAddress = null;

   /** Alle bisher gespeicherten Adressen */
   private List<Address> addressList = null;

   /** Konstruktor */
   public AddressController() {
      addressList = new ArrayList<Address>();
   }

   /**
    * Diese Aktion legt eine neue Adresse an und leitet auf
    * die Seite 'AddressDetail.jsp' weiter.
    */
   public String createAddress() {
      currentAddress = new Address();
      return OUTCOME_EDIT;
   }
```

Listing 6.9 (Forts.)
Der AddressController

```
    /**
     * Diese Aktion bricht das Erfassen einer neuen Adresse ab und
     * kehrt zur Liste aller gespeicherten Adressen zurück.
     */
    public String cancel() {
       currentAddress = null;
       return OUTCOME_LIST;
    }

    /**
     * Diese Methode übernimmt die Daten der neu erfassten Adresse
     * in die Liste und stellt anschließend die 'addressList.jsp'
     * dar.
     */
    public String save() {
       addressList.add(currentAddress);
       currentAddress = null;
       return OUTCOME_LIST;
    }

    // Ab hier folgen Getter und Setter für Variablen dieser JavaBean
    // damit diese per Daten-Bindungssyntax in die JSP eingebunden
    // werden können.

    public Address getCurrentAddress() {
       return currentAddress;
    }
    public void setCurrentAddress(Address currentAddress) {
       this.currentAddress = currentAddress;
    }

    public List<Address> getAddressList() {
       return addressList;
    }
    public void setAddressList(List<Address> addressList) {
       this.addressList = addressList;
    }
}
```

Wie Sie sehen, unterstützt dieser Controller die drei Operationen `createAddress()`, `cancel()` sowie `save()` und initialisiert die ebenfalls lokal gespeicherten Variablen der JSP gemäß den jeweiligen Anforderungen der nachfolgenden Seite.

Die größten Unterschiede im Vergleich zur Struts Action sind dabei:

- Es handelt sich um eine gewöhnliche JavaBean (POJO).
- Der Controller ist Struts Action und ActionForm in einem.
- Bei den zurückgegebenen symbolischen Links (*Outcomes*), die die jeweils als Nächstes darzustellende Seite symbolisieren, handelt es sich um gewöhnliche Strings und nicht um spezielle `ActionForwards`.

6.2.4 Konfiguration

Bevor Sie Ihr Werk gleich testen können, heißt es allerdings – wie beim Struts Framework – zunächst einmal konfigurieren. Hierfür dient die Datei *faces-config.xml* unterhalb von *WEB-INF*.

Listing 6.10
Konfiguration der Beispielanwendung

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE faces-config PUBLIC
   "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
   "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">

<faces-config>

   <!-- Definition der unterstützten Locales -->
   <application>
      <locale-config>
         <default-locale>de</default-locale>
         <supported-locale>en</supported-locale>
      </locale-config>
   </application>

   <!-- Definition des Workflows in Form von Navigationsregeln -->
   <navigation-rule>
      <navigation-case>
         <from-outcome>listAddresses</from-outcome>
         <to-view-id>/jsp/addressList.jsp</to-view-id>
      </navigation-case>
      <navigation-case>
         <from-outcome>editAddress</from-outcome>
         <to-view-id>/jsp/addressDetail.jsp</to-view-id>
      </navigation-case>
   </navigation-rule>

   <!-- Einbinden des Adress-Controllers -->
   <managed-bean>
      <managed-bean-name>addressController</managed-bean-name>
      <managed-bean-class>
         de.akdabas.javee.jsf.AddressController</managed-bean-class>
      <managed-bean-scope>session</managed-bean-scope>
   </managed-bean>
</faces-config>
```

Dies genügt schon vollkommen, um die Beispielanwendung zu konfigurieren! Die drei Toplevel-Elemente haben dabei folgende Funktion:

- `<application>` Hier werden z.B. die unterstützten Locales angegeben.
- `<navigation-rule>` Diese bilden das Herz des Controllers, da hier der Workflow der Applikation festgelegt wird. Diese Elemente werden Sie im nachfolgenden Absatz über Navigationsregeln genauer kennen lernen.
- `<managed-bean>` Hier beschreiben Sie die FormBeans Ihrer Applikation und legen neben der implementierenden Klasse auch deren Gültigkeitsbereich fest.

6.3 Navigationsregeln

Die Konfigurationsdatei *faces-config.xml* steuert, abhängig vom Rückgabewert der Aktionen, welche JSP als Nächstes angezeigt werden soll. Sie sind damit für den Workflow der Anwendung verantwortlich.

Aktionen werden durch `UICommand`-Elemente wie den `CommandButton` oder den `CommandLink` ausgelöst und führen über einen Request auf eine neue

Seite (bzw. eine andere Anzeigekomponente). Die in Listing 6.10 definierten Navigationsregeln sind dabei von der allgemeinsten Form:

- Sie sind allgemein und in der gesamten Anwendung gültig, egal, von welcher Seite der entsprechende Link aufgerufen wurde.
- Sie sind unabhängig von der Aktion, die zuvor ausgeführt wurde.
- Sie werden einzig über eine Zeichenkette identifiziert, die von der jeweiligen Action-Methode zurückgegeben wurde.

Da die Aktionen in diesem Beispiel immer nur an eine fest definierte JSP weiterleiten, hätten Sie Navigationsregeln auch folgendermaßen realisieren können.

Listing 6.11
Alternative Konfiguration der Navigationsregeln

```
...
   <!-- Alternative Konfiguration der Navigationsregeln -->
   <navigation-rule>
     <from-view-id>/jsp/addressList.jsp</from-view-id>
     <navigation-case>
       <from-action>#{addressController.createAddress}</from-action>
       <to-view-id>/jsp/addressDetail.jsp</to-view-id>
     </navigation-case>
   </navigation-rule>
   <navigation-rule>
     <from-view-id>/jsp/addressDetail.jsp</from-view-id>
     <navigation-case>
       <from-action>#{addressController.save}</from-action>
       <to-view-id>/jsp/addressList.jsp</to-view-id>
     </navigation-case>
     <navigation-case>
       <from-action>#{addressController.cancel}</from-action>
       <to-view-id>/jsp/addressList.jsp</to-view-id>
     </navigation-case>
   </navigation-rule>
...
```

Achtung

Alle Navigationsregeln sind *optional*. Geben Sie keine Regel an, wird der Request anschließend auf die Seite unter der URL `<from-view-id>` weitergeleitet. Auch wenn Sie gar keine Regel (`navigation-rule`) angeben, kommen Sie wieder auf die Seite, von der der Request gesendet wurde, zurück. Wenn Sie also trotz Betätigung eines `CommandButton` immer wieder auf der gleichen Seite landen, überprüfen Sie, ob Sie die gewünschten Regeln bereits implementiert haben.

Das Tag `<from-view-id>` beschreibt die Ausgangsseite (URL), von der aus das Kommando ausgeführt wurde. Anschließend können Sie einen oder mehrere Workflows (`<navigation-case>`) definieren, die drei Parameter enthalten:

- `<from-action>` Name der gerufenen Aktion
- `<from-outcome>` Rückgabewert der Action
- `<to-view-id>` URL der nächsten anzuzeigenden View-Komponente

6.3.1 Verwendung von Wildcards

Gerade in größeren Applikationen mit vielen Komponenten ist es nicht sinnvoll, für jede aufzurufende Seite einen Ausgangspunkt `<from-view-id>` zu definieren, da mit jeder neuen Komponente alle bestehenden Regeln erweitert werden müssten. Aus diesem Grund gestattet das JSF-Framework die Verwendung von Platzhaltern (Wildcards).

Um beispielsweise auf eine zentrale Hilfeseite zu verweisen, die prinzipiell von jedem Punkt der Anwendung und aus jeder Aktion erreicht werden kann, definieren Sie:

Listing 6.12
Definition einer statischen Navigation-Rule

```
...
   <navigation-rule>
      <from-view-id>*</from-view-id>
      <navigation-case>
         <from-outcome>help</from-outcome>
         <to-view-id>/help.jsp</to-view-id>
      </navigation-case>
   </navigation-rule>
...
```

Wie Sie sehen, wird auch die Angabe einer Aktion nicht zwingend benötigt. Hierbei kommt die so genannte *statische Navigationssteuerung* ins Spiel, welche immer dann sinnvoll ist, wenn ein Link ohne Aktion aufgerufen werden soll. Mit obiger Navigationsregel können Sie den nun folgenden `CommandButton` an beliebiger Stelle einsetzen.

Listing 6.13
Eine Hilfe-Schaltfläche

```
...
   <!-- Ein beliebig platzierbarer Hilfe-Button -->
   <h:commandButton action="help" value="Hilfe"/>
...
```

Wenn Sie diese Schaltfläche mit den Schaltflächen Ihrer bisherigen JSPs vergleichen, werden Sie feststellen, dass für den Wert des Attributs `action` diesmal keine Datenbindungssyntax, sondern eine statische Zeichenkette `help` eingesetzt wurde. Dies ist sinnvoll, da nicht jede FormBean eine besondere Aktion für die Hilfe benötigt. Für das Controller-Servlet hat dieser Link immer den gleichen Wert.

Sie können die Wildcards auch verwenden, um die entsprechenden Links auf eine bestimmte Domäne (z.B. ein Verzeichnis) Ihrer Applikation zu beschränken:

Listing 6.14
Einschränken einer statischen Navigation-Rule auf eine Domäne

```
...
   <navigation-rule>
      <from-view-id>/login/*</from-view-id>
      <navigation-case>
         <from-outcome>help</from-outcome>
         <to-view-id>/loginHelp.jsp</to-view-id>
      </navigation-case>
   </navigation-rule>
...
```

In diesem Fall gilt die Regel nur für URLs innerhalb der Domäne `/login`.

6.3.2 Standardverhalten

Neben dem Verhalten auf spezifische Rückgabewerte Ihrer Aktionen können Sie innerhalb der Navigationsregel auch ein Standardverhalten definieren, das immer dann zum Einsatz kommt, wenn kein `<from-outcome>`-Parameter passt. Hierzu lassen Sie die Angabe `<from-outcome>` einfach weg.

Listing 6.15
Standardverhalten einer Navigation-Rule definieren

```
...
   <navigation-rule>
      <from-view-id>/jsp/addressDetail.jsp</from-view-id>
      <navigation-case>
         <from-action>#{addressController.save}</from-action>
         <from-outcome>listAddresses</from-outcome>
         <to-view-id>/jsp/addressList.jsp</to-view-id>
      </navigation-case>

      <!-- Falls keine der Regeln passt, wird diese gerufen -->
      <navigation-case>
         <to-view-id>/failure.jsp</to-view-id>
      </navigation-case>
   </navigation-rule>
...
```

In diesem Fall wird der Anwender, wenn die Aktion `AddressController.save` nicht gerufen wird oder einen anderen Wert als `listAddresses` zurückliefert, immer auf die Seite *failure.jsp* weitergeleitet.

6.3.3 Prioritäten der Verarbeitung

Sie ahnten es sicher schon: Wenn innerhalb einer Navigation-Rule mehrere `<navigation-case>`-Elemente existieren, kann es zu Überschneidungen kommen. Aus diesem Grund definiert das JSF-Framework die Reihenfolge, in der nach einem passenden Fall gesucht wird:

- Zunächst wird ein Fall gesucht, bei dem sowohl der Name der Aktion als auch Ihr Rückgabewert übereinstimmen.
- Wird keine entsprechende Regel gefunden, versucht es das Framework unabhängig vom Namen der gewählten Aktion nur anhand des Rückgabeparameters.
- Schließlich wird nur das `<from-action>`-Element überprüft.
- Hat das Framework anschließend immer noch keinen passenden Fall gefunden, wird der Request wieder auf die Ausgangsseite zurückgeleitet.

6.4 Validieren von Daten

Wo gehobelt wird, da fallen auch Späne: Immer dann, wenn Benutzereingaben zur Weiterverarbeitung genutzt werden sollen, müssen diese zuerst auf Korrektheit überprüft werden, um beispielsweise ein Fehlverhalten der Anwendung aufgrund eines Tippfehlers zu vermeiden.

Um Daten zu validieren, stellt das JSF-Framework eine Reihe von Validatoren bereit, die es bereits bei der Erstellung der JSPs ermöglichen, anzugeben, ob ein Feld ein Pflichtfeld ist und welcher Wertebereich zulässig ist. Die eigentliche Validierung findet anschließend serverseitig statt, wobei mit Hilfe eigener Renderer auch clientseitige Validierungen vorgenommen werden können.

Die Validierung der Daten findet nach einem Request, jedoch vor der Aktualisierung des zugrunde liegenden Datenmodells (`FormBean`) statt. Entsprechen die angegebenen Daten nicht den Vorgaben, wird der Request zurück auf die Ausgangsseite geleitet, damit der Anwender seine Daten korrigieren kann.

6.4.1 Standardvalidatoren

Das JSF-Framework kommt zunächst mit drei Standardvalidatoren, die Sie jedoch um eigene erweitern können.

Tabelle 6.1 Validatoren des JSF-Framework

Validator	Beschreibung
`validateLength`	Überprüft die Länge einer Zeichenkette.
`validateLongRange`	Überprüft eine Zahl vom Datentyp `long` auf einen vorgegebenen Wertebereich.
`validateDoubleRange`	Überprüft eine Zahl vom Datentyp `double` auf einen vorgegebenen Wertebereich.

Um ein Eingabefeld mit einem Validator zu verknüpfen, fügen Sie einfach ein entsprechendes Tag in seinen Rumpf ein:

Listing 6.16 Beispiel für die Verwendung eines Validators

```
...
   <h:inputText id="name" required="true"
                value="#{addressController.currentAddress.name}"/>
      <f:validateLength minimum="3" maximum="20"/>
   </h:inputText>
...
```

In diesem Fall muss der Nachname zwischen 3 und 20 Stellen haben. Bei den Validatoren für `Long` und `Double` begrenzen `minimum` und `maximum` hingegen den gültigen Wertebereich. Fehlt eines der beiden Attribute, ist der Wertebereich bzw. die Länge auf einer Seite unbeschränkt.

6.4.2 Pflichtfelder deklarieren

Um ein Eingabefeld als Pflichtfeld zu deklarieren, könnten Sie einen String-Validator mit der Mindestlänge 1 deklarieren:

Listing 6.17 Input zum Pflichtfeld machen (Version 1)

```
...
   <h:inputText id="name"
                value="#{addressController.currentAddress.name}"/>
      <f:validateLength minimum="1"/>
   </h:inputText>
...
```

Es geht jedoch auch eleganter, indem Sie einfach das Attribut `required` auf `true` setzen.

Listing 6.18 Input zum Pflichtfeld machen (Version 2)

```
...
   <h:inputText id="name" required="true"
             value="#{addressController.currentAddress.name}"/>
...
```

Während erstere Variante nur bei String-Werten funktioniert, kann die Überprüfung per `required`-Attribut unabhängig vom Datentyp eingesetzt werden. Letzterer Weg sollte im Interesse einer konsistenten Schreibweise vorgezogen werden.

6.4.3 Validierungsfehler ausgeben

Nun ist das Validieren von Eingaben sicherlich eine nützliche Sache, die Ihnen eine Reihe von Standardaufgaben abnehmen kann, aber wenn der Benutzer nicht auf seinen Fehler hingewiesen wird, kann er ihn in der Regel auch nicht beheben. Bei Struts bedienen Sie sich bei Validierungsfehlern des `ActionMessages`-Objekts, dessen Inhalt (die Fehlermeldungen) über das gleichnamige Tag ausgegeben werden konnten.

Auch für das JSF-Framework existiert ein solches Tag, das Sie nach Belieben in Ihre JSPs integrieren können:

Listing 6.19 Ausgabe von (Fehler-) Meldungen

```
...
   <h:messages/>
...
```

Achtung

Dieses Tag gibt jedoch alle Meldungen und nicht nur jene über aufgetretenen Validierungsfehler aus. Nützlicher ist es, die einzelnen Fehlermeldungen zum spezifischen Tag auszugeben. Dazu erweitern Sie die Datei *jsp/addressDetail.jsp* (Listing 6.7) nun folgendermaßen:

Listing 6.20 Um Validierungsmeldungen erweiterte addressDetail.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<html>
<body>
<f:view>
   <h:form>

      <%-- Einlesen des Resource-Bundles und binden an die lokale
           Variable msg --%>
      <f:loadBundle basename="resources.application" var="msg"/>

      <fieldset>
         <%-- Ausgabe eines Labels aus dem Ressource-Bundle --%>
         <h3><h:outputText value="#{msg.detail_heading}"/></h3>

         <%-- Deklaration eines Grids, welches nach jeweils 3
              Elementen eine neue Zeile beginnt.--%>
         <h:panelGrid columns="3">
```

Listing 6.20 (Forts.)
Um Validierungsmeldungen erweiterte addressDetail.jsp

```
            <%-- Ausgabe eines Labels aus den Ressourcen --%>
            <h:outputLabel for="firstName"
                           value="#{msg.address_firstName}"/>
            <h:inputText id="firstName" required="true"
                         value="#{addressController
                                  .currentAddress.firstName}"/>
            <h:message for="firstName"/>

            <h:outputLabel for="name" value="#{msg.address_name}"/>
            <h:inputText id="name" required="true"
                         value="#{addressController
                                  .currentAddress.name}"/>
            <h:message for="name"/>

           <h:outputLabel for="mail" value="#{msg.address_eMail}"/>
            <h:inputText id="mail" required="true"
                         value="#{addressController
                                  .currentAddress.mail}">
            </h:inputText>
            <h:message for="mail"/>

          </h:panelGrid>

          <h:commandButton action="#{addressController.cancel}"
                           value="#{msg.detail_cancel}"
                           immediate="true"/>
          <h:commandButton action="#{addressController.save}"
                           value="#{msg.detail_save}"/>
      </fieldset>
   </h:form>
</f:view>
</body>
</html>
```

Zunächst müssen nun alle Eingabefelder mit einem Wert belegt werden (`required="true"`). Ist ein Feld leer, schlägt die Validierung fehl und es wird eine entsprechende Fehlermeldung erzeugt.

Diese feldspezifische Fehlermeldung kann über das nun folgende Tag in der gleichen Zeile wie das Eingabefeld selbst ausgegeben werden.

```
<h:message for="ID des Feldes"/>
```

Listing 6.21
Ausgabe einer spezifischen Fehlermeldung

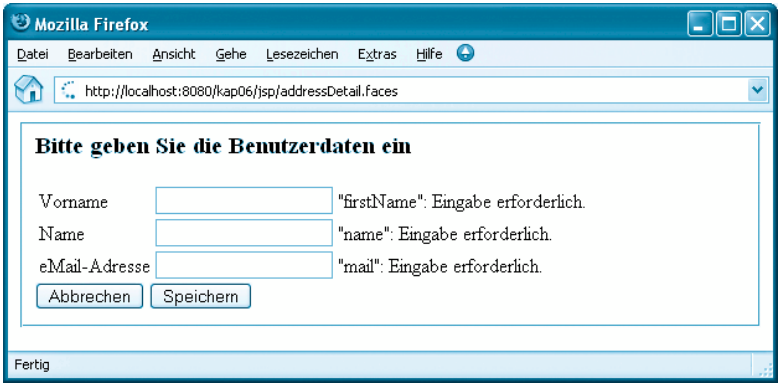


Abbildung 6.3
Ausgabe spezifischer Validierungsfehler

6.4.4 Überschreiben der vorgefertigten Fehlermeldungen

Wie Sie in Listing 6.4 sehen, liefert das JSF-Framework bereits standardisierte Ressourcen für Validierungsfehler mit. Das sind:

Tabelle 6.2 Standardfehlermeldungen des JSF-Framework

Schlüssel	Standardtext
`javax.faces.validator.NOT_IN_RANGE`	Meldung: Spezifiziertes Attribut liegt nicht zwischen {0} und {1}.
`javax.faces.validator.DoubleRangeValidator.LIMIT`	Meldung: Spezifiziertes Attribut kann nicht in einen Double-Typ umgewandelt werden.
`javax.faces.validator.DoubleRangeValidator.MAXIMUM`	Meldung: Wert ist größer als zulässiges Maximum "{0}".
`javax.faces.validator.DoubleRangeValidator.MINIMUM`	Meldung: Wert ist kleiner als zulässiges Minimum "{0}".
`javax.faces.validator.DoubleRangeValidator.TYPE`	Meldung: Wert ist nicht vom richtigen Datentyp.
`javax.faces.validator.LengthValidator.LIMIT`	Meldung: Spezifiziertes Attribut kann nicht in korrekten Typ umgewandelt werden.
`javax.faces.validator.LengthValidator.MAXIMUM`	Meldung: Wert ist größer als zulässiges Maximum "{0}".
`javax.faces.validator.LengthValidator.MINIMUM`	Meldung: Wert ist kleiner als zulässiges Minimum "{0}".
`javax.faces.component.UIInput.REQUIRED`	Meldung: Wert wird benötigt.
`javax.faces.component.UISelectOne.INVALID`	Meldung: Wert nicht gültig.
`javax.faces.component.UISelectMany.INVALID`	Meldung: Wert nicht gültig.
`javax.faces.validator.RequiredValidator.FAILED`	Meldung: Wert wird benötigt.
`javax.faces.validator.LongRangeValidator.LIMIT`	Meldung: Spezifiziertes Attribut kann nicht in korrekten Typ umgewandelt werden.
`javax.faces.validator.LongRangeValidator.MAXIMUM`	Meldung: Wert ist größer als zulässiges Maximum "{0}".
`javax.faces.validator.LongRangeValidator.MINIMUM`	Meldung: Wert ist kleiner als zulässiges Minimum "{0}".
`javax.faces.validator.LongRangeValidator.TYPE`	Meldung: Wert ist nicht vom richtigen Datentyp.

Sollten Ihnen die Texte nicht zusagen, können Sie diese natürlich auch überschreiben. Dazu müssen Sie den Schlüssel des zu überschreibenden Fehlertextes einfach in Ihr Ressourcen-Bündel kopieren und mit einem beliebigen Text überschreiben, wie es Ihnen die folgenden Beispiele demonstrieren:

```
javax.faces.component.UIInput.REQUIRED=Bitte geben Sie den
benötigten Wert an!

javax.faces.validator.LengthValidator.MINIMUM=Die Länge des angege-
benen Wertes ist kleiner als die erforderliche Mindestlänge von {0}
Zeichen.
```

Listing 6.22
Beispiele für überschriebene Fehlermeldungen

6.4.5 Implementieren eines eigenen Validators

Analog zum Struts-Framework unterstützen auch JavaServer Faces die Implementierung von `validate()`-Methoden innerhalb der Managed Beans. Diese Methode des Validierens eignet sich natürlich nur für spezifische Anforderungen einer Bean. Um allgemeingültige Elemente wie die E-Mail-Adresse in jeder JSP zu überprüfen, verwenden Sie eigene Validatoren. Diese sind unabhängig von einer Managed Bean und können leicht wieder verwendet werden.

Implementierung des Validators

Ein Validator ist eine Java-Klasse, die das Interface `javax.faces.validator.Validator` und die dort deklarierte Methode `validate()` implementiert. Diese unterscheidet sich, wie Sie gleich sehen werden, kaum von den `validate()`-Methode der FormBean. Außer, dass sie bei einer ungültigen E-Mail-Adresse eine `ValidatorException` wirft, deren Konstruktor die `FaceMessage` erhält. Diese wird vom Framework abgefangen und die Nachricht anschließend ausgegeben.

Tipp

Dies ist kein Buch über reguläre Ausdrücke und so werden wir erst gar nicht versuchen, das hier beschriebene Muster zu dokumentieren. Wenn Sie einmal einen bestimmten Ausdruck suchen und ihn partout nicht zusammenbekommen, sei Ihnen die Website *http://www.regexlib.com* ans Herz gelegt.

Listing 6.23
Ein eigener E-Mail-Validator

In diesem Beispiel verwenden Sie einen Ressourcen-Bündel für die spezifischen Ressourcen der Applikation sowie für die wieder verwendbaren Ressourcen, wie den E-Mail-Validator. Lagern Sie diese in der Praxis in ein eigenes Ressourcen-Bündel aus und verpacken Sie Validator und Ressourcendatei in ein gemeinsames JAR-File um eine vollständige wieder verwendbare Komponente zu erzeugen.

```
package de.akdabas.javee.jsf;
import java.util.regex.Pattern;
import java.util.regex.Matcher;
import java.util.ResourceBundle;

import javax.faces.context.FacesContext;
import javax.faces.component.UIComponent;
import javax.faces.application.FacesMessage;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;

/** Validiert eMail-Adressen auf syntaktische Korrektheit */
public class EmailValidator implements Validator {

   /** Validiert den Wert des übergebenen Objekt 'obj' */
   public void validate(FacesContext ctx, UIComponent component,
                        Object obj)
         throws ValidatorException {

      // Ressourcen-Bundle laden um die Fehlermeldung auszulesen
      ResourceBundle bundle = ResourceBundle.getBundle(
            "resources.application",
            ctx.getViewRoot().getLocale()
      );

      // Fehlermeldung auslesen
      String errorKey = "errors_email";
      String errorMsg = bundle.getString(errorKey);

      // Fehlermeldung im Kontext speichern, damit diese ggf. auch
      // in der JSP zur Verfügung steht
      FacesMessage msg = new FacesMessage(errorMsg);

      // 'null' ist ebenfalls ein ungültiger Wert
      if (obj == null) {
         throw new ValidatorException(msg);
      }

      // Muster gültiger eMail-Adressen als Regular Expression
      String pattern =
            "^[\\w-]+(?:\\.[\\w-]+)*@(?:[\\w-]+\\.)+[a-zA-Z]{2,7}$";

      // Pattern-Instanz erzeugen und auf Gültigkeit überprüfen
      Pattern p = Pattern.compile(pattern);
      Matcher m = p.matcher(obj.toString());
      if (!m.matches()) {
         // Wert passt nicht auf den Ausdruck
         throw new ValidatorException(msg);
      }
   }
}
```

Registrieren des Validators

Bevor Sie Ihren Validator verwenden können, müssen Sie diesen im JSF-Framework unter einer eindeutigen ID registrieren. Fügen Sie dazu folgende Tags in ihre Konfigurationsdatei (*faces-config.xml*) ein:

Listing 6.24
Registrieren des eMail-Validators im JSF-Framework

```
...
   <!-- Einbinden eines eigenen Validators -->
   <validator>
      <validator-id>eMailValidator</validator-id>
      <validator-class>
         de.akdabas.javee.jsf.EmailValidator</validator-class>
   </validator>
...
```

Verwenden des E-Mail-Validators in der JSP

Nun können Sie den Validator unter Angabe seiner ID nach Belieben einsetzen. Das folgende Listing zeigt die Anpassungen in der *input.jsp*.

Listing 6.25
Einsatz des Validators

```
...
   <h:inputText id="mail" required="true"
                value="#{addressController.currentAddress.mail}">
      <f:validator validatorId="eMailValidator"/>
   </h:inputText>
...
```

Ein weiterer Vorteil in der Verwendung eines separaten Validators besteht darin, dass nun auch eine spezifische, diesem Eingabefeld zugeordnete Fehlermeldung existiert, die über das Tag `<h:message>` ausgegeben werden kann.

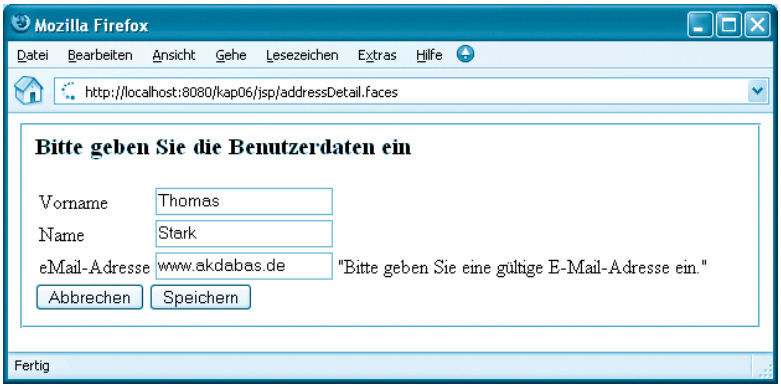


Abbildung 6.4
Externer Validator in Aktion

6.5 Konverter

HTML kennt keine Datentypen, eine Managed Beans schon. Dies kann ein Dilemma werden, wenn man bedenkt, dass sämtliche Eingaben im Frontend in Textform und in der Bean in ihrem ursprünglichen Datentyp vorliegen. Um die Daten aus der Präsentations- in die Datenschicht zu überführen, finden deshalb Konvertierungen statt. In der Regel übernimmt die Komponente selbst die Datenkonvertierung, so dass sich ein Anwendungsentwickler nicht darum kümmern muss. Manchmal ist es jedoch sinnvoll, die Art der Konvertierung und damit auch Ausgabe- und Eingabeformate selbst festzulegen.

Das JSF-Framework stellt einige vordefinierte Konverter bereit, die sich alle im Paket `javax.faces` befinden:

- `BigDecimalConverter`
- `BigIntegerConverter`
- `BooleanConverter`
- `ByteConverter`
- `CharacterConverter`
- `DateTimeConverter`

- DoubleConverter
- FloatConverter
- IntegerConverter
- LongConverter
- NumberConverter
- ShortConverter

Alle diese Konverter implementieren das Interface javax.faces.convert.Convert, das die Methoden getAsObject() und getAsString() deklariert, die für die Konvertierung der Daten eingesetzt werden können.

Um bei den Standardkomponenten des JSF-Framework Konverter explizit anzugeben, können Sie zwei Möglichkeiten nutzen:

- Geben Sie in der Eingabekomponente mit Hilfe von deren Attribut convert den zu verwendenden Konverter an.
- Fügen Sie einen Konverter mithilfe von Konverter-Tags hinzu.

6.5.1 Das Converter-Attribut

Das converter-Attribut kann verwendet werden, um einen Konverter explizit anzugeben. Sinnvoll ist dies, wenn Eingaben zwingend in ein bestimmtes Format umgewandelt werden müssen, dabei aber keine weiteren Informationen bezüglich Formatierung oder konverterspezifischer Attribute übergeben werden müssen.

Für die Eingabe einer Postleitzahl könnte beispielsweise ein Integer-Konverter explizit angegeben werden:

Listing 6.26 Einsatz eines Integer-Konverters für Postleitzahlen

```
...
   <!-- Konvertieren einer Postleitzahl -->
   <h:inputText id="zip" value="#{EineFormBean.zip}"
                required="true" converter="javax.faces.Integer">
      <f:validateLongRange maximum="99999" minimum="10000"/>
   </h:inputText>
...
```

Sollte nun ein nicht dem Datentyp entsprechender Wert eingegeben worden sein, wird nicht etwa eine Exception geworfen, sondern innerhalb eines <message>- oder <messages>-Element eine Fehlermeldung ausgegeben.

6.5.2 Das Converter-Tag

Wenn Sie an Datumswerte denken, werden Sie sicherlich zustimmen, dass es hier unzählige Möglichkeiten gibt, wie ein Datum ausgedrückt werden kann. Dementsprechend dürfte es schwierig sein, die entsprechenden Möglichkeiten mit nur einem Attribut auszudrücken.

Aus diesem Grund sind mehrere Konverter-Tags eingeführt worden, die derartige Aufgaben erleichtern:

- ConvertDate zur Konvertierung in Datumswerte
- ConvertNumber zur Konvertierung in numerische Werte

Konvertieren eines Datumswerts

Angenommen, Sie wollten die Eingabe eines Datums erlauben, dann könnte dies so umgesetzt werden:

Listing 6.27
Verwendung eines Konverter-Tag für ein Datum

```
...
   <!-- Konvertieren eines Datumsfeldes -->
   <h:inputText id="birth" value="#{UserData.birth}"
                required="true">
      <f:convertDateTime dateStyle="short" type="date"/>
   </h:inputText>
...
```

Der `DateTime`-Konverter bietet folgende Attribute:

Tabelle 6.3
Attribute des DateTime-Konverters

Attribut	Typ	Bedeutung
`dateStyle`	`String`	Legt das Datumsformat fest. Mögliche Wert sind: ■ `default` ■ `short` ■ `medium` ■ `long` Wird kein Wert angegeben, wird `default` angenommen.
`parseLocale`	`String` oder `Locale`	Gibt an, welche `Locale` verwendet werden kann. Per Default wird die `Locale` verwendet, die im aktuellen `FacesContext` definiert wird.
`pattern`	`String`	Legt das Muster für ein benutzerdefiniertes Datums- und Zeitformat fest. Wenn ein Muster angegeben worden ist, werden die Werte der Attribute `dateStyle`, `timeStyle` und `type` ignoriert.
`timeStyle`	`String`	Legt das Format für die Zeit fest. Mögliche Werte sind: ■ `default` ■ `short` ■ `medium` ■ `full` Wird kein Wert angegeben, wird `default` angenommen.
`type`	`String`	Gibt an, ob der Wert als Datum, Zeit oder eine Kombination aus beidem interpretiert wird. Mögliche Werte sind: ■ `date` ■ `time` ■ `datetime` Wird hier kein Wert angegeben, wird `date` verwendet.

Konvertierung eines numerischen Werts

Wollten Sie einen numerischen Wert konvertieren, können Sie auf einen `Number`-Konverter zurückgreifen. Am Beispiel einer Preisangabe könnte dies so aussehen:

Listing 6.28 Konvertierung eines Number-Werts

```
...
   <!-- Konvertierung eines nummerischen Wertes -->
   <h:inputText id="price" value="" required="true">
      <f:convertNumber type="currency"
         maxFractionDigits="2" maxIntegerDigits="3"
         minFractionDigits="2" minIntegerDigits="1"/>
   </h:inputText>
...
```

Achtung

Vorsicht: Erlaubte Währungssymbole sind von der `Locale`-Einstellung abhängig – in der Euro-Zone wird nur das Euro-Symbol akzeptiert, andere Währungssymbole erzeugen Fehlermeldungen.

Hier eingegebene Werte dürften maximal drei Vorkomma- und maximal zwei Nachkommastellen haben. Andererseits muss auch mindestens eine Vorkommastelle vorhanden sein und da auch der Typ der Eingabe (`currency`) vorgegeben ist, muss diese zusätzlich über ein Währungssymbol verfügen.

Folgende Attribute stehen beim Number-Konverter zur Verfügung:

Tabelle 6.4 Attribute des Number-Konverters

Attribut	Typ	Bedeutung
`currencyCode`	`String`	ISO 4217-Ländercode, der jedoch nur bei der Formatierung von Währungen zur Anwendung kommt
`currencySymbol`	`String`	Währungssymbol; kommt nur bei der Formatierung von Währungen (`currency`) zur Anwendung
`groupingUsed`	`Boolean`	Gibt an, ob Tausender-Trenner verwendet werden
`integerOnly`	`Boolean`	Gibt an, ob nur ganzzahlige Werte verwendet werden
`maxFractionDigits`	`int`	Maximale Anzahl der Nachkommastellen, die formatiert werden
`maxIntegerDigits`	`int`	Maximale Anzahl der Vorkommastellen, die formatiert werden
`minFractionDigits`	`int`	Minimale Anzahl der Nachkommastellen, die formatiert werden

Tabelle 6.4 (Forts.) Attribute des Number-Konverters

Attribut	Typ	Bedeutung
`minIntegerDigits`	`int`	Minimale Anzahl der Vorkommastellen, die formatiert werden
`parseLocale`	`String` oder `Locale`	Gibt an, welche `Locale` verwendet werden kann.
`pattern`	`String`	Legt ein benutzerdefiniertes Muster fest
`type`	`String`	Gibt an, ob der Wert als Nummer, Währung oder Prozentangabe interpretiert werden soll. Mögliche Werte sind: ■ `number` ■ `currency` ■ `percentage` Die Default-Angabe ist `number`.

Die beschriebenen Konverter können (und sollen) nicht nur für die Eingabe, sondern auch für die Ausgabe von Daten verwendet werden:

Listing 6.29 Verwendung von Konvertern zur Ausgabe

```
...
   <!-- Formatiert ein Datum für die Ausgabe -->
   <h:outputText value="#{UserData.birth}">
      <f:convertDateTime dateStyle="short" type="date"/>
   </h:outputText>

   <!-- Formattiert eine Zahl in einen Währungs-String -->
   <h:outputText value="#{UserData.price}">
      <f:convertNumber maxFractionDigits="2" maxIntegerDigits="3"
                       minFractionDigits="2" minIntegerDigits="1"
                       type="currency"/>
   </h:outputText>
...
```

6.6 Zusammenfassung

Mit diesem Kapitel verlassen wir den Bereich webbasierter Frontend-Technologien und widmen uns den dahinter liegenden Schichten für Geschäftslogik und Persistenz. Die vorangegangnen Kapitel sollten Ihnen einen Einblick in die verschiedenen, aufeinander aufbauenden Technologien gegeben haben und Sie in die Lage versetzen, nun auch komplexe Webanwendungen mit Java EE zu realisieren.

Während Struts seinen Schwerpunkt auf die klare Implementierung des Model-View-Controller-Pattern (MVC) fokussiert, weicht das JSF-Framework die daran geknüpften Bedingungen auf und gestattet z.B. die Zusammenfassung von Action und ActionForm. Dies ermöglicht es Ihnen ohne großen Aufwand schnell, ansehnliche Weboberflächen zu gestalten und diese mit Leben zu füllen.

Die Roadmap für Struts sieht zwar das Zusammenwachsen von Struts und JSF vor, wobei Ersteres im Wesentlichen für den Workflow und die Abbildung der Daten zuständig ist, während die Benutzerschnittstelle mit JavaServer Faces implementiert werden können, dennoch werden JavaServer Faces wahrscheinlich nie die einzigen View-Komponenten für Struts-Applikationen sein.

Grob vereinfacht lässt sich sagen, dass die Stärken von Struts in der Controller-Komponente (`Action`) liegen und sich gerade bei komplexen Applikation bemerkbar machen, während die JSF-Spezifikation sich mehr und mehr in Richtung vorgefertigter Komponenten entwickelt, die beispielsweise auch über Designer-Tools, wie Macromedias Dreamweaver, zusammengefügt werden könnten.

7

Java Naming and Directory Interface

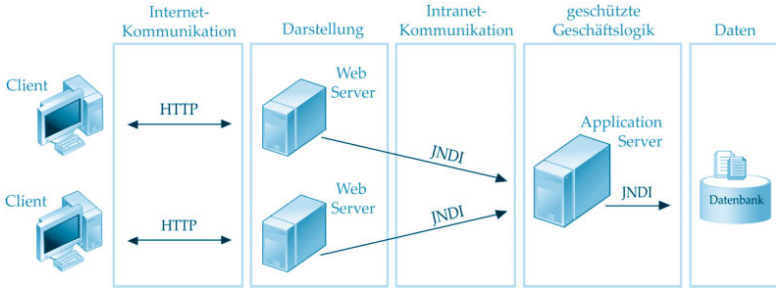
Mit diesem Kapitel bewegen Sie sich im Server-Client-Modell nun eine Schicht weiter. Wir verlassen den Bereich der direkten Interaktion mit dem Benutzer und widmen uns den dahinter liegenden Prozessen, die entweder innerhalb eines einzelnen Servers ablaufen oder auch auf verschiedene Rechner verteilt sein können (*Distributed Computing*).

Info

Unter *Distributed Computing* (dt. verteiltes Rechnen) versteht man Programmiertechniken, bei denen eine Anwendung in verschiedene Prozesse unterteilt wird, die anschließend von *verschiedenen Rechnern* ausgeführt werden können.

So kann bei Bedarf ein Webserver für die grafische Aufbereitung der Daten optimiert werden, die ihm wiederum von einem dahinter liegenden Application Server über Dienste bereitgestellt werden. Bei einer solchen Architektur ist es dann wiederum möglich, weitere Webserver hinzuzufügen, die entweder die Aufgabe haben, den ersten zu unterstützen (Cluster-Betrieb), oder auch zu einer völlig anderen Webanwendung gehören können. Der zweite Fall ist dabei vor allem dann von Vorteil, wenn die Geschäftlogik der (Web-)Anwendung komplex ist oder hohe Sicherheitsansprüche hat und auf einem speziell gesicherten und von außen nicht zu erreichenden Backend-System bereitgestellt werden soll.

Abbildung 7.1
Aufgabenverteilung und Bereitstellung von Diensten



Es hängt natürlich vom konkreten Einzelfall ab, inwieweit es sich lohnt, einzelne Prozesse auf unterschiedliche Server zu verteilen. So ist es für kleinere Webapplikationen, bei denen die benötigte Rechenkapazität von einem einzelnen Server erbracht werden kann, oftmals wenig sinnvoll, die Logikschicht, z.B. in Form von den im nächsten Kapitel besprochenen *Enterprise Java Beans (EJB)*, auf einen anderen Rechner auszulagern und den ganzen hierdurch entstehenden Verwaltungsaufwand in Kauf zu nehmen. Stattdessen können Sie die Applikationslogik in diesem Fall in Form »einfacher« JavaBeans kapseln, um den Code von der Darstellung zu trennen.

Info

Unter *Load Balancing* (dt. Lastverteilung) versteht man Verfahren, die ressourcenintensive Aufgaben auf mehrere parallel arbeitende Systeme verteilen. Beim Load Balancing zwischen Webservern müssen Sie dabei darauf achten, dass die Session des Clients auch auf dem Server verfügbar ist, der den aktuellen Request dieses Clients bearbeitet.

Bei größeren Projekten hingegen, bei denen die Rechenlast auf mehrere Server verteilt werden soll, sind *Load Balancing* und die im nächsten Kapitel besprochenen Enterprise JavaBeans (EJBs) als Service unverzichtbar.

Doch egal, ob Sie Ihren Code nun von einem oder einhundert Servern berechnen lassen: Was ich Ihnen in jedem Fall empfehle, ist die bereits in Kapitel 1 beschriebene Dreiteilung Ihrer Anwendung in *Darstellung*, *Logik* und *Datenhaltung*. Denn glauben Sie mir, nichts ist nervenaufreibender, als die gesamte Anwendung nach verstreuten SQL-Statements zu durchsuchen, nur weil Sie sich entschieden haben, eine Tabelle Ihrer Datenbank zu normalisieren.

Aufgaben von Namensdiensten

Eine der zentralen Aufgaben innerhalb großer Java-Applikationen ist die Bereitstellung von Informationen und Diensten. Und da diese in der Regel von eigenständigen Teilsystemen erbracht wird, besteht eine der größten Herausforderungen darin, diese Dienste und Informationen aufzufinden –

unabhängig davon, ob es sich dabei um die Authentifizierung eines Anwenders am Netzwerk oder den Zugriff auf eine Datenbank handelt.

Damit einzelne Rechner oder Applikationen untereinander kommunizieren und Objekte austauschen können, müssen sie wissen, welches System welchen Dienst erbringen kann und wo die Informationen abgelegt sind. Hierfür bedienen sich viele Anwendungen eines so genannten *Namens-* oder *Verzeichnisdienstes*.

7.1 Einführung in Namensdienste

Was verbirgt sich nun also hinter dem *Java Naming and Directory Interface (JNDI)*? Nun zunächst handelt es sich dabei, wie es der Name schon andeutet, lediglich um einen Satz von Schnittstellen (engl. Interfaces), die definieren, wie man mit Java auf verschiedene Namens- und Verzeichnisdienste zugreifen kann.

Info

An dieser Stelle begegnet Ihnen das Konzept der Trennung von API und SPI aus Kapitel 1 wieder.

Sun liefert also nur das Regelwerk, welches durch verschiedene Gremien definiert wurde, sowie eine gemeinsame Plattform, auf der Anbieter und Anwender von Diensten zusammenarbeiten können, während die Hersteller von Namens- bzw. Verzeichnisdiensten dazu angehalten sind, diese Normen mit Implementierungen auszufüllen. Die folgende Abbildung verdeutlicht dies schematisch.

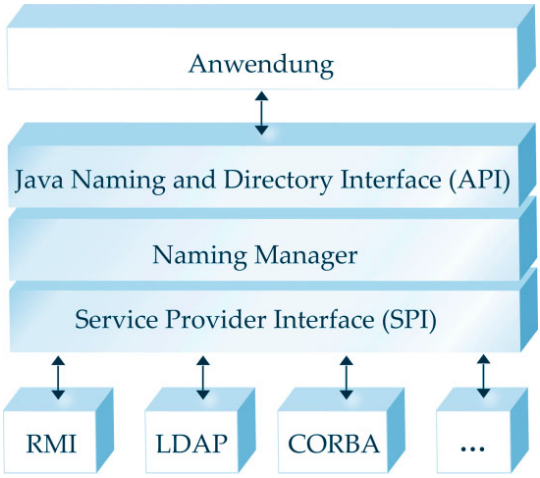


Abbildung 7.2
JNDI-Architektur (schematisch)

Während Sie Ihre Anwendung auf Grundlage des *Application Programming Interface (API)* entwickeln, programmieren Anbieter von Namensdiensten entsprechende Schnittstellen auf Basis eines so genannten *Service Provider Interface (SPI)*. Sun stellt lediglich den so genannten *Naming Manager*, der die Aufgabe hat, zwischen beiden Schichten zu vermitteln.

Info

JNDI ist im Übrigen nicht die einzige Technologie, bei der Sun lediglich eine Spezifikation für API und SPI bereitstellt. Diese Technik hat sich in vielen Bereichen etabliert, da sie den Standardisierungsprozess unterstützt und beiden Seiten (Herstellern wie Anwendern) Sicherheit bietet. Sie werden dieser Technik also noch an vielen Stellen in diesem Buch wiederbegegnen.

Der große Vorteil dieser Architektur besteht in der einheitlichen Schnittstelle für Java-Programmierer. Egal, ob Sie nun auf ein *LDAP-Verzeichnis*, die Funktionalität einer *Enterprise JavaBean (EJB)* oder gar eine Mail-Session zugreifen wollen, solange der entsprechende Service von einem SPI implementiert wird, können wir diesen über die JNDI ansprechen.

7.1.1 Ein API – zwei Services

Nachdem Sie nun verstanden haben, dass das JNDI-API immer nur das eine Ende eines Kanals sein kann und Sie stets einen Partner benötigen, der den Service erbringt, können wir uns den eigentlichen Diensten widmen, auf die Sie später zugreifen werden.

- Ein Namensdienst (*Naming Service*)

  Hierunter versteht man die Abbildung eines Namens auf ein damit assoziiertes Objekt. So ordnet ein Dateisystem beispielsweise jedem Dateinamen ein entsprechendes Dateiobjekt zu.

- Ein Verzeichnisdienst (*Directory Service*)

  Ein Verzeichnisdienst ist im Prinzip die Erweiterung eines Namensdienstes, indem er neben der Zuordnung von Namen zu Objekten auch die Angabe von Attributen zum Objekt erlaubt. Das können beispielsweise Lese- und Schreibrechte zu einer Datei sein.

Beide Dienste können dabei mit einem abstrakten Dateisystem verglichen werden, das es Ihnen ermöglicht, Objekte abzulegen, zu suchen, aufzulisten und gegebenenfalls auch wieder zu entfernen. Dabei ist es auch möglich, dass ein Dienst statt des Objekts selbst lediglich eine *Referenz* (Adresse) auf dieses Objekt zurückliefert. So übersetzt der wohl meistbekannte Namensdienst *Domain Name Services (DNS)* einen URL wie *google.de* in die zugehörige IP-Adresse des Servers, die der Browser anschließend verwendet, um den eigentlichen Request abzuschicken.

7.1.2 Der Kontext

Ein zentraler Begriff, um den sich bei Namens- und Verzeichnisdiensten fast alles dreht, ist der *Kontext* (`Context`). Der Kontext wird dabei durch eine Anzahl von Bindungen zwischen Namen und Objekten gebildet. So bildet die Liste aller Namen von Dateien und Unterverzeichnissen eines Ordners den Kontext dieses Ordners. Ein Unterverzeichnis können Sie in diesem Zusammenhang dann als Subkontext auffassen, wodurch ein hierarchisches System von verschiedenen Kontexten entsteht.

Ein Kontext ermöglicht Ihnen die Navigation durch die Hierarchie der Kontexte, die Suche (`lookup()`) und Auflistung (`list()`) von Objekten sowie das Erzeugen (`bind()`) und das Löschen (`unbind()`) von Einträgen.

Info

Namens- und Verzeichnisdienste lassen sich mit einem verallgemeinerten Dateisystem vergleichen. Der Kontext bildet dabei einen Ordner, in dem Objekte sowie weitere Ordner abgelegt werden können.

Wie Sie sehen, ist das JNDI also nichts anderes als ein verallgemeinertes Dateisystem, bei dem Sie sich stets an einer bestimmten Stelle im Verzeichnisbaum (engl. *Tree*) befinden, von der aus Sie gegebenenfalls weiter nach oben oder unten navigieren können. An der jeweils aktuellen Position können Sie sich die dort gespeicherten Elemente ausgeben lassen, neue Elemente erstellen oder vorhandene löschen.

7.1.3 Namensdienste

Ein Namensdienst (engl. Naming Service) verbindet also eine Menge von Namen mit Objekten eines bestimmten Typs. Er ermöglicht es Ihnen, diese in einem Kontext abzulegen und später, eventuell zusammen mit anderen Objekten dieses Typs, wieder zu extrahieren. Damit gehören Naming Services zu den fundamentalsten Diensten in allen Computersystemen, da sie es ermöglichen, Objekte in komplexen, hierarchischen Strukturen abzulegen und über verständliche Namen wieder zu extrahieren.

Aufbau von Namen

Die Namen innerhalb eines Namensdienstes folgen bestimmten zuvor definierten Syntaxregeln, so genannten *Naming Conventions*, die sich von Service zu Service unterscheiden können. Beim Domain Name Service (DNS) des Internets bestehen die Namen aus Zeichenketten, die durch Punkte (.) getrennt werden und keine Leerzeichen enthalten dürfen, wohingegen in Ihrem lokalen Dateisystem die einzelnen Verzeichnisse, je nach Betriebssystem, entweder durch einen Schrägstrich bzw. Slash (/) oder Backslash (\) getrennt werden.

Info

Namen eines Namensdienstes gehorchen den jeweiligen Konventionen über zulässige Zeichen, Leserichtung etc.

Auch die Leserichtung der Namen ist von Service zu Service unterschiedlich. Während der URL *java.sun.com* von rechts nach links aufgelöst wird, entschlüsseln Sie den Pfad der Datei */home/java/jndi* von links nach rechts. Auch Mischformen zwischen beiden Konventionen sind möglich, wobei diese dann von unterschiedlichen Services aufgelöst werden.

Listing 7.1
Ein zusammengesetzter Name verschiedener Dienste

Die folgende Zeichenkette zeigt beispielsweise eine Mischform zwischen Domain Name System (DNS) und Dateisystem:

```
java.sun.com/products/jndi/index.jsp
```

Info

Tim Berners-Lee hat in einem Interview mit Impact Labs übrigens einmal gestanden, dass er, wenn er das WWW noch einmal erfinden würde, auf den einleitenden doppelten Schrägstrich verzichten und die Leserichtung umkehren würde.

Die einzelnen Bestandteile des Namens werden nach folgenden Kriterien unterschieden:

- Atomic Name

  Der *Atomic Name* ist der *elementare* Name eines Objekts in einem bestimmten Kontext. Dieser Name ist nicht weiter zerlegbar. Die Zeichenkette `sun` bildet im obigen Beispiel einen solchen Atomic Name.

- Compound Name

  Als *Compound Name* (engl. zusammengesetzter Name) bezeichnet man einen aus Atomic Names zusammengesetzten hierarchischen Namen eines Objekts. Dieser Name muss den Namenskonventionen des jeweiligen Namensdienstes folgen. `java.sun.com` bzw. `/products/jndi/index.jsp` sind Beispiele für Compound Names.

- Composite Name

  Ein *Composite Name* (engl. zusammengefasster Name) ist schließlich das Resultat verschiedener Compound Names, z.B. `java.sun.com/products/jndi/index.jsp`. Ein Composite Name besteht aus den Compound Names unterschiedlicher Namensdienste und wird von verschiedenen Service Providern gemeinsam aufgelöst.

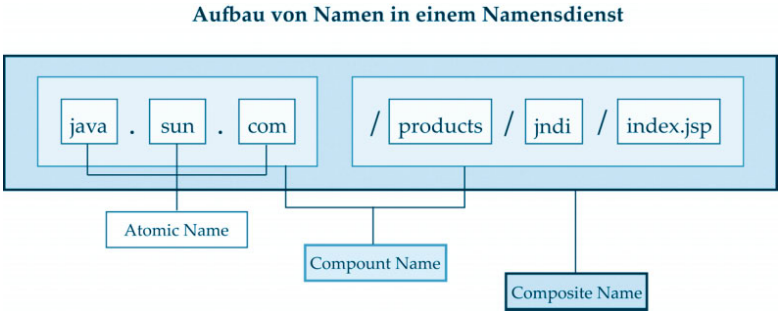


Abbildung 7.3
Aufbau von Namen

Namenskonventionen

Sun beschreibt in der Java-EE-Spezifikation auch eine Namenskonvention, unter der die verschiedenen Dienste angeboten werden sollen. Danach sollen die Namen der Objekte, die über die JNDI referenziert werden können, mit folgenden Präfixen (die wir später als Sub-Kontexte identifizieren werden) beginnen:

- `java:comp/env/jdbc`
  Datenbankverbindungen, die über JNDI als `javax.sql.DataSource` verfügbar gemacht werden, sollen unter diesem Kürzel abgelegt werden.
- `java:comp/env/url`
  Dieses Kürzel dient der Ablage von `java.net.URL`-Objekten.
- `java:comp/env/jms`
  Den Java Message Service (JMS) lernen Sie in Kapitel 9 kennen.
- `java:comp/env/mail`
  Auch der JavaMail-Service bekommt einen eigenen Kontext.
- `java:comp/env/eis`
  Enterprise Information Systems (EIS) haben (wie der Name es sagt) die Aufgabe, Informationen bereitzustellen. Dies können z.B. Kundendaten für eBusiness-Anwendungen oder Wissensmanagement-Systeme für eLearnig-Plattformen sein.
- `java:comp/env/eis/JAXR`
  Hinter dem Kürzel JAXR verbirgt sich das Java API for XML Registry. Dieses ermöglicht das Auffinden und den Zugriff auf die Universal Description Discovery and Integration (UDDI)-Services. Mehr hierzu erfahren Sie im Kapitel über Webservices.

Beispiel: der Domain Name Service (DNS)

Der wohl meistbenutzte Namensdienst ist der schon eingangs erwähnte *Domain Name Service (DNS)* des Internets, der die fantasievollen Domain-Namen auf feste IP-Adressen abbildet, die von Gremien wie dem DENIC (*www.denic.de*) verwaltet werden.

Info

Das Deutsche Network Information Center (DENIC) ist für die Registrierung und Verwaltung von Domains zuständig, die sich innerhalb der Top-Level-Domain .de befinden, also z.B. *google.de*.

Nun können sich die meisten Menschen Namen deutlich einfacher merken als scheinbar willkürliche Zahlenkolonnen und auch die Werbung hätte wesentlich mehr Schwierigkeiten, statt *eBay* die Zeichenkette *66.135.192.71* zu vermarkten. Wie Sie sehen, sind *Namensdienste* mittlerweile so alltäglich für uns geworden, dass wir ihre Verwendung kaum noch bemerken, aber vielleicht denken Sie bei der Erstellung ihrer nächsten Java-Datei ja daran, dass sich dahinter, genau genommen, nichts weiter als ein auf bestimmte Weise magnetisierter Bereich Ihrer Festplatte verbirgt, den Sie anschließend mit einer Bezeichnung assoziieren.

7.1.4 Verzeichnisdienste

Nachdem sich Namensdienste einfach aus der Neigung des Menschen entwickelt haben, Objekte oder Referenzen auf Objekte unter logisch aufgebauten Namen abzulegen, wollte man diesen Einträgen auch bald bestimmte Attribute zuweisen, um sie genauer zu spezifizieren oder nach ihren Eigenschaften gruppieren zu können. Zu diesem Zweck wurden die Namensdienste zu Verzeichnisdiensten (Directory Service) erweitert.

Ein klassisches Beispiel für einen Directory Service ist das Telefonbuch. In diesem Verzeichnis sind Personen – die Verzeichnisobjekte – anhand ihres Nachnamens aufgelistet. Zusätzlich sind diesen Personen dann die Attribute Straße (optionales Attribut) und Telefonnummer zugeordnet, wobei jedem der Attribute auch mehrere Werte zugeordnet sein können.

Verzeichnisdienst vs. Datenbank

Wenn Sie bereits mit Datenbanken vertraut sind, würden Sie das städtische Telefonbuch vielleicht auch eher mit einem relationalen Datenbanksystem (RDBMS) als mit einem Verzeichnisdienst assoziieren und tatsächlich gibt es eine Reihe von Analogien zwischen beiden Technologien, wie beispielsweise Transaktionsunterstützung, Skalierbarkeit und Multi-User-Zugriffe. Daneben existieren jedoch, wie die nachfolgende Tabelle zeigt, auch eine Reihe markanter Unterschiede:

Tabelle 7.1
Vergleich zwischen Datenbank und Verzeichnisdienst

	RDBMS	Verzeichnisdienst
Abbildung der Daten	In Tabellen und Schemata	In hierarchischen Knoten
Art des Zugriffs	Lesende und schreibende Zugriffe	Überwiegend lesende Zugriffe

	RDBMS	Verzeichnisdienst
Such-Kriterien	Nach fest definierten Attributen	Nach beliebigen Attributen
Häufigkeit der Attributwerte	Existiert zu jedem Datensatz genau einmal oder ist undefiniert (null)	Jedem Attribut können beliebig viele Werte zugeordnet sein.
Beziehungen zwischen Datensätzen	Können über einen JOIN ermittelt werden	Es können lediglich Aussagen über die Hierarchie gemacht werden.
Charakteristika	Komplexe Anfragen, lange Verbindungen mit dem Client	Triviale Anfragen (Existenz), kurze Client-Verbindungen
Typische Anwendungen	Datenspeicher für Anwendungen	Verzeichnisse (Recherche)

Tabelle 7.1 (Forts.)
Vergleich zwischen Datenbank und Verzeichnisdienst

Info

Verzeichnisdienste können auch als hierarchische Datenbanken aufgefasst werden.

Sie können sich einen Verzeichnisdienst also durchaus auch als eine über das Netzwerk verteilte, hierarchische Datenbank vorstellen, die auf einem Client-Server-Prinzip basiert. Und tatsächlich existieren mit *IBM Research* oder *Open LDAP* auch LDAP-Implementierungen, die auf einem RDBMS basieren.

LDAP – ein Verzeichnisdienst

Es existieren eine große Anzahl von Verzeichnisdiensten die für unterschiedlichste Aufgaben vorgesehen sind. Zu den am weitesten verbreiteten gehören:

- *Active Directory* (ADS) in Windows-2000-Netzwerken
- *Novell Directory Services* (NDS) in Novell-Netzwerken
- *Network Information Service* (NIS) auf Unix-Systemen
- *Lightweight Directory Access Protocol* (LDAP) als offener Standard mit vielen Implementierungen

Dabei war auch LDAP zunächst nur ein Protokoll, das 1995 an der Universität Michigan entwickelt wurde, um auf Daten im X.500-Standard zugreifen zu können. Der X.500-Standard beschreibt dabei den grundsätzlichen Aufbau eines Verzeichnisdienstes über den gesamten *ISO/OSI Stack*. Dies machte die Implementierung des Dienstes sehr aufwändig.

Info

Das *OSI-Modell (Open Systems Interconnection Reference Model)* ist ein in den 70er Jahren entwickeltes Modell, das den Aufbau von Netzwerkprotokollen (TCP/IP, AppleTalk) beschreibt. Jede zu bewältigende Aufgabe wird dabei einer einzelnen, auf den anderen Layern aufbauenden, Schicht übertragen. Die einzelnen Schichten befassen sich dabei (von unten nach oben) mit der physischen Übertragung (Physical Layer), der Fehlerkorrektur (Data Link Layer), der Netzwerkkommunikation (Network Layer), dem Datentransport (Transport Layer), der Sitzung (Session Layer), der Darstellung (Presentation Layer) und schließlich der darauf aufbauenden Anwendung (Application Layer). Und wer sich die Reihenfolge partout nicht merken kann, dem hilft vielleicht der Merksatz: »*Please do not throw salami pizza away.*«

Das LDAP-Protokoll verzichtete auf die Beschreibung der Transport- und Netzwerkschichten und verwendete dafür das TCP/IP-Protokoll, daher auch der Name Lightweight (dt. leichtgewichtig). Während vollwertige *Directory Access Protocols (DAP)* Mechanismen zur Internationalisierung, eine vollständige Benutzerauthentifizierung und ein Verzeichnis-Management implementieren, unterstützt LDAP im Wesentlichen nur den Zugriff auf Verzeichnisse mit diesem Standard und emuliert die anderen Funktionen lediglich. Mit anderen Worten, der LDAP-Standard erfand das Rad nicht von Grund auf neu, sondern konzentrierte sich auf die Hauptaufgaben.

Wie spannend dieses Thema ist, lässt sich auch an der Anzahl der entsprechenden Publikationen erkennen. So finden Sie mehr zum Thema LDAP unter anderem in den RFCs 1777, 1823 und 1959.

7.2 API und SPI – Download und Installation

Sie haben bereits gelernt, dass die JNDI-Spezifikation nur einen Teil der erforderlichen Software darstellt, der um die zum jeweiligen Speichermedium gehörende Service-Implementierung ergänzt werden muss. Dieser Abschnitt befasst sich kurz mit den wichtigsten Bezugquellen und der Installation und gibt einen kurzen Überblick über die wichtigsten SPI-Implementierungen.

7.2.1 Download des API

Die für das *Java Naming and Directory Interface* benötigten APIs sind kein Bestandteil des Standard-Java-EE-Pakets, das von Sun zum Download angeboten wird. Dies liegt daran, dass Sun JNDI als ein eigenes Produkt

und eine Kerntechnologie (*Core Java*) sieht, die eher im J2SE-Umfeld angesiedelt ist und deren Kommunikation mit Java Enterprise-Services ermöglicht.

Sie können die API-Spezifikation und deren Dokumentation sowie ein umfangreiches Tutorial von Suns Produktseite unter *http://java.sun.com/products/jndi/* herunterladen. Anschließend entpacken Sie die ZIP-Dateien in ein Verzeichnis Ihrer Wahl, binden die JAR-Bibliotheken in den *Classpath*, fertig.

Dass Sun JNDI als eigenständiges Produkt, außerhalb der Standard (J2SE) oder Enterprise Java Edition, ansieht, können Sie auch anhand der eigenen Versionsnummern erkennen, die die einzelnen JNDI-Releases unterscheiden. Die derzeit aktuelle Version, auf der die Beispiele in diesem Kapitel aufbauen, trägt die Bezeichnung JNDI 1.2.1 und wird für alle gängigen Java-Versionen als *Booster Pack* angeboten. Da sich die API zwischen den Releases 1.1 und 1.2 jedoch etwas verändert hat, sollten Sie zuvor überprüfen, welche Version der Anbieter Ihres Dienstes unterstützt, und diese ebenfalls einsetzen.

Info

JNDI ist kein Bestandteil des Java EE Software Development Kit, sondern wird als separates Produkt vertrieben.

7.2.2 Einige SPI-Implementierungen

Neben der API stellt Ihnen Sun auf der oben genannten Webseite auch einige kostenfreie SPI-Implementierungen zum Download bereit. Folgende Provider stehen dabei zur Auswahl.

Lightweight Directory Access Protocol (LDAP)

Der Klassiker unter den Verzeichnisdiensten. Suns Implementierung dieser Schnittstelle wird sowohl für die JNDI-Version 1.1 als auch 1.2 angeboten und unterstützt die LDAP-Versionen 2 und 3.

Domain Name System (DNS)

Und auch für die Königin unter den Namensdiensten stellt Sun eine Implementierung bereit. Damit können Sie direkt mit dem Domain Name Service des Internets in Verbindung treten, um etwa eine DNS-Anfrage abzusenden.

Corba Naming Service (COS)

Da CORBA-Schnittstellen im Wesentlichen dazu dienen, auf entfernte Objekte beliebiger, objektorientierter Programmiersprachen zuzugreifen, wird dieser Standard der *Open Management Group (OMG)* vor allem bei RPC-Diensten interessant.

Info

Hinter der *Open Management Group (OMG)* verbirgt sich ein 1989 gegründetes Konsortium, das sich mit Standards für systemunabhängige objektorientierte Programmierung beschäftigt. Zu den über 800 Mitgliedern zählen unter anderem Firmen wie *IBM*, *Apple* und *Sun*.

Remote Method Invocation Registry (RMI Registry)

Einst Suns Antwort auf CORBA, nähern sich beide Dienste hinsichtlich ihrer Übertragungsprotokolle immer mehr an. Im Gegensatz zu CORBA unterstützt RMI auch lieb gewonnene Funktionen wie beispielsweise den *Garbage Collector*. Dafür bleibt die Kommunikation ausschließlich auf Java-Objekte beschränkt.

Network Information System (NIS)

Von Sun ursprünglich unter dem Namen »Yellow Pages« (YP) entwickelt, dient dieser Verzeichnisdienst vor allem der Verwaltung von Benutzerkonten, Rechnernamen und anderen in einem Computernetzwerk benötigten Daten.

Da der Begriff »Yellow Pages« (dt. Gelbe Seiten) jedoch sowohl in Großbritannien als auch in Deutschland marktrechtlich geschützt ist, änderte Sun ihn in NIS.

Directory Service Markup Language (DSML)

Als jüngstes Kind in Suns Directory-Service-Implementierungen versucht dieser von vielen kommerziellen Anbietern (u.a. Netscape, IBM, Oracle und Sun) vorangetriebene, offene Standard, Verzeichnisdienste über XML-Strukturen abzubilden. Er soll dabei den Datenaustausch zwischen Server-Systemen ermöglichen und damit einfachere eBusiness- und eCommerce-Anwendungen ermöglichen.

Ein Dateisystem (File System)

Am Anfang des Kapitels haben Sie gelernt, dass auch Ihr Dateisystem im Prinzip nichts anderes als einen Verzeichnisdienst darstellt und Sun selbstverständlich eine entsprechende Service-Implementierung anbietet – obwohl Sie Ihre Dateien natürlich auch weiterhin über die Klassen des Package `java.io` bzw. `java.nio` verwalten können.

Der Vorteil dieses Dienstes ist, dass er praktisch auf jedem PC verfügbar ist und sich damit hervorragend eignet, um eigene Erfahrungen im Umgang mit Naming Services zu sammeln. Sie werden ihn später für Ihre ersten Gehversuche mit dem JNDI verwenden.

Novell Directory Service (NDS)

Wenn Sie vorhaben, mit Ihrem Verzeichnisdienst auf ein großes Unternehmensnetzwerk zuzugreifen, und dieses keinen LDAP-Service verwendet, dann haben Sie es mit großer Wahrscheinlichkeit mit diesem Vertreter zu tun. Neben den klassischen Merkmalen eines Verzeichnisdienstes stattet Novell diesen Ableger mit zusätzlichen Features wie *der redundanten Ablage* von Informationen und der Möglichkeit zur Replikation aus.

Info

Unter *Redundanz* versteht man das mehrfache (überflüssige) Ablegen oder Übermitteln der gleichen Information. Diese dient in der Nachrichtenübertragung der Fehlerkorrektur und ermöglicht es Nachrichtendiensten, auch bei Ausfall eines Servers, den Service über andere Kanäle aufrechtzuerhalten. Um die Änderungen an einer Kopie der redundanten Information auf alle anderen zu übertragen und das System damit in einem konsistenten (widerspruchsfreien) Zustand zu halten, muss die Aktualisierung an alle Kopien übermittelt werden. Diesen Vorgang bezeichnet man als *Replikation*.

Die SPI-Implementierung kann kostenlos von Novells Website unter *http://developer.novell.com/ndk* bezogen werden.

Windows Registry

Und auch hinter der *Windows Registry* verbirgt sich nichts anderes als ein weiterer Verzeichnisdienst, der es den verschiedenen Applikationen ermöglicht, ihre Konfigurationsdaten zentral zu verwalten, ohne auf Dateien zurückgreifen zu müssen. Die Content Logic Corporation bietet unter *http://www.contentlogic.com* eine kommerzielle Implementierung dieses SPI an.

7.3 Arbeiten mit dem JNDI

Achtung

In diesem Abschnitt werden Sie sich die Grundlagen erarbeiten und lernen, mit der API auf einen Namensdienst zuzugreifen. Um die entscheidenden Stellen besser herauszuarbeiten, verzichten wir dabei auf vollständige Java-Klassen mit `main()`-Methoden und arbeiten stattdessen mit Listing-Fragmenten, die Sie anschließend in Ihre Java-Klassen einfügen können. Auf der CD finden Sie die Listings in vollständige Klassen eingebettet.

Da es durch das JNDI ein Leichtes ist, auf bereits bestehende Namens- und Verzeichnisdienste zuzugreifen und Objekte in diesen abzulegen oder zu extrahieren, gibt es wohl kaum noch Java-EE-Applikationen, die sich ausschließlich damit beschäftigen. Vielmehr ermöglicht es die API Ihnen, flexible Verbindungen zwischen Ihren Komponenten zu schaffen und diese dadurch zu entkoppeln.

7.3.1 Erzeugen eines initialen Kontexts

Zuallererst benötigen Sie immer einen Einstiegspunkt in das System des Services, von dem aus Sie anschließend weiter navigieren können. Beispiele für einen solchen Einstiegspunkt, so genannte *initiale Kontexte* sind beispielsweise das *home*-Verzeichnis eines Benutzers unter Linux oder der Ordner *C:* unter Windows-Betriebssystemen.

Der initiale Kontext für die Standardumgebung

Um den initialen Kontext der Standardumgebung zu erzeugen, genügt es, den leeren Konstruktor der Klasse `javax.naming.InitialContext` aufzurufen, wie es das folgende Listing zeigt.

Listing 7.2 Initialisieren des Standardkontexts (TemplateListing_1.java)

```
// Importieren der erforderlichen Klassen der JNDI-API
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
...
   try {
      // Erzeugen eines Initialen Kontexts für die Standardumgebung
      Context ctx = new InitialContext();

      // ...
      // Hier können Sie mit dem Dienst arbeiten und die in
      // Abschnitt 7.3.2ff beschriebenen Methoden verwenden.
      // ...

      // Am Ende müssen Sie die Verbindung wieder schließen,
      // um belegte Ressourcen freizugeben.
      ctx.close();

   } catch (NamingException nex) {
      nex.printStackTrace();
   }
...
```

Die Standardumgebung wird beispielsweise häufig durch den verwendeten Application-Server (z.B. *JBoss*) initialisiert und ermöglicht Ihnen Zugriff auf dessen Namensdienst.

Anpassen der Standardumgebung über die Kommandozeile

Sie können die jeweilige SPI-Implementierung, die als Standardumgebung verwendet wird, auch über einen Parameter bei Start der Applikation festlegen. Sie steuern das Verhalten über die Kommandozeilen-Option `-D`.

Listing 7.3 Setzt LDAP als Standardumgebung

```
// Setzen der Standard JNDI-Umgebung per Kommandozeile
java -Djava.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFac-
tory AuszuführendeJavaKlasse
```

Achtung

Natürlich muss die gewählte Klasse auch über den Classpath eingebunden sein. Anderenfalls erzeugt die JVM eine `ClassNotFoundException`.

Anpassen der Standardumgebung zur Laufzeit

Sie können die zu verwendende SPI-Implementierung natürlich auch erst zur Laufzeit bestimmen, indem Sie den Konstruktor des initialen Kontexts über eine `Hashtable` konfigurieren.

Listing 7.4
Konfiguration von Suns LDAP-Services zur Laufzeit (TemplateListing_2.java)

```
// Importieren der erforderlichen Klassen
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
...
   try {
      // Erzeugen einer neuen Hashtable zur Konfiguration
      Hashtable<String,String> env = new Hashtable<String,String>();

      // Spezifizieren des zu verwendenden JNDI Services
      // hier Suns LDAP Service Implementation
      env.put(Context.INITIAL_CONTEXT_FACTORY,
              "com.sun.jndi.ldap.LdapCtxFactory");

      // Spezifizieren des zu verwendenden Provider URLs
      env.put(Context.PROVIDER_URL, "ldap://localhost:389");

      // Erzeugen des Initialen Kontextes mit Parametern
      Context ctx = new InitialContext(env);


      // Hier können Sie auf den Dienst zugreifen (Siehe 7.3.2) ...


      // Freigeben der Ressourcen und Schließen der Verbindung
      ctx.close();
   } catch (NamingException ne) {
      ne.printStackTrace();
   }
...
```

Konfiguration des Dienstes

Wie Sie in obigem Listing sehen, kann der zu verwendende Namens- oder Verzeichnisdienst über eine `Hashtable` konfiguriert werden. Hierfür definiert Ihnen das Interface `javax.naming.Context` eine Reihe von Konstanten.

Tabelle 7.2
Konstanten zur Konfiguration des initialen Kontexts

Schlüssel	Verwendung
`INITIAL_CONTEXT_FACTORY`	Schnittstelle zum SPI. Mit diesem Schlüssel geben Sie die Klasse an, die den Service implementiert.
`PROVIDER_URL`	URL, unter dem der Service erreichbar ist

Tabelle 7.2 (Forts.) Konstanten zur Konfiguration des initialen Kontexts

Schlüssel	Verwendung
URL_PKG_PREFIXES	Liste der Präfixe der Java-Package, die nach einer Implementierung durchsucht werden. Der Standardwert com.sun.jndi.url bleibt stets erhalten und wird hinten angefügt
LANGUAGE	Eine nach RFC 1766 kodierte Zeichenkette, mit der Sie die von Ihnen bevorzugte Sprache einstellen.
SECURITY_PROTOCOL	Typ des Verschlüsselungsprotokolls, zum Beispiel »SSL«
SECURITY_AUTHENTICATION	Gibt die Art des Authentifizierungsschemas an. Standard: simple.
SECURITY_PRINCIPAL	Benutzername ...
SECURITY_CREDENTIALS	... und das zugehörige Passwort

Machen Sie sich keine Sorgen, wenn Ihnen einige der Schlüssel und ihre Funktionen nicht vertraut sind: In aller Regel werden Namens- und Verzeichnisdienste von Systemadministratoren vorkonfiguriert und besitzen eine dokumentierte Konfiguration. Außer zum Testen der Beispiele in diesem Buch werden Sie wahrscheinlich keinen Dienst selbst konfigurieren müssen. Die wichtigsten Schlüssel sind INITIAL_CONTEXT_FACTORY, mit dem Sie den zu verwendenden Dienst angeben, und PROVIDER_URL, der die Adresse des Service enthält.

Info

Je nach verwendetem Service stehen die einzelnen Konstanten zur Verfügung oder auch nicht. So macht die Angabe von Benutzername und Passwort beim Zugriff auf das Dateisystem beispielsweise wenig Sinn.

Umgebung für den JBoss

Ein wichtiges Anwendungsgebiet für Namensdienste stellen Application Server dar, die diese nutzen, um die verschiedenen bereitgestellten Services miteinander zu verknüpfen. Sollten Sie die folgenden Beispiele mit dem JBoss Application Server testen wollen, können Sie mit der in Listing 7.5 angegebenen Konfiguration arbeiten:

Listing 7.5 Konfiguration der Umgebung für den JBoss Application Server (TemplateListing_3.java)

```
// Importieren der erforderlichen Klassen
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
...
   try {
      // Konfigurieren des Service-Providers (JBoss)
      Hashtable<String,String> env = new Hashtable<String,String>();
```

```
        // Verwenden der JBoss JNDI Factory
        env.put(Context.INITIAL_CONTEXT_FACTORY,
                "org.jnp.interfaces.NamingContextFactory");

        // URL des Namensdienstes (JBoss-Standard-Port: 1099)
        env.put(Context.PROVIDER_URL, "localhost:1099");

        // Konfiguration der URL Package Präfixe
        env.put(Context.URL_PKG_PREFIXES,
                "org.jboss.naming:org.jnp.interfaces");

        // Erzeugen des Initialen Kontexts
        Context ctx = new InitialContext(env);


        // Hier können Sie auf den Dienst zugreifen (Siehe 7.3.2) ...


        // Freigeben der Ressourcen und Schließen der Verbindung
        ctx.close();
      } catch (NamingException nex) {
        nex.printStackTrace();
      }
...
```

Listing 7.5 (Forts.)
Konfiguration der Umgebung für den JBoss Application Server (TemplateListing_3.java)

Anschließend müssen Sie die mit der JBoss-Distribution ausgelieferte Datei *jbossall-client.jar* in den Classpath einbinden. Sie enthält unter anderem die oben angegebenen Klassen der Service-Provider-Implementierung.

7.3.2 Auf den Dienst zugreifen ...

Nachdem Sie das initiale `Context`-Objekt, den Einstiegspunkt für den Namensdienst, erzeugt haben, können Sie mit diesem auf den Dienst zugreifen und in diesem navigieren. Hierfür stellt Ihnen das Interface folgende Basismethoden bereit, deren Verwendung Ihnen die nächsten Abschnitte näher bringen:

Tabelle 7.3
Basisoperationen eines Namensdienstes

Operation	Beschreibung
`bind()`	Unter dem Binden (engl. *bind*) versteht man das Ablegen eines Objekts im aktuellen Kontext.
`unbind()`	Diese Methode ist das Pendant zu `bind` und löscht ein Objekt aus dem Kontext.
`rebind()`	Verwenden Sie einen bereits vorhandenen Namen, um ein Objekt über `bind` abzulegen, wird das alte Objekt nicht automatisch gelöscht. Stattdessen wirft Java zur Sicherheit eine entsprechende `Exception`. `rebind()` unterbindet dies, indem es das alte Objekt überschreibt.
`createSubcontext()` und `destroySubcontext()`	Mit diesen Operationen erzeugen und entfernen Sie neue Subkontexte. Diese Operationen sind vergleichbar mit dem Anlegen und Löschen neuer Verzeichnisse im Dateisystem.

Tabelle 7.3 (Forts.) Basisoperationen eines Namensdienstes

Operation	Beschreibung
`list()`	Diese Operation listet alle gebundenen Objekte (inklusive Subkontexte) des aktuellen Kontexts auf. Sie entspricht damit den Befehlen `dir` (Windows) bzw. `ls` (Linux).
`listBindings()`	Diese Methode liefert im Gegensatz zu `list()` gleichzeitig auch Referenzen auf das gebundene Objekt.
`lookup()`	Das ist die vielleicht interessanteste Methode: Sie liefert Ihnen nämlich ein gebundenes Objekt zurück. Auch Subkontexte werden so gefunden.
`rename()`	Bindet ein bereits gebundenes Objekt unter einem neuen Namen.
`close()`	Beendet den Zugriff auf den Namens- und Verzeichnisdienst. Jeden Versuch, nach dem Aufruf dieser Methode eine weitere Operation auf dem aktuellen Kontext auszuführen, quittiert Java mit einer Fehlermeldung.

Das sind die Basisoperationen jedes Kontexts und des dahinter liegenden Namensdienstes.

Info

Alle Operationen erzeugen bei Laufzeitfehlern eine `javax.naming.NamingException` (bzw. eine Subklasse hiervon), so dass Sie diese Fehler global und gemeinsam abfangen können.

7.3.3 Ausgabe der Elemente eines Kontexts

Als Erstes werden Sie alle Elemente eines `Context`-Objekts ausgeben. Dazu verwenden Sie die oben beschriebene Methode `list()`.

Info

Dieses Beispiel verwendet zu Demonstrationszwecken und, weil nicht jeder Leser über einen LDAP-Verzeichnisdienst verfügt, *Suns File System*-Implementierung (`com.sun.jndi.fscontext.RefFSContextFactory`). Sie können jedoch auch jede andere Implementierung verwenden. Suns File System-Implementierung bietet dabei einen auf JNDI basierenden Zugriff auf das lokale Dateisystem, bei dem Ordner Kontexte darstellen. Die ausgeführten Operationen können auf diese Weise leicht nachvollzogen werden.

Listing 7.6
Auflisten der Elemente im InitialContext (ListContext.java)

```
package de.akdabas.javaee.jndi;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NameClassPair;
import javax.naming.NamingEnumeration;
import javax.naming.NamingException;

/** Listet alle an einen Context gebundenen Elemente auf */
public class ListContext {

   /** Listet die Elemente des InitialContext auf */
   public static void main(String[] args) {
      try {
         // JNDI - Konfiguration
         Hashtable<String, String> env =
                 new Hashtable<String, String>();

         // Konfigurieren des Service-Providers (File System)
         env.put(Context.INITIAL_CONTEXT_FACTORY,
                 "com.sun.jndi.fscontext.RefFSContextFactory");

         // Erzeugen des Initialen Kontexts
         Context ctx = new InitialContext(env);

         // Auslesen des Inhaltes des aktuellen Kontextes
         NamingEnumeration list = ctx.list(".");

         // Auflisten der im Kontext gebundenen Elemente
         while (list.hasMore()) {
            NameClassPair ncp = (NameClassPair) list.next();
            System.out.println(ncp);
         }

         // Freigeben der Ressourcen und Schließen der Verbindung
         ctx.close();
      } catch (NamingException nex) {
         nex.printStackTrace();
      }
   }
}
```

Nachdem Sie das `InitialContext`-Objekt erzeugt haben, können Sie über die Methode `list()` dessen Inhalt auslesen.

Listing 7.7
Auslesen des Inhalts eines Context-Objekts

```
...
   // Auflisten der gebundenen Elemente eines Kontextes
   javax.naming.NamingEnumeration list = ctx.list(".");
...
```

Die Klasse `NamingEnumeration` ist dabei eine Subklasse von `java.util.Enumeration`, die beim Lesen eines Kontext aufgetretene Fehler zunächst zurückhält und erst dann weitergibt, wenn alle Elemente der *Enumeration* durchlaufen wurden und die Methode `hasMore()` den Wert `false` zurückliefert. Die Elemente der zurückgegebenen Enumeration sind dabei vom Typ `javax.naming.NameClassPair`, der den Namen und die Klasse des gebundenen Objekts enthält.

Achtung

Das Beispiel zeigt Ihnen noch eine weitere Analogie zwischen JNDI und einer Datenbank: Nachdem Sie mit dem Service gearbeitet haben und die geöffnete Verbindung nicht mehr benötigen, sollten Sie diese über die Methode `close()` auch wieder schließen, um wichtige Ressourcen freizugeben.

Ausführen der Klasse

Da sich die Beispiele in diesem Kapitel alle von der Kommandozeile ausführen lassen, können Sie die benötigten JAR-Bibliotheken manuell angeben oder in Ihren Classpath einbinden. Um das obige Listing, nachdem Sie es erfolgreich mit dem Ant-Script übersetzt haben, starten zu können, wechseln Sie in das Verzeichnis *classes* und verwenden Sie am einfachsten folgenden Aufruf:

Listing 7.8 Aufruf des Beispiels

```
java -cp ../lib/jndi.jar;../lib/fscontext.jar;../lib/providerutil.jar; de.akdabas.javaee.jndi.ListContext
```

Die Bibliotheken *jndi.jar* und *providerutil.jar* enthalten dabei das API bzw. SPI, während die Implementierung in der Datei *fscontext.jar* steckt.

Tipp

Listing 7.8 zeigt demonstriert den Einsatz unter Windows-Systemen. Für Unix/Linux-Systeme tauschen Sie bitte das Semikolon im *Classpath* (`-cp`) gegen Doppelpunkte aus.

Das Resultat

Der initiale Kontext des File-System-Dienstes ist übrigens in der Regel das Laufwerks-Stammverzeichnis (*C:*, *D:*) unter Windows bzw. *home* unter Linux und da die Methode `list()` in diesem Service dem Befehl `dir` bzw. `ls` entspricht, erhalten Sie beispielsweise folgende Ausgabe:

Listing 7.9 Ausgabe des initialen Kontextes

```
Programme: javax.naming.Context
ReadMe.txt: java.io.File
Recycled: javax.naming.Context
Projekte: javax.naming.Context
Music: javax.naming.Context
```

Wie Sie sehen listet das obige Beispiel sowohl Dateien als auch enthaltenen Ordner auf, die dann wiederum Subkontexte darstellen.

7.3.4 Zugreifen auf gebundene Elemente

Die im vorherigen Beispiel verwendete `NamingEnumeration` ist vor allem dazu gedacht, den *Inhalt* eines Context-Objekts auszulesen. Die einzelnen Elemente vom Typ `NameClassPair` enthalten dabei zwar Informationen über das gebundene Objekt, jedoch *keine* Referenz auf das gebundene Objekt selbst. Dazu dient die Methode `listBindings()`.

Die Klasse

Die Klasse zum gleichzeitigen Referenzieren aller gebundenen Objekte ähnelt stark dem vorherigen Beispiel – nur dass Sie jetzt die Methode `listBinding()` aufrufen, die Ihnen eine *Enumeration* von `Binding`-Objekten zurückliefert.

Listing 7.10
Ausgabe der Elemente im InitialContext (ListBindings.java)

```
package de.akdabas.javaee.jndi;
import java.util.Hashtable;
import javax.naming.Binding;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingEnumeration;
import javax.naming.NamingException;

/** Gibt alle an einen Context gebundenen Elemente aus */
public class ListBindings {

   /** Gibt die Elemente eines Kontextes aus */
   public static void main(String[] args) {
      try {
         // JNDI - Konfiguration
         Hashtable<String, String> env =
                 new Hashtable<String, String>();

         // Konfigurieren des Service-Providers (File System)
         env.put(Context.INITIAL_CONTEXT_FACTORY,
                 "com.sun.jndi.fscontext.RefFSContextFactory");

         // Erzeugen des Initialen Kontexts
         Context ctx = new InitialContext(env);

         // Auslesen aller Elemente eines Kontextes
         NamingEnumeration list = ctx.listBindings(".");

         while (list.hasMore()) {
            Binding binding = (Binding) list.next();
            System.out.println(binding.getName() +
                               " : " + binding.getObject());
         }

         // Freigeben der Ressourcen und Schließen der Verbindung
         ctx.close();
      } catch (NamingException nex) {
         nex.printStackTrace();
      }
   }
}
```

Der Unterschied mag Ihnen akademisch vorkommen, zumal die hier für die Objekte verwendete Klasse `Binding` eine Subklasse von `NameClassPair` ist. Doch genau dieser kleine Unterschied erlaubt es Ihnen nun, über die `Binding`-Methode `getObject()` auch auf die einzelnen Objekte des Binding zuzugreifen.

Listing 7.11
Referenzieren eines Objekts

```
...
   // Zugriff auf die in einem Kontext gebundenen Objekte
   System.out.println(binding.getName() +":"+ binding.getObject());
...
```

Das Resultat

Nachdem Sie auch dieses Beispiel erfolgreich übersetzt und mit einem dem Klassennamen entsprechend abgewandelten Aufruf gestartet haben (Listing 7.8), erhalten Sie die folgende Ausgabe:

Listing 7.12 Ausgabe von gebundenen Objekten

```
Programme : com.sun.jndi.fscontext.RefFSContext@863399
ReadMe.txt : D:\.\AdobeWeb.log
Recycled : jndi.fscontext.RefFSContext@5ac072
Projekte : com.sun.jndi.fscontext.RefFSContext@4a65e0
Music : com.sun.jndi.fscontext.RefFSContext@1cf8583
```

Wie Sie an den Hash-Werten und der Ausgabe der Datei deutlich erkennen können, handelt es sich jetzt um »echte« Objekte innerhalb der Java Virtual Machine (VM), mit der Sie auf Wunsch sofort weiterarbeiten können.

7.3.5 Umbenennen von Objektbindungen

Das nächste Listing zeigt Ihnen, wie Sie bereits gebundene Objekte umbenennen.

Listing 7.13 Umbenennen von Referenzen (Rename.java)

```
package de.akdabas.javaee.jndi;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/** Bindet eine Objekt unter einem neuen Namen */
public class Rename {

   /** Benennt die Datei 'ReadMe.txt' in 'LiesMich.txt' um */
   public static void main(String[] args) {
      try {
         // JNDI - Konfiguration
         Hashtable<String, String> env =
                 new Hashtable<String, String>();

         // Konfigurieren des Service-Providers (File System)
         env.put(Context.INITIAL_CONTEXT_FACTORY,
                 "com.sun.jndi.fscontext.RefFSContextFactory");

         // Erzeugen des Initialen Kontexts
         Context ctx = new InitialContext(env);

         // Umbenennen der Datei 'ReadMe.txt' in 'LiesMich.txt'
         ctx.rename("ReadMe.txt", "LiesMich.txt");

         // Freigeben der Ressourcen und Schließen der Verbindung
         ctx.close();
      } catch (NamingException nex) {
         nex.printStackTrace();
      }
   }
}
```

Dateioperationen leicht gemacht: Anschließend heißt die Datei *ReadMe.txt* nun *LiesMich.txt*.

Achtung

Damit Listing 7.13 funktioniert, muss eine entsprechende Datei *ReadMe.txt* natürlich zunächst im entsprechenden Kontext existieren.

7.3.6 Entfernen von Objekten

Und natürlich können Sie gebundene Objekte auch aus dem Kontext entfernen. Beim File System Provider entspricht das dem Löschen der Datei oder des Verzeichnisses.

Listing 7.14
Löschen einer Referenz (Unbind.java)

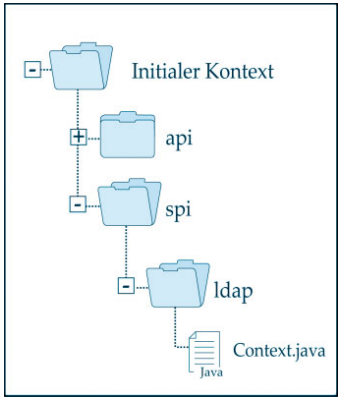
```
package de.akdabas.javaee.jndi;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/** Entfernt ein Objekt aus dem Kontext */
public class Unbind {

   /** Löscht die Datei 'LiesMich.txt' */
   public static void main(String[] args) {
      try {
         // JNDI - Konfiguration
         Hashtable<String, String> env =
                  new Hashtable<String, String>();

         // Konfigurieren des Service-Providers (File System)
         env.put(Context.INITIAL_CONTEXT_FACTORY,
                  "com.sun.jndi.fscontext.RefFSContextFactory");

         // Erzeugen des Initialen Kontexts
         Context ctx = new InitialContext(env);

         // Löschen der Datei 'LiesMich.txt'
         ctx.unbind("LiesMich.txt");

         // Freigeben der Ressourcen und Schließen der Verbindung
         ctx.close();
      } catch (NamingException nex) {
         nex.printStackTrace();
      }
   }
}
```

7.3.7 Verschieben von gebundenen Objekten

Bisher haben Sie sich stets im initialen Kontext bewegt und auf den dort gebundenen Objekten operiert. Dabei haben Sie auch gesehen, dass an das `InitialContext`-Objekt weitere Objekte des Typs `Context`, so genannte Subkontexte, gebunden sind. Sie könnten diese auch extrahieren, zum `Context` casten und entsprechend verwenden, doch es geht auch einfacher: Bestimmen Sie einfach einen *Pfad* durch die Kontexte.

Ein Beispiel: Nehmen wir einmal an, Sie befänden sich im Stammverzeichnis Ihres Laufwerks und hätten folgende Ordnerstruktur:

Abbildung 7.4
Eine Struktur im Dateisystem



Wenn Sie nun die Datei *Context.java* aus dem Verzeichnis *spi/ldap* in das Verzeichnis *api* bewegen wollen, führen Sie beispielsweise folgenden Befehl aus.

Listing 7.15
Kopieren einer Datei in Subkontexten

```
// Verschieben einer Datei unter Windows
move .\spi\ldap\Context.java .\api

// Pendant unter Linux
mv ./spi/ldap/Context.java ./api
```

Sie geben einfach einen Pfad in den gewünschten Kontext an. Dieser besteht aus einem so genannten Compound Name, der sich wiederum aus einzelnen Atomic Names zusammensetzt. Die Regeln, nach denen diese Namen aufgebaut werden, sind dabei von Service zu Service verschieden (siehe Abschnitt 7.1.3).

Verwenden von Pfaden

Auch bei der Verwendung des JNDI können Sie Pfade verwenden, um Objekte in Subkontexten zu referenzieren. Um das obige Beispiel mit dem File System Provider zu implementieren, erweitern Sie Listing 7.13 folgendermaßen:

Listing 7.16
Bewegen von Objekten in Subkontexten (Move.java)

```
package de.akdabas.javaee.jndi;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/** Verschiebt ein Objekt von einem Kontext in einen Anderen */
public class Move {

   /** Verschiebt die Datei analog zu Listing 7.15 */
   public static void main(String[] args) {
      try {
         // JNDI - Konfiguration
         Hashtable<String, String> env =
                 new Hashtable<String, String>();

         // Konfigurieren des Service-Providers (File System)
         env.put(Context.INITIAL_CONTEXT_FACTORY,
                 "com.sun.jndi.fscontext.RefFSContextFactory");
```

Listing 7.16 (Forts.)
Bewegen von Objekten in Subkontexten (Move.java)

```
            // Erzeugen des Initialen Kontexts
            Context ctx = new InitialContext(env);

            // Verschieben einer Datei von einem Kontext zum Nächsten
            ctx.rename("./spi/ldap/Context.java","./api/Context.java");

            // Freigeben der Ressourcen und Schließen der Verbindung
            ctx.close();
        } catch (NamingException nex) {
            nex.printStackTrace();
        }
    }
}
```

Das Verschieben (*move*) einer Datei bedeutet im File System Context also nichts anderes als das Umbenennen (*rename*) einer Referenz.

7.4 Speichern einer Datenbankverbindung

Nachdem Sie nun erste Erfahrungen mit dem `Context`-Objekt und seinen Methoden gesammelt haben, ist es an der Zeit, ein eigenes Objekt in einem Kontext abzulegen. In diesem Abschnitt werden Sie lernen, wie Sie eine Datenbankverbindung über das JNDI verfügbar machen. Dazu werden Sie die Verbindung zunächst erzeugen und in einem dafür vorgesehenen Kontext ablegen. Dieser Teil könnte beispielsweise von einem Service-Dienst Ihrer Applikation übernommen werden, um den Rest der Applikation von der Datenhaltung zu trennen und diese auch nachträglich anpassen zu können.

Im nachfolgenden Beispiel werden Sie diese Verbindung schließlich wieder aus dem Kontext extrahieren und für eine Datenbankanfrage verwenden. Dieser in Java-EE-Applikationen wesentlich häufigere Anwendungsfall zeigt Ihnen, wie wir einen Service ansprechen und die von ihm bereitgestellten Ressourcen in (Client-)Anwendungen verwenden. Doch zunächst müssen Sie Ihr Wissen um Datenbankverbindungen erweitern.

7.4.1 Vom DriverManager zur DataSource

Durch die Einführung des *DriverManager* (`java.sql.DriverManager`) vereinfachte Sun den Zugriff auf Datenbanken erheblich. Sofern der Hersteller seinerseits einen entsprechenden Treiber zur Verfügung stellte, konnte man ein einheitliches Interface dazu verwenden, die unterschiedlichsten Typen von Datenbanken anzusprechen. Kommt Ihnen das System bekannt vor?!

Der DriverManager ermöglichte es zwar, verschiedene Datenbanken auf unterschiedlichen Servern anzusprechen, jedoch war die Datenzugriffsschicht weiterhin an spezifischen Code gebunden. Um die Verbindung zu einer MySQL-Datenbank aufzubauen, verwenden Sie beispielsweise folgende Codefragmente:

Listing 7.17
Aufbau einer Datenbankverbindung (klassisch)

```
// Datenbank - Treiber via Classloader laden
Class.forName("com.mysql.jdbc.Driver");

// Verbindung mit der Datenbank herstellen
java.sql.Connection connection =
   java.sql.DriverManger.getConnection
      ("jdbc:mysql://localhost:3306/mySQL", "username", "password");
```

Nun ist der Driver-Manager aufgrund seiner Architektur nicht geeignet, über einen Kontext verfügbar gemacht zu werden. Daher erweiterte Sun seine SQL-Unterstützung um eine verallgemeinerte Datenquelle: `javax.sql.DataSource`. Eine Datenquelle ist im Wesentlichen eine verallgemeinerte Datenbank und erfüllt die gleichen Aufgaben wie ein `DriverManager`. Mit dem Unterschied, dass die `DataSource` explizit dafür entwickelt wurde, um über einen Namensdienst verfügbar gemacht zu werden. Das folgende Listing zeigt die Schritte, die notwendig sind, um eine Datenbankverbindung aufzubauen. Diese sind zum Teil herstellerabhängig und können von Implementierung zu Implementierung variieren.

Info

Objekte vom Typ `DataSource` können im Gegensatz zu Objekten von Typ `DriverManager` oder `Connection` über Namensdienste verfügbar gemacht werden.

Listing 7.18
Aufbau einer Datenbankverbindung via DataSource (aus BindDataSource.java)

```
// Importieren der benötigten Klassen
import java.sql.Connection;
import com.mysql.jdbc.jdbc2.optional.MysqlDataSource;
...
   // Erzeugen einer neuen Datenquelle (DataSource)
   MysqlDataSource dataSource = new MysqlDataSource();

   // Konfigurieren des Server-Namens
   dataSource.setServerName("localhost");

   // Konfigurieren des Datenbank-Namens
   dataSource.setDatabaseName("test");

   // Angabe des Datenbank-Ports
   dataSource.setPortNumber(3306);

   // Setzen der Zugangs-Informationen
   dataSource.setUser("userName");
   dataSource.setPassword("password");

   // Aufbau der Datenbank-Verbindung via DataSource
   Connection connection = dataSource.getConnection();
...
```

7.4.2 Ablegen der Datenquelle

Nachdem Sie jetzt wissen, wie Sie ein `DataSource`-Objekt erzeugen, müssen Sie nur noch lernen, wie Sie es an einen JNDI-Kontext (`Context`) binden und über diesen verfügbar machen.

Listing 7.19
Binden einer DataSource in einen JNDI-Kontext (aus BindDataSource.java)

```
package de.akdabas.javaee.jndi;
import java.util.Hashtable;
import javax.sql.DataSource;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

// Hersteller-spezifische Klassen
import com.mysql.jdbc.jdbc2.optional.MysqlDataSource;

/** Bindet ein konfiguriertes DataSource-Objekt an einen Kontext */
public class BindDataSource {

    /** Bindet die DataSource via File Service Provider */
    public static void main(String[] args) {
      try {
         // Erzeugen einer neuen Datenquelle (DataSource)
         MysqlDataSource dataSource = new MysqlDataSource();

         // Konfiguration der Datenbank-Verbindung
         dataSource.setServerName("localhost");
         dataSource.setDatabaseName("test");
         dataSource.setPortNumber(3306);
         dataSource.setUser("user");
         dataSource.setPassword("password");

         // Erzeugen des InitialContext
         Hashtable<String, String> env =
                 new Hashtable<String, String>();
         env.put(Context.INITIAL_CONTEXT_FACTORY,
                 "com.sun.jndi.fscontext.RefFSContextFactory");
         Context ctx = new InitialContext(env);

         // Binden der DataSource im Kontext: 'comp/env/jdbc/mysql/'
         // unter dem symbolischen Namen 'dataSource'
         ctx.rebind("comp/env/jdbc/mysql/dataSource", dataSource);

         // Freigeben der Ressourcen und Schließen der Verbindung
         ctx.close();
      } catch (NamingException nex) {
         nex.printStackTrace();
      }
   }
}
```

Dieses Listing legt die konfigurierte `DataSource` im Kontext *comp/env/jdbc/mysql/* unter dem Namen *dataSource* ab. Die Methode `rebind()` überschreibt dabei gegebenenfalls einen bereits unter dem gleichen Namen existierenden Eintrag, ohne einen Fehler zu erzeugen. Wenn Sie dies nicht wünschen, verwenden Sie das Pendant `bind()`.

Listing 7.20
Definition von Namen und Kontext für die Datenquelle

```
...
   // Binden der DataSource im Kontext
   ctx.rebind("comp/env/jdbc/mysql/dataSource", dataSource);
...
```

Achtung

Zum Übersetzen und Ausführen von Listing 7.19 muss der Classpath neben den in Listing 7.8 genannten Dateien auch die Datei *mysql.jar* enthalten, welche Bestandteil der MySQL-Distribution ist.

7.4.3 Auslesen der Datenbankverbindung

Es nutzt Ihnen in einer verteilten Applikation natürlich nichts, lediglich Objekte zu binden, wenn Sie diese später nicht auch extrahieren können. In diesem Abschnitt werden Sie die im Beispiel zuvor abgelegte `DataSource` aus dem Kontext extrahieren, um anschließend eine Datenbankverbindung zu öffnen.

Listing 7.21 Extrahieren eines Objekts aus einem Kontext (aus LookupDataSource.java)

```
package de.akdabas.javaee.jndi;
import java.sql.Connection;
import java.sql.SQLException;
import java.util.Hashtable;
import javax.sql.DataSource;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/** Extrahiert das DataSource-Objekt aus dem Kontext */
public class LookupDataSource {

   /** Extrahiert die DataSource */
   public static void main(String[] args) {
      try {
         // Erzeugen des Initialen Kontext
         Hashtable<String, String> env =
                  new Hashtable<String, String>();
         env.put(Context.INITIAL_CONTEXT_FACTORY,
                  "com.sun.jndi.fscontext.RefFSContextFactory");
         Context ctx = new InitialContext(env);

         // Extrahieren der DataSource
         DataSource datasource = (DataSource)
            ctx.lookup("comp/env/jdbc/mysql/dataSource");

         // Freigeben der Ressourcen und Schließen der Verbindung
         ctx.close();

         // Öffnen einer Datenbank-Verbindung
         Connection connection = datasource.getConnection();

         //...
         // Hier können Sie mit der Datenbank-Verbindung arbeiten
         // ...

         // Schließen der Datenbank-Verbindung
         connection.close();
      } catch (NamingException nex) {
         nex.printStackTrace();
      } catch (SQLException exc) {
         exc.printStackTrace();
      } catch (ClassCastException cce) {
         cce.printStackTrace();
      }
   }
}
```

Zunächst erzeugen Sie wie gewohnt ein `InitialContext`-Objekt, um mit dem Service kommunizieren zu können. Dann extrahieren Sie das gewünschte Objekt über seinen Pfad mit der Methode `lookup()`.

```
...
   // Referenzieren einer Datenquelle aus einem JNDI-Kontext
   DataSource datasource =
      (DataSource) ctx.lookup("comp/env/jdbc/mysql/dataSource");
...
```

Listing 7.22
Extrahieren und Casten eines Objekts aus dem JNDI

Da die Methode `lookup()` stets nur ein Objekt der Basisklasse `Object` zurückgeben kann, müssen Sie dieses anschließend entsprechend casten. Nachdem Sie alle gewünschten Objekte extrahiert haben, können Sie die Verbindung zu diesem Dienst schließen und anschließend mit den Objekten weiterarbeiten. In obigem Listing öffnen Sie beispielsweise eine Verbindung (`java.sql.Connection`) zur Datenbank, die Sie abschließend auch unbedingt wieder schließen sollten.

Namensdienste und ClassCastExceptions

In Listing 7.21 arbeiten Sie zum ersten Mal mit einem vom Namensdienst bereitgestellten Objekt, welches Sie hierfür zunächst in eine `DataSource` casten müssen.

Auch wenn es die Schnittstelle nicht zwingend erfordert, empfehle ich Ihnen beim Casten von Objekten, insbesondere wenn diese über einen Verzeichnisdienst bereitgestellt werden, eine `ClassCastException` abzufangen. Schließlich können Sie nicht mit Sicherheit sagen, ob an dieser Stelle im Directory tatsächlich ein Objekt vom Typ `DataSource` liegt. So können Konfigurationsfehler unseres Verzeichnisdienstes ein seltsames Verhalten unserer Anwendung bewirken ...

7.5 JNDI und Verzeichnisdienste

Nachdem Sie die Grundkonzepte von Namensdiensten verstanden haben, können Sie sich nun den Verzeichnisdiensten zuwenden. Verzeichnisdienste (*Directory Services*) stellen eine Erweiterung von Namensdiensten dar, bei denen den gebundenen Objekten Attribute zugeordnet werden können. Auf diese Weise können Sie in Verzeichnisdiensten nach Objekten mit bestimmten Eigenschaften suchen, ohne deren genauen Pfad kennen zu müssen. Es ist sogar möglich, Einträge zu erstellen, die nur aus Attributen bestehen.

7.5.1 Attribute des Verzeichnisdienstes LDAP

Da sich die nächsten Beispiele auf den wohl bekanntesten Verzeichnisdienst LDAP beziehen, folgt hier eine kurze Einführung in die gängigsten Attribute. LDAP-Einträge bestehen aus einer Reihe von Attributen, die zusammen den so genannten *Distinguished Name* (`dn`) bilden. Jedes der Attribute hat dabei einen bestimmten Typ, der in der Regel durch ein gängiges Kürzel definiert wird. Die folgende Tabelle listet die gängigsten Kürzel auf:

Tabelle 7.4
Gebräuchliche Kürzel für LDAP-Einträge

Kürzel	Bezeichnung	Beschreibung
`dc`	Domain Context	Der Domain-Kontext, also ein *Atomic Name* des Domänennamens
`cn`	Common Name	Der vollständige Name einer Person
`sn`	Surname	Der Familienname der Person; in der Regel ein Teil des `cn`
`c`	Country	Aus zwei Buchstaben bestehendes Länderkürzel, z.B. `de`
`st`	State	Der Bundstaat oder das Bundesland
`l`	Locality	Ein dem Eintrag zugeordneter Ort
`o`	Organisation	Eine Firma oder Organisation
`ou`	Organisation Unit	Eine Organisationseinheit oder Abteilung.
`title`	Title	Ein Titel, wie CEO, Dr., ...
`mail`	eMail	Eine E-Mail-Adresse

Dabei kann jedes Attribut auch mehrmals mit unterschiedlichen Werten vorhanden sein, z.B. wenn eine Person an zwei Orten arbeitet oder für unterschiedliche Unternehmen tätig ist. Die gültigen Werte, die ein Attributeintrag annehmen kann, hängen stark vom jeweiligen Typ ab. So akzeptiert das Attribut `mail` eine E-Mail-Adresse, während sich hinter `photo` beispielsweise eine JPEG-Datei verbergen könnte.

Ein Beispiel für einen LDAP-Eintrag

Einem LDAP-Eintrag sind Sie übrigens schon bei der Konfiguration des Tomcat (Kapitel 1) begegnet, als Sie sich über das Java-Programm `keytool` ein HTTPS-Zertifikat erzeugten. Das Programm erfragte verschiedene Informationen von Ihnen und erzeugte damit ein auf Ihren Namen ausgestelltes Zertifikat. Der entsprechende Eintrag könnte dabei etwa folgende Form gehabt haben:

Listing 7.23
Ein Distinguished Name eines HTTPs-Zertifikats

```
dn: cn=Thomas Stark, ou=Autors, o="Addison-Wesley", l=Berlin,
st=Deutschland, c=DE
```

Diesen Eintrag können Sie in den nächsten Beispielen nutzen, um nach bestimmten Attributen zu suchen.

7.5.2 Erzeugen eines DirContext-Objekts

Prinzipiell funktioniert ein Verzeichnisdienst wie ein Namensdienst und so gleicht der Code zur Erzeugung des `InitialDirContext` dem Code zur Erzeugung eines `InitialContext`, nur dass Sie jetzt einige Klassen des

Package `javax.naming.`*directory* benötigen, einen anderen Provider verwenden und den URL des Dienstes angeben müssen, was beim File System Context nicht erforderlich ist.

Listing 7.24
Erzeugen eines InitialDirContext (aus InitLdap.java)

```
package de.akdabas.javaee.jndi;
import java.util.Hashtable;
import javax.naming.NamingException;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;

/** Erzeugt ein DirContext-Objekt analog zum InitialContext */
public class InitLdap {

   /** Erzeugt ein InitialDirContext-Objekt */
   public static void main(String[] args) {
      try {
         // Konfiguration der JNDI-Umgebung
         Hashtable<String, String> env =
                  new Hashtable<String, String>();

         // Setzen des Service-Providers (LDAP)
         env.put(DirContext.INITIAL_CONTEXT_FACTORY,
                  "com.sun.jndi.ldap.LdapCtxFactory");

         // Setzen des Provider-URLs
         env.put(DirContext.PROVIDER_URL,
                  "ldap://localhost:389/o=Authors");

         // Erzeugen des InitialDirContext
         DirContext ctx = new InitialDirContext(env);

         // ...
         // Hier können Sie analog mit dem DirContext arbeiten
         // ...

         // Freigeben der Ressourcen und Schließen der Verbindung
         ctx.close();
      } catch (NamingException nex) {
         nex.printStackTrace();
      }
   }
}
```

7.5.3 Binden von Objekten

`DirContext` stellt Ihnen die gleichen Methoden zum Suchen, Binden und Verschieben von Objekten zur Verfügung wie das `Context`-Objekt – außer dass es sich bei den Pfaden jetzt um *Distinguished Names* handelt. Das folgende Listing-Fragment speichert beispielsweise ein Applet unter dem Common Name (`cn`) `applet`.

Listing 7.25
Speichern eines Applet im LDAP

```
...
   // Binden eines Applets im Verzeichnisdienst
   java.applet.Applet applet = ...    // Erzeugen des Applets
   ctx.bind("cn=applet", applet);     // Ablegen mit dem cn=applet
...
```

7.5.4 Suche nach Objekten mit bestimmten Attributen

Neben den Standardmethoden des `Context`-Objekts stellt Ihnen ein `DirContext`-Objekt auch komfortable Methoden zum Suchen nach bestimmten Objekten zur Verfügung.

Definieren von Attributen

Um anhand von Attributen suchen zu können, müssen Sie diese natürlich zunächst definieren. Dazu erzeugen Sie zunächst ein `Attributes`-Objekt und füllen es anschließend mit den gewünschten Parametern.

Listing 7.26 Erzeugen eines Attributes-Objekts (aus AttributeSearch.java)

```
import javax.naming.directory.Attributes;
...
   // Erzeugen des Wrapper-Objektes für die Attribute
   Attributes atts = new BasicAttributes(true);

   // Befüllen des Attribut-Wrappers
   atts.put("ou", "Authors");
   atts.put("l", "Berlin");
...
```

Mit diesem `Attributes`-Objekt können Sie nun beispielsweise nach allen Einträgen suchen, deren Organisationseinheit (`ou`) `Authors` ist und die einen Locality-Eintrag (`l`) für Berlin besitzen. Der optionale Parameter des Konstruktors gibt dabei an, ob bei den Attributen (nicht den Werten) die Groß- und Kleinschreibung ignoriert werden soll oder nicht. In obigem Beispiel ist es folglich egal, ob die Organisationseinheit mit dem Attribut `ou` oder `OU` hinterlegt wurde.

Eine Klasse zum Durchsuchen des Kontexts

Nachdem Sie nun wissen, wie Sie Attribute erzeugen können, ist die Suche damit ein Kinderspiel (siehe auch Listing 7.10).

Listing 7.27 Durchsuchen eines Kontexts (aus AttributeSearch.java)

```
package de.akdabas.javaee.jndi;
import java.util.Hashtable;
import javax.naming.NamingEnumeration;
import javax.naming.NamingException;
import javax.naming.directory.Attributes;
import javax.naming.directory.DirContext;
import javax.naming.directory.SearchResult;
import javax.naming.directory.BasicAttributes;
import javax.naming.directory.InitialDirContext;

/** Durchsucht einen Verzeichniskontext anhand von Attributen */
public class AttributeSearch {

   /** Durchsucht den InitialDirContext anhand von Attributen */
   public static void main(String[] args) {
      try {
         // Konfiguration der JNDI-Umgebung
         Hashtable<String, String> env =
                  new Hashtable<String, String>();
         env.put(DirContext.INITIAL_CONTEXT_FACTORY,
                  "com.sun.jndi.ldap.LdapCtxFactory");
```

Listing 7.27 (Forts.)
Durchsuchen eines Kontexts (aus AttributeSearch.java)

```
            env.put(DirContext.PROVIDER_URL,
                    "ldap://localhost:389/o=JNDITutorial");

            // Erzeugen des InitialDirContext
            DirContext ctx = new InitialDirContext(env);

            // Erzeugen des Wrapper-Objektes für die Attribute
            Attributes atts = new BasicAttributes(true);

            // Befüllen des Attribut-Wrappers
            atts.put("ou", "Authors");
            atts.put("l", "Berlin");

            // Suche nach passenden Objekten des Kontextes
            NamingEnumeration list = ctx.search(".", atts);

            // Ausgeben der gefundenen Objekte
            while (list.hasMore()) {
               SearchResult result = (SearchResult) list.next();
               System.out.println(result.getName() + " : " +
                                  result.getObject());
            }

            // Freigeben der Ressourcen und Schließen der Verbindung
            ctx.close();
         } catch (NamingException nex) {
            nex.printStackTrace();
         }
      }
}
```

Die Methode `search()` des `DirContext`-Objekts übernimmt einfach einen Kontextpfad, sowie das `Attributes`-Objekt und liefert die Ergebnismenge in Form einer `NamingEnumeration` zurück, die Sie schon aus den vorangegangenen Beispielen kennen. Diese enthält jedoch dieses Mal Objekte vom Typ `SearchResult`, einer Subklasse von `Binding`. Eines dieser `SearchResult`-Objekte könnte dabei zum Beispiel den Eintrag aus Listing 7.23 enthalten, da dieser auf die angegebenen Attribute passt.

Tipp

Das analoge Beispiel beim Zugriff auf einen reinen Namensdienst finden Sie in Listing 7.6.

7.6 Eine JNDI Lookup-Klasse

In den nächsten Kapiteln werden Ihre Beispiele hin und wieder auf den Namensdienst des JBoss Application Servers zugreifen, um Objekte abzulegen oder auszulesen. Und da wir uns auch dort wieder auf das Wichtigste konzentrieren und Sie nicht mit überlangen Beispielen langweilen möchten, werden wir in diesen Beispielen auf die JNDI-`Lookup`-Klasse zurückgreifen, die in Listing 7.28 zu sehen ist und deren Funktionsweise Ihnen nach dem Studium dieses Kapitels keine Kopfschmerzen mehr bereiten sollte.

Listing 7.28 Eine Hilfsklasse zum Durchsuchen eines Namensdienstes

```
package de.akdabas.javaee.jndi;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/** Extrahiert ein Objekt aus dem JNDI Namensdienst */
public class Lookup {

   /** JNDI Standard-Umgebungskonfiguration des JBoss */
   public static Hashtable<String, String> JBOSS_ENV;

   // Initialisierung von JBOSS_ENV mit Statischem Initialisierer
   static {
      // Initialisieren der Variable JBOSS_ENV
      JBOSS_ENV = new Hashtable<String, String>();

      // Setzen der JBoss JNDI-Factory
      JBOSS_ENV.put(Context.INITIAL_CONTEXT_FACTORY,
                    "org.jnp.interfaces.NamingContextFactory");

      // URL des Namensdienstes (JBoss-Standard-Port: 1099)
      JBOSS_ENV.put(Context.PROVIDER_URL, "localhost:1099");

      // Konfiguration der URL Package Präfixe
      JBOSS_ENV.put(Context.URL_PKG_PREFIXES,
                    "org.jboss.naming:org.jnp.interfaces");
   }

   /** Extrahiert das Objekt unter dem angegebenen Pfad */
   public static Object lookup(Hashtable<String, String> env,
                               String path) {
      Object result  = null;
      try {
         // Erzeugen des Initialen Kontextes mit der übergebenen
         // Konfiguration 'env'
         InitialContext ctx = new InitialContext(env);

         // Suchen des Objektes innerhalb des Namensdienstes
         result = ctx.lookup(path);

         // Freigeben der JNDI Ressourcen
         ctx.close();
      } catch (NamingException nex) {
         nex.printStackTrace();
      }
      return result;
   }
}
```

Dieses Klasse arbeitet mit verschiedenen Namensdiensten und Konfigurationen zusammen, die ebenfalls von dieser Klasse bereitgestellte statische Variable `JBOSS_ENV` enthält die Standardparameter für den Zugriff auf den Namensdienst des JBoss, die Sie bereits in Listing 7.5 kennen gelernt haben.

Info

Diese Klasse initialisiert die Variable mithilfe des *statischen Initialisierers*, mitunter auch als statischer Konstruktor bezeichnet. Dieser hat die Form `static {...}` und kann an beliebiger Stelle zwischen den Methoden platziert werden. Er wird gerufen, sobald die Klasse vom *ClassLoader* geladen wird.

Nun können Sie diese Klasse beispielsweise über folgendes Statement verwenden.

```
...
   // Aufruf der Klasse Lookup mit dem JBoss Standard-Environment
   Object result = Lookup.lookup(Lookup.JBOSS_ENV, "Pfad im JNDI");
...
```

Listing 7.29
Aufruf der Klasse Lookup mit dem JBoss Environment

Diese kapselt im Augenblick alle mit dem Namensdienst verbundenen Operationen und Ausnahmen und gibt Ihnen im Augenblick entweder das gefundene Objekt oder `null` zurück. Wenn Sie diese Beispielklasse in Ihren Anwendungen einsetzen wollen, sollten Sie das Fehlermanagement natürlich anpassen und gegebenenfalls eine eigene `Exception` werfen. In diesem Buch hält das Beispiel jedoch die Beispiele der nachfolgenden Kapitel kurz und übersichtlich.

Info

In echten Produktionsumgebungen sollten Sie darauf achten, ein effektives Fehlermanagement zu etablieren. Fehler werden weiter geworfen, bis effektiv auf sie reagiert werden kann.

7.7 JNDI und Webanwendungen

JNDI wird natürlich vor allem bei großen verteilten Applikationen eingesetzt, um dort beispielsweise *Enterprise JavaBeans (EJB)* oder *Java Message Service (JMS)*-Kanäle abzulegen. Aber auch, wenn Sie es lediglich mit reinen Webanwendungen zu tun haben, kann Ihnen der Einsatz des Java Naming and Directory Service Vorteile bieten.

7.7.1 Datenquelle mit Apache Tomcat im Standalone-Betrieb

Auch wenn Sie den Apache Tomcat als einfachen Webserver und nicht integriert in den JBoss einsetzen und dieser kein »vollwertiger« Application Server im Sinne der Java EE-Spezifikation ist, stellt er Ihnen dennoch eine Reihe von nützlichen Diensten zur Verfügung – so zum Beispiel einen JNDI-Service, der allerdings nur den lokalen Webapplikationen zur Verfügung steht. Dieser Service ermöglicht es Ihnen, Datenquellen und andere Ressourcen über den Webserver vorzukonfigurieren, und erleichtert Ihnen so die Arbeit mit verschiedenen Umgebungen (Test, Produktion ...) durch:

- Integration des Service in den Apache Tomcat

  Da der vom Tomcat emulierte JNDI-Service bereits in den Webcontainer integriert ist, wird kein zusätzlicher Anbieter benötigt.

- Pooling der Verbindungen

  Bei der Verwaltung von Datenquellen verwendet der Apache Tomcat automatisch einen `ConnectionPool`, in dem Datenbankverbindungen

zur späteren Wiederverwendung zwischengespeichert werden können. Das spart Ressourcen für den Auf- und Abbau der Verbindungen und bringt einen deutlichen Geschwindigkeitsgewinn.

Tipp

Mehr über die Unterschiede zwischen dem Apache Tomcat Webserver und dem JBoss Application Server erfahren Sie in Kapitel 8.

Doch wie immer bekommt man nichts geschenkt in dieser Welt und so erkaufen Sie sich diese Vorteile, indem Sie sich in eine Abhängigkeit vom Apache Tomcat begeben.

So wird eine derart erstellte Webapplikation nach der Verwendung dieser Services nicht mehr ohne Anpassungen auf einem einfachen Webserver wie *Mort Bay's Jetty* laufen. Solche Abhängigkeiten treten allerdings bei der Verwendung aller Webserver-spezifischen Dienste auf und wenn Sie sich einmal für einen solchen entschieden haben, steht einer Verwendung dieser Services nichts mehr im Wege.

Info

Jetty (*http://jetty.mortbay.org*) ist ein ebenfalls in Java geschriebener Webserver und Servlet-Container, der unter Apache-Lizenz (Open-Source) verfügbar, jedoch kein Produkt der Apache Software Foundation ist.

Konfiguration des Apache Tomcat

Um die DataSource vom Apache Tomcat verwalten zu lassen, müssen Sie diesen natürlich entsprechend konfigurieren und mit der Datenquelle bekannt machen. Dazu fügen Sie den folgenden Eintrag in die Konfigurationsdatei *conf/server.xml* ein:

```
<Server port="8005" shutdown="SHUTDOWN" debug="0">

<Resource name="jdbc/myoracle" auth="Container"
          type="javax.sql.DataSource"
          driverClassName="oracle.jdbc.OracleDriver"
          url="jdbc:oracle:thin:@127.0.0.1:1521:mysid"
          username="scott" password="tiger"
          maxActive="20" maxIdle="10" maxWait="-1"/>
```

Listing 7.30
Konfiguration einer DataSource im Apache Tomcat (server.xml)

```
<!-- Globale JNDI - Ressourcen -->
<GlobalNamingResources>

   <!-- Definition der DataSource -->
   <Resource name="jdbc/mySQL" auth="Container"
             type="javax.sql.DataSource"
             driverClassName="com.mysql.jdbc.Driver"
             url="jdbc:mysql://localhost:3306/mySQL"
             username=" Username der Datenbank-Verbindung "
             password=" Passwort der Datenbank-Verbindung "
             maxActive="20" maxIdle="10" maxWait="-1" />
```

```
   </GlobalNamingResources>
...
   <!-- Konfiguration des Tomcat-Service -->
   <Service name="Tomcat-Standalone">
...
      <!-- Konfiguration des Containers -->
      <Engine name="Standalone" defaultHost="localhost" debug="0">
...
         <!-- Konfiguration des virtuellen Hosts -->
         <Host name="localhost" debug="0" appBase="webapps">

            <!-- Konfiguration des Kontextes -->
            <Context path="/kap07/" docBase"webapps">
...
               <!-- Einbinden der DataSource -->
               <ResourceLink name="jdbc/applicationDB"
                             global="jdbc/mySQL"
                             type="javax.sql.DataSource"/>
...
         </Context>
      </Host>
    </Engine>
  </Service>
</Server>
```

Listing 7.30 (Forts.)
Konfiguration einer DataSource im Apache Tomcat (server.xml)

Sie können im Top-Level-Element `<GlobalNamingResources>` beliebig viele JNDI-Ressourcen und Datenquellen ablegen und mit einem symbolischen Namen (hier `jdbc/mySQL`) versehen. Diese werden dann über das Element `<ResourceLink>` in den Kontext der jeweiligen Webapplikation eingebunden und dort verfügbar gemacht.

Tipp

Mehr über die Verwendung von Datenquellen und Namensdiensten im Apache Tomcat erfahren Sie unter: *http://tomcat.apache.org/tomcat-5.5-doc/*.

7.7.2 Datenquelle im JBoss Application Server

Natürlich können Sie Datenquellen auch einsetzen, wenn Ihre Webanwendung im JBoss Application Server und dessen integriertem Tomcat ausgeliefert wird. Allerdings wird in diesem Fall der Namensdienst des JBoss genutzt und so ist auch dieser für die Konfiguration verantwortlich.

Um die Datenquelle aus Listing 7.30 im JBoss zu konfigurieren, kopieren Sie einfach eine auf *-ds.xml* endende Datei (z.B. *mySql-ds.xml*), wie sie in Listing 7.31 zu sehen ist, in das Verzeichnis *$JBOSS/server/default/deploy*.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<datasources>
   <local-tx-datasource>

      <!-- Name der DataSource im Namensdienst.Ihm wird
           automatisch das Prefix 'java:/' vorangestellt -->
      <jndi-name>comp/env/jdbc/applicationDB</jndi-name>

      <!-- URL unter der die Datenbank erreichbar ist -->
```

Listing 7.31
Konfiguration der Datasource im JBoss (mySql-ds.xml)

Listing 7.31 (Forts.) Konfiguration der Datasource im JBoss (mySql-ds.xml)

```
      <connection-url>
         jdbc:mysql://localhost:3306/mySQL
      </connection-url>

      <!-- Zu verwendender Datenbanktreiber -->
      <driver-class>com.mysql.jdbc.Driver</driver-class>

      <!-- Passwort und Benutzername der DB-Verbindung -->
      <user-name>Username der Datenbank-Verbindung</user-name>
      <password>Passwort der Datenbank-Verbindung</password>

   </local-tx-datasource>
</datasources>
```

7.7.3 Zugriff aus der Applikation

In der Webanwendung können Sie nun wie gewohnt auf die Datenquelle zugreifen, wobei die Standard-JNDI-Umgebung natürlich die des Apache Tomcat bzw. JBoss ist.

Listing 7.32 Zugriff aus der Webanwendung

```
...
   // Dieses Beispiel zeigt Ihnen, wie Sie z.B. in Servlets auf
   // zuvor konfgurierte Datenquellen zugreifen.
   Context initCtx = new InitialContext();

   // Zum richtigen Kontext navigieren ...
   Context envCtx = (Context) initCtx.lookup("java:comp/env/jdbc");

   // ... die Datenquelle auslesen ...
   DataSource dataSource =
      (DataSource) envCtx.lookup("applicationDB");

   // und die Datenbank-Verbindung öffen.
   Connection connection = dataSource.getConnection();
...
```

Tipp

Den Standardkontext haben Sie in Listing 7.2 kennen gelernt. Für Webanwendungen ist der Standardkontext für den jeweiligen Server konfiguriert. Sie können natürlich auch eine andere Konfiguration verwenden.

Innerhalb dieses Dienstes sind die Datenquellen innerhalb des Subkontexts `java:com/env` abgelegt, den Sie über die Methode

Listing 7.33 Suche des richtigen Kontexts der Datenquelle

```
...
   // Zum richtigen Kontext navigieren ...
   Context envCtx = (Context) initCtx.lookup("java:comp/env/jdbc");
...
```

finden. Und nachdem Sie den richtigen Kontext gefunden haben, ist das Extrahieren der Datenquelle ein Kinderspiel: Sie müssen nur das Objekt mit dem symbolischen Namen auslesen und zur `DataSource` casten – fertig.

Listing 7.34 Auslesen der zuvor konfigurierten Datenquelle

```
...
   // ... die Datenquelle auslesen ...
   DataSource dataSource =
      (DataSource) envCtx.lookup("applicationDB");
...
```

Info

Wenn Sie Ihre `DataSource` nicht im Kontext finden, lassen Sie sich zu Debug-Zwecken doch einmal alle Objekte im Subkontext `java:comp/env` anzeigen.

7.7.4 Vorteile für Webanwendungen

Wenn Sie Ihre Datenbankumgebungen über den Webserver verwalten, haben Sie einen weiteren Vorteil: Größere Applikationen durchlaufen vor dem Produktiveinsatz häufig zunächst eine Art Testsystem und greifen dabei auf eine Testdatenbank zu. Ist der jeweilige URL der Datenbank dabei fest in den Quelltext einkodiert oder beispielsweise im Web Deployment Descriptor hinterlegt, laufen Sie immer Gefahr, versehentlich auf die falsche Datenbank zuzugreifen, oder Sie müssen Ihre Anwendung zwischen Test und Produktion sogar neu übersetzen.

Hinterlegen Sie Ihre Datenbankverbindungen jedoch in den jeweiligen Webservern, minimieren Sie diese Gefahr ganz beträchtlich, da diese nur sehr selten umkonfiguriert werden. Außerdem können Sie auf ein erneutes Übersetzen bei der Überführung in die Produktionsumgebung verzichten. Stattdessen müssen Sie lediglich das Verzeichnis der Anwendung von einem Server auf den anderen übertragen und schon greift die Anwendung auf die Produktionsdatenbank zu.

Dieses Kapitel sollte Ihnen an praxisnahen Beispielen verdeutlichen, wie Sie JNDI einsetzen können, um Ihre Anwendungen effizienter und weniger fehleranfällig zu machen. Natürlich bietet der Apache Tomcat, wie viele andere Webserver auch, ein weitaus größeres Spektrum zur Nutzung von JDNI an. Eine umfassende Dokumentation über die Möglichkeiten erhalten Sie unter *http://jakarta.apache.org/tomcat/tomcat-5.0-doc/jndi-resources-howto.html*.

7.8 Zusammenfassung

Zugegeben, für sich genommen gibt es wenig wirklich sinnvolle Anwendungsfälle für das *Java Naming and Directory Interface*, umso wichtiger ist seine Rolle jedoch für verteilte Applikationen, in denen lose gekoppelte Systeme sicher zusammenarbeiten sollen. Schließlich ist ein gut ausgebautes Mobilfunknetz für sich genommen auch recht nutzlos, doch wer möchte heute noch auf den damit verbundenen Komfort verzichten?

JNDI ist ein einfach zu verwendendes und nahezu universales Werkzeug, um verschiedene Systeme oder Systemkomponenten zu verbinden und so wird Ihnen diese Technologie auch in den folgenden Kapiteln immer wieder begegnen

8

Enterprise JavaBeans (EJB)

Beans! (dt. Bohnen) – so lautet Suns Antwort auf die Frage nach wiederverwendbaren Software-Komponenten. Wenn Sie bisher Webanwendungen entwickelt haben, dann haben Sie JavaBeans vielleicht dazu verwendet, um so genannte Business-Objekte wie die Adresse einer Person abzubilden. Über verschiedene Getter- und Setter-Methoden ermöglichen Sie es einem Anwender, diese Daten zu setzen bzw. auszulesen, wobei die interne Repräsentation der Daten verborgen bleibt. Auf diese Weise wird die JavaBean von der umgebenden Anwendung entkoppelt, so dass beide unabhängig voneinander weiterentwickelt werden können, solange die jeweiligen Schnittstellen der JavaBean erhalten bleiben.

Alle in einer JavaBean gesetzten Variablenwerte bilden dabei den *Zustand* oder die *Entität* der JavaBean. Diese unterscheidet sie von anderen Objekten dieses Typs. In diesem Kapitel soll es nun um die populärsten Vertreter von JavaBeans gehen: die so genannten *Enterprise JavaBeans (EJB)*.

Enterprise JavaBeans der dritten Generation

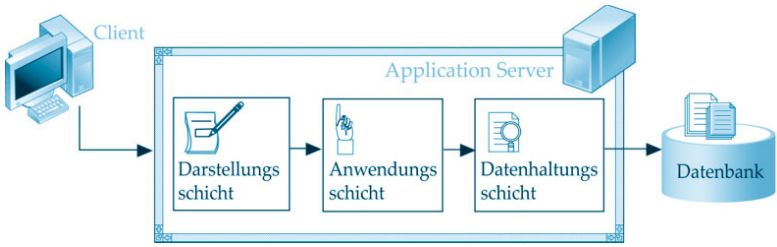
Mit der Spezifikation Java EE 5 unterstützt Sun nun EJBs der dritten Generation (EJB 3.0) und gerade dieser Bereich wurde im Vergleich zu seinen Vorgängern massiv vereinfacht und entschlackt. Nichtsdestotrotz befinden sich auch noch viele EJB-2.x-Anwendungen im Einsatz und es ist gut möglich, früher oder später auch mit dieser »alten« Technologie konfrontiert zu werden. Deshalb werden Sie die Grundlagen im ersten Teil des Kapitels auch so in diese Technologie einführen, dass Sie in der Lage sind, auch EJB-2.x-Anwendungen in Ihrer Struktur zu verstehen und gegebenenfalls zu warten.

Die praktischen Übungen im zweiten Teil konzentrieren sich anschließend natürlich auf die viele neuen Möglichkeiten und zeigen Ihnen, wie leicht sich EJBs inzwischen einsetzen lassen.

8.1 Aufgabe von Enterprise JavaBeans

Gute und robuste Anwendungen sind häufig dreischichtig aufgebaut und unterteilen sich in die Darstellung, die Geschäfts- oder Business-Logik und die Datenhaltung, welche häufig mit objektrelationalen Datenbanken realisiert wird.

Abbildung 8.1
Schematische Darstellung einer dreischichtigen Anwendung



Diese Form der Abstraktion lässt sich nun natürlich beliebig fortsetzen und so könnten Sie beispielsweise eine Schicht einführen, welche über das Java Naming and Directory Interface (JNDI) mit anderen Systemen kommuniziert oder die Speicherung der Daten auf verschiedene Systeme verteilt. Auf diese Weise gelangen Sie von 3-Schicht-Applikationen (Three-Tier-Applications) zu N-Tier-Applications.

Info

Ab diesem Kapitel benötigen die Beispiele einen Java Application Server wie beispielsweise den JBoss. Wenn Sie in den vorangegangenen Beispielen mit dem Standalone Tomcat gearbeitet haben, müssen Sie nun »umsteigen«.

Sun unterteilt JavaBeans hin und wieder in Objekte und Komponenten, wobei Letztere eine eigenständige Datenstruktur und Logik besitzen, wohingegen Objekte meist nur der Speicherung von zusammengehörenden Daten dienen. Obwohl der genaue Unterschied zwischen einem Objekt und einer Komponente nirgendwo fest definiert ist, handelt es sich bei Enterprise JavaBeans stets um Komponenten. Diese werden in einen *Container* integriert, der die einzelnen Komponenten verwaltet und die Dienste der enthaltenen Komponenten anbietet.

8.1.1 Kapselung der Geschäftslogik

Die Geschäftslogik bildet den zentralen Bestandteil aller Applikationen. Egal, ob Bank, Rentenversicherer oder Mautbetreiber: Auch wenn sich Darstellung und Datenhaltung hin und wieder ändern – ohne die in Code gegossenen Formeln zur Berechnung (Workflow), hinter denen sich häufig wichtige Betriebsgeheimnisse dieser Firmen verbergen, können Sie nicht arbeiten.

Enterprise JavaBeans ermöglichen es Ihnen, Ihre Business-Logik in einem zentralen Dienst zu kapseln und für alle darauf zugreifenden Frontend-Systeme zur Verfügung zu stellen.

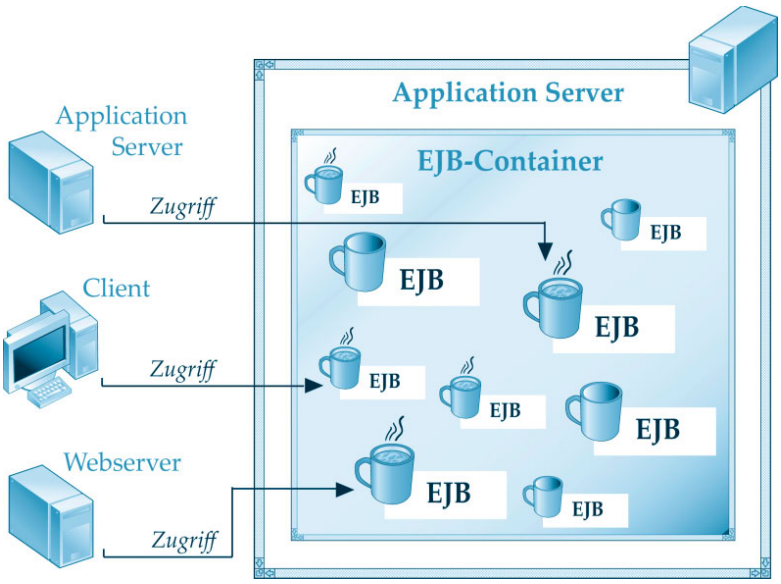


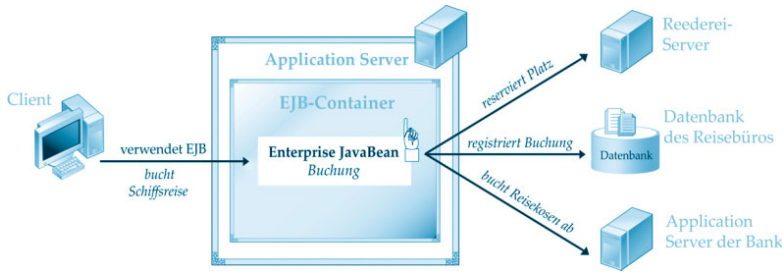
Abbildung 8.2
Kapselung des Workflow

8.1.2 Transaktionsmanagement

Wenn auf die Eingabe eines Benutzers verschiedene Dinge parallel geschehen müssen und entweder alle oder keines ausgeführt werden soll, benötigen Sie ein Transaktionsmanagement-System. Angenommen, ein Reisebüro bietet online die Reservierung für Kreuzfahrten an. Findet nun eine solche Reservierung statt, muss sowohl der Betrag von einem Konto eingezogen als auch die Reservierung bei der Reederei erfolgen. Dabei dürfen Sie weder nur das eine noch nur das andere tun. Kommt es bei einer der Aktionen zu einem Fehler (falsche Kontodaten ...), muss auch die andere storniert werden.

Viele Datenbanken unterstützen bereits Transaktionen, der Vorteil von Enterprise JavaBeans besteht darin, dass deren Transaktionen nicht allein auf Datenbanken beschränkt sind, sondern beispielsweise über den Java Message Service (JMS), den Sie in einem späteren Kapitel kennen lernen werden, auch Nachrichten an weit entfernte Teile einer verteilten Applikation (Server der Reederei) senden können. Kann das Geld nicht vom Einzugskonto abgebucht werden, ist sichergestellt, dass auch die (Buchungs-) Nachricht an die Reederei nicht gesendet wird, da sie Bestandteil der aktuellen Transaktion ist.

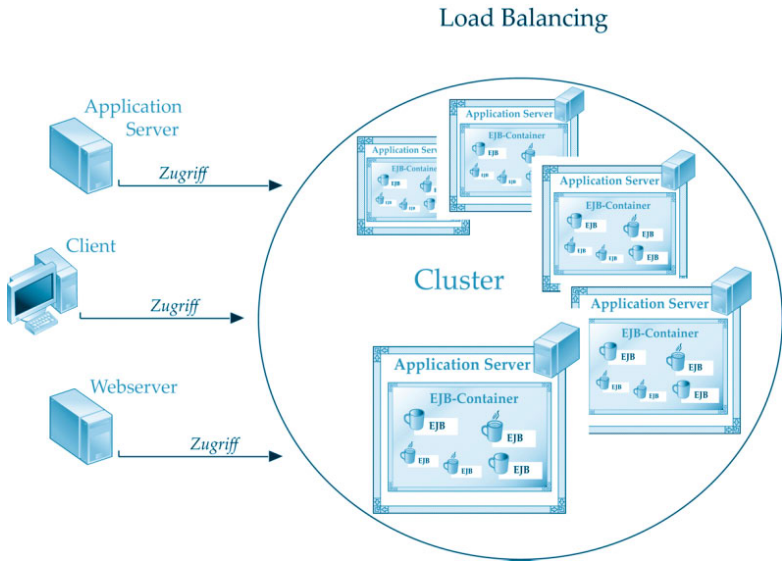
Abbildung 8.3
Transaktionsmanagement in verteilten Systemen



8.1.3 Loadbalancing

Unter Loadbalancing (dt. Lastverteilung) versteht man Verfahren, bei denen eine große Menge von Aufgaben auf verschiedene Server verteilt werden, um die Überlastung eines einzelnen Servers zu vermeiden. Nach außen hin verhält sich dieser Rechnerverbund (Cluster) wie ein einzelner Rechner, so dass Clientsysteme nicht wissen, mit welchem Rechner sie gerade verbunden sind.

Abbildung 8.4
Ein Rechnerverbund (Cluster) verteilt die Aufgaben



Da sich alle Rechner des Cluster nach außen hin identisch verhalten müssen, werden die Dienste des Cluster häufig mit Enterprise JavaBeans realisiert.

8.2 Vom Webserver zum Application-Server

Wenn in den vorangegangenen Kapiteln von einem Server die Rede war, war damit in der Regel ein Webserver gemeint. Dieser enthielt zum Beispiel den *Servlet-Container*, der die Laufzeitumgebung für Servlets und JSPs bildete und auch in einen Application Server eingebettet sein konnte.

Genau wie Servlets benötigen auch Ihre *Enterprise Java Beans* einen Platz, an dem sie erzeugt, mit Aufgaben betraut und schließlich wieder entfernt werden. Diese Laufzeitumgebung ist der so genannte *EJB-Container*.

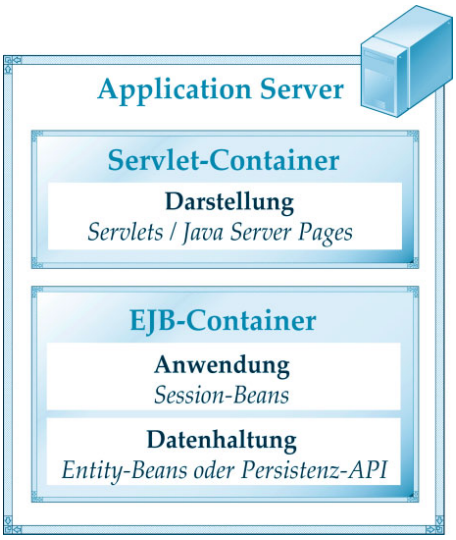


Abbildung 8.5
Aufbau eines Java Application Servers

Nun gibt es kaum einen Hersteller, der einen reinen EJB-Container anbietet. Dies liegt daran, dass eine Enterprise JavaBean als Logik-Container, ohne Darstellungs- und Datenhaltungsschicht meist reichlich nutzlos wäre. So statten inzwischen auch viele Anbieter von Datenbanken, Transaktionsmonitoren oder CORBA-ORBs ihre Server mit einem EJB-Container aus.

Info

CORBA steht für *Common Object Request Broker Architecture* und definiert plattformunabhängige Kommunikationsprotokolle, die nicht an eine bestimmte Programmiersprache gebunden sind. Über einen so genannten *Interface Definition Language (IDL)*-Compiler werden dabei zunächst formale Objekte definiert und anschließend um die gewünschte Funktionalität erweitert, dazu wird diese dann in einer konkreten Programmiersprache implementiert.

Der entfernte Client operiert auf einem leeren *Stub-Interface*, dessen Methodenaufrufe dann über einen *Object Request Broker (ORB)* an das eigentliche Business-Objekt weitergeleitet werden, das vom erzeugten Skeleton ableitet. Abschnitt 8.4 führt Sie genauer in die Technik der Methodenfernaufrufe ein.

Suns Java-EE-Spezifikation fordert sogar, dass ein konformer Application Server sowohl einen Servlet- als auch einen EJB-Container und außerdem eine Reihe von unterstützenden APIs enthält: Damit scheidet der Apache Tomcat jedoch aus, da er zwar eine Reihe der Normen erfüllt und zahlreiche Dienste anbietet, aber eben keinen EJB-Container enthält.

8.2.1 Portabilität von Enterprise JavaBeans

Java, das steht auch für *Write once, run anywhere!* und natürlich sollte es laut Spezifikation möglich sein, EJB-Komponenten, die unter einem EJB-Container entwickelt wurden, problemlos auf einen anderen zu übertragen. Ganz analog wie Sie auch von JSPs erwarten, dass das resultierende HTML-Dokument des *Jetty-Servers* identisch zu dem des *Apache Tomcat* ist.

So weit die Theorie, doch in Wirklichkeit entwickelt jeder Anbieter seinen Java Application Server so, dass er möglichst harmonisch mit den anderen Komponenten zusammenarbeitet, und passt die Spezifikation entsprechend an. So können Oracle-Benutzer beispielsweise ihre EJBs wahlweise im *Internet Application Server IAS* (vormals OAS) oder direkt in der Datenbank zum Einsatz bringen, wodurch ein deutlicher Performance-Gewinn erreicht werden kann und die aufwändige Administration des OAS entfällt. Man muss sich schon sehr zwingen, statt der vom Hersteller angebotenen Abkürzung immer den EJB-konformen Weg zu gehen.

Ob und inwieweit Sie Ihren EJB-Container austauschen können, hängt also stark davon ab, wie weit sich die Anbieter und *Sie* sich an die EJB-Spezifikation halten.

Auf der jährlich in San Francisco stattfindenden *JavaWorld* werden von einer unabhängigen Jury die besten Produkte in zehn Kategorien gewählt. Die Finalisten der Kategorie *Best Java Application Server* lauten dabei seit Jahren:

- JBoss (*http://www.jboss.org*)
- BEA WebLogic (*http://www.bea.com*)
- IBM WebSpere (*http://www.ibm.com*)

Info

Die *Lesser General Public License (LGPL)* ist eine Software-Lizenz der *Free Software Foundation*, unter der OpenSource-Programme vertrieben werden. Ein wesentlicher Unterschied zur *General Public License (GPL)* besteht darin, dass auf LGPL basierende Applikationen auch proprietär vertrieben werden dürfen.

Diese drei Kandidaten liefern sich jedes Jahr ein knappes Kopf-an-Kopf-Rennen, was ein weiteres Mal zeigt, dass OpenSource-Software durchaus mit kommerziell entwickelten Produkten konkurrieren kann. Nicht ohne Grund haben wir uns in diesem Buch für den unter LGPL-Lizenz verfügbaren JBoss entschieden.

8.2.2 Aufgaben eines EJB-Containers

Ein EJB-Container dient als Laufzeitumgebung für Ihre Enterprise JavaBeans (EJBs). Das verpflichtet ihn, neben der Bereitstellung der oben beschriebenen APIs folgende Dienste anzubieten:

Überwachung des Lebenszyklus von EJBs

Genau wie Servlets durchlaufen auch EJBs einen bestimmten Lebenszyklus. Und analog zum Servlet-Container ist der EJB-Container in diesem Fall für den Übergang von einem Zustand in einen anderen verantwortlich und steuert diesen.

Instanzen-Pooling

Im Gegensatz zu Servlets, von denen zur Laufzeit stets nur eine einzige Instanz existiert, kann ein EJB-Container mehrere Instanzen einer *Enterprise JavaBean* auf Vorrat erzeugen und über einen Pool verwalten.

Namens- und Verzeichnisdienst

EJB-Container bieten die Dienste der integrierten Enterprise JavaBeans über einen Namensdienst an, der in der Regel vom Server selbst verwaltet wird. Mehr über Namens- und Verzeichnisdienste erfahren Sie im Kapitel über das Java Naming and Directory Interface (JNDI).

Transaktionsdienst

Komplexe Aufgaben bestehen in der Regel aus einer Menge von Teilaufgaben, die etwa ganz oder gar nicht ausgeführt werden. So besteht eine Überweisung beispielsweise aus dem Abbuchen eines Betrags vom Belastungskonto und der anschließenden Gutschrift dieses Betrags auf dem Gutschriftkonto. Misslingt der zweite Schritt, etwa weil das Konto nicht existiert oder der verantwortliche Server nicht erreichbar ist, muss auch der erste Teil der Transaktion rückgängig gemacht werden.

Sie können die Abarbeitung einzelner EJBs auch zu Transaktionen zusammenfassen, die entweder alle erfolgreich ablaufen (*Commit*) oder alle rückgängig gemacht werden (*Rollback*).

Nachrichtendienst (Message Service)

Mit der Spezifikation 2.0 führte Sun die *Message Driven Beans* ein. Diese kommunizieren nicht via Prozedurfernaufruf, sondern über den *Java Message Service (JMS)* miteinander und ermöglichen so erstmals einen asynchronen Objektaustausch.

Persistenz

Hin und wieder kommt es vor, dass ein EJB-Container neu gestartet wird oder selten genutzte Komponenten aus dem Arbeitsspeicher ausgelagert werden, um benötigte Ressourcen freizugeben. In diesem Fall kann es erforderlich sein, dass ein Enterprise JavaBean nach der Reinstanziierung den gleichen Zustand wie vorher wiedererlangt.

8.3 Beans, Beans, Beans

Beans, bei diesem Wort sollte es inzwischen schon in Ihren Ohren klingeln, denn schließlich begegnen Ihnen diese Software-Komponenten auf Schritt und Tritt. Doch Bean ist nicht gleich Bean und deshalb sollten Sie die eine von der anderen unterscheiden können.

Dieser Abschnitt stellt Ihnen die Konzepte hinter JavaBeans und Enterprise JavaBeans vor und zeigt Ihnen, worin sie sich unterscheiden. Denn obwohl beide augenscheinlich starke Ähnlichkeit in der Namensgebung besitzen, verfolgen sie gänzlich andere Ziele und stehen für verschiedene Konzepte der Software-Entwicklung, die sich allerdings vortrefflich ergänzen.

8.3.1 Beans

Unter einer *Bean* verstehen Sie in Java nichts anderes als eine in sich geschlossene Software-Komponente, die einer Handvoll Bedingungen genügt. Dabei kann es sich prinzipiell um alles handeln, von einer Wrapper-Class, die nichts anderes tut, als ein Objekt zu kapseln und den Zugriff darauf zu steuern, bis hin zur vollständigen ERP-Lösung inklusive Buchhaltung und Lagerverwaltung.

Info

ERP steht in der Informatik für *Enterprise Resource Planning* und bezeichnet Prozesse, die versuchen, alle in einem Unternehmen verfügbaren Ressourcen (z.B. Betriebsmittel, Kapital) möglichst effizient zu koordinieren und einzusetzen.

8.3.2 JavaBeans

Der wohl meist zitierte Satz zum Thema *JavaBeans* ist:

> *Eine JavaBean ist eine wiederverwendbare Software-Komponente, die mit einem Builder-Tool visuell manipuliert werden kann.*

Dieser stammt aus Suns Java Beans-Spezifikation 1.01 aus dem Jahre 1997. Doch was verbirgt sich dahinter und was versteht Sun unter einem Builder-Tool? Nun, wenn Sie je mit *Visual Basic* oder anderen Microsoft-verwandten Programmiersprachen und ihren meist grafischen IDEs zu tun

hatten, dürfte der Begriff *Property* (Eigenschaft) keine Schwierigkeiten bereiten. Eine JavaBean ist dementsprechend nichts anderes als die Sammlung von Eigenschaften einer (grafischen) Komponente.

Für alle anderen kommt hier ein kurzer (und sehr persönlich interpretierender) Erklärungsversuch. Ende der neunziger Jahre wurde eine neue Art der Software-Entwicklung populär. Grafische Benutzeroberflächen setzten sich mehr und mehr durch und hielten insbesondere Einzug auf Computern von Informatikstudenten. Es wuchs eine neue Generation von Informatikern heran, für die Lochkarten und Assembler-Programmierung Saurier einer längst vergangenen Computer-Steinzeit waren. Mit den weit verbreiteten Hochsprachen *C/C++* und *Borland Pascal* ließen sich zwar alle möglichen und unmöglichen Algorithmen relativ komfortabel realisieren, doch die Programmierung von grafischen Benutzeroberflächen war weiterhin eine schwer zu meisternde Königsdisziplin.

Ebendiese Lücke zwischen Programmieranspruch und Realität erkannte Microsoft und setzte sich mit seinen grafischen IDEs schnell in bestimmten Märkten durch. Die vordefinierten Bibliotheken gestatteten es, relativ einfach zu erlernenden Basic-Code mit anspruchsvollen grafischen Elementen zu verknüpfen und diesen per Mausklick bestimmte Eigenschaften zuzuweisen und eine gewisse Zeit lang war man der Meinung, dies sei die Zukunft der Programmierung (und darüber kann wieder jeder eigener Meinung sein). Doch egal, ob es sich dabei nun um eine Antwort Suns auf Microsofts Standard oder lediglich um eine Lösung für einen ähnlichen Problemkreis handelte: *JavaBeans* sind gewissermaßen das Java-Gegenstück zu diesen *Properties*.

Durch die Namenskonventionen `<Typ> get<Eigenschaft>()` und `void set<Eigenschaft>(<Typ>)` war man in der Lage, per Reflection (`java.lang.reflect`) auch zur Laufzeit auf die Eigenschaften zuzugreifen und über einen Dialog zu verändern. Und so lässt sich zusammenfassen: Eine JavaBean

- ist ein Daten-Container, der bestimmte Eigenschaften eines Objekts repräsentiert,
- erlaubt es, durch eine besondere Namenskonvention diese Eigenschaften auch dynamisch zu erkennen und per Reflection zu manipulieren,
- konzentriert sich im Wesentlichen auf die Persistenz dieser Eigenschaften.

8.3.3 Enterprise JavaBeans (EJB)

EJBs entstanden wiederum aus einem ganz anderen Software-Blickwinkel. Ihre Heimstätten waren nicht heimische PCs, auf denen Anwender optisch ansprechende Dialoge realisieren wollten, sondern die Mainframe-Rechner großer Recheninstitute.

Info

Als *Mainframe* (dt. Großrechner) bezeichnet man komplexe Computersysteme, die sich vor allem durch Zuverlässigkeit und Datendurchsatz auszeichnen. Die Kapazitäten eines solchen Rechners liegen typischerweise weit über denen eines PCs oder Servers, wobei viele Komponenten redundant ausgelegt sind, um einen durchgängigen Betrieb zu gewährleisten. Im Gegensatz zu den so genannten Supercomputern, die sich vor allem durch eine hohe Rechenleistung auszeichnen, sollen Mainframes vor allem robust sein. Dazu statten Anbieter von Mainframes (z.B. IBM, Siemens und Sun) ihre Systeme meist mit selbst entwickelten und auf den Server zugeschnittenen Betriebssystemen aus. Typische Nutzer von Mainframes sind vor allem Banken, Versicherungen und öffentliche Verwaltungen.

Eines der großen Probleme serverseitiger Prozesse ist deren *Skalierbarkeit*. Darunter versteht man, wie gut ein System auf steigende Belastung reagiert bzw. wie viel mehr Ressourcen es benötigt. Ein gut skalierendes System benötigt, um die zehnfache Leistung zu erbringen, auch zehnmal mehr Ressourcen, während ein schlecht skalierendes System diese Ressourcen bereits bei doppelter Leistung beanspruchen würde und bei zehnfacher Last komplett versagt.

Die EJB-Spezifikation beschreibt nun ein Framework, das die verteilte Verarbeitung von *transaktionsorientierter Geschäftslogik* ermöglicht. Dabei bleiben die Enterprise JavaBeans im Gegensatz zu »reinen« JavaBeans für den Benutzer stets verborgen, da der EJB-Container für ihn eine Blackbox darstellt und er keinen Zugriff auf die tatsächliche Implementierung hat. EJBs sind also weniger reine Daten-Container als vielmehr Service-Objekte, deren Dienste über einen Methodenfernaufruf in Anspruch genommen werden können.

Dabei verwenden EJBs zur Transaktion von Daten JavaBeans als Daten-Container, wobei diese dann oftmals auch als *Value-Beans* oder *Wert-Objekte* bezeichnet werden.

8.4 Methodenfernaufruf

Kern der EJB-Technologie ist der Methodenfernaufruf (engl.: *Remote Methode Invocation*, *RMI*). Darunter versteht man den Aufruf von Befehlen, die auf einem entfernten Rechner oder einer anderen Java Virtual Machine abgearbeitet werden.

Die Befehle liegen dabei in Form einer Methode vor, die von einem Objekt im Speicher des entfernten Rechners implementiert wird. Sie kommunizieren mit diesem Objekt über eine vom entfernten Server bereitgestellte Schnittstelle, die diese Methoden deklariert und deren Aufruf über den Application Server an das reale Objekt weiterleitet.

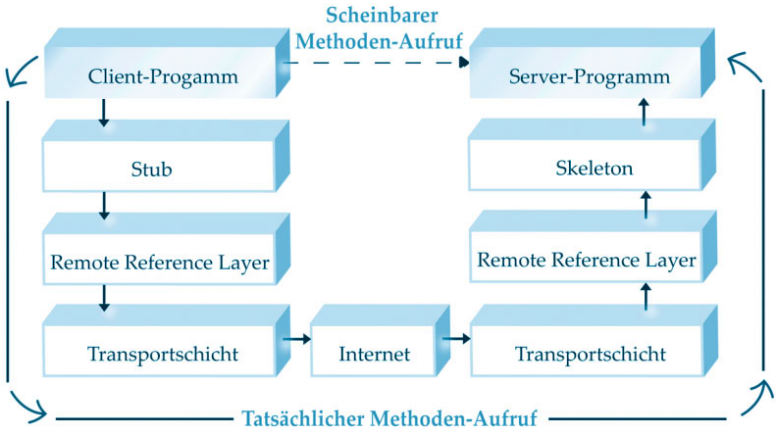


Abbildung 8.6
Remote Method Invocation (RMI) schematisch

Der dem Client lokal verfügbare, abstrakte Rumpf der Klasse wird als *Stub* (dt. Stumpf) bezeichnet und entspricht in Java einem `Interface`. Zu diesem gehört ein passendes *Skeleton*: eine abstrakte Klasse, die einerseits dem `Interface` des Clients genügt und andererseits als Basis für die »echte« Klasse auf dem Server dient. Diese implementiert dann die gewünschte Funktionalität und liefert schließlich das »Fleisch« um das nackte Skelett (*Skeleton*), das im Wesentlichen nur die Methodensignaturen enthält.

8.4.1 Verschiedene Broker-Architekturen

Um die Funktionalität der entfernten Klasse bereitzustellen, bedarf es eines Maklers (*Broker*), der zwischen abstraktem Objekt und tatsächlicher Implementierung vermittelt. Bekannte Broker-Architekturen sind dabei:

Common Object Request Broker Architecture (CORBA)

Dieser wohl älteste Standard wurde von der Open Management Group (OMG) entworfen und ermöglicht den Datenaustausch und die Kommunikation zwischen Komponenten unterschiedlicher Programmiersprachen.

Server, die CORBA-Dienste anbieten, werden als Object Request Broker (ORB) bezeichnet.

Remote Method Invocation (RMI)

Dieser bereits seit der Version 1.1 im JDK enthaltene Standard ermöglicht die Kommunikation von verteilten Java-Komponenten. Dies hat gegenüber CORBA den Vorteil, dass auch spezielle Java-Charakteristika (z.B. der Garbage Collector) berücksichtigt werden konnten, außerdem entfällt das Erzeugen formaler Interface Definition Language (IDL)-Objekte, da ausschließlich Java-Komponenten ausgetauscht werden (können).

Distributed Component Object Model (DCOM)

Auch Microsofts Implementierung ermöglicht die Kommunikation von Komponenten, die in verschiedenen Sprachen geschrieben wurden. Hierfür wird allerdings die nicht auf allen Plattformen verfügbare DCOM-Laufzeitbibliothek benötigt.

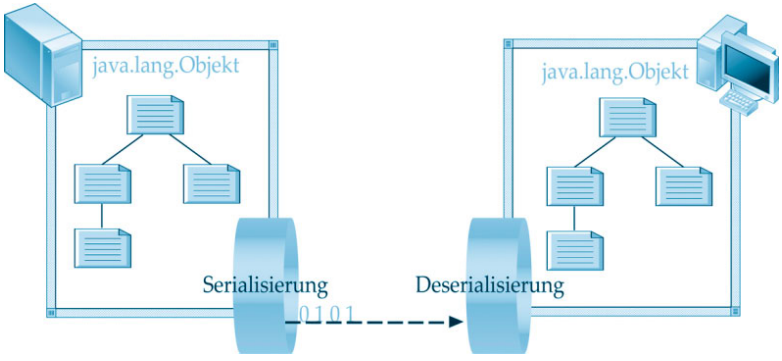
Info

Seit dem JDK 1.3 stellte Sun die Kommunikation seiner Architektur auf das CORBA zugrunde liegende Internet *Inter-ORB Protocol (IIOP)* um, wodurch es prinzipiell (wenn auch mit einigem Aufwand) möglich ist, über RMI auch Objekte anzusprechen, die nicht in Java implementiert sind (RMI-IIOP). Um entfernte Objekte anzusprechen, bedienen sich die meisten Broker-Architekturen eines Namens- oder Verzeichnisdienstes (siehe Kapitel über JNDI). Der bedeutendste Service ist dabei der COS Naming Service.

8.4.2 Austausch von Objekten durch Serialisierung

Wie Sie wissen, werden durch RMI die über ein `Interface` bereitgestellten Methoden eines entfernten Objekts auf einem entfernten Rechner zur Ausführung gebracht. Solange alle Methoden den Rückgabewert `void` haben, existiert bis dato auch noch kein Problem. Doch was passiert mit zurückgegebenen Objekten, seien es nun Strings oder eigene JavaBeans? Das Zauberwort heißt *Serialisierung*.

Abbildung 8.7
Objektserialisierung



Dabei wird ein Objekt bzw. seine Repräsentation im Speicher des entfernten Rechners in einen Datenstrom (z.B. über einen `java.io.DataOutputStream`) umgewandelt und an den lokalen Rechner gesendet. Dieser deserialisiert das empfangene Objekt anschließend wieder und übergibt dem Benutzer des `Interface` eine Referenz auf dieses – ganz so, als sei der Auf-

ruf der Methode lokal erfolgt. Mit der Serialisierung und Deserialisierung kommt der Benutzer in der Regel nicht in Berührung. Dies erledigen die Klassen des Package `java.rmi.*` für ihn.

Nun hat zwar jedes Objekt eine Repräsentation im Speicher, dennoch können nicht alle Objekte serialisiert werden. So stellt eine JDBC-Verbindung beispielsweise eine lokal gebundene Referenz auf einen Datenbankdienst dar und kann damit nicht übertragen oder gar auf einem persistenten Speichermedium festgehalten werden.

Damit Ihre Objekte übertragen werden können, müssen sie das Marker-Interface `java.io.Serializable` implementieren. Versuche, ein nicht serialisierbares Objekt per RMI zu übertragen, quittiert Java mit einer entsprechenden Ausnahme.

8.5 Verschiedene Typen von Enterprise JavaBeans

Die EJB-Spezifikation sieht verschiedene Typen von JavaBeans vor, welche unterschiedliche Aufgaben übernehmen. Dies sind:

- Entity Beans

  Diese Form von EJBs bildete vor der Einführung der EJB-Spezifikation 3.0 die Schnittstelle zur Datenbank und repräsentiert die Datensätze einer Tabelle als Java-Objekte. Sie wurden in der Regel von Session Beans dazu verwendet, um Datensätze zu lesen oder zu manipulieren.

  Seit Java EE 5 übernimmt dies das neu geschaffene Persistenz API, mit dem sich das übernächste Kapitel beschäftigt. Entity Beans werden damit zunehmend seltener anzutreffen sein.

- Session Beans

  Session Beans sind die eigentlichen Stars der EJBs. Sie kapseln die Geschäftslogik und stellen diese anderen Anwendungen (Clients) in Form von Services bereit. Wenn Sie für die Erbringung von Diensten auf Datenbanken zugreifen müssen, tun Sie dies entweder in Form von Entity Beans (EJB 2.x) oder mithilfe des Persistenz API.

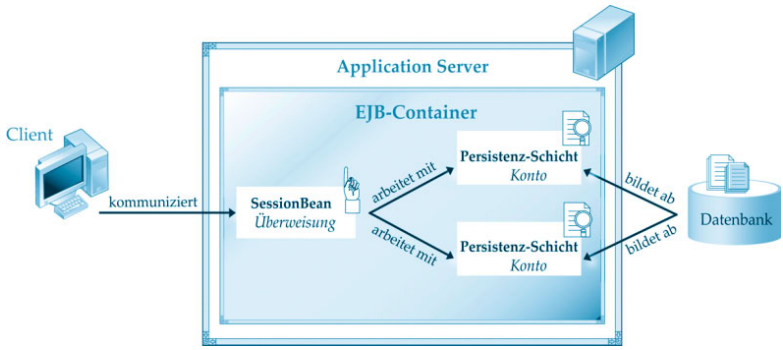
  Abbildung 8.8 verdeutlicht dies noch einmal anhand einer Überweisung (Workflow), die Daten zwischen zwei Konten (Datensätzen) transferiert.

- Message Driven Beans

  Diese Beans sind die jüngsten Ableger in Suns Bohnenzucht. Seit der EJB-Spezifikation 2.0 (2001) ermöglichen diese eine asynchrone Kommunikation von EJB-Komponenten über den *Java Message Service (JMS)*.

Message Driven Beans werden Sie im nächsten Kapitel begegnen, welches Sie ausführlich in die Techniken des asynchronen Nachrichtenaustauschs einführt.

Abbildung 8.8
Funktionsweise von Session Beans



8.6 EJBs in freier Wildbahn

Zeit, eigene Erfahrungen zu sammeln! Und da sich die nächsten beiden Kapitel mit Messaging und Persistenz beschäftigen, konzentrieren wir uns in diesem Kapitel auf die Session Beans. Dies genügt auch vollauf, da es gleich zwei verschiedene Typen von Session Beans gibt.

8.6.1 Stateless Session Beans

Die Methoden einer Stateless Session Bean arbeiten, wie es der Name schon suggeriert, zustandslos. Genau wie Servlets vergessen diese Beans nach einer Transaktion den Client sofort wieder und widmen sich dem nächsten. Wenn ein Client zwei aufeinander folgende Transaktionen durchführt, ist es gut möglich, dass er es dabei mit unterschiedlichen Instanzen dieses Objekts zusammenarbeitet.

Tipp

Verwenden Sie Stateless Session Beans immer dann, wenn Sie die Session-Informationen bereits an anderer Stelle speichern können, z.B. wenn es sich beim Client um ein Servlet handelt, welches ja ebenfalls über ein Objekt zur Speicherung benutzerspezifischer Informationen handelt.

Weder der Client noch die Stateless Session Bean können ihren jeweiligen Partner von anderen unterscheiden oder gar wiedererkennen, genauso wenig, wie ein Servlet einen Request von einem anderen unterscheiden kann. Und da für den EJB-Container alle Instanzen dieses Objekts identisch sind, können sie von ihm sehr ressourcenschonend über einen gemeinsamen Pool verwaltet werden.

8.6.2 Stateful Session Beans

Stateful Session Beans sind in der Lage, ihren Zustand in Instanz-Variablen zu speichern. Dieser Speicher wird auch als *Konversationsgedächtnis* bezeichnet, da sich die Bean darüber den Zustand ihrer Interaktion mit dem Client »merkt«.

Info

Stateful Session Beans verfügen über ein Konversationsgedächtnis.

Im Gegensatz zu Stateless Session Beans, bei denen einem Client für jede Transaktion ein beliebiges Bean-Objekt zugewiesen wird, arbeitet dieser im Fall von Stateful Session Beans stets mit der gleichen Instanz, was vergleichbar ist mit dem *Session-Objekt* von Servlets, die ebenfalls über verschiedene Requests des Clients hinweg eineindeutig waren.

Ein typisches EJB-Szenario für den Einsatz von Stateful Session Beans ist ein virtueller Warenkorb, der natürlich einem Benutzer zugeordnet sein muss. Die Produkte in diesem Warenkorb könnten dann entweder Entity-Beans oder (da sie alle gleichwertig sind) Stateless Session Beans sein.

Da sich der innere Zustand dieser Beans nun von Instanz zu Instanz unterscheidet und diese bei einer Transaktion nicht mehr beliebig gegeneinander ausgetauscht werden können, muss der EJB-Container wesentlich mehr Aufwand betreiben, um diese Beans zu verwalten. Mein Tipp an dieser Stelle ist: Wenn Sie bereits über ein Session-Objekt für Ihren Benutzer verfügen – etwa weil dieser über ein Web-Interface mit Ihnen kommuniziert und somit bereits ein `HttpSession` vom Servlet-Container verwaltet wird –, ersparen Sie Ihrem EJB-Container diesen Zusatzaufwand. Er wird es Ihnen durch einen sparsameren Ressourcenverbrauch danken.

8.6.3 Hello World – eine Stateless Session Bean

Um Ihnen zunächst zu zeigen, wie einfach sich Enterprise JavaBeans erstellen lassen, werden wir einer guten alten Tradition erster Beispiele folgen. Die EJB, die Sie gleich erstellen werden, macht demzufolge nichts anderes, als eine fest definierte Zeichenkette zurückzugeben, und da sie sich hierfür keinerlei Zustände merken muss, realisieren wir sie als Stateless Session Bean.

Info

Die »Hello World«-Tradition geht auf ein internes C-Programmierhandbuch zurück, welches Brian Kernighan (194) für die Bell Laboratories verfasste. Richtig berühmt wurde es jedoch erst durch das Standardwerk »The C Programming Language«, das er zusammen mit Dennis Ritchie verfasste.

Das Interface

Da die spätere Implementierung vor dem Client verborgen werden soll und dieser nur auf einem Interface arbeitet, empfehle ich Ihnen, auch mit diesem zu beginnen und somit die gesamte »äußere« Funktionalität der EJB zusammenzutragen.

Listing 8.1 Remote Interface der Stateless Session Bean

```
package de.akdabas.javaee.ejb;

/**
 * Remote Interface der Stateless Session Bean;
 * Dieses Interface wird allen Clients zur Verfügung gestellt, die
 * später auf den Dienst zugreifen sollen.
 */
public interface HelloWorld {

   /** Von der EJB bereitgestellte Service-Methode */
   public String getGreeting();
}
```

Schon wieder ein Plain Old Java Object (POJO)! Mit Einführung der neuen EJB-Spezifikation 3 handelt es sich bei den Remote Interfaces um einfache Java-Klassen, die kein spezielles Interface erweitern müssen. Die einzige Aufgabe dieses Interface ist es, die nach außen zur Verfügung gestellten Methoden zu deklarieren.

Der Name Remote Interface ist übrigens historisch bedingt: Vor EJB 3 mussten alle EJBs mit Ausnahme der Message Driven Beans zwei Schnittstellen definieren. Das so genannte Home Interface definierte Methoden, die es dem Application Server ermöglichten, eine (lokale) Instanz der Bean zu erzeugen, während das oben beschriebene Remote Interface für den entfernten Zugriff durch den Client bestimmt war. Aus heutiger Sicht würde man das Remote Interface vielleicht als Business Interface bezeichnen.

Die Implementierung

Nachdem das Interface deklariert ist, können Sie sich an die Implementierung machen und auch diese sieht auf den ersten Blick wie eine gewöhnliche Java-Klasse aus.

Listing 8.2 Implementierung der Stateless Session Bean

```
package de.akdabas.javaee.ejb;

import javax.ejb.Remote;
import javax.ejb.Stateless;
import java.io.Serializable;
import java.rmi.RemoteException;

/**
 * Implementierung der Stateless Session Bean (EJB)
 */

@Stateless
@Remote(de.akdabas.javaee.ejb.HelloWorld.class)
public class HelloWorldBean implements HelloWorld, Serializable {

   /**
    * Gibt die Grußbotschaft zurück
```

```
     */
    public String getGreeting() throws RemoteException;
       return "Hello World!";
    }
}
```

Listing 8.2 (Forts.)
Implementierung der Stateless Session Bean

Vor der EJB-Spezifikation 3.0 musste jede EJB ein vom Framework vordefiniertes Interface (`SessionBean`) und darin definierte Methoden des EJB-Lebenszyklus implementieren. Außerdem musste die Bean anschließend über einen Deployment Descriptor namens *ejb-jar.xml* konfiguriert werden, wie es Listing 8.3 zeigt.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC
   '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN'
   'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>

<ejb-jar>
  <enterprise-beans>
   <session>
      <ejb-name>HelloWorldBean</ejb-name>
      <remote>de.akdabas.javaee.ejb.HelloWorld</remote>
      <ejb-class>de.akdabas.javaee.ejb.HelloWorldBean</ejb-class>
      <session-type>Session</session-type>
      <transaction-type>Bean</transaction-type>
   </session>
  </enterprise-beans>
</ejb-jar>
```

Listing 8.3
Deployment Descriptor früherer EJBs

Die unter EJB 2.x erforderlichen Methoden des EJB-Lebenszyklus haben oft eine generische Struktur und erfordern eine Menge zusätzlichen Aufwand. Dies schreckte in den letzten Jahren auf der einen Seite viele potenzielle Anwender von der Technologie ab und führte auf der anderen Seite zur Entwicklung von Code-Generatoren wie xDoclet, die die aufwändigen Standardkonfigurationen automatisch erzeugten.

Info

Sie können EJBs auch über verschiedene Deployment-Descriptors in Form von XML-Dateien konfigurieren.

Sun erkannte dies und ermöglicht es seit der Spezifikation 3.0, auf diesen Overhead zu verzichten und stattdessen mit Java-Annotationen zu arbeiten. Wenngleich es, schon aus Gründen der Abwärtskompatibilität, weiterhin möglich ist, EJBs über die verschiedenen Deployment Descriptors zu konfigurieren.

Wir zeigen Ihnen das an dieser Stelle nur, damit Sie nicht verwirrt sind, wenn Sie bei einer EJB-2.x-Anwendung mit diesen Deployment Descriptors in Berührung kommen. Deren Aufgabe ist es, den Application Server mit Meta-Informationen über die EJB zu versorgen, damit dieser sie korrekt bereitstellen kann. Wenn Sie Ihre eigenen EJBs hingegen nach der neuen Spezifikation 3 implementieren, können Sie die Deployment Descriptors auch gleich wieder vergessen, da es nun einen sehr eleganten Weg gibt, Implementierung und Meta-Informationen zusammen zu deklarieren.

Java Annotations

Annotations stellen eine einfache und elegante Möglichkeit dar, Informationen und Meta-Informationen einer Klasse in einer Datei zu hinterlegen. Es existieren dabei unterschiedliche Arten von Meta-Informationen.

Info

Annotationen enthalten Meta-Informationen zu einer Klasse.

- Meta-Informationen, die von speziellen Zusatzprogrammen verwendet werden können, um z. B. weitere Ressourcen zu erstellen. Alte Bekannter für solche Meta-Informationen sind beispielsweise Javadoc-Tags, die verwendet werden, um die vom gleichnamigen Werkzeug erzeugte Dokumentation zu erzeugen.
- Meta-Informationen, die von einem speziellen Compiler verwendet werden können, um zusätzliche Anweisungen in den Bytecode einzufügen. Dies ist vergleichbar mit Präpozessor-Anweisungen von C/C++ oder der Bytecode-Anreicherung von Java Data Objects (JDO).
- Meta-Informationen, die als solche in den Bytecode übernommen werden und zur Laufzeit über das Java Reflection API ausgelesen werden können. Diese dienen dazu, Frameworks und Containern Informationen über die Klasse bereitzustellen.

Info

Bei den *Java Data Objects (JDO)* handelt es sich um eine Java-EE-Spezifikation, die es gestattet, JavaBeans auf einfache Weise mit Datenbanken zu synchronisieren. Sie wurde vor der Einführung des Persistenz API vor allem dort eingesetzt, wo die Größe der Anwendung den Einsatz von Entity Beans nicht rechtfertigte.
Mit Java EE 5 und dem enthaltenen Persistenz API verliert diese Technologie zunehmend an Bedeutung.

Bei den EJB-Annotationen handelt es sich natürlich um die zuletzt genannte Art und Sie teilen dem Application Service die folgenden Bereitstellungsinformationen mit:

Listing 8.4
Annotationen in Listing 8.2

```
...
import javax.ejb.Remote;
import javax.ejb.Stateless;
...
@Stateless
@Remote(de.akdabas.javaee.ejb.HelloWorld.class)
public class HelloWorldBean implements HelloWorld, Serializable {
...
```

- `@Stateless` – bei dieser EJB handelt es sich um eine Stateless Session Bean.
- `@Remote` – die Klasse, die als Remote Interface für diese EJB dient.

Sie werden den verschiedenen Annotationen in diesem und den nächsten Kapiteln noch wiederbegegnen, da nahezu alle neuen Technologien der Java Enterprise Edition sich hierdurch konfigurieren lassen. Der große Vorteil liegt dabei darin, dass Information und Meta-Information in einer Datei abgelegt und zusammen gewartet werden können.

Die Remote Exception

Da sich EJB und Client auch auf unterschiedlichen Servern befinden können und somit über RMI miteinander kommunizieren müssen, ist es auch möglich, dass es dabei zu so genannten `RemoteExceptions` kommt. Sie können diese im Interface deklarieren, um den Client anzuhalten, eine entsprechende Behandlungsroutine bereitzustellen.

Das Auslösen dieser `Exception` ist jedoch dem Application Server vorbehalten. Wenn Ihre Geschäftslogik eigene Ausnahmen benötigt, deklarieren Sie diese zusätzlich.

Das Interface Serializable

Schließlich implementiert die Session Bean in Listing 8.2 das Marker-Interface `Serializable`. Sie sollten dies tun, um nachfolgende Programmierer darauf aufmerksam zu machen, dass sich Objekte dieses Typs per RMI übertragen lassen müssen. Wann immer Sie auf eine solche Klasse stoßen, sollten Sie zwei Dinge im Kopf behalten:

Ein Marker-Interface dient in der Regel dazu, Objekte einer Objektfamilie zuzuordnen, ohne diese zu verpflichten, von etwaigen Basisklassen ableiten zu müssen.

1. Nicht alle Objekte sind serialisierbar! So stellen beispielsweise Datenbankverbindungen (`Connection`) eine lokale Referenz dar, die nicht auf einen anderen Rechner übertragen werden kann.

   Wenn eine Klasse eine nicht serialisierbare Objektvariable enthält, kann diese nicht übertragen werden, bis Sie die entsprechende Variable als `transient` kennzeichnen.

2. Serialisierbare (übertragbare) Klassen sollen verhältnismäßig kleine Objekte sein, weil sonst bei der Übertragung auf einen anderen Rechner viel Overhead erzeugt wird.

Info

Als Marker-Interface bezeichnet man ein Interface, das die implementierende Klasse nicht zur Implementierung einer oder mehrerer Methoden verpflichtet. Der Rumpf eines solchen Interface ist bis auf etwaige Konstanten leer.
Mit Java 5 könnte auch diese Aufgabe prinzipiell von Annotations übernommen werden.

Bereitstellung und Implementierung

Auch die Bereitstellung der EJB hat sich dank des Einsatzes von Java Annotations stark vereinfacht:

- Kompilieren Sie Remote Interface und Session Bean in *class*-Dateien.
- Verpacken Sie die Klassen in ein Java-Archiv (*jar*-Datei).
- Kopieren Sie die Datei in das Verzeichnis *$JBOSS/server/default/deploy*.
- Starten Sie den Application Server.

Der JBoss untersucht in der Standardinstallation automatisch das Verzeichnis *deploy* nach bereitzustellenden EJBs und macht diese über seinen Namensdienst unter dem Namen `BeanName/remote` verfügbar.

Um also mithilfe von JNDI auf die Bean zugreifen zu können, müssen Sie nur das folgende Code-Fragment in Ihren Client – z.B. ein Servlet – einfügen.

Listing 8.5
Code-Fragment, um auf die EJB zugreifen zu können.

Die in diesem Listing verwendete Klasse Lookup haben Sie in Kapitel 7 entwickelt.

```
...
import de.akdabas.javaee.jndi.Lookup;
import de.akdabas.javaee.ejb.HelloWorld;
...
   // Zugriff auf die EJB über den Namensdienst
   HelloWorld bean = (HelloWorld)
      Lookup.lookup(Lookup.JBOSS_ENV, "HelloWorldBean/remote");
...
   // Ab hier können Sie die EJB wie eine "ganz gewöhnliche" Bean
   // verwenden.
   System.out.println(bean.getGreeting());
...
```

8.6.4 Hello World – eine Stateful Session Bean

Bevor Sie sich gleich einem praktisch anwendbaren Client in Form eines Servlet widmen und damit Ihre Webanwendung mit EJBs verknüpfen, werden wir das Beispiel aus dem vorangegangenen Absatz zu einer vollwertigen Stateful Session Bean erweitern.

Das Interface

Da der Client nicht entscheiden kann, ob es sich bei der aktuellen EJB um eine Stateless Session Bean oder ihr Pendant mit einem Konversationsgedächtnis handelt, unterscheiden sich die Remote Interfaces auch nur wenig voneinander.

Listing 8.6
Remote Interface der Stateful Session Bean

```
package de.akdabas.javaee.ejb;
import java.rmi.RemoteException;

/**
 * Remote Interface der Stateful Session Bean;
 * Dieses Interface wird allen Clients zur Verfügung gestellt, die
 * später auf den Dienst zugreifen sollen.
 */
public interface HelloWorld {
```

Listing 8.6 (Forts.)
Remote Interface der Stateful Session Bean

```
    /**
     * Dies Methode erlaubt es die Grußnachricht zu personalisieren
     */
    public void setName(String name) throws RemoteException;

    /**
     * Gibt die Grußbotschaft zurück
     */
    public String getGreeting() throws RemoteException;
}
```

Die zusätzliche Methode soll es dem Client dabei erlauben, die zurückgegebene Grußbotschaft zu personalisieren.

Die Session Bean-Implementierung

Auch die erforderlichen Änderungen an der Bean-Implementierung sind marginal.

Listing 8.7
Stateful Session Bean-Implementierung

```
package de.akdabas.javaee.ejb;

import javax.ejb.Remote;
import javax.ejb.Stateful;
import java.io.Serializable;

/**
 * Implementierung der Stateful Session Bean (EJB)
 */

@Stateful
@Remote(de.akdabas.javaee.ejb.HelloWorld.class)
public class HelloWorldBean implements HelloWorld, Serializable {

    /** Diese Variable speichert den Zustand der EJB */
    private String name = "Unbekannter Benutzer";

    /**
     * Dies Methode erlaubt es die Grußnachricht zu personalisieren
     */
    public void setName(String name) {
      this.name =  name;
    }

    /**
     * Gibt die Grußbotschaft zurück
     */
    public String getGreeting() {
       return "Hello " + name + "!";
    }
}
```

Bereitstellung

Auch bei Stateless Session Beans beschränkt sich die Bereitstellung dank des Einsatzes von Java Annotations auf vier Schritte: Übersetzen der Java-Klassen, Verpacken in eine *jar*-Datei, Kopieren der Datei ins Verzeichnis *$JBOSS/server/default/deploy* und Starten des Servers.

8.6.5 Ein Client

Zum Abschluss dieses Kapitels werden Sie schließlich einen vollwertigen Client in Form eines Servlet schreiben, welches die EJB verwendet, um eine HTML-Seite zu erzeugen.

Listing 8.8 Ein Servlet-Client

```
package de.akdabas.javaee.servlets;

import de.akdabas.javaee.jndi.Lookup;
import de.akdabas.javaee.ejb.HelloWorld;

import java.io.*;
import javax.servlet.http.*;
import javax.servlet.ServletException;

/** Dieses Servlet greift über JNDI auf eine EJB zu */
public class HelloWorldEJB extends HttpServlet {

   /** Erzeugt eine einfache HTML-Seite */
   public void doGet(HttpServletRequest request,
                     HttpServletResponse response)
      throws IOException, ServletException {

      response.setContentType("text/html;charset=ISO-8859-1");
      PrintWriter out = response.getWriter();

      // Auslesen der Stateful Session Bean 'HelloWorld'
      HelloWorld ssb = (HelloWorld)
          Lookup.lookup(Lookup.JBOSS_ENV, "HelloWorldBean/remote");

      // Setzen des Benutzernamens
      ssb.setName("Thomas");

      // Erzeugen von HTML-Code ...
      out.println("<html><body>");
      out.println("  Verwendung einer Stateless Session Bean!");
      out.println("  Resultat der EJB: " + ssb.getGreeting());
      out.println("</body></html>");
   }
}
```

Um dieses Servlet zu übersetzen, benötigen Sie natürlich zumindest das Remote Interface `HelloWorld` der EJB. Am einfachsten verwenden Sie dafür die zuvor erstellte *jar*-Datei. Bei der Bereitstellung der Mini-Webanwendung im JBoss ist jedoch darauf zu achten, dass die EJB-Klassen kein Bestandteil der resultierenden WAR-Datei sind, weil es sonst aufgrund der ClassLoader-Struktur des JBoss zu einer `ClassCastExceptions` kommen kann.

Info

Wenn Sie beim Aufruf des Servlet mit `ClassCastExceptions` konfrontiert werden, überprüfen Sie, dass die EJB-Klassen kein Bestandteil der Webanwendung sind.

Abbildung 8.9
Resultat Ihrer EJB

Enterprise Archive EAR

Wenn Sie Webanwendungen und EJBs gemeinsam bereitstellen möchten, können Sie diese zu einem Enterprise Archive (EAR) zusammenfügen. Dabei handelt es sich um eine ZIP-Datei mit der Endung `.ear`, die neben verschiedenen Webarchiven (WAR) und EJB-JARs auch ein Verzeichnis *META-INF* und darin eine Datei *application.xml* enthält. Letztere bildet einen Deployment Descriptor, der dem Application Server mitteilt, wie die einzelnen enthaltenen Anwendungen bereitzustellen sind.

Ein Enterprise Archive, welches die WAR-Datei und den EJB-Service dieses Beispiels zusammenfasst und über die Webanwendung unter der URL kap08 verfügbar macht, könnte beispielsweise folgende Form haben.

Listing 8.9
Deployment Descriptor für ein Enterprise Archive (EAR)

```
<application>
   <!-- Symbolischer Name dieser Enterprise Anwendung -->
   <display-name>Masterclass Java EE</display-name>

   <!-- Konfguration des EJB Moduls -->
   <module>
      <ejb>kap08.jar</ejb>
   </module>

   <!-- Einbinden der Webanwendung unter der URL 'kap08' -->
   <module>
      <web>
         <web-uri>kap08.war</web-uri>
         <context-root>/kap08</context-root>
      </web>
   </module>
</application>
```

8.7 Restriktionen bei der Implementierung von EJBs

Bevor wir dieses Kapitel mit einer kurzen Zusammenfassung abschließen, möchte ich Sie noch auf einige Beschränkungen hinweisen, auf die Sie bei der Erstellung von EJBs achten müssen. Damit Ihre Enterprise JavaBean sicher von einem EJB-Container verwaltet werden kann, darf sie unter keinen Umständen eines der folgende Dinge tun:

- Mit Objekten aus dem Package `java.io` bzw. `java.nio` auf Dateien zugreifen
- Einen `ServerSocket` erstellen oder manipulieren
- Eigene Threads erstellen oder starten
- Native Bibliotheken laden
- Über Klassen des AWT oder Swing direkt mit dem Anwender in Verbindung treten

Diese Beschränkungen sind nötig, da eine vom EJB-Container verwaltete EJB keine Annahmen darüber machen kann, wann und in welchem Kontext sie erstellt wird.

8.8 Zusammenfassung

Dieses Kapitel zeigte Ihnen die grundlegenden Techniken, die notwendig sind, um Enterprise Java Beans 3 mit dem JBoss Application Server bereitzustellen. Dabei haben wir uns bewusst auf den einfachsten Weg beschränkt, um Sie einerseits nicht mit Konfigurationsanweisungen zu überfluten und andererseits nicht vom Einsatz dieser Technologie abzuschrecken.

EJBs waren und sind ein Kernbestandteil der Java-EE-Spezifikation und waren aufgrund der angesammelten Altlasten früherer Versionen auch häufig ein zentraler Kritikpunkt. Doch Sun hat aus den Fehlern der Vergangenheit – umfangreicher Lebenszyklus von EJBs, viele notwendige Konfigurationsdateien etc. – gelernt und Ihnen Mittel an die Hand gegeben, um ohne viel Mehraufwand leistungsfähige Komponenten zu erstellen, die von verschiedenen Anwendungen parallel verwendet werden können.

Der Servicegedanke steht bei EJBs mehr im Vordergrund als bei jeder anderen Technologie. Statt Utility-Klassen zu schreiben und diese in Form von Bibliotheken in jede Webanwendung einzeln einzubinden, können Sie diese nun als Service etablieren und gemeinsam nutzen, verbunden mit dem Vorteil, dass Sie den Service nur an einer einzigen Stelle administrieren müssen und z.B. auch in einem Cluster bereitstellen können.

Zusammen mit den Technologien der nächsten beiden Kapitel bilden Enterprise JavaBeans einen wichtigen Grundpfeiler für robuste, leistungsfähige Anwendungen.

9

Java Message Service

Egal, ob zwischen Browser und Servlet, Client und Enterprise JavaBean oder Applikation und Persistenzschicht, der Austausch von Informationen spielt im Alltag eines Programmierers eine große Rolle. Je verteilter das System, desto zuverlässiger muss die Nachrichtenübermittlung arbeiten, denn ein »Vielleicht« ist bei der Zustellung einer Nachricht inakzeptabel.

Wenn wir bisher von miteinander kommunizierenden Komponenten sprachen, meinten wir in der Regel, dass eine Komponente Methoden einer anderen Komponente aufruft und dann auf die Antwort wartet. Dies ist vergleichbar mit einem Telefongespräch, bei dem nur einer der beiden Partner reden kann, während der andere warten muss. Da sich beide Kommunikationsteilnehmer in gewisser Weise synchronisieren müssen, wird diese Art der Kommunikation als *synchrone Kommunikation* bezeichnet.

Achtung

Wir werden in diesem Kapitel manchmal eine Brücke zur gewohnten Kommunikation via E-Mail schlagen, um die Arbeitsweise nachrichtenbasierter Systeme zu verdeutlichen. Nichtsdestotrotz ist der *Java Message Service* kein Ersatz für das JavaMail-API. Stattdessen wird die hier beschriebene Technik für die Kommunikation von getrennten Prozessen und zur asynchronen Abarbeitung von Befehlen (z.B. in großen Datenbanksystemen) verwendet.

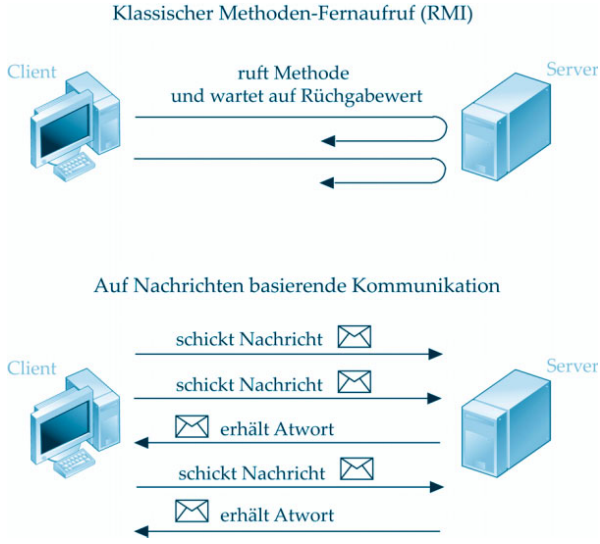
Wenn Sie beispielsweise an ein Servlet denken, das eine bestimmte HTML-Seite bereitstellt, muss dieses auf einen entsprechenden Request warten, auf den es antworten kann. Der Browser des Clients wiederum ist nach dem Absenden eines Request so lange blockiert, bis er die Antwort des Servers empfangen hat. Ist das Servlet gerade beschäftigt, kann er nichts anderes tun als warten.

Beim Messaging hingegen spricht man von *asynchroner Kommunikation*, weil der Informationsaustausch zwischen beiden Parteien nicht notwendigerweise in regelmäßigem Wechsel, sondern versetzt erfolgt: ein Verfah-

ren, das eher an einen Briefwechsel oder eine E-Mail-Kommunikation erinnert. Dabei kann der Informationssender seine Nachricht verfassen, abschicken und sich weiteren Aufgaben widmen, während der Informationsempfänger nur regelmäßig sein Postfach auf neue Eingänge überprüfen muss.

Die Idee hinter Message-Systemen ist dabei ganz einfach: Statt dass Sender und Empfänger eines Methodenaufrufs direkt miteinander in Verbindung treten, wie es etwa bei der *Remote Method Invocation (RMI)* der Fall ist, kommunizieren beide mit einem Vermittler, dem so genannten *Message Broker*. Dieser nimmt die Anforderung entgegen und stellt sicher, dass diese zu einem späteren Zeitpunkt dem Empfänger zugestellt wird.

Abbildung 9.1
Klassischer Methodenfernaufruf vs. Messaging



9.1 Asynchrone Kommunikation

Obwohl nachrichtenbasierte Kommunikation ein sehr altes Thema der Informatik ist und die Wurzeln *Message Oriented Middleware (MOM)* bis in die frühen achtziger Jahre zurückreichen, gab es bis 1998 kein einheitliches Java API für die Kommunikation mit Message Broker-Systemen. Zwar bieten einige Hersteller eigene Bibliotheken für den Zugriff auf ihr System an und hin und wieder gab es auch eigene Implementierungen, den großen Durchbruch für Java-Entwickler brachte jedoch erst Suns Spezifikation für den *Java Message Service (JMS)*.

Die Herausforderung bestand diesmal nicht in der Schaffung einer guten Referenzimplementierung, sondern darin, ein API zu definieren, das eine möglichst große Anzahl bereits etablierter Systeme unterstützen kann. So entstand eine sehr allgemeine und stark abstrahierende Bibliothek, die von Dienst zu Dienst mit individuellen Eigenschaften versehen werden kann.

Inzwischen ist JMS ein fester Bestandteil des Java-EE-Kerns und Grundlage für weitere Technologien, wie die *Message Driven Beans*, die Sie am Ende dieses Kapitels kennen lernen werden. Und so fordert Sun inzwischen von jedem Java-EE-konformen Application Server, dass dieser mindestens über einen Message Service verfügt.

9.1.1 Download und Installation

Wie das Java Naming and Directory Interface ist auch der Java Message Service kein Bestandteil der Java-EE-Distribution, sondern kann als eigenständiges Produkt inklusive Spezifikation und Tutorial über die JMS-Webseite *http://java.sun.com/products/jms/* heruntergeladen werden.

Auch eine kostenlose Referenzimplementierung bleibt Sun Ihnen in diesem Fall schuldig und so bleibt Ihnen (sofern Sie nicht anderweitig über einen Message Broker verfügen) nichts anderes übrig, als den auch schon im vorangegangenen Kapitel verwendeten JBoss oder eine andere freie Implementierung wie beispielsweise *OpenJMS* (*http://openjms.sourceforge.net/*) zu verwenden.

9.1.2 Message Oriented Middleware

Den Kern nachrichtenorientierter Kommunikation bildet der so genannte Message Broker, der die Nachrichten des Senders entgegennimmt und für ihre Zustellung verantwortlich ist. Dafür stellt er verschiedene Kanäle bereit, die die Funktion von E-Mail-Adressen erfüllen.

Das System aus Message Broker und Nachrichtenkanälen wird dabei als Message Oriented Middleware bezeichnet. Deren Konfiguration und Basis unterscheidet sich von Anbieter zu Anbieter, lediglich der Zugriff aus Java-Programmen heraus ist standardisiert.

9.1.3 Vorteile asynchroner Kommunikation

Asynchrone Kommunikation hat gegenüber synchroner vor allem folgende Vorteile:

- Lose Kopplung der Systeme

  Wenn Sie einen Dienst auf zwei unterschiedliche Systeme verteilen, die beispielsweise über `ServerSockets` oder Remote Procedure Invocation (RMI) *synchron* miteinander kommunizieren, und eines der beiden Systeme ausfällt, ist davon gleichzeitig auch das andere System betroffen.

  Bei asynchroner, nachrichtengesteuerter Kommunikation hingegen arbeiten beide Systeme vollkommen autark und können auch getrennt voneinander reinitialisiert werden. Vom anderen System hinterlegte Nachrichten verbleiben bis zur Zustellung im System, lediglich die Auslieferung verzögert sich.

- Transparente Aufgabenverteilung

  Da der Sender einer Nachricht diese nicht an einen konkreten Empfänger sendet, sondern lediglich in einem »Eingangskorb« ablegt, kann er auch nicht entscheiden, wer die Nachricht letztendlich entnimmt und bearbeitet. Für ihn ist es vollkommen transparent, ob am anderen Ende der Leitung ein einzelner oder eine Gruppe von gleichberechtigten Empfängern die Bearbeitung der Informationen übernimmt.

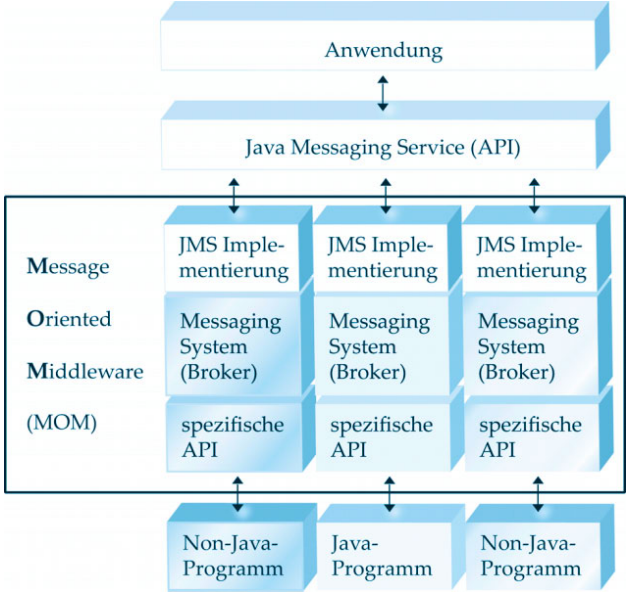
  Hierdurch können auf einfache Art und Weise gut skalierende Systeme implementiert werden, die anstehende Aufgaben zunächst sammeln und ab einer genügend großen Anzahl in einem Rutsch erledigen. Ein Beispiel hierfür sind Datenbank- oder Logging-Operationen.
- Hohe Integrationsfähigkeit

  Durch die lose Kopplung ist Messaging vor allem für die plattform- und programmiersprachenübergreifende Kommunikation von Anwendungen geeignet. So kann ein verteiltes System leicht um neue Komponenten erweitert werden oder bestehende Teile lassen sich austauschen und durch andere Systeme ersetzen.

9.2 Das Konzept

Analog zum Java Naming and Directory Interface ist JMS als allgemeines API spezifiziert, dessen Implementierung vom Anbieter eines Message-Diensts gestellt wird. Die Anbieter implementieren auf Grundlage eines Service Provider Interface (SPI) Adapter, die z.B. auch den Zugriff auf Message-Dienste anderer Programmiersprachen, wie etwa *IBMs MQSeries*, ermöglichen.

Abbildung 9.2
Aufbau und Zugriff auf Messaging-Systeme



Im Gegensatz zu JNDI werden durch die JMS-Spezifikation allerdings herstellerspezifische Erweiterungen ermöglicht und sogar erwünscht.

9.2.1 Kommunikationspartner

Eines der wichtigsten Konzepte beim Aufbau eines nachrichtenbasierten Systems ist die Anonymität. Sender und Empfänger kennen einander nicht, sondern kommunizieren einzig und allein mit ihrem Request-Broker. Zwischen diesen Brokern können dann wiederum weitere Broker liegen.

Da somit auch kein klassisches Frage-Antwort-Schema entsteht, bei dem auf einen Request eine Response erfolgen muss, verschwimmen auch die Grenzen zwischen Client und Server. Oftmals sind Nachrichtenempfänger gleichzeitig auch Sender und umgekehrt. Um dies zu verdeutlichen, werden wir in diesem Kapitel nicht mehr zwischen Client und Server unterscheiden, sondern nur noch von Nachrichtensendern (*Message Producer*) und Nachrichtenempfängern (*Message Consumer*) sprechen.

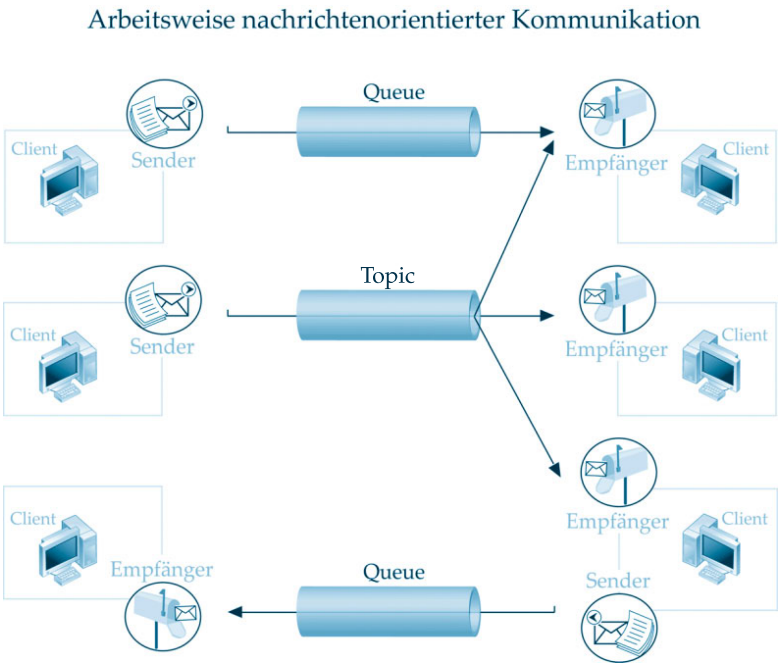


Abbildung 9.3
Kommunikation via Java Message Service

Zwar kann ein Nachrichtenempfänger nicht zwischen verschiedenen Sendern unterscheiden, wohl aber zwischen den verschiedenen Kanälen, über die er Nachrichten empfangen kann. Diese sind in zwei Gruppen (*Queue* und *Topic*) aufgeteilt, die zu unterschiedlichen Messaging-Konzepten gehören.

9.2.2 Nachrichtenkonzepte

Wenn Sie eine Frage oder ein bestimmtes Implementierungsproblem haben, können Sie in der Regel auf zwei Wegen zu einer Lösung gelangen:

- Sie kennen die E-Mail-Adresse von jemandem, der die Antwort mit Sicherheit weiß und Ihnen helfen wird.
- Sie kennen ein Forum, in dem Probleme dieser Art diskutiert werden.

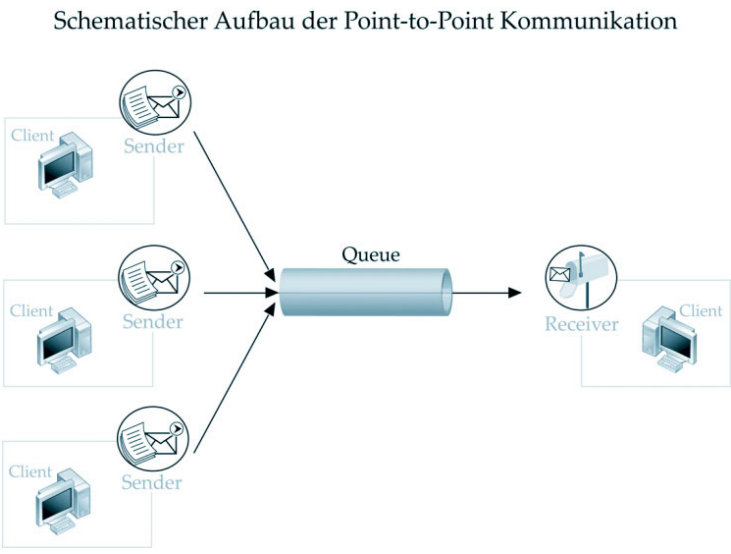
Im ersten Fall verfassen Sie eine konkrete (persönliche) E-Mail an den Empfänger und hoffen auf baldige Antwort. Da bei dieser Art der Kommunikation sowohl Ausgangs- als auch Endpunkt genau definiert sind, wird sie als Punkt-zu-Punkt-Kommunikation (*Point-to-Point*) bezeichnet.

Im zweiten Fall versuchen Sie, das Problem einerseits zu verallgemeinern und trotzdem möglichst genau zu beschreiben, und veröffentlichen es beispielsweise in einer Newsgroup (*Publish*). Diese kann von verschiedenen Empfängern abonniert werden (*Subscribe*), wobei einer der Abonnenten Ihnen hoffentlich weiterhelfen kann. Eine Besonderheit dieses *Publish-and-Subscribe*-Systems ist, dass Sie auf eine Anfrage unter Umständen mehrere Antworten erhalten.

Point-to-Point-Kommunikation

Beim *Point-to-Point*-Modell wird Ihre Nachricht an genau einen Empfänger bzw. an genau ein dafür konfiguriertes Postfach versandt. Der Empfänger kann den Sender der Nachricht allerdings nicht direkt identifizieren und da der Empfänger in der Regel nur eine Nachricht bearbeiten kann, bevor er sich der nächsten zuwendet, gilt die Regel: Wer zuerst schreibt, wird zuerst bearbeitet – alle anderen stellen sich hinten an. Aufgrund dieses Verhaltens wird auch der Kanal, über den diese Nachricht versendet wird, als `Queue` (dt. Warteschlange) bezeichnet.

Abbildung 9.4
Point-to-Point-Kommunikation



Publish-and-Subscribe

Beim *Publish-and-Subscribe*-Modell müssen Sie lediglich darauf achten, die Nachricht an die richtige Gruppe von Konsumenten zu schicken. Da sich jede dieser Gruppen in der Regel mit genau einem Thema beschäftigt, werden diese `Topic` (dt. Thema) genannt.

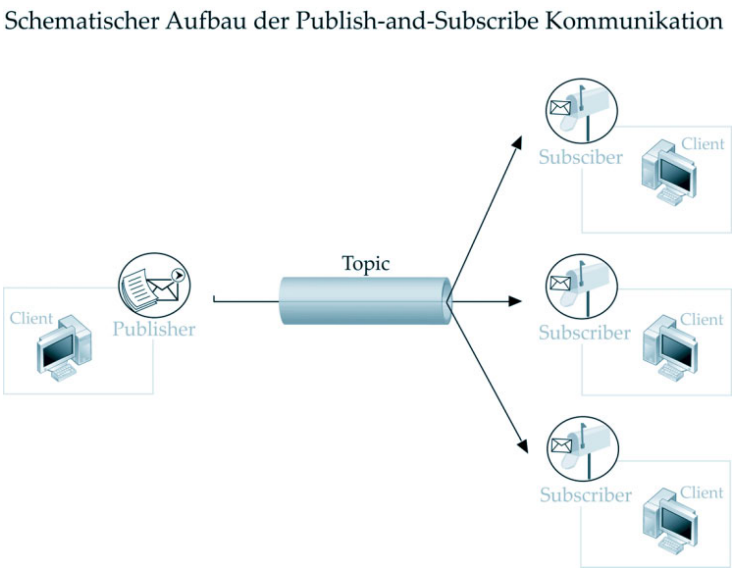


Abbildung 9.5
Publish-and-Subscribe-Modell

Unterstützung der unterschiedlichen Kommunikationsmodelle

Da nicht alle Nachrichtendienste beide Kommunikationsmodelle unterstützen, enthält das API immer zwei Ausprägungen aller Klassen und Interfaces, die von einer gemeinsamen Basisklasse ableiten. So gibt es beispielsweise eine Klasse `TopicConnection` für Verbindungen zu einem Publish-and-Subscribe-Service und eine Klasse `QueueConnection`, die für Point-to-Point-Verbindungen zuständig ist. Da beide Klassen jedoch ähnlich arbeiten, verwenden die folgenden Beispiele mal die eine, mal die andere Implementierung.

9.2.3 Konfiguration und Eigenschaften der Dienste

Über einen Message Service können in der Regel beliebig viele Topics bzw. Queues gleichzeitig verwaltet werden, wobei die Konfiguration der Kanäle sich von Anbieter zu Anbieter unterscheidet. Unterschieden werden die einzelnen Kanäle dabei, ähnlich einer E-Mail-Adresse, nach unterschiedlichen Namen, die in folgendem JNDI-Kontext bzw. Subkontexten definiert werden sollten:

Listing 9.1 JNDI-Kontext für JMS-Services

```
// Standard JNDI-Kontext für Objekte des Java Message Service
java:comp/env/jms
```

Mehr über den JNDI-Kontext und die Bedeutung seiner Pfade erfahren Sie im Kapitel über das *Java Naming and Directory Interface (JNDI)*.

9.3 Bestandteile des API

Bevor Sie sich an das Versenden von Nachrichten machen, lernen Sie in diesem Absatz jedoch zunächst die wichtigsten daran beteiligten Klassen und ihre Funktion kennen. Dabei besprechen wir immer das Basis-Interface, welches später durch die eine der beiden Ausprägungen (siehe Abschnitt 9.2.2) erweitert wird. Beide Ausprägungen verhalten sich gegenüber dem Client identisch.

9.3.1 javax.jms.ConnectionFactory

Die Klasse `ConnectionFactory` dient dazu, Verbindungen (`Connection`) zum Nachrichten-Service aufzubauen. Häufig wird die Klasse `ConnectionFactory` über einen Namens- oder Verzeichnisdienst zur Verfügung gestellt, obgleich nicht alle MOM-Hersteller einen solchen Namensdienst unterstützen.

Die `ConnectionFactory` hat außerdem die Aufgabe, dem Administrator die Möglichkeit zur Konfiguration des Dienstes zu geben, weswegen sie auch als *Administered Object* bezeichnet wird.

9.3.2 javax.jms.Connection

Achtung

Um keine Verwirrungen aufkommen zu lassen, sei an dieser Stelle noch einmal ausdrücklich darauf hingewiesen, dass es sich bei den Erklärungen wie E-Mail und Internetanschluss um Vergleiche mit umgänglichen Technologien handelt, die die Situation anschaulich darstellen sollen. Der Java Message Service hat weder etwas mit dem Versenden von E-Mails noch mit einem Zugang zum Internet zu tun, sondern dient einzig und allein der Kommunikation zwischen (Java-)Programmen.

Eine `Connection` ist vergleichbar mit einer geöffneten Datenbankverbindung (`java.sql.Connection`). Sie wird über die Klasse `ConnectionFactory` erzeugt und ermöglicht Ihnen die Kommunikation mit anderen Teilnehmern.

Mit einer geöffneten `Connection` ausgestattet, sind Sie im Message Service unter einer eindeutigen Adresse (ähnlich einer IP-Adresse) bekannt und können sich nun der Aufgabe widmen, Nachrichten zu versenden oder zu empfangen. Mit einer geöffneten `Connection` besitzen Sie an dieser Stelle zwar einen Internetanschluss, Sie verfügen allerdings noch nicht über eine gültige E-Mail-Adresse.

9.3.3 javax.jms.Session

Eine geöffnete Verbindung (`Connection`) ermöglicht es Ihnen nun, eine oder mehrere *Sessions* zu öffnen. Eine `Session` bildet dabei eine konkrete Verbindung mit einem oder mehreren (Nachrichten-)Kanälen. Dabei wird eine `Session` unter folgenden Gesichtspunkten konfiguriert:

1. Transaktionskontrolle

   Über diesen Parameter entscheiden Sie, ob die Session an Transaktionen teilnehmen soll oder nicht. Diese sind vergleichbar mit Datenbanktransaktionen.

2. Bestätigungsmodus

   Natürlich kann es sowohl für den Sender (*Producer*) als auch für den Message Service wichtig sein, zu erfahren, ob die verschickte Nachricht inzwischen beim Empfänger (*Consumer*) angekommen ist und erfolgreich verarbeitet werden konnte oder ob sie sich noch in irgendeiner Warteschlage befindet und auf ihre Zustellung wartet. Mit dieser Einstellung legen Sie fest, wer für die Bestätigung zuständig ist.

Um eine Nachricht zu versenden, benötigen Sie anschließend drei Objekte, die Sie alle über das `Session`-Objekt erzeugen können.

- Eine Zieladresse (`Destination`)

  Damit die Nachricht zugestellt werden kann, benötigt sie ein konkretes Ziel (`Destination`). Dieses erhalten Sie in der Regel über einen Namens- oder Verzeichnisdienst, der auch als Adressbuch funktioniert.

  Bei der Zieladresse handelt es sich dabei, je nach verwendetem Kommunikationsmodell, entweder um eine `Queue` oder eine `Topic`.

- Einen Sender (`MessageProducer`)

  Dieser hat die Aufgabe, die Nachricht zu versenden. Dabei versucht er zunächst, die angegebene Zieladresse zu erreichen und die Nachricht zu übermitteln. Gelingt ihm dies nicht, hinterlässt er die Mitteilung, dass eine Nachricht vorliegt und abgeholt werden kann, oder er versucht es nach einer gewissen Zeitspanne noch einmal.

  Über den `MessageProducer` können Sie die Nachricht auch konfigurieren und ihr beispielsweise eine Priorität zuordnen oder die Zeitspanne angeben, bis zu der die Nachricht ausgeliefert sein muss (*TimeOut*). Wird diese Zeitspanne überschritten, verfällt die Nachricht und wird vom `MessageProducer` ohne Auslieferung gelöscht.

- Eine Nachricht (`Message`)

  Welchen Sinn hätte die Kommunikation ohne Objekte, die Sie übermitteln wollen. Der zentrale Bestandteil bleibt natürlich die Nachricht selbst. Dabei ist die Nachricht einer der wenigen Bestandteile des Framework, bei dem Sie nicht zwischen Point-to-Point-Kommunikation und Publish-and-Subscribe-Modell unterscheiden müssen. Eine Information bleibt eben eine Information.

Um Nachrichten empfangen zu können, erzeugen Sie mit dem `Session`-Objekt einen Empfänger (`MessageConsumer`), dem eine bestimmte `Topic` oder `Queue` zugeordnet sein muss und der auf eingehende Nachrichten wartet. Schließlich ermöglicht die Session es Ihnen auch, über einen `QueueBrowser` die in einer `Queue` gespeicherten Nachrichten aufzulisten oder nach bestimmten Kriterien zu filtern.

9.3.4 javax.jms.Destination

Eine `Destination` ist ein Zielpunkt, an den Nachrichten weitergeleitet werden. Er besteht entweder aus einer `Queue`, dem Pendant einer E-Mail-Adresse, oder einer `Topic`, was einer Newsgroup entspricht. Auch diese Objekte sind, wie die `ConnectionFactory`, zentrale Bestandteile der Infrastruktur und werden deshalb in der Regel von einem Administrator verwaltet und angelegt (Administered Objects).

9.3.5 javax.jms.Message

Die `Message` ist der Kernbestandteil der Kommunikation, um die sich beim JMS letztendlich alles dreht. Sie dient dem Austausch von Nachrichten und Objekten und kann sogar zum Versand von XML-Dokumenten verwendet werden. Und zwar über verschiedene Computer, Betriebssysteme und Programmiersprachen hinweg.

Jede `Message` setzt sich dabei aus folgenden drei Bestandteilen zusammen:

- Header

  Haben Sie sich schon einmal die Header einer empfangenen E-Mail ausgeben lassen? Im Kopf einer Nachricht hinterlässt dabei jeder am Versand beteiligte Server (für den Nutzer meist unsichtbare) Informationen, über die eine Nachricht identifiziert und ihr Weg (Route) rekonstruiert wird.

- Properties

  Während Sie beim Versand einer E-Mail in der Regel auf das Attribut `Priorität` beschränkt sind, ermöglicht es Ihnen JMS, jeder Nachricht eine beliebige Anzahl von Attributen mitzugeben, um diese beispielsweise beim Empfang nach beliebigen Kriterien filtern zu können.

- Body

  Der Body enthält den eigentlichen Inhalt der Nachricht. Dabei wird zwar nicht zwischen `Queue` und `Topic`, wohl aber nach dem Inhaltstyp unterschieden.

Die verschiedenen Nachrichtentypen

Die aktuelle JMS-Spezifikation unterscheidet zwischen fünf verschiedenen Ausprägungen einer Nachricht, deren Anwendungsgebiete sich teilweise überschneiden:

- `StreamMessage`

  Das Versenden einer Nachricht mit einer `StreamMessage` ist vergleichbar mit der synchronen Übertragung der Daten per `java.io.DataOutputStream`. Dieser Nachrichtentypus stellt Ihnen Methoden zur Verfügung, mit denen Sie elementare Datentypen und serialisierbare Objekte (`java.io.Serializable`) effektiv von A nach B übertragen können.

  Genau wie beim Einsatz von `DataOutputStream` müssen die Objekte in der gleichen Reihenfolge, in der sie geschrieben wurden, auch wieder ausgelesen werden. Sender und Empfänger müssen sich also vorher über diese einigen.

- `TextMessage`

  Diese Nachricht gleicht am ehesten einer E-Mail. Sie nimmt einen Text (`String`) auf und übermittelt diesen an den Empfänger. Dieser Nachrichtentypus ist außerdem dazu geeignet, XML-Dokumente zu übertragen. So existieren inzwischen auch Parser, die die enthaltenen Dokumente direkt aus einer `TextMessage` parsen.

- `ObjectMessage`

  Diese Klasse ermöglicht Ihnen die Übertragung eines beliebigen Java-Objekts, wobei dieses das Interface `java.io.Serializable` implementieren muss.

- `ByteMessage`

  Wenn Sie sich jetzt fragen, worin der Unterschied zwischen einer `ByteMessage` und einer `StreamMessage` liegt, kann ich sie beruhigen: Solange Sie lediglich mit (Ihren eigenen) Java-Clients kommunizieren, werden Sie diesen Nachrichtentypus nicht benötigen und können stattdessen immer auf eine `StreamMessage` zurückgreifen.

  Insbesondere bei der Kommunikation mit C/C++-Modulen gibt es jedoch Fälle, bei denen die Methoden der Klasse `StreamMessage` nicht ausreichen. So unterscheiden C/C++ beispielsweise zwischen dem Datentyp `char` und `unsigned char`, was eine `StreamMessage` nicht tut.

  Nachrichten, die über eine `ByteMessage` empfangen und verschickt werden, werden als Rohdaten behandelt und auf keine Weise von der Java Virtual Machine interpretiert.

- `MapMessage`

  Egal, ob es sich nun um Datenbankzugangsdaten oder die Ergebnisse einer statistischen Auswertung handelt: Manchmal besteht eine Nachricht aus nichts anderem als einer Menge von Attributen. Für diese Name-Wert-Paare ist mit der Spezifikation 1.1 die MapMessage eingeführt worden.

9.3.6 Zusammenfassung des API

Nachdem Sie nun alle Bestandteile des Nachrichtenverkehrs kennen, fasst die folgende Tabelle diese noch einmal in ihrer jeweiligen Ausprägung (`Topic` oder `Queue`) zusammen

Tabelle 9.1 Bestandteile des Nachrichtenversands

Allgemeine Basisklasse	Point-to-Point-Modell	Publish-and-Subscribe-Modell
`ConnectionFactory`	`QueueConnectionFactory`	`TopicConnectionFactory`
`Connection`	`QueueConnection`	`TopicConnection`
`Session`	`QueueSession`	`TopicSession`
`Destination`	`Queue`	`Topic`
`MessageProducer`	`QueueSender`	`TopicPublisher`
`MessageConsumer`	`QueueReceiver`	`TopicSubscriber`
`Message`	`StreamMessage`, `TextMessage`, `ObjectMessage`, `ByteMessage`, `MapMassage`	

9.4 Senden einer Nachricht

In diesem Abschnitt werden Sie eine erste Nachricht an eine Queue senden. Das Vorgehen beim Publish-and-Subscribe-Modell ist ganz analog, doch werden Sie sich dies für eine kleine Chat-Applikation später in diesem Kapitel aufsparen.

Info

Die Beispiele in diesem Kapitel greifen auf die Lookup-Klasse zurück, die Sie bereits in Kapitel 7 kennen gelernt haben.

9.4.1 Der schematische Ablauf

Bevor Sie sich in die Implementierung stürzen, wenden wir uns zunächst noch einmal dem schematischen Aufbau zu:

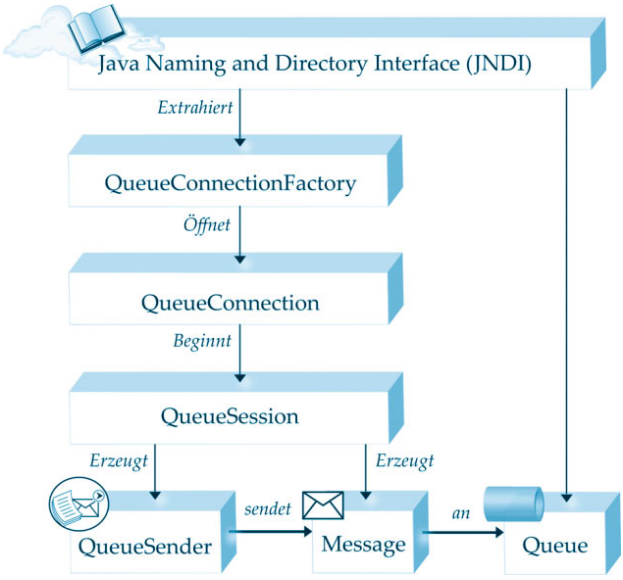


Abbildung 9.6
Schematischer Ablauf des Nachrichtenversands

Sowohl die `QueueConnectionFactory` wie auch die Zieladresse (`Queue`) werden dabei als *Administered Objects* vom Administrator des Service erstellt und über den Namensdienst verfügbar gemacht. Die `ConnectionFactory` ermöglicht es Ihnen, eine `Connection` zu öffnen und sich damit ins System »einzuwählen«. Dabei werden, abhängig von der Konfiguration des Dienstes, unter Umständen ein Benutzername und ein zugehöriges Passwort benötigt.

Über die geöffnete `Connection` können Sie anschließend beginnen, verschiedene Sessions zu starten. Eine `Session` ermöglicht es Ihnen, Nachrichtensender (`MessageProducer`) und Nachrichten (`Message`) zu erzeugen und diese an die vorgegebene Zieladresse (`Queue`) zu senden.

Wenn Sie sich jetzt fragen, warum das alles so komplex gestaltet ist und Sie Ihre Nachricht nicht beispielsweise direkt über eine `Connection` versenden und empfangen können, denken Sie daran, dass Sie innerhalb einer Applikation zwischen verschiedenen Nachrichten mit unterschiedlichen Prioritäten unterscheiden können: Einige Nachrichten sollen zusammen als Pulk versendet werden, wofür Sie eine Transaktion benötigen. Für andere können Sie sich die dafür benötigten Ressourcen sparen. Bei wichtigen Nachrichten kann es erforderlich sein, eine Empfangsbestätigung (in Form einer weiteren Nachricht) zu erhalten, etc. Um all diese komplexen Bedürfnisse mit einem API befriedigen zu können, muss dieses eben auch zwischen verschiedenen Kanälen unterscheiden können.

Eine `Session` ist mit einem E-Mail-Client vergleichbar, mit dem Sie Nachrichten verfassen und anschließend versenden oder Postfächer auf neue Nachrichten überprüfen können.

9.4.2 Ein Nachrichtensender (Message Producer)

Nachdem Sie den schematischen Ablauf kennen, müssen Sie diesen nur noch in Java gießen. Um das Beispiel übersichtlich zu halten und an gute Traditionen anzuknüpfen, handelt es sich bei Ihrer ersten Nachricht um eine `TextMessage` mit einer ganz bestimmten Botschaft.

Listing 9.2 Senden einer TextMessage (jms/MessageProducer.java)

```
package de.akdabas.javaee.jms;

import javax.jms.Queue;
import javax.jms.Session;
import javax.jms.Message;
import javax.jms.QueueSender;
import javax.jms.QueueSession;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.JMSException;

import de.akdabas.javaee.jndi.Lookup;

/** Erzeugt neue Text - Nachrichten */
public class MessageProducer {

   /** Sendet eine Nachricht an die Queue 'mcQueue' */
   public static void main(String [] args) {

      QueueConnection qConnection = null;
      QueueSession qSession = null;
      QueueSender qSender = null;
      try {

         // Auslesen der ConnectionsFactory aus dem Namensdienst
         QueueConnectionFactory qcFactory = (QueueConnectionFactory)
            Lookup.lookup(Lookup.JBOSS_ENV, "ConnectionFactory");

         // Auslesen der (konfigurierten) Ziel-Adresse (Queue)
         String destination = "queue/mcQueue";
         Queue queue = (Queue)
            Lookup.lookup(Lookup.JBOSS_ENV ,destination);

         // Aufbau der Verbindung
         qConnection = qcFactory.createQueueConnection();

         // Erzeugen der Session
         qSession = qConnection.createQueueSession(false,
            Session.AUTO_ACKNOWLEDGE);

         // Erzeugen des Senders
         qSender = qSession.createSender(queue);

         // Erzeugen der Nachricht
         String msg = "Hello Queue!";
         Message message = qSession.createTextMessage(msg);

         // Senden der Nachricht
         qSender.send(message);

      } catch (JMSException jmx) {
```

```
            jmx.printStackTrace();
        } finally {
            try { qSender.close(); } catch (Exception exc) { }
            try { qSession.close(); } catch (Exception exc) { }
            try { qConnection.close(); } catch (Exception exc) { }
        }
    }
}
```

Listing 9.2 (Forts.)
Senden einer TextMessage (jms/MessageProducer.java)

Vielleicht ist es Ihnen ja gar nicht aufgefallen, aber dieses Listing enthält nicht einmal den Operator `new`. Alle verwendeten Objekte sind entweder vorgegebene Administered Objects oder werden mithilfe solcher erzeugt. Ansonsten sollte Ihnen der Quelltext nach den vorangegangenen Bemerkungen keine Schwierigkeiten bereiten.

Freigeben von Ressourcen im finally-Block

Info

`finally`-Blöcke sind dazu gedacht, in einem `try`-Block beanspruchte Ressourcen wieder freizugeben. Der Java-Interpreter garantiert dabei, dass, sofern der `try`-Block betreten wurde, abschließend die Kontrolle an den `finally`-Block übergeben wird. Kommt es während der Verarbeitung des `try`-Blocks zu einer Ausnahme, so kann diese zunächst durch eine passende `catch`-Klausel behandelt werden. Anschließend wird, unabhängig davon, ob eine Ausnahme abgefangen oder weitergereicht wurde, zunächst der `finally`-Block ausgeführt. Die einzige Ausnahme von dieser Regel bildet der Aufruf `System.exit()`, wodurch der Java Interpreter angehalten und die Virtual Machine beendet wird.

An dieser Stelle sollten Sie sich einige Gedanken zum Thema Ressourcenmanagement machen: Da sowohl die `Connection` als auch die `Session` und der *Sender* eigene Threads im Speicher des Application Servers darstellen, belegen sie dabei eine ganze Reihe von Ressourcen. Zwar werden die Threads nach einem definierten TimeOut in der Regel beendet und die belegten Ressourcen wieder freigegeben. Doch auch zu viele schlafende Threads können einen Server in die Knie zwingen. Besonders häufig tritt dies auf, wenn ein Programm einen Fehler enthält und an einer bestimmten Stelle terminiert.

Sie sollten sich deshalb, wie bei einer Datenbankverbindung, darum bemühen, alle gestarteten Threads durch die Verwendung diverser `close()`-Methoden auch wieder zu beenden. Dabei müssen Sie vor allem auf die richtige Reihenfolge des Schließens achten (zum Beispiel die `Session` vor der `Connection`, die Sender vor der `Session` usw.) und wiederum eigene Try-Catch-Blöcke verwenden, da auch beim Schließen der einzelnen Ressourcen Fehler auftreten können.

Achtung

Um uns auf das Wichtigste konzentrieren zu können und die Listings einfach zu halten, werden wir die Fehler beim Schließen einer Verbindung in diesem Buch ignorieren. In Ihren Anwendungen sollten Sie diese jedoch zumindest ausgeben, um einen Ansatz für die Fehlersuche zu erhalten.

Bemerkungen zum Ressourcenmanagement

Bei der Frage, wann die geöffneten Ressourcen wieder freigeben werden sollten, begeben wir uns auf eine schwierige Gratwanderung: Zum einen benötigt der Aufbau einer Verbindung und die Erzeugung eines neuen `Session`-Thread natürlich Ressourcen, zum anderen können Sie Nachrichten auch über Kanäle (`Queue` / `Topic`) versenden, an denen gerade kein Empfänger aktiv lauscht. Diese verbleiben, solange ihre Lebensdauer nicht überschritten wird, in diesem Kanal und werden anschließend bei erneuter Verbindung des Clients zugestellt.

Empfängt oder sendet ein Client regelmäßig Nachrichten, empfiehlt es sich, seine Verbindung und die damit verbundenen Sessions offen zu halten und erst beim Abschluss der Operation zu beenden. Werden Nachrichten jedoch zum Beispiel empfangen, um (seltene) Vorgänge wie Fehlermeldungen zu protokollieren, und wird keine direkte Reaktion des Empfängers erwartet, können Sie auf ein aktives Lauschen des Clients verzichten.

9.4.3 Konfiguration der Queue

Natürlich müssen Sie die `Queue` und die `QueueConnectionFactory` als Administered Objects auch über Ihren Service Provider zur Verfügung stellen. Dies erledigen Sie beim JBoss durch folgenden Eintrag in die Datei *%JBOSS_HOME%/server/default/deploy/jms/jbossmq-destination-service.xml*, in der alle für das Messaging definierten Ziele als MBeans konfiguriert werden.

Info

Bei einer *MBean* handelt es sich um eine Komponente des Java Management Extension Framework (JMX), die die Aufgabe hat, alle Informationen und Operationen einer zu verwaltenden Komponente oder eines Service offen zu legen.

```
...
   <!-- JBoss Konfiguration der Queue 'mcQueue' -->
   <mbean code="org.jboss.mq.server.jmx.Queue"
          name="jboss.mq.destination:service=Queue,name=mcQueue">

      <depends optional-attribute-name="DestinationManager">
         jboss.mq:service=DestinationManager
      </depends>
   </mbean>
...
```

Listing 9.3
Konfiguration und Bereitstellung der Queue

Die Konfiguration von Zielen für JMS ist abhängig vom Container und nicht standardisiert. Beim JBoss befinden sich alle für JMS wichtigen Dateien unterhalb des Ordners *$JBOSS/server/default/deploy/jms*.

Damit wird eine Queue unter dem JNDI-Pfad *queue/mcQueue* bereitgestellt, die Sie in Listing 9.2 aus dem Namensdienst auslesen. Wenn Sie alles korrekt gemacht haben, erhalten Sie bei einem Neustart des JBoss-Servers anschließend folgende Ausgabe:

```
...
INFO  [mcQueue] Bound to JNDI name: queue/mcQueue
...
```

Listing 9.4
Statusmeldung beim Start des JBoss

9.4.4 Test der Anwendung

Nun steht einem Test nichts mehr im Wege. Hierfür übersetzen Sie die Klasse und führen sie anschließend mit folgenden Bibliotheken (JAR) im Verzeichnis */classes* aus:

```
java -cp ..\lib\jms.jar;..\lib\jndi.jar;..\lib\jbossall-cli-
ent.jar; de.akdabas.javaee.jms.MessageProducer
```

Listing 9.5
Test des Nachrichtenversands

Da Sie in Listing 9.2 keine weiteren Ausgaben definiert haben, können Sie eine erfolgreiche Übertragung zunächst nur daran erkennen, dass keine Fehlermeldung erscheint.

9.5 Empfangen einer Nachricht

Nachdem Sie nun durch (mehrmaliges) Ausführen der Klasse `MessageProducer` eine oder mehrere Nachrichten in der Queue hinterlegt haben, möchten Sie diese nun bestimmt auch ausgeben. Die dazu notwendigen Objekte sind ganz analog zum Versand. Nur, dass Sie über die geöffnete `Session` (Abbildung 9.6) diesmal keinen *Sender*, sondern einen *Receiver* erzeugen.

```
package de.akdabas.javaee.jms;

import javax.jms.*;
import de.akdabas.javaee.jndi.Lookup;

/** Empfängt Text-Nachrichten über den Message Service des JBoss */
public class MessageConsumer {

   /** Empfängt eine Nachricht aus der Queue 'mcQueue' */
   public static void main(String [] args) {

      QueueConnection qConnection = null;
      QueueSession qSession = null;
      QueueReceiver qReceiver = null;
      try {
         // Auslesen der ConnectionsFactory aus dem Namensdienst
```

Listing 9.6
Empfang einer Nachricht (jms/MessageConsumer.java)

Listing 9.6 (Forts.)
Empfang einer Nachricht (jms/MessageConsumer.java)

```
            QueueConnectionFactory qcFactory = (QueueConnectionFactory)
               Lookup.lookup(Lookup.JBOSS_ENV, "ConnectionFactory");

            // Auslesen der Ziel-Adresse (Queue)
            String destination = "queue/mcQueue";
            Queue queue = (Queue)
               Lookup.lookup(Lookup.JBOSS_ENV, destination);

            // Aufbau der Verbindung
            qConnection = qcFactory.createQueueConnection();

            // Erzeugen der Session
            qSession = qConnection.createQueueSession(false,
               Session.AUTO_ACKNOWLEDGE);

            // Erzeugen des Empfängers
            qReceiver = qSession.createReceiver(queue);

            // Empfang von Nachrichten starten
            qConnection.start();

            // Text-Nachricht empfangen (Timeout = 10 * 1000 ms)
            TextMessage tMsg =
               (TextMessage) qReceiver.receive((long) 10 * 1000);

            // Ausgabe des Nachrichteninhalts
            if (tMsg != null) {
               System.out.println("Neue Nachricht: " + tMsg.getText());

               // Erfolgreichen Empfang der Nachricht bestätigen
               tMsg.acknowledge();
            } else {
               System.out.println("Keine Nachrichten in der Queue.");
            }

         } catch (JMSException jmx) {
            jmx.printStackTrace();
         } finally {
            try { qReceiver.close(); } catch (Exception exc) {}
            try { qSession.close(); } catch (Exception exc) {}
            try { qConnection.close(); } catch (Exception exc) {}
         }
      }
   }
```

Wie Sie sehen, ist das Vorgehen bis zur Erzeugung der `Session` identisch zum Versenden einer Nachricht. Anschließend erzeugen Sie einfach einen Receiver für die angegebene `Queue` und beginnen mit dem »Empfang neuer Nachrichten« (`start()`). Diese fragen Sie über die Methode `receive()` ab, wobei Sie über den Parameter optional ein Timeout festlegen können, bei dem der Aufruf, gegebenenfalls auch ohne eine Nachricht empfangen zu haben, zurückkehrt.

Da Sie durch das definierte Timeout nicht entscheiden können, ob die Rückkehr der Methode `receive()` durch eine eingehende Nachricht oder ein Timeout veranlasst wurde, überprüfen Sie zunächst, ob tatsächlich eine Nachricht vorliegt, und können anschließend mit deren Verarbeitung beginnen.

Zuletzt schließen Sie den Empfang durch eine Empfangsbestätigung (`acknowledge()`) ab. Diese werden Sie später in diesem Kapitel noch genauer kennen lernen.

Test des Empfangs

Auch der Aufruf dieser Klasse erfolgt analog zum `MessageProducer`, nur mit dem Unterschied, dass Sie diesmal auf jeden Fall eine Ausgabe erhalten:

Listing 9.7
Test des Message Consumer

```
// Resultat, wenn zuvor Listing 9.2 ausgeführt wurde
java -cp ..\lib\jms.jar;..\lib\jndi.jar;..\lib\jbossall-cli-
ent.jar; de.akdabas.javaee.jms.MessageConsumer
Neue Nachricht: Hello Queue!

// Resultat, wenn keine Nachrichten verfügbar sind
java -cp ..\lib\jms.jar;..\lib\jndi.jar;..\lib\jbossall-cli-
ent.jar; de.akdabas.javaee.jms.MessageConsumer
Keine neuen Nachrichten in der Queue.
```

Empfang, ohne zu warten

In einigen Applikationen kann es sinnvoll sein, ganz auf den Timeout der Methode `revceive()` zu verzichten und nur aktuell vorliegende Nachrichten zu verarbeiten, etwa weil die Applikationslogik die verschiedenen `receive()`-Anfragen des Clients schon ausreichend verzögert. Für diese Fälle existiert die Methode `receiveNoWait()`, welche mit oder ohne Nachricht sofort zurückkehrt.

Listing 9.8
Verzögerungsfreier Empfang einer Nachricht

```
// Nichtblockierender Empfang von Nachrichten
MessageConsumer.receiveNoWait()
```

Da jedoch jede Anfrage mit einer gewissen Netzlast verbunden ist, sollten Sie diese Methode nur einsetzen, wenn die daraus resultierenden Anfragen wirklich verzögert sind, sonst erzeugt Ihr Client womöglich einen Denial-of-Service-Fehler, bei dem Ihre Nachricht gar nicht mehr zugestellt wird.

9.6 Empfangsverfahren: Pull vs. Push

Listing 9.6 zeigt Ihnen auch gleich eine Krux des Nachrichtenempfangs: Sie müssen aktiv lauschen. Wenn Sie mehrere Nachrichten empfangen möchten und nicht genau wissen, wann diese Sie erreichen, könnten Sie hierfür vielleicht folgende Schleife implementieren:

Listing 9.9
Empfang mit Pull

```
...
   // Schleife für aktives Warten auf Nachrichten (schlecht)
   boolean ack = false;
   while (!ack) {
      TextMessage msg = (TextMessage) qReceiver.receive((long) 100);
      if (msg != null) {
         System.out.println(msg.getText());
         msg.acknowledge();
         ack = true;
      }
   }
...
```

Dieses Codefragment überprüft ständig, ob neue Nachrichten vorliegen, und erzeugt hierdurch womöglich eine hohe Netzlast. Dieses aktive Lauschen wird als *Pull* bezeichnet.

9.6.1 Push, wenn der Postmann zweimal klingelt

Viel komfortabler, als ständig aktiv zu überprüfen, ob bereits neue Nachrichten vorliegen, wäre es doch, wenn Sie der Message Service informieren würde, ob es etwas zu empfangen gibt. Um an diesem Push-Verfahren teilzunehmen, implementieren Sie einfach das Interface `MessageListener`, über dessen Callback-Methoden Sie bei Eingang einer neuen Nachricht informiert werden. Das folgende Listing zeigt einen entsprechend abgewandelten `MessageConsumer`:

Listing 9.10 Empfang im Push-Verfahren (jms/PushReceiver.java)

```
package de.akdabas.javaee.jms;

import javax.jms.*;
import de.akdabas.javaee.jndi.Lookup;

/** Empfängt Text-Nachrichten im Push-Modus */
public class PushReceiver implements MessageListener {

   /** Die JMS Connection */
   private static QueueConnection qConnection = null;

   /** Die JMS Session */
   private static QueueSession qSession = null;

   /** Der JMS Empfänger */
   private static QueueReceiver qReceiver = null;

   /** Empfängt Nachrichten aus der Queue 'mcQueue' */
   public static void main(String[] args) {
      try {
         // Erzeuge einen neuen 'PushReceiver'
         PushReceiver receiver = new PushReceiver();

         // Lausche für 1 Minute passiv
         long deadline = System.currentTimeMillis() + 60 * 1000;
         while (System.currentTimeMillis() < deadline) {}

      } catch (JMSException jmx) {
         jmx.printStackTrace();
      } finally {
         try { qReceiver.close(); } catch (Exception exc) {}
         try { qSession.close(); } catch (Exception exc) {}
         try { qConnection.close(); } catch (Exception exc) {}
      }
   }

   /** Constructor, beginnt nach seiner Erzeugung zu Lauschen */
   public PushReceiver() throws JMSException {

      // Auslesen der ConnectionsFactory aus dem Namensservice
      QueueConnectionFactory qcFactory = (QueueConnectionFactory)
            Lookup.lookup(Lookup.JBOSS_ENV, "ConnectionFactory");

      // Auslesen der Ziel-Adresse (Queue)
      String destination = "queue/mcQueue";
      Queue queue = (Queue)
            Lookup.lookup(Lookup.JBOSS_ENV, destination);
```

Listing 9.10 (Forts.)
Empfang im Push-Verfahren (jms/PushReceiver.java)

```
        // Aufbau der Verbindung
        qConnection = qcFactory.createQueueConnection();

        // Erzeugen der Session
        qSession = qConnection.createQueueSession(false,
              Session.AUTO_ACKNOWLEDGE);

        // Erzeugen des Consumers
        qReceiver = qSession.createReceiver(queue);

        // Registriert sich selbst als MessageListener
        qReceiver.setMessageListener(this);

        // Empfang starten
        qConnection.start();
    }

    /** Wird gerufen, wenn eine neue Nachricht vorliegt */
    public void onMessage(Message message) {
        try {
            // Casten der Message
            TextMessage tMsg = (TextMessage) message;

            // Ausgabe der Mitteilung
            System.out.println("Neue Nachricht: " + tMsg.getText());

            // Empfang bestätigen
            tMsg.acknowledge();
        } catch (JMSException jmx) {
            jmx.printStackTrace();
        }
    }
}
```

In diesem Listing erzeugen Sie einen `MessageListener`, welcher eingehende Nachrichten über die Methode `onMessage()` ausgibt. Zwar läuft auch dieser Thread für eine Minute aktiv, allerdings werden dafür jetzt die Ressourcen des Clients und nicht mehr die des Servers verwendet. Außerdem können Sie mit dieser Technik z.B. auch Ihre Servlets zu einem Empfänger für Nachrichten machen.

9.7 Topic vs. Queue

Während eine Queue in der Regel nur einen einzigen Empfänger hat, erfüllt eine Topic die Aufgabe eines Schwarzen Bretts oder einer Newsgroup, bei der eingehende Nachrichten an alle registrierten Empfänger versendet werden.

9.7.1 Ein konsolenbasierter Chat

Um die Analogie zu einer Queue zu zeigen, können Sie mit Ihrem jetzigen Wissen über das Senden und Empfangen von Nachrichten auch einen einfachen, konsolenbasierten Chat implementieren.

Listing 9.11
Ein konsolenbasierter Chat (jms/Chat.java)

```
package de.akdabas.javaee.jms;

import javax.jms.*;
import java.util.Scanner;
import de.akdabas.javaee.jndi.Lookup;

/** Empfängt Text - Nachrichten einer Topic */
public class Chat implements MessageListener {

   /** Konstante unter der der Name des Anwenders abgelegt wird */
   public final static String NAME = "name";

   /** Konstante unter der der Beitrag abgelegt wird */
   public final static String MESSAGE = "message";

   /** Kommando zum Verlassen des Chat */
   public final static String EXIT = "exit";

   /** Die geöffnete Topic-Connection */
   private static TopicConnection tConnection = null;

   /** Die geöffnete Topic-Session */
   private static TopicSession tSession = null;

   /** Ein Subscriber zum Emfangen von Nachrichten */
   private static TopicSubscriber tSubscriber = null;

   /** Ein Publisher zum Versenden von Nachrichten */
   private static TopicPublisher tPublisher = null;

   /** Name dieses Chat - Teilnehmers */
   private static String name = null;

   /** Ermöglicht das Chatten per Kommandozeile */
   public static void main(String [] args) {
      try {
         // Name dieses Teilnehmers auslesen
         if (args.length < 1) {
            System.out.println("Starten mit %>java Chatname");
            System.exit(0);
         } else {
            name = args[0];
         }

         // Erzeuge einen neuen Push-Listener
         Chat chat = new Chat();

         // Kommandos werden von der Kommandozeile gelesen
         Scanner scanner = new Scanner(System.in);
         boolean doChat = true;

         // Chatten, bis das Kommando EXIT erfolgt
         while (doChat){
            if (scanner.hasNextLine()) {
               String command = scanner.nextLine();
               if(command.equalsIgnoreCase(EXIT)){
                  doChat = false;
               } else {
                  chat.writeMessage(command);
               }
            }
         }

      } catch (Exception exc) {
         exc.printStackTrace();
      } finally {
```

Listing 9.11 (Forts.)
Ein konsolenbasierter Chat (jms/Chat.java)

```
            try { tPublisher.close(); } catch (Exception exc) {}
            try { tSubscriber.close(); } catch (Exception exc) {}
            try { tSession.close(); } catch (Exception exc) {}
            try { tConnection.close(); } catch (Exception exc) {}
        }
    }


    /** Constructor, Chat beginnt nach der Erzeugung zu Lauschen */
    public Chat() throws JMSException {

        // Auslesen der ConnectionsFactory aus dem Namensservice
        TopicConnectionFactory tcFactory = (TopicConnectionFactory)
            Lookup.lookup(Lookup.JBOSS_ENV, "ConnectionFactory");

        // Auslesen der Ziel-Adresse (Queue)
        Topic chatTopic = (Topic)
              Lookup.lookup(Lookup.JBOSS_ENV, "topic/mcTopic");

        // Aufbau der Verbindung
        tConnection = tcFactory.createTopicConnection();

        // Erzeugen der Topic-Session
        tSession = tConnection.createTopicSession(false,
            Session.AUTO_ACKNOWLEDGE);

        // Erzeugen des Publishers zum versenden von Nachrichten
        tPublisher = tSession.createPublisher(chatTopic);

        // Erzeugen des Subscribers zum Emfangen von Nachrichten
        tSubscriber = tSession.createSubscriber(chatTopic);

        // Setzen des MessageListener
        tSubscriber.setMessageListener(this);

        // Empfang starten
        tConnection.start();
    }


    /** Wird gerufen, wenn eine neue Nachricht vorliegt */
    public void onMessage(Message message) {
        try {
            // Casten der Message
            MapMessage mMessage = (MapMessage) message;

            // Ausgabe der Mitteilung
            System.out.println(mMessage.getString(NAME) + " : " +
                               mMessage.getString(MESSAGE));

            // Empfang bestätigen
            mMessage.acknowledge();
        } catch (JMSException jmx) {
            jmx.printStackTrace();
        }
    }

    /** Sendet einen neuen Beitrag an den Chat */
    private void writeMessage(String messageText)
        throws JMSException {
        MapMessage mMessage = tSession.createMapMessage();
        mMessage.setString(NAME, name);
        mMessage.setString(MESSAGE, messageText);
        tPublisher.publish(mMessage);
    }
}
```

Wie Sie sehen, enthält dieser Chat sowohl einen `TopicSubscriber` (Pendant zu `QueueReceiver`) als auch einen `TopicPublisher` (Pendant zu `QueueSender`), um sowohl Nachrichten senden als auch empfangen zu können. Sonst ist der Aufbau identisch zu den bisherigen Sendern und Empfängern von Nachrichten, wobei das jeweilige `Topic`-Pendant verwendet wird (siehe auch Tabelle 9.1).

9.7.2 Konfiguration und Test des Chat

Da der Chat auf einer Topic aufbaut, muss diese natürlich entsprechend vorkonfiguriert werden. Dazu erweitern Sie die Datei *%JBOSS_HOME%/server/default/deploy/jms/jbossmq-destinations-service.xml* um folgenden Eintrag:

Listing 9.12 Konfiguration einer Topic

```
...
   <!-- JBoss Konfiguration der Topic 'chatTopic' -->
   <mbean code="org.jboss.mq.server.jmx.Topic"
          name="jboss.mq.destination:service=Topic,name=chatTopic">

      <depends optional-attribute-name="DestinationManager">
         jboss.mq:service=DestinationManager
      </depends>
   </mbean>
...
```

Nun ist nach einem Neustart des JBoss die Topic `topic/chatTopic` verfügbar und kann mit folgendem Aufruf des Clients getestet werden.

Listing 9.13 Starten des Chats

```
java -cp ..\lib\jms.jar;..\lib\jndi.jar;..\lib\jbossall-cli-
ent.jar; de.akdabas.javaee.jms.Chat Thomas
```

9.7.3 Topic vs. Queue

Während Topics von Haus aus als Rundschreiben gedacht sind und die Funktion von Newsgroups und Schwarzen Brettern erfüllen, besitzen Queues in der Regeln nur einen einzigen Empfänger, der die dort eingehenden Nachrichten verarbeitet.

Gleichberechtigte Empfänger einer Queue

Doch was passiert, wenn der Empfänger einer Queue sich durch einen Programmierfehler in einer Endlosschleife befindet oder der Server, in dessen Speicher sich der Empfänger-Thread befindet, nicht erreichbar oder überlastet ist? Aus diesem Grund ermöglichen es heute viele JMS-Implementierungen, mehrere Threads (Empfänger) an einer Queue zu registrieren. Dabei ist sichergestellt, dass nur jeweils einer der registrierten MessageConsumer tatsächlich mit der Verarbeitung der Nachricht betraut wird, wobei alle Empfänger gleichberechtigt sind.

In der Praxis wird dieses Verfahren oft zur Verteilung der Rechenlast (Skalierung) auf verschiedene Systeme verwendet. Dabei kann der Client nach dem Versenden einer Nachricht nicht entscheiden, welches Teilsystem die Nachricht erhält.

9.8 Optionen für Nachrichten

Dieser Abschnitt beschäftigt sich, jenseits von Queues und Topics, mit den Eigenschaften von Nachrichten im Allgemeinen. Für diese sieht die JMS-Spezifikation eine Reihe von Konfigurationsmöglichkeiten vor, mit denen eine Nachricht mit bestimmten Attributen versehen werden kann.

9.8.1 Priorität einer Nachricht

Normalerweise gilt für alle Nachrichten: Wer zuerst kommt, wird zuerst gelesen. Nun sind manche Nachrichten jedoch wichtiger als andere. Beispielsweise ist in einer nachrichtenbasierten Datenbankapplikation die Mitteilung: »Tabelle ist voll, Speicherplatz muss erweitert werden« viel wichtiger, als das Speichern von Datensätzen in eben dieser Tabelle.

Es muss also einen Mechanismus geben, mit dem wichtige Nachrichten sich an der Schlange von unwichtigen Nachrichten vorbeischummeln können, um zum nächstmöglichen Zeitpunkt ausgeliefert zu werden. Dies realisieren Sie durch die Zuteilung von *Prioritäten*. Dazu verwenden Sie die Methode `setJMSPriority()`:

Listing 9.14
Setzen einer Priorität

```
...
   // Definieren einer Priorität für Ihre Nachricht
   Message message = tSession.createTextMessage(text);
   message.setJMSPriority(20);
...
```

Dabei werden Nachrichten mit einer höheren Priorität vor Nachrichten einer niedrigeren Priorität ausgeliefert.

9.8.2 Das Verfallsdatum einer Nachricht festlegen

Info

Java repräsentiert jedes Datum intern durch die Anzahl der vergangenen Millisekunden seit Mitternacht des 01. Januar 1970. Durch den verwendeten Datentyp long wird der erste Zeitüberlauf erst in 292 Mio. Jahren stattfinden, es ist also vorerst kein Jahr-2000-Problem in Sicht.

Während einige Nachrichten eine unbegrenzte Lebensdauer besitzen und in jedem Fall zugestellt werden sollen, sind die Informationen bestimmter Nachrichten nur für eine begrenzte Zeitspanne gültig und verfallen zu einem bestimmten Zeitpunkt. Dies ist vergleichbar mit den Cookies, die Sie in Zusammenhang mit Servlets kennen gelernt haben und die ebenfalls eine definierte Gültigkeitsdauer besitzen (siehe Kapitel 3.4).

Den Zeitpunkt, bis zu dem der Inhalt einer Nachricht gültig sein soll, legen Sie über die Methode `setJMSExpiration()` fest, wobei zu beachten ist, dass es sich bei dem übergebenen Parameter nicht um eine Zeitspanne, sondern um den *tatsächlichen Zeitpunkt* nach GMT (repräsentiert in Millisekunden seit 1970) handelt.

Um eine Nachricht mit einer tatsächlichen Lebensdauer von 5 Minuten (300.000 Millisekunden) auszustatten, schreiben Sie:

Listing 9.15 Festlegen des Verfallsdatums

```
...
   // Definieren einer Lebensdauer von 5 Minuten
   Message message = tSession.createTextMessage(text);
   long expirationDate = System.currentTimeMillis() + 5 * 60 * 1000;
   message.setJMSExpiration(expirationDate);
...
```

Soll die Nachricht hingegen unbegrenzt gültig sein, setzen Sie ihr Verfallsdatum einfach auf `0`.

Beim Erreichen oder Überschreiten des Verfallsdatums einer Nachricht entfernt der Message Broker diese aus der `Topic` bzw. `Queue`. Die Nachricht wird nicht mehr zugestellt. Dabei ist zunächst kein Benachrichtigungsmechanismus vorgesehen, der Sie über die Löschung informiert, obwohl einige Service Provider einen solchen implementieren.

Info

Da sich die verschiedenen Implementierungen der einzelnen Anbieter in Bezug auf das Verfallsdatum sehr unterschiedlich verhalten und dieses bei einigen Produkten überhaupt nicht vorgesehen ist, schreibt die JMS-Spezifikation tatsächlich die Löschung der Nachricht nicht zwingend vor. Das *Expiration* Date hat eher einen empfehlenden Charakter ohne Garantie.

9.8.3 Identifizierung einer Nachricht

Auch wenn die JMS-Spezifikation vorsieht, dass sich weder Sender noch Empfänger einer Nachricht kennen, gibt es dennoch eine Möglichkeit, verschiedene Nachrichten eindeutig einem bestimmten Sender zuzuordnen. Dies ist immer dann nötig, wenn zwei aufeinander folgende Nachrichten miteinander in Verbindung stehen (korreliert sind).

Jede Nachricht wird von Haus aus mit einer Korrelationsnummer ausgestattet, die laut Vereinbarung mit dem Kürzeln `ID:` beginnt. Die Korrelationsnummer können Sie bei Bedarf jedoch beliebig ändern, beispielsweise um verschiedene Nachrichten zu Gruppen zusammenzufassen.

Achtung

Bei diesem Verfahren müssen Sie natürlich sicherstellen, dass alle Nachrichten von *einem* `MessageConsumer` empfangen werden. Für Queues, an denen mehrere Empfänger registriert sind, ist es demnach nicht geeignet.

Tipp

Natürlich ist es über die Korrelations-ID auch möglich, Nachrichten, verschiedener Clients zu einer Gruppe zusammenzufassen, indem alle Sender die gleiche Korrelations-ID verwenden. Bei größeren Gruppen ist es jedoch besser, für diese einen eigenen Kanal (`Topic / Queue`) zu öffnen und ausschließlich über diesen zu kommunizieren. Eine weitere Möglichkeit ist die Verwendung eines Attributs im Nachrichten-Header, das über Filter ausgewertet werden kann.

9.9 Filtern einer Nachricht

Sie wissen bereits, dass jede Nachricht aus Kopfteil (Header), Eigenschaften (Properties) und dem Inhalt (Body) der Nachricht besteht, hinter dem sich je nach Nachrichtentyp beispielsweise ein Objekt, ein Ausgabestrom oder auch eine Map verbergen kann. Und eben diese Properties können Sie dazu verwenden, die Nachrichten zu filtern und über eine Queue beispielsweise nur die Nachrichten zu empfangen, die bestimmten Bedingungen genügen.

Tipp

Prinzipiell sollten Sie Korrelations-IDs verwenden, um zusammengehörende Nachrichten eines Senders zu kennzeichnen.

9.9.1 Setzen und Auslesen von Eigenschaften

Properties sind ein zentraler Bestandteil von Nachrichten und jede Nachricht stellt Ihnen verschiedene Methoden zum Setzen und Auslesen dieser zur Verfügung. Da JMS jedoch dafür gedacht ist, die Kommunikation über verschiedene Programmiersprachen hinweg zu gewährleisten, deren Objekte (`java.lang.Object`) in der Regel zueinander inkompatibel sind, beschränken sich die Properties auf bestimmte (überall verfügbare) Basistypen. Tabelle 9.2 zeigt die dabei unterstützten primitiven Datentypen.

Tabelle 9.2
Unterstützte Objekttypen für Nachrichten-Properties

Datentyp	gesetzt als	gelesen als
`boolean`	`boolean, String`	`boolean, String`
`byte`	`byte, String`	`byte, short, int, long, String`
`short`	`short, String`	`short, int, long, String`
`int`	`int, String`	`int, long, String`
`long`	`long, String`	`long, String`
`float`	`float, String`	`float, double, String`
`double`	`double, String`	`double, String`
`String`	`String`	`String`

Der Versuch, ein Attribut unter in einem nicht unterstützten Format auszulesen, resultiert dabei stets in einer `MessageFormatException`. Dies gilt insbesondere für die Typumwandlung beim Setzen als String.

Neben den Setter- und Getter-Methoden für Basistypen (z.B. `setFloatProperty()`, `getFloatProperty()`) existieren auch Methoden für Objekttypen.

```
// Setzen von Message-Attributen
setObjectProperty(String name, java.lang.Object object)
getObjectProperty(String name)
```

Diese könnten Sie zu der Annahme verleiten, JMS würde auch diese Attribute unterstützen, dabei sind sie nur für die Wrapper-Klassen der jeweiligen Basistypen (`java.lang.Long`, `java.lang.Float` usw.) gedacht. Der Versuch, andere Objekte über diese Typen zu setzen, wird ebenfalls unweigerlich zu einer `MessageFormatException` führen.

9.9.2 Ausgabe von Message-Attributen

Nachdem Sie eine Nachricht mit verschiedenen Properties ausgestattet haben, möchten Sie diese natürlich auch auswerten können. Dazu stellt Ihnen das `Message`-Objekt zunächst natürlich einige direkte Methoden zum Auslesen bereit.

```
// Auslesen von Message-Attributen
boolean getBooleanProperty(String name)
byte getByteProperty(String name)
usw.
```

Da Ihnen diese Methoden jedoch ebenfalls Basistypen zur Verfügung stellen, kann Ihnen Java das Fehlen des entsprechenden Attributs nicht durch Zurückgabe eines `null`-Werts signalisieren. Stattdessen würden Sie mit dem Initialisierungswert des Basistyps (bei vielen Datentypen 0) konfrontiert werden und könnten nicht unterscheiden, ob dieser jetzt explizit gesetzt oder einfach vergessen wurde.

Um daraus resultierendes Fehlverhalten des Empfängers zu vermeiden, schreibt die JMS-Spezifikation in diesem Fall das Erzeugen einer `JMSException` vor und um diese zu vermeiden, können Sie auf zwei weitere Methoden zurückgreifen:

- `existParam(String name)`
  Mit dieser Methode können Sie das Vorhandensein eines bestimmten Parameters überprüfen. Sie sagt jedoch nichts über den Typ des verwendeten Attributs aus.
- `getObjectProperty(String name)`
  Über diese Methode können Sie alle vorhandenen (und nicht vorhandenen) Eigenschaften einer Nachricht auslesen. Existiert ein bestimmter Parameter nicht, gibt die Methode einfach `null` zurück und erfüllt somit die Anforderungen der Spezifikation.

  Der zurückgegebene Typ ist dabei immer derselbe, unter dem der Wert auch abgelegt wurde – also `Float`, wenn `setFloatProperty()` verwendet wurde, usw.

Ausgabe aller vorhandenen Attribute

Natürlich können Sie sich auch eine Liste aller vorhandenen Attribute erzeugen. Um diese anschließend ausgeben zu können, verwenden Sie beispielsweise das folgende Code-Fragment.

Listing 9.16
Ausgabe aller Attribute einer Nachricht

```
...
   // Ausgabe aller Properties einer Nachricht
   Enumeration enum = message.getPropertyNames();
   while (enum.hasMoreElements()) {
      String propertyName = (String) enum.nextElement;
      System.out.println(message.getStringProperty(propertyName));
   }
...
```

9.9.3 Filtern anhand von Attributen

Nun ist das Setzen und Lesen von Attributen sicherlich eine schöne Angelegenheit, doch da Ihnen seit der Version 1.1 über die `MapMessage` ein eigens dafür eingerichteter Nachrichtentyp zur Verfügung steht, wären Properties (fast) nutzlos.

Das Besondere an den Properties des Header ist, dass Ihnen die JMS-Spezifikation einen eigenen Abfragemechanismus für diese Attribute zur Verfügung stellt, mit dem Sie die Nachrichten einer Queue oder Topic auf einfache Art und Weise kategorisieren können. Dazu verwenden Sie so genannte Attributfilter, über die Sie festlegen, welche Nachrichten Sie erhalten möchten.

Ein einfaches Beispiel für die Verwendung solcher Attribute ist eine Protokollanwendung, die priorisierte Mitteilungen auf unterschiedlichen Speichermedien festhält. Dazu definieren Sie drei Kategorien `low`, `default` und `high`, die die Priorität der Protokollausgabe symbolisieren, und Sie lassen sich vom Administrator des Namensdienstes *eine* Topic erzeugen, an die alle Protokollinformationen der Anwendung gesendet werden sollen.

Jetzt müssen Sie nur noch darauf achten, alle ausgehenden Nachrichten vom Sender (`MessageProducer`) mit folgendem Attributpaar zu versehen.

Listing 9.17
Setzen des 'prio'-Attributs einer Nachricht

```
...
   // Setzen des Attributes 'prio' einer Nachricht
   Message message = tSession.createTextMessage(text);
   message.setStringProperty("prio", "high");
...
```

Anschließend erzeugen Sie zwei unterschiedliche Protokollierer, um die übermittelten Informationen auszugeben. So könnte ein Protokollierer alle mittleren und hohen Informationen auf einem Terminal ausgeben, wohingegen wichtige Informationen über einen zweiten Protokollierer zusätzlich in eine Datenbank geschrieben werden.

Anstatt jede eingehende Nachricht jetzt innerhalb der Methode `onMessage()` auf ihre Relevanz zu überprüfen, können Sie die jeweiligen `Mes-`

sageConsumer auch gleich so konfigurieren, dass diese nur die für sie bestimmten Informationen empfangen. Listing 9.18 zeigt einen MessageListener, den Sie aus Listing 9.10 ableiten können, um lediglich Nachrichten mittlerer und hoher Priorität zu empfangen.

Listing 9.18
Filtern von Nachrichten mittlerer und hoher Priorität

```
package de.akdabas.javaee.jms;

import javax.jms.*;
import de.akdabas.javaee.jndi.Lookup;

/**
 * Empfängt Text-Nachrichten einer Topic, die definierten
 * Filterkriterien genügen.
 */
public class TopicPushReceiver implements MessageListener {

   /** Die JMS Connection */
   private static TopicConnection tConnection = null;

   /** Die JMS Session */
   private static TopicSession tSession = null;

   /** Der Empfänger */
   private static TopicSubscriber tSubscriber = null;

   /** Empfängt Nachrichten aus der Topic 'mcTopic' */
   public static void main(String[] args) {
      try {
         // Erzeuge einen neuen 'TopicPushReceiver'
         TopicPushReceiver receiver = new TopicPushReceiver();

         // Lausche für 1 Minute passiv
         long deadline = System.currentTimeMillis() + 60 * 1000;
         while (System.currentTimeMillis() < deadline) {}

      } catch (JMSException jmx) {
         jmx.printStackTrace();
      } finally {
         try { tSubscriber.close(); } catch (Exception exc) {}
         try { tSession.close(); } catch (Exception exc) {}
         try { tConnection.close(); } catch (Exception exc) {}
      }
   }

   /** Constructor; beginnt nach seiner Erzeugung zu Lauschen */
   public TopicPushReceiver() throws JMSException {

      // Auslesen der ConnectionsFactory aus dem Namensservice
      TopicConnectionFactory tcFactory = (TopicConnectionFactory)
            Lookup.lookup(Lookup.JBOSS_ENV, "ConnectionFactory");

      // Auslesen der Ziel-Adresse (Topic)
      String destination = "topic/mcQueue";
      Topic topic = (Topic)
            Lookup.lookup(Lookup.JBOSS_ENV, destination);

      // Aufbau der Verbindung
      tConnection = tcFactory.createTopicConnection();

      // Erzeugen der Session
      tSession = tConnection.createTopicSession(false,
            Session.AUTO_ACKNOWLEDGE);
```

Listing 9.18 (Forts.)
Filtern von Nachrichten mittlerer und hoher Priorität

```
        // Definition der Filter-Attribute
        String filter = "prio IN ('default', 'high') ";

        // Flag,ob dieser Listener eigene Nachrichten empfangen soll
        boolean noLocal = true;

        // Erzeugen des Listeners mit Filter-Attributen
        tSubscriber =
              tSession.createSubscriber(topic, filter, noLocal);

        // Registriert sich selbst als MessageListener
        tSubscriber.setMessageListener(this);

        // Empfang starten
        tConnection.start();
    }

    /**
     * Wird gerufen, wenn eine neue Nachricht vorliegt und diese den
     * Filter-Kriterien genügt.
     */
    public void onMessage(Message message) {
        try {
            TextMessage tMsg = (TextMessage) message;
            System.out.println("Neue Nachricht: " + tMsg.getText());
            tMsg.acknowledge();
        } catch (JMSException jmx) {
            jmx.printStackTrace();
        }
    }
}
```

Wie schon bei so vielen anderen in diesem Buch vorgestellten Technologien, existiert auch für die Definition von Filterregeln eine eigene Syntax. Dabei lehnen sich die auch als *MessageSelector* bezeichneten Zeichenketten stark an die *Structured Query Language (SQL)* an, die sie vielleicht von Ihrer Datenbank her kennen.

Tipp

Sie können Nachrichtenfilter natürlich auch auf Queues anwenden, wobei die nicht empfangenen Nachrichten in der Queue verbleiben, bis sie verfallen oder anderweitig empfangen werden. Das Attribut `noLocal` gibt es bei Queues hingegen nicht.

9.9.4 Filterelemente

Die in Filtern verwendete Syntax entspricht zwar im Wesentlichen dem SQL-92-Standard, doch da Sie hier statt mit Tabellen mit Attributen arbeiten, gibt es einige Unterschiede in den verwendeten Elementen.

Zeichenketten

Zeichenketten werden stets von einfachen Hochkommata (`'`), so genannten MessageSelectors, eingeschlossen, was die Schreibweise im Java-Quellcode deutlich vereinfacht.

Zeichenketten werden vor allem verwendet, um Attributwerte auf bestimmte Werte zu überprüfen. Um in obigem Beispiel nur die Nachrichten mit dem Attributpaar (prio/high) herauszufiltern, könnten Sie folgenden Filter verwenden.

```
String filter = "prio = 'high'";
```

Attributnamen

Natürlich verwenden typische Filter auch Attributnamen, deren Werte schließlich überprüft werden sollen. Dabei müssen Sie auf die korrekte Schreibweise achten. So unterscheidet Java beispielsweise zwischen den Attributen `prio` und `Prio`. Und folgende Filter würden keiner Nachricht aus Listing 9.18 genügen.

```
String filter = "Prio = 'high'"; // Das funktioniert nicht
String filter = "prio = 'High'"; // Das ebenfalls nicht
```

Arithmetische, logische und Vergleichsoperatoren

Um den Wert eines Attributs mit einem anderen Attributwert vergleichen zu können, benötigen Sie schließlich noch entsprechende Operatoren. Dabei unterscheidet man die Vergleichsoperatoren =, >, >=, <, <= und <>, die arithmetischen Operatoren +, –, *, / und natürlich die logischen Operatoren `AND`, `OR` und `NOT`.

Sie hätten in obigem Beispiel also auch folgenden Filter verwenden können.

```
String filter = "prio = 'default' OR prio = 'high'"
```

Achtung

Sie sollten besonders bei den *arithmetischen* und *Vergleichsoperationen* auf sinnvolle Typen achten, da die Multiplikation eines Strings mit einer Zahl kein Ergebnis bringt. Deshalb dürfen Sie bei diesen Attributen auch *nicht* die Methode `setStringProperty()` verwenden.

Sonstige Operatoren

SQL bietet neben den oben genannten Operatoren noch einer ganze Reihe weiterer Operatoren, von denen einige übernommen wurden.

- BETWEEN zahl AND zahl

  Dieser Operator eignet sich hervorragend, um Wertebereiche von Zahlen einzugrenzen. So ist der Filterausdruck `"id >= 5 AND id <= 8"` identisch mit der Form `"id BETWEEN 5 AND 8"`.

- IN ('Zeichenkette 1', 'Zeichenkette 2', ...)

  Dieser Operator dient vor allem dazu, mehrere mögliche Zeichenketten zusammenzufassen. Wie wir bereits gesehen haben, eignet er sich dazu, den Filter `"prio = 'default' OR prio = 'high'"` zu folgender Form zu verkürzen `"prio IN ('default', 'high')"`.

- LIKE

  Mit diesem Ausdruck können Sie die gesuchte Zeichenkette mit regulären Ausdrücken beschreiben. Dabei steht der _ (Unterstrich) für einen beliebigen einzelnen Buchstaben und % (Prozentzeichen) für eine beliebige Buchstabenfolge.

  Um also lediglich Nachrichten zu empfangen, deren `name`-Attribute mit `T` beginnt und mit einem `S` aufhört, verwenden Sie den Filter `"name LIKE 'T%S'"`.

- IS NULL

  Abschließend können Sie natürlich auch Nachrichten auf das prinzipielle Vorhandensein bestimmter Attribute überprüfen. So würden Sie beispielsweise, um alle Nachrichten ohne das Attribut `prio` zu empfangen, schreiben `"prio IS NULL"`, dessen Pendant natürlich `"prio IS NOT NULL"` ist.

Wie Sie sehen, ermöglichen es Filter, auch die komplexesten Attributkombinationen schon bei der Zustellung der Nachricht sicherzustellen. Dabei haben Sie allerdings nur Zugriff auf den Property-Teil der Nachricht. Attribute, die Sie im Body einer `MapMessage` übermitteln, bleiben außen vor und können nicht mit einbezogen werden.

Empfangen eigener Nachrichten

Filter werden oftmals in chat-basierten Message-Systemen angewendet, bei denen viele Clients über einen gemeinsamen Kanal – in der Regel eine Topic – miteinander kommunizieren, aber nur die wirklich für sie relevanten Informationen tatsächlich erhalten sollen. Da innerhalb dieser Topic alle Clients sowohl als `MessageProducer` als auch als `MessageConsumer` registriert sind, existiert ein besonderer Filter, um zu verhindern, dass diese ihre eigenen Nachrichten selbst noch einmal empfangen und so eventuell in eine Endlosschleife geraten. Das Flag `noLocal` aus Listing 9.18 steuert dieses Verhalten und hat in der Regel den Wert `true`.

9.10 Transaktionen und Empfangsbestätigungen

In diesem Abschnitt lernen Sie das Transaktionsmanagement kennen, bei dem es darum geht, eine unteilbare Menge von Operationen vorzubereiten und anschließend entweder vollständig auszuführen (`Commit`) oder den gesamten Vorgang abzubrechen (`Rollback`).

Beim Java Message Service (JMS) müssen Sie hierbei sogar noch einmal zwischen Transaktionen beim Senden und Empfangen von Nachrichten unterscheiden.

9.10.1 Transaktionen beim Senden

Bei jeder neuen, über eine `Connection` geöffneten `Session` müssen Sie sich entscheiden, ob diese der Transaktionskontrolle des Message Service unterliegen soll oder nicht. Wenn Sie sich für die Transaktionskontrolle entscheiden, werden alle Nachrichten, die Sie anschließend über die `Session` an `Queues` bzw. `Topics` senden, zunächst zurückgehalten, bis Sie den Versand mit einem `Commit` bestätigen. Erst dann wird die Nachricht durch den Message Service übertragen und im »Eingangskorb« des Empfängers abgelegt.

Unterliegt ein `Session`-Objekt allerdings erst einmal der Transaktionskontrolle, können Sie diese nicht mehr ausschalten. Jedes `Commit` bzw. `Rollback` einer aktuellen Transaktion öffnet sofort eine neue Transaktion, so dass keine Operation ohne abschließende Bestätigung ausgeführt wird. Sind Sie sich hingegen nicht mehr sicher, ob Sie sich innerhalb einer Transaktion befinden, die mit einem `Commit` abgeschlossen werden muss, können Sie dies über folgenden Methodenaufruf in Erfahrung bringen:

Listing 9.19 Überprüfen des Transaktionsstatus dieser Session (Senden)

```
// Überprüfen, ob die aktuelle Session mit
// Transaktionskontrolle arbeitet
public boolean Session.getTransacted()
```

9.10.2 Transaktionen beim Nachrichtenempfang

Natürlich können Sie transaktionsgesteuerte `Sessions` auch zum Empfang von Nachrichten verwenden. In diesem Fall wirkt sich die Transaktion jedoch nicht auf das Versenden, sondern auf den Status der Nachricht aus.

So können Sie innerhalb einer Transaktion nacheinander verschiedene Nachrichten empfangen und ihren Erhalt bestätigen, ohne dass diese bereits endgültig aus der `Queue` bzw. `Topic` entfernt werden. Auf diese Weise sind Sie in der Lage, zusammengehörende Nachrichten zu bearbeiten und ihren Empfang erst nach vollständigem Abschluss der Operation zu quittieren.

Wird eine Nachricht zunächst über eine transaktionsgesteuerte Session empfangen und anschließend über ein Rollback zurückgesetzt, beginnt der `Session`-Thread von neuem, diese zuzustellen. Dabei kann es sich im Fall einer `Queue` mit mehreren registrierten Empfängern dann auch um einen anderen `MessageConsumer` mit einer möglicherweise nicht transaktionsgesteuerten `Session` handeln. Ob die aktuelle Nachricht bereits einmal empfangen und nach einem Rollback erneut zugestellt wurde, erfahren Sie über die Methode:

Listing 9.20 Überprüfen, ob diese Nachricht schon einmal zugestellt wurde

```
// Überprüfen, ob diese Nachricht schon einmal zugestellt wurde
public boolean Message.getJMSRedelivered()
```

Wobei auch hier wieder einmal gilt, dass die JMS-Spezifikation dieses Verhalten zwar empfiehlt, in Hinblick auf die große Anzahl der unterstützten Message-Systeme jedoch nicht zwingend vorschreibt.

9.10.3 Empfangsbestätigungen

Wenn die `Session` eines `MessageConsumer` keiner Transaktionskontrolle unterliegt, müssen andere Mechanismen dafür sorgen, dass

1. die Nachricht zugestellt wird,
2. die Nachricht nach erfolgreicher Bearbeitung aus der `Queue` / `Topic` entfernt wird,
3. die Nachricht nach einem fehlgeschlagenen Bearbeitungsversuch erneut (an einen anderen Client) versendet wird.

Die Verantwortung für diese Verwaltungsaufgaben verteilen Sie durch das Setzen des `Acknowledge`-Modus beim Erzeugen der `Session`:

Listing 9.21
Festlegen des Bestätigungsmodus

```
// Definieren, wer verantwortlich für die Empfangsbestätigung ist
Session session =
   connection.createSession(transacted, acknowledgeMode);
```

In der aktuellen Spezifikation werden dabei drei verschiedene Modi unterstützt, durch die Sie die Verantwortung entweder in die Hände der empfangenden `Session` oder des Clients selbst legen.

javax.jms.Session.AUTO_ACKNOWLEDGE

Dieser Modus dürfte den Anforderungen der meisten nicht transaktionsgesteuerten Sessions genügen, da er die Verantwortung für das Entfernen einer Nachricht der empfangenden `Session` überträgt.

Dabei wird eine Nachricht sofort aus der entsprechenden `Queue` bzw. `Topic` entfernt, nachdem

- die `onMessage()`-Methode eines registrierten `MessageConsumer` erfolgreich (d.h. ohne eine `Exception`) beendet werden konnte.
- zum ersten Mal die Methode `receive()` erfolgreich ausgeführt wurde.

In diesem Modus entspricht jede Nachricht einer atomaren Transaktion, die nicht rückgängig gemacht werden kann. Die `acknowledge()`- und `recover()`-Methoden des Clients werden in diesem Modus ignoriert.

javax.jms.Session.CLIENT_ACKNOWLEDGE

Vielleicht haben Sie sich schon gefragt, warum alle bisherigen Empfangsmethoden stets mit dem Aufruf `acknowledge()` beendet wurden?! Im `Client_Acknowledge`-Modus teilen Sie der Session durch diesen Aufruf ein erfolgreiches Verarbeiten der Nachricht mit, woraufhin diese die Nachricht umgehend aus der `Topic` (`Queue`) entfernt.

Das Gegenstück zur `acknowledge()`-Methode bildet die Methode `recover()`, welche mit einem Rollback des Empfangs vergleichbar ist. Alle seit dem letzten `acknowlegde()`-Kommando ausgelieferten Nachrichten werden in diesem Fall wieder vorn in die `Topic` (`Queue`) eingestellt und anschließend erneut ausgeliefert.

Tipp

Übernimmt eine `Session` hingegen selbst die Kontrolle über die `Topic` (Queue), werden die `acknowledge()`- und `recover()`-Aufrufe einfach ignoriert. Mit diesem Wissen können Sie also getrost alle Clients mit einem `acknowledge()`-Aufruf ausstatten (was den Vorteil hat, dass Sie den Empfangsmodus auch im Nachhinein noch wechseln können).

javax.jms.DUPS_OK_ACKNOWLEDGE

Dieser Modus hat den seltsamen Namen *Dups ok*. Dies ist die Abkürzung für *Duplicates ok!* und beschreibt das Verhalten der Anwendung sehr treffend. Eine solche `Session` verhält sich prinzipiell genau wie im Modus `AUTO_ACKNOWLEDGE` und steuert das Entfernen der Nachrichten selbst. Der einzige Unterschied besteht darin, dass diese `Session` keine Aussage darüber macht, wann sie dies tut. So kann es in diesem Modus vorkommen, dass die Nachricht einer `Queue` (welche ja eigentlich nur für einen einzigen Empfänger gedacht ist), trotz erfolgreicher Verarbeitung durch den ersten Client, zusätzlich an andere Empfänger ausgeliefert wird, weil sie sich noch in der Warteschlange befindet.

Um den Vorteil dieses Modus zu verstehen, müssen Sie wissen, dass eine `Session` in der Regel aus mehreren Threads für unterschiedliche Aufgaben (Empfangen, Senden, Löschen) besteht. Im `AUTO_ACKNOWLEDGE`-Modus kann dabei der eher unwichtige Lösch-Thread nicht vom zentralen Empfangsmechanismus getrennt werden, da jede Nachricht nach der Zustellung sofort entfernt werden muss. Stellt hingegen das mehrfache Zustellen einer bereits verarbeiteten Nachricht kein Problem dar, können beide Mechanismen voneinander losgelöst agieren – wodurch das Verhalten der Anwendung performanter wird.

9.11 Hin und zurück – Synchrone Kommunikation mit JMS

Die bisherigen Nachrichten bauten darauf auf, dass der `MessageProducer` den Message Service kontaktiert, eine Nachricht erzeugt und an ein vordefiniertes Ziel sendet. Dort angekommen, empfängt sie ein registrierter `MessageConsumer` und verarbeitet sie nach besten Wissen und Gewissen. Damit wurde die Nachricht ihrer Bestimmung gerecht und konnte gelöscht werden.

Zweifelsohne gibt es eine ganze Reihe von Anwendungen, bei denen genau dieses Verhalten erwünscht ist, doch daneben gibt es auch Fälle, bei denen Sender und Empfänger in beiden Richtungen miteinander kommunizieren müssen. So gibt es große Application Server, die ihren Clients auf Wunsch komplexe Berechnungen abnehmen und die Ergebnisse anschlie-

ßend über eine weitere Mitteilung zurücksenden. Dazu müssen jedoch sowohl Client als auch Server über einen gemeinsamen Kanal verfügen.

Hierfür sind die bereits eingangs erwähnten temporären Queues und Topics gedacht, die der Sender dem Empfänger über das `ReplyTo`-Feld zugänglich macht.

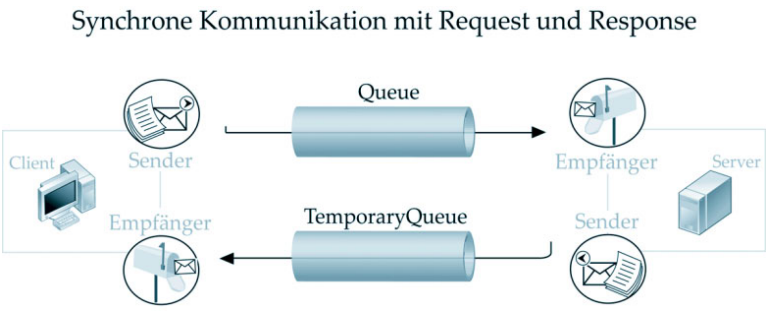


Abbildung 9.7
Request und Response mit JMS

Da das manuelle Erzeugen und aktive Warten auf die Antwort auf Dauer recht umständlich wird, sieht die JMS-Spezifikation dafür gleich eine Klasse vor, die Ihnen (fast) die ganze Arbeit abnimmt.

9.11.1 Ein QueueRequestor

Um die Funktionsweise synchronisierter Nachrichten demonstrieren zu können, benötigen Sie zunächst einen Client, der einen "Request" in Form einer Nachricht an einen "Server" sendet und anschließend auf dessen Antwort wartet. Das folgende Listing zeigt einen solchen Client:

Listing 9.22
Ein blockierender Client

```
package de.akdabas.javaee.jms;
import javax.jms.*;
import de.akdabas.javaee.jndi.Lookup;

/**
 * Sendet eine Nachricht an den Server und gibt dessen Antwort aus
 */
public class RequestClient {

   /**
    * Sendet eine Nachricht an den Server (Request) und warten auf
    * eine Antwort (Response) von diesem
    */
   public static void main(String [] args) {

      QueueConnection qConnection = null;
      QueueSession qSession = null;
      QueueRequestor qRequestor = null;
      try {
         // Auslesen der ConnectionsFactory aus dem Namensservice
         QueueConnectionFactory qcFactory = (QueueConnectionFactory)
            Lookup.lookup(Lookup.JBOSS_ENV, "ConnectionFactory");

         // Auslesen der Ziel-Adresse (Queue)
         Queue serverQueue = (Queue)
            Lookup.lookup(Lookup.JBOSS_ENV, "queue/serverQueue");
```

Listing 9.22 (Forts.)
Ein blockierender Client

```
            // Aufbau der Verbindung
            qConnection = qcFactory.createQueueConnection();

            // Erzeugen der Session
            qSession = qConnection.createQueueSession(false,
               Session.AUTO_ACKNOWLEDGE);

            // Empfang von Nachrichten starten
            qConnection.start();

            // Erzeugen der Nachricht
            Message message =
               qSession.createTextMessage("Dies ist ein Request!");

            // Erzeugen eines QueueRequestors
            qRequestor = new QueueRequestor(qSession, serverQueue);

            // Senden der Nachricht und empfangen der Antwort
            TextMessage answer =
               (TextMessage) qRequestor.request(message);

            // Ausgabe der Anwort
            System.out.println(answer.getText());
         } catch (JMSException jmx) {
            jmx.printStackTrace();
         } finally {
            try { qRequestor.close(); } catch (Exception exc) { }
            try { qSession.close(); } catch (Exception exc) { }
            try { qConnection.close(); } catch (Exception exc) { }
         }
      }
   }
```

Wie Sie sehen ist das einzige neue Element der `QueueRequestor`, der an die Stelle des `QueueSender` tritt. Der QueueRequestor verschickt Nachrichten dabei nicht über die Methode `send()`, sondern via `request()` und blockiert anschließend so lange bis er eine Antwort (`Message`) erhalten hat. Diese empfängt er über eine temporäre `Queue`, die er automatisch im Hintergrund erzeugt und an das `ReplyTo`-Feld anhängt.

Dabei müssen Sie darauf achten, dass die Session dieses Clients nicht unter Transaktions-Kontrolle steht.

9.11.2 Ein Empfänger, der antwortet

Das Gegenstück zum `RequestClient` (Listing 32) bildet natürlich ein `MessageConsumer`. Dieser muss nun aber auch wissen, dass er auf die eingehende Nachricht antworten soll. Dazu modifizieren Sie Ihren `PushReceiver` (Listing 11) wie folgt:

Listing 9.23
Ein Response-Client

```
package de.akdabas.javaee.jms;
import javax.jms.*;
import de.akdabas.javaee.jndi.Lookup;

/** Empfängt Text - Nachrichten und gibt eine Antwort zurück*/
public class ResponseClient implements MessageListener {

   /** Die Connection */
   private static QueueConnection qConnection = null;

   /** Die Session */
   private static QueueSession qSession = null;
```

Listing 9.23 (Forts.)
Ein Response-Client

```
/** Der Empfänger */
private static QueueReceiver qReceiver = null;

/** Empfängt eine Nachricht aus der Queue */
public static void main(String [] args) {
   try {
      // Erzeuge Push - Listener
      ResponseClient respClient = new ResponseClient();

      // Lausche für 1 Minute passiv
      long deadline = System.currentTimeMillis() + 60 * 1000;
      while (System.currentTimeMillis() < deadline) {}

   } catch (JMSException jmx) {
      jmx.printStackTrace();
   } finally {
      try { qReceiver.close(); } catch (Exception exc) { }
      try { qSession.close(); } catch (Exception exc) { }
      try { qConnection.close(); } catch (Exception exc) { }
   }
}

/** Constructor, beginnt zu lauschen */
public ResponseClient() throws JMSException {

   // Auslesen der ConnectionsFactory aus dem Namensservice
   QueueConnectionFactory qcFactory = (QueueConnectionFactory)
      Lookup.lookup(Lookup.JBOSS_ENV, "ConnectionFactory");

   // Auslesen der Ziel-Adresse (Queue)
   Queue destinationQueue = (Queue)
      Lookup.lookup(Lookup.JBOSS_ENV, "queue/serverQueue");

   // Aufbau der Verbindung
   qConnection = qcFactory.createQueueConnection();

   // Erzeugen der Session
   qSession = qConnection.createQueueSession(false,
      Session.AUTO_ACKNOWLEDGE);

   // Erzeugen des Consumers
   qReceiver = qSession.createReceiver(destinationQueue);

   // Setzen des MessageListener
   qReceiver.setMessageListener(this);

   // Empfang starten
   qConnection.start();
}


/** Wird gerufen, wenn eine neue Nachricht vorliegt */
public void onMessage(Message message) {
   try {
      // Casten der Message
      TextMessage tMsg = (TextMessage) message;

      // Ausgabe der Mitteilung
      System.out.println("Empfange Nachricht: "+tMsg.getText());

      // Antwort erzeugen
      TextMessage answer =
         qSession.createTextMessage("Request erhalten!");

      // Extrahieren der temporären Queue des anfragenden Clients
      Queue reply = (Queue) message.getJMSReplyTo();
```

Listing 9.23 (Forts.)
Ein Response-Client

```
            // Einen Sender erzeugen
            QueueSender replySender = qSession.createSender(reply);

            // Antwort versenden
            replySender.send(reply, answer);

            // Sender schließen
            replySender.close();

            // Empfang bestätigen
            message.acknowledge();
         } catch (JMSException jmx) {
            jmx.printStackTrace();
         }
      }
   }
```

Nachdem er eine Nachricht empfangen hat, verhält sich dieser Empfänger wie ein klassischer Nachrichten-Sender (`MessageProducer`), indem er nach der "Verarbeitung" der Nachricht den Rückgabe-Kanal ausliest, einen `QueueSender` erzeugt und die "verarbeitete" Antwort zurückschickt. Da der temporäre Kanal für die Antwort nur dem Sender und dem Empfänger bekannt ist, wird eine störungsfreie Antwort ermöglicht.

9.11.3 Request und Reply vs. Remote Procedure Call

Über den `QueueRequestor` (`TopicRequestor`) sind Sie nun in der Lage, auch klassische *Remote Procedure Call (RPC)* - Szenarien zu entwickeln, bei denen der Client seine Verarbeitung bis zum Eintreffen der Antwort unterbricht (synchrone Kommunikation).

Der Vorteil des Message Service gegenüber RPC ist dabei, dass Client und Server nicht direkt miteinander in Kontakt treten, sondern nur über den Message Broker miteinander kommunizieren. Hierdurch kann dieser die eingehenden Requests auf verschiedene Server verteilen, wodurch hoch skalierende und ausfallsichere Systeme realisiert werden können. Ein Anwendungsgebiet für diese Technik bilden die Message Driven Beans.

9.12 Message Driven Beans

Mit den Message Driven Beans verheiratete Sun zwei erfolgreiche und etablierte Technologien. Diese Enterprise JavaBeans sind vergleichbar mit Stateless Session-Beans und dienen wie diese zur Abbildung von Geschäftsprozessen.

Der Vorteil der Message Driven Beans besteht darin, dass sie im Vergleich zu normalen Stateless Session-Beans bei gleicher Leistungsfähigkeit meist leichter zu erzeugen und zu konfigurieren sind, weil sie weder Home Interface noch Remote Interface benötigen.

Info

Das Remote Interface von MDBs ist das Interface `javax.jms.MessageListener`.

Message Driven Beans implementieren nach außen genau eine Schnittstelle – die des Interface `MessageListener` – und unterscheiden sich für den Client durch nichts von anderen Komponenten nachrichtengesteuerter Prozesse. Sie werden mit Hilfe einer `ConnectionFactory` über den ohnehin in den Application Server integrierten Namensdienst zur Verfügung gestellt und empfangen Serviceanfragen in Form von Nachrichten.

Ein typisches Anwendungsgebiet für Message Driven Beans sind dabei zum Beispiel die Auswertungen von Datenbeständen umfangreicher Datenbanken. Dabei werden die einzelnen Operationen bzw. deren Ergebnis in Form einer Message hinterlegt und zu einem späteren Zeitpunkt abgearbeitet.

Info

Dieses Beispiel setzt einige Grundlagen des vorangegangenen Kapitels über Enterprise JavaBeans (EJB) voraus.

Für den EJB-Container ist eine Message Driven Bean nichts anderes als eine Stateless Session-Bean mit einer speziellen Signatur und da sie immer die gleiche Schnittstelle implementiert, benötigt sie dafür weder Home Interface noch Remote Interface. Das folgende Listing zeigt die für alle Message Driven Beans einheitliche Schnittstelle schematisch:

Listing 9.24
Eine einfache Message Driven Bean (MessageDrivenBeanSchema.java)

```
package de.akdabas.javaee.ejb;
import javax.jms.Message;
import javax.jms.TextMessage;
import javax.jms.JMSException;
import javax.jms.MessageListener;

import javax.ejb.MessageDriven;
import javax.ejb.ActivationConfigProperty;

/**
 * Diese Annotationen konfigurieren die MDB
 */
@MessageDriven(activationConfig =
   {
      // Typ des Nachrichtenkanals: 'Queue' oder 'Topic'
      @ActivationConfigProperty(propertyName = "destinationType",
                                propertyValue = "javax.jms.Queue"),

      // Name der Queue an die diese MDB gebunden ist
      @ActivationConfigProperty(propertyName = "destination",
                                propertyValue = "queue/mcQueue")
   })
```

Listing 9.24 (Forts.) Eine einfache Message Driven Bean (MessageDrivenBeanSchema.java)

```
/**
 * Ab hier kann die Message Driven Bean auch als einfacher
 * MessageListener interpretiert werden, wie Sie ihn auch
 * in den vorangegangenen Beispielen kennengelernt haben
 */
public class SimpleMessageDrivenBean implements MessageListener {

   /**
    * Diese Methode wird vom Interface 'MessageListener' deklariert
    * und gerufen, wenn eine Nachricht in der Queue vorliegt.
    */
   public void onMessage(Message recvMsg) {

      // Casten zur Textmessage, nur erlaubt wenn es sich auch
      // tatsächlich um eine Textmessage handelt
      TextMessage textMsg = (TextMessage) recvMsg;

      try {
         // Ausgabe des enthaltenen Textes
         System.out.println("Received Message: "+textMsg.getText());
      } catch (JMSException jme) {
         jme.printStackTrace();
      }
   }
}
```

Wie Sie sehen, eignen sich Annotationen auch zur Konfiguration von Message Driven Beans und diesmal werden auch die Annotationen selbst von weiteren Annotationen parametrisiert.

Listing 9.25 Konfiguration der Message Driven Bean

```
...
@MessageDriven(activationConfig =
   {
      // Typ des Nachrichtenkanals: 'Queue' oder 'Topic'
      @ActivationConfigProperty(propertyName = "destinationType",
                                propertyValue = "javax.jms.Queue"),

      // Name der Queue an die diese MDB gebunden ist
      @ActivationConfigProperty(propertyName = "destination",
                                propertyValue = "queue/mcQueue")
   })
...
```

Diese Zeilen teilen dem Application Server mit, dass diese MDB an der zuvor in Listing 9.3 konfigurierten Queue `mcQueue` lauscht. Die Bereitstellung erfolgt dabei analog zu den Session Beans des vorangegangenen Beispiels:

- Übersetzen der Java-Klasse in Bytecode (*class*-Dateien)
- Verpacken der *class*-Datei in ein Java-Archiv (*jar*-Datei)
- Kopieren des Java-Archivs in das Verzeichnis *$JBOSS/server/default/deploy*
- Starten des JBoss Application Servers

Während des Startvorgangs parst der JBoss das Verzeichnis *deploy* nach annotierten MDBs und macht diese über den Namensdienst verfügbar. Wenn Sie die standardmäßigen Protokollierungseinstellungen nicht verändert haben, können Sie dies an folgender Ausgabezeile überprüfen:

```
...
[EJBContainer] STARTED EJB: de.akdabas...ejb.SimpleMessageDrivenBean
               ejbName: SimpleMessageDrivenBean
...
```

Listing 9.26
Bereitstellung einer annotierten MDB

Da auch der `MessageProducer` aus Listing 9.2 seine Nachrichten an die Queue `queue/mcQueue` sendet, können Sie die Funktionsweise Ihrer MDB durch Starten dieses Clients überprüfen. Bei Erfolg sollten Sie folgende Ausgabezeile des JBoss sehen.

```
...
INFO  [STDOUT] Received Message: Hello Queue!
...
```

Listing 9.27
Erfolgreicher Empfang der Nachricht aus Listing 9.2

9.12.1 Drei erfolgreich verheiratete Java-EE-Technologien

Listing 9.24 zeigt Ihnen außerdem, wie Sie drei zentrale Java-EE-Technologien gemeinsam einsetzen, um ihre jeweiligen Stärken zu verbinden und einfache wie flexible Backend-Dienste bereitzustellen:

- Enterprise JavaBeans (EJB)

  Der Application Server nimmt Ihnen das Erzeugen und Entfernen einzelner Beans ab. Außerdem kann er durch parallel arbeitende Instanzen eine Lastverteilung auf verschiedene Systeme bewirken.

- Java Message Service (JMS)

  JMS dient als Proxy zwischen Server und Client. Durch die lose Kopplung können beide unabhängig voneinander weiterentwickelt und sogar im laufenden Betrieb ausgetauscht werden.

- Java Naming and Directory Interface (JNDI)

  Dieser Service bringt beide Partner schließlich zueinander. Als abstrakter Objektspeicher gestattet er es, nahezu beliebige Objekte abzulegen und zu rekonstruieren. Auch hier kann die konkrete Implementierung nachträglich ausgetauscht werden.

9.12.2 Alternative Konfiguration mit Deployment Descriptor

Abschließend zeigt Ihnen dieser Abschnitt noch, wie Sie Ihre Message Driven Bean über einen EJB Deployment Descriptor einbinden können. Diese Technik war die einzige Möglichkeit der Konfiguration vor Einführung von EJB 3 und ist auch heute noch in vielen Systemen anzutreffen. Wie Sie sehen, beinhaltet der Descriptor dabei die gleichen Informationen wie die Java-Annotationen.

```
<!--
     Alternative Konfiguration mit einem Deployment Descriptor.
-->
<ejb-jar>
...
   <enterprise-beans>
```

Listing 9.28
Konfiguration der Message Driven Bean (ejb-jar.xml)

Listing 9.28 (Forts.)
Konfiguration der Message Driven Bean (ejb-jar.xml)

```
      ...
      <message-driven>
         <ejb-name>SimpleMessageDrivenBean</ejb-name>
         <ejb-class>
            de.akdabas.javaee.ejb.SimpleMessageDrivenBean
         </ejb-class>
         <transaction-type>Container</transaction-type>
         <message-driven-destination>
            <destination-type>javax.jms.Queue</destination-type>
         </message-driven-destination>
      <message-driven>
      ...
   </enterprise-beans>
...
</ejb-jar>
```

Analog zu den Session Beans des vorangegangenen Kapitels müssten Sie ohne den Einsatz von Annotationen diese Zeilen in den EJB Deployment Descriptor *ejb-xml.jar* eintragen und diese Datei im Verzeichnis *META-INF* des Java-Archivs bereitstellen.

9.13 Zusammenfassung

Der Java Message Service (JMS) eröffnet Java-Entwicklern den Zugang zur großen, weiten Welt der nachrichtenorientierten Systeme und unterstützt die beiden gängigen Kommunikationsverfahren gleichermaßen:

- Point-to-Point-Kommunikation, bei der ein Sender seine Nachricht über eine Queue an *genau einen Empfänger* sendet.
- Publish-and-Subscribe-Kommunikation, bei der jede Nachricht über eine Topic an *alle registrierten Empfänger* übermittelt wird.

JMS ist dabei kein Ersatz für das JavaMail-API, sondern ausschließlich für die Kommunikation zwischen verschiedenen Programmteilen verteilter Applikationen bestimmt, wie es etwa bei Enterprise JavaBeans der Fall ist. Von synchronen Kommunikationsverfahren, wie beispielsweise Remote Method Invocation (RMI), unterscheiden sich Nachrichten-Services durch:

1. Asynchrone Verarbeitung, ohne permanentes Polling
2. Indirekte Adressierung über Queues und Topics, wodurch Sender und Empfänger einer Nachricht einander nicht kennen
3. Zeitlich versetzte Bearbeitung, bei der die Bearbeitung einer Nachricht auch zu einem späteren Zeitpunkt erfolgen kann, falls der gewünschte Service beispielsweise vorübergehend nicht erreichbar ist
4. Gleichberechtigung aller Partner, bei der keine klare Trennung zwischen Client und Server mehr vorgenommen werden kann und jeder Nachrichtenempfänger auch gleichzeitig Sender sein kann

Damit eignen sich asynchrone Kommunikationssysteme vor allem für heterogene Systeme auf Basis verschiedener Plattformen oder für Situationen, in denen ein Partner nicht dauerhaft über ein Netzwerk erreicht

werden kann, wie es etwa bei mobilen Endgeräten und Notebooks der Fall ist. Die eingehenden Nachrichten werden dann einfach vom Message Broker zwischengespeichert und zu einem späteren Zeitpunkt zugestellt.

Message Driven Beans sind eine gute Alternative zu den Stateless Session-Beans, da sie neben dem Vorteil der asynchronen Verarbeitung durch das standardisierte Home- und Remote-Interface in der Regel einfacher zu erstellen und zu konfigurieren sind.

10

Persistenz – JavaBeans und Datenbanken

Persistenz – also die dauerhafte Speicherung von Daten – ist im Augenblick Thema vieler Fachartikel und auf dem Weg, sich in den nächsten Jahren zu einem ähnlich großen Hype zu entwickeln, wie es bei XML derzeit der Fall ist.

Dieses Kapitel führt Sie in das vollkommen neu entwickelte Java Persistenz API ein, das es Ihnen gestattet, Ihre relationalen Datenbankeinträge in einfache JavaBeans zu überführen und wie gewöhnliche Java-Objekte zu verwalten. Sie werden sehen, wie Sie Tabellen einer Datenbank über Listen in Java verwalten und wie einfach es sein kann, das Erzeugen, Manipulieren und Löschen von Java-Objekten mit Datenbanken zu synchronisieren.

10.1 Etablierte Persistenz-Standards

Der Zugriff auf Datenbanken und das Erzeugen von Einträgen mit Java-Programmen ist natürlich schon lange möglich. Im Laufe der Jahre haben sich dabei verschiedene Technologien entwickelt, welche auch heute noch im Einsatz sind und von vielen Java-Einsteigerbüchern vermittelt werden.

Info

Da diie *Java Database Connectivity (JDBC)* mittlerweile in jedem guten Java-Buch beschrieben wird, konzentriert sich dieses Kapitel auf die weitergehenden Standards.

Da Sun schon immer darauf geachtet hat, die Kopplung von Java und Datenbanken flexibel und erweiterbar zu halten, können diese Technologien auch immer noch verwendet werden und – zumindest für eine gewisse Übergangszeit – auch ein berechtigtes Nischendasein führen. Die-

ser Absatz stellt die verschiedenen Technologien kurz vor und ermöglicht es Ihnen, das neue Persistenz API in diesen Kontext einzuordnen.

10.1.1 Java Database Connectivity (JDBC)

Bereits seit den frühen Anfängen von Java gestattet es die Java Database Connectivity (JDBC) Java-Programmen, über eine standardisierte Schnittstelle mit SQL auf Datenbanken zuzugreifen. Hierfür definiert Sun ein einheitliches API, dessen Implementierung von vielen Datenbankherstellern bereitgestellt und weiterentwickelt wird.

Info

Open Database Connectivity ist eine von Microsoft entwickelte standardisierte Schnittstelle, die es ähnlich JDBC gestattet, Datenquellen zu manipulieren. Die Datenquellen reichen dabei von der Excel-Datei bis zum Datenbanksystem.

Das API kennt dabei unterschiedliche Typen von Implementierungsebenen, welche sich vor allem in der Leistungsfähigkeit und Flexibilität unterscheiden. Sie ermöglichen es beispielsweise, über eine JDBC-ODBC-Brücke jede ODBC-Datenquelle anzusprechen und somit sogar MS-Excel-Dateien mit Java auszulesen oder über hoch angepasste Spezialtreiber die volle Leistungsfähigkeit einer Oracle-Datenbank zu nutzen.

Größter Kritikpunkt von JDBC ist die Verwendung von SQL-Statements zur Abfrage der Datenbank, die vom Anwendungsentwickler sowohl Know-how im Bereich objektorientierter Programmierung als auch relationaler Datenbanken erfordert. Da die Schnittstelle – angelehnt an die Datenbank – mengenorientiert arbeitet, benötigt die Überführung der Daten von und in JavaBeans viel Java-Code.

Info

Die *Standard Query Language (SQL)* ist eine an die englische Umgangssprache angelehnte Abfragesprache für *Relationale Datenbanksysteme*. Sie ermöglicht neben der Abfrage (Query) auch die Manipulation der Datenbank bezüglich der Datensätze (Insert, Update, Delete) und hinsichtlich ihrer Tabellen (Spalten löschen oder hinzufügen, Tabellen erstellen, Ändern von Rechten).

Heute wird die JDBC im Grunde nur noch von den nachfolgend beschriebenen Frameworks eingesetzt oder kommt bei Anwendungen zum Einsatz, bei denen eine hohe Optimierung der SQL-Statements erreicht werden muss. Da die Frameworks dem Anwendungsentwickler eine komfortable, objektorientierte Schnittstelle zur Verfügung stellen und die Entwicklung von Applikationen stark beschleunigen können, werden diese allerdings zunehmend häufiger eingesetzt.

10.1.2 EJBs Entity Beans

Eine der ersten Möglichkeiten, Datensätze in Form von JavaBeans zu manipulieren, waren spezielle Enterprise JavaBeans, die als Entity Beans bezeichnet werden. Entity Beans benötigen einen EJB Container als Laufzeitumgebung und besitzen einen fest vordefinierten Lebenszyklus. Hierdurch gestatten Sie es dem Entwickler der EJB, die Datenbankzugriffslogik zu standardisieren und für den Anwendungsprogrammierer zu kapseln.

Mit der später hinzugekommenen Container Managed Persistence (CMP) ist es sogar möglich, ganz auf den Einsatz von SQL zu verzichten und die Verknüpfung von Variablen und Datenbankspalten durch Deployment Descriptors in Form von XML-Dateien zu definieren.

EJB Entity Beans werden mit der Einführung des in diesem Kapitel vorgestellten, einheitlichen Persistenz API als solches nicht mehr benötigt und werden aufgrund des mit ihrem komplexen Lebenszyklus verbundenen Aufwands zunehmend seltener anzutreffen sein. Lassen Sie sich also nicht verwirren: Auch wenn die JavaBeans des neuen Persistenz API gelegentlich als Entity Beans bezeichnet werden, haben diese in punkto Lebenszyklus, Konfiguration und Beschränkung auf einen Application Server nichts mit ihren Namensvettern aus der EJB 2.x-Spezifikation gemein.

10.1.3 Java Data Objects

Wenn man vor Java EE 5 die automatische vom EJB Container überwachte Persistenz auch außerhalb eines Application Servers einsetzen wollte, konnte man auf diese Technologie zurückgreifen.

Die *Java Data Objects* (JDO)-Spezifikation beschreibt ein Framework bestehend aus einheitlichem API, einem Persistenz-Manager und einer anbieterspezifischen Implementierung (SPI), welche es gestattete, den Zustand von JavaBeans in einem Hintergrundspeicher festzuhalten.

Hierfür wurde der kompilierte Bytecode mit einem Postprozessor angereichert und so mit der benötigten Datenbankunterstützung versehen. Der große Vorteil dieser Technologie war, dass es sich bei den JavaBeans um gewöhnliche Java-Objekte (POJO) ohne speziellen Lebenszyklus handelte, die einfach zweimal übersetzt wurden und anschließend mit der Datenbank synchronisiert werden konnten. Der Entwickler sah also reinen Java-Code und benötigte keinerlei SQL-Know-how.

Mit den Möglichkeiten von Annotationen gibt es heute andere Technologien, um Java-Objekte mit Metadaten über Datenbanken zu versehen, weshalb JDO mit seinen angereicherten Klassen wohl ebenfalls zunehmend von dem neuen Persistenz API verdrängt werden wird. Außerdem muss diese nun nicht mehr zwischen Persistenz im Application Server und in Desktop-Java-Programmen unterscheiden, so dass ein Entwickler nur noch eine einzige Technologie beherrschen muss.

10.1.4 Freie OpenSource Frameworks

Wegen der oben genannten Einschränkungen der von Sun spezifizierten Frameworks und da sich neue Ideen vor allem in der OpenSource-Szene durchsetzen können, haben sich im Laufe der letzten Jahre auch eine Vielzahl freier Persistenz-Frameworks entwickelt, von denen hier das wohl bekannteste Framework Hibernate (*www.hibernate.org*) vorgestellt werden soll.

Hibernate gestattet es wie JDO, einfache JavaBeans zur Abbildung objektrelationaler Daten zu verwenden. Da das Framework hierfür keine besondere Laufzeitumgebung benötigt, kann es sowohl für einfache Java-Programme als auch innerhalb eines Application Servers eingesetzt werden.

Info

Der JBoss Application Server verwendet übrigens intern das Hibernate Framework, um Persistenz-Dienste bereitzustellen. Wenn Sie also innerhalb des JBoss mit dem Persistenz API arbeiten, greifen Sie indirekt auch auf Hibernate zurück.

Da sich Hibernate inzwischen fest etabliert hat und – da es nicht an Gremien und Standards gebunden ist – neue Ideen und Techniken schneller umsetzen kann, wird es nach Meinung des Autors in den nächsten Jahren zu einer friedlichen Koexistenz zwischen Hibernate und dem neuen Persistenz API kommen, zumal beide auf ganz ähnlichen Technologien basieren.

10.1.5 Das Persistenz API

Man könnte sagen: Was JavaServer Faces für Struts sind, ist das neue Persistenz API für Hibernate. Genau wie Sun beim JSF Framework viele Ideen und Ansätze von Struts übernommen hat, standardisiert auch das neue Persistenz API unter Verwendung in der OpenSource-Szene entwickelter Ideen den »objektorientierten« Zugriff auf Datenbanken

Info

Das neue Persistenz API gestattet es, Objekte in nahezu jeder Umgebung und Datenbank zu speichern, ganz nach dem Motto »*Write once, store anyway!*«.

Der große Vorteil des Persistenz API gegenüber älteren Technologien von Sun ist dabei, dass die Anforderungen an Konfiguration und Lebenszyklus wesentlich vereinfacht wurden und nun eine einheitliche Schnittstelle für alle Arten von Java-Programmen verwendet werden kann.

Auch das Persistenz API greift auf das bewährte Muster aus Anwendungsschnittstelle (API), Framework und Implementierung (SPI) zurück und ist so in der Lage, mit vielen verschiedenen Anbietern zusammenzuarbeiten.

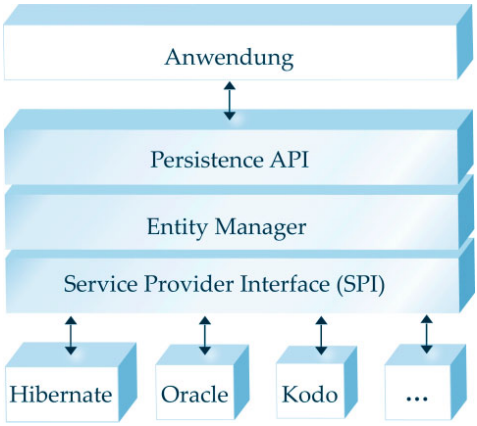


Abbildung 10.1
Aufbau des Persistenz API

10.2 Bezug und Installation

Um Ihnen den Umgang mit dem Persistenz API in einer beliebigen Java-EE-Umgebung und damit außerhalb eines Application Servers zu demonstrieren, werden die ersten Beispiele auf der Hibernate-Implementierung arbeiten. Zwar verwendet der JBoss diese auch intern, dennoch enthält er nicht alle benötigten Bibliotheken.

Bitte laden Sie sich deshalb ein aktuelles Hibernate Release von der Homepage *http://www.hibernate.org* herunter und entpacken Sie es in ein Verzeichnis Ihrer Wahl. Anschließend müssen Sie lediglich die enthaltenen Java-Bibliotheken (JAR) zum Beispiel über den Ordner *lib* im Projektverzeichnis im Classpath verfügbar machen.

Tipp

Ein zum Zeitpunkt der Auflage aktuelles Hibernate-Release sowie ein vorkonfiguriertes Projektverzeichnis finden Sie natürlich auch auf der beiliegenden CD.

10.2.1 Hypersonic SQL Database (HSQLDB)

Um die dauerhafte Speicherung von Daten über die Laufzeit eines Java-Programms zu demonstrieren, benötigen Sie natürlich eine Datenbank und da wir nicht davon ausgehen, dass auf Ihrem System bereits eine solche installiert ist, greifen wir in diesem Buch auf die Hypersonic SQL Database (HSQLDB) zurück. Dabei handelt es sich um eine kleine, vollständig in Java geschriebene Datenbank, welche auch vom JBoss verwendet und zusammen mit diesem ausgeliefert wird.

Tipp

Die HSQLDB ist eine vollständig in Java implementierte Datenbank, welche von vielen OpenSource-Produkten, wie beispielsweise OpenOffice, als Hintergrundspeicher eingesetzt wird.
Wenn Sie bereits über eine installierte Datenbank wie beispielsweise MySQL verfügen, können Sie zum Test der Beispiele natürlich auch diese verwenden.

Um die HSQLDB zu installieren, können Sie entweder die im Verzeichnis *$JBOSS/server/default/lib* enthaltene Datei *hsqldb.jar* den Hibernate-Klassen hinzufügen oder sich auch hier das vollständige Release von der Homepage *http://www.hsqldb.org/* herunterladen.

10.3 Eine einfache JavaBean

Wie so oft, beginnen wir vor der eigentlichen Implementierung mit einer einfachen JavaBean. Diese soll anschließend für die nachfolgenden Beispiele stellvertretend als Geschäftsobjekt dienen und was sollte sich in diesem Fall besser eignen als die schon bekannte `Address`-Bean. Um Sie allerdings nicht mit bereits Bekanntem zu langweilen, werden wir die Leistungsfähigkeit der `Address`-Bean in diesem Kapitel nochmals erweitern und nun auch mehrere E-Mail-Adressen pro Kontakt zulassen. Hierfür beschneiden wir diese allerdings zunächst einmal auf das Wesentlichste.

Listing 10.1 Eine einfache Address-Bean

```
package de.akdabas.javaee.persistence.bean;

import javax.persistence.Id;
import javax.persistence.Column;
import javax.persistence.Entity;

@Entity
public class SimpleAddress {

   private String lastName;
   private String firstName;

   /**
    * Standard-Konstruktor; Dieser wird vom Persistenz-Framework
    * zur Verwaltung der Klasse mit dem Reflection-API benötigt;
    */
   public SimpleAddress() {}

   @Id
   public String getLastName() {
      return lastName;
   }
   public void setLastName(String lastName) {
      this.lastName = lastName;
   }

   @Column
   public String getFirstName() {
      return firstName;
```

```
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
}
```

Listing 10.1 (Forts.)
Eine einfache Address-Bean

Tipp

Annotationen können über den Gettern und Settern der Attribute einer JavaBean stehen. Es genügt aber vollkommen, sie über die Getter zu schreiben.

Wie Sie sehen, handelt es sich auch bei dieser Klasse um eine ganz gewöhnliche JavaBean ohne speziellen Lebenszyklus, deren Bindung an die Datenbank vollständig über Java-Annotationen aus dem Package `javax.persistence` realisiert wird. Diese haben dabei folgende Funktionen:

- `@Entity`
  Ähnlich wie die Annotation `@Stateful` bei den EJBs, markiert diese Annotation die Klasse als persistierbares Objekt.
- `@Id`
  Diese Annotation markiert ein Attribut als Primärschlüssel des Datensatzes, der eindeutig über alle Datensätze sein muss. In diesem Fall würden die Nachnamen in die Spalte `LASTNAME` der Tabelle eingetragen werden.
- `@Column`
  Natürlich können nicht alle Attribute einer JavaBean Primärschlüssel (ID) sein. Die Annotation `@Column` teilt dem Framework mit, dass dieses Attribut ebenfalls mit einer Spalte der Tabelle synchronisiert werden soll. In diesem Beispiel würde der Vorname eines Datensatzes in der gleichnamigen Spalte `FIRSTNAME` gespeichert werden.

Diese drei Annotationen reichen bereits aus, um die JavaBean mit der Tabelle einer Datenbank zu verknüpfen, wobei das Framework die Namen für Tabellen und Spalten standardmäßig aus den Namen der Klassen und Attribute ableitet. Für die Bean aus Listing 10.1 würde demzufolge eine Tabelle mit dem Namen `SIMPLEADDRESS` und den Spalten `FIRSTNAME` und `LASTNAME` angelegt werden.

10.3.1 Verfeinerungen

Obwohl sich die `SimpleAddress`-Bean bereits für alle Beispiele einsetzen ließe und vom Framework kommentarlos akzeptiert werden würde, weist sie dennoch einige Beschränkungen und Sicherheitslücken auf. Deshalb empfehle ich Ihnen in echten Projekten, von einem solchen Minimalismus abzusehen und einige zusätzliche Angaben zu machen.

Tipp

Listing 10.1 zeigt Ihnen lediglich die Minimalanforderungen an eine persistierbare JavaBean. Für produktive Anwendungen sollten Sie sich deshalb eher an Listing 10.2 orientieren.

Einige sehr interessante Verbesserungen, die Sie im wirklichen Leben vor Kopfschmerzen und aufwändiger Fehlersuche bewahren können, sind folgende Meta-Informationen:

- Deklarieren Sie den Namen der Tabelle und Spalten explizit.

  Auf diese Weise können Sie die Attribute der JavaBean später auch gefahrlos umbenennen, ohne alle bisher gespeicherten Datensätze zu verlieren.

- Verwenden Sie eine generische ID.

  In obigem Beispiel verwenden Sie den Nachnamen der Adresse als eindeutiges Identifizierungsmerkmal. Wenn Sie allerdings an die vielen Schmidts, Meiers und Schulzes denken, die Sie im Telefonbuch finden, erkennen Sie schnell die Limitation dieses Ansatzes.

  Sollte die Geschäftsschnittstelle Ihrer JavaBean kein Attribut enthalten, welches über alle Datensätze hinweg eindeutig ist, fügen Sie dieses einfach in Form einer *generischen ID* hinzu. Bei einer generischen ID handelt es sich häufig um ein numerisches Attribut (`int`, `long`), welches für die Geschäftslogik nicht zwingend erforderlich ist, aber eine einfache und sichere Unterscheidung verschiedener Datensätze ermöglicht und versehentliches Überschreiben bestehender Datensätze vermeidet.

  Da sich einfache, numerische Attribute sowohl auf Datenbank- als auch auf JavaBean-Ebene besser indizieren und vergleichen lassen, sind generische IDs auch dann von Vorteil, wenn es sich beim Primärschlüssel um eine Zeichenkette (`String`) oder eine Kombination verschiedener Attribute, also einen zusammengesetzten Schlüssel handelt.

Info

Für *generische ID* wird in der Literatur oft auch der Begriff »stellvertretender Schlüssel« (engl. surrogate key) verwendet.

- Veröffentlichen Sie nur Konstruktoren, die alle Pflichtfelder der JavaBean enthalten.

  Die `SimpleAddress`-Bean mit obiger Schnittstelle lässt das Erzeugen von Datensätzen zu, deren Attribute nur aus `null`-Werten bestehen. Bei unachtsamer Programmierung kann dies zu schwer identifizierbaren Fehlern führen. Zwingen Sie den Anwendungsprogrammierer deshalb dazu, die Werte in einem Minimalkonstruktor zu übergeben.

- Deklarieren Sie Eigenschaften wie `nullable` für die Tabellen der Datenbank.

  Auch die Tabellen einer Datenbank weisen Eigenschaften wie »Ist Pflichtfeld«, »Kann aktualisiert werden« oder die maximale Länge für einen Eintrag auf. Wann immer Sie diese Eigenschaften kennen, sollten Sie diese auch angeben. Das Framework kann diese Konsistenzbedingung dann überprüfen und Sie mit aussagekräftigen Fehlermeldungen auf fehlerhafte Datensätze hinweisen.

Um diese und andere Eigenschaften einer persistierbaren JavaBean zu spezifizieren, existieren eine Reihe von weiteren Annotationen und Parametern. Diese sind – wie gesagt – optional, verbessern jedoch das Design Ihrer Anwendungen und machen diese sicherer. Listing 10.2 zeigt eine `Address`-Bean, die einige weitere, wenn auch nicht alle, möglichen Annotationen enthält.

Listing 10.2
Eine erweiterte AddressBean

```
package de.akdabas.javaee.persistence.bean;

import javax.persistence.Id;
import javax.persistence.Entity;
import javax.persistence.Column;
import javax.persistence.Table;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;

@Entity
@Table(name = "ADDRESSES")
public class Address {

   private int id;
   private String lastName;
   private String firstName;

   /**
    * Standard-Konstruktor; Dieser wird vom Persistenz-Framework
    * zur Verwaltung der Klasse mit dem Reflection-API benötigt;
    */
   protected Address() {}

   /**
    * Minimaler Konstruktor; Dieser kann von Clients zur Erzeugung
    * neuer Objekte verwendet werden und enthält alle Pflichtfelder
    */
   public Address(String lastName){
      this.lastName = lastName;
   }

   /**
    * Vollständiger Kontruktor; Dieser enthält alle Attribute dieses
    * Objektes und erleichtert die Erzeugung von neuen Objekten
    */
   public Address(String lastName, String firstName){
      this.lastName = lastName;
      this.firstName = firstName;
   }

   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   @Column(name="id")
   public int getId() {
      return id;
```

Listing 10.2 (Forts.) Eine erweiterte AddressBean

```
    }
    protected void setId(int id) {
        this.id = id;
    }

    @Column(name="last_name", nullable=false, length=120)
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    @Column(name="first_name", length=120)
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    /** Nur für Debug-Zwecke und für die nachfolgenden Beispiele */
    public String toString(){
        return "Address [" + id +"]: " + firstName + " " + lastName;
    }
}
```

Konstruktoren

Wie Sie sehen, ist der leere Standardkonstruktor geblieben, da dieser vom Persistenz-Framework benötigt wird. Er ist nun allerdings geschützt (`protected`) und kann von Anwendungsprogrammierern nicht mehr verwendet werden. Für diese stehen zwei weitere Konstruktoren bereit, von denen der eine alle zwingend benötigten und der andere der Einfachheit halber alle Attribute einer Adresse übernimmt.

Eine generische ID

Außerdem ist ein neues Attribut `id` hinzugekommen, hinter dem sich natürlich nichts anderes als der generische Schlüssel dieses Datensatzes verbirgt. Bemerkenswert an dieser Stelle ist, dass die ID zwar von jedem Objekt ausgelesen (`public`), jedoch nicht von außen verändert werden kann (`protected`). Da die IDs eines Datensatzes ausschließlich vom Framework verwaltet werden sollten, ist dies ein weiterer Schutz vor missbräuchlichem Gebrauch der JavaBean.

Dies bedeutet natürlich auch, dass eine neue erzeugte `Address`-Bean zunächst ohne ID ausgestattet ist und diese beim Speichern vom Framework erzeugt werden muss. Um dies anzuzeigen, verwenden Sie einfach die folgende Annotation:

Listing 10.3 Automatisches Erzeugen von IDs

```
...
    @GeneratedValue(strategy = GenerationType.AUTO)
...
```

Angabe des Tabellennamens

Da sich Tabellennamen in der Nomenklatur oftmals von Java-Objekten unterscheiden, ermöglicht es die Annotation `@Table`, diesen und andere Eigenschaften wie Unique Constraints anzugeben.

Info

Ein *Unique Constraint* stellt sicher, dass der Wert einer Spalte oder die Kombination von Werten einer Gruppe von Spalten über die gesamte Tabelle eindeutig für einen Datensatz sind.
Als einfachster Unique Constraint kann der Primärschlüssel einer Spalte verstanden werden, wobei dieser zusätzlich nicht `Null` sein darf.

```
...
@Table(name = "ADDRESSES")
...
```

Listing 10.4
Hinterlegen von Eigenschaften der Tabelle

Attribute einer Datenbankspalte

Schließlich enthalten die `@Column`-Annotationen in Listing 10.2 einige zusätzliche, optionale Angaben über die jeweilige Datenbankspalte, von denen Ihnen Tabelle 10.1 die wichtigsten zeigt.

Name	Typ	Beschreibung	Standardwert
`name`	`String`	Name der Tabellenspalte	Name des Bean-Attributs
`length`	`int`	Maximale Länge des Eintrags	`255`
`table`	`String`	Name einer Tabelle	Name der Tabelle dieser JavaBean
`nullable`	`boolean`	Sind `null`-Werte erlaubt?	`true`
`insertable`	`boolean`	Darf dieser Wert mit INSERT Statements gepflegt werden?	`true`
`updateable`	`boolean`	Darf dieser Wert mit UPDATE Statements gepflegt werden?	`true`

Tabelle 10.1
Die wichtigsten Attribute der @Column-Annotation

Die Methode toString()

Diese Methode wird natürlich nicht zwingend für die nachfolgenden Beispiele benötigt, sie erleichtert das Nachvollziehen jedoch ungemein und ist auch für die schnelle Ausgabe von Debug-Meldungen von Vorteil. Da Ihnen diese Methode gerade bei der Fehlersuche wertvolle Zeit sparen kann, empfehle ich Ihnen, den Mehraufwand für die Implementierung in Kauf zu nehmen, wobei die zurückgegebene Zeichenkette alle relevanten Informationen der JavaBean enthalten sollte.

Statt die Methode `toString()` zu überschreiben, könnten Sie die Debug-Informationen auch in einer separaten Methode kapseln und diese zur Ausgabe verwenden. Der Vorteil der Methode `toString()` besteht jedoch darin, dass diese automatisch gerufen wird, sobald ein Objekt – zum Beispiel über einen Ausgabestrom – serialisiert wird. Auf diese Weise ist es ein Leichtes, die Informationen beispielsweise über einen Logger zu protokollieren.

10.3.2 Konfiguration

Nachdem Sie die JavaBean nun mit allen empfohlenen Annotationen versehen und kompiliert haben, müssen Sie das Persistenz-Framework noch einmalig konfigurieren. Hierzu dient der so genannte Persistence Descriptor (`persistence.xml`), welcher nach dem Kompilieren im Verzeichnis *META-INF/classes* abgelegt werden muss.

Listing 10.5
Der Persistence Descriptor (persistence.xml)

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!-- Persistence Descriptor zur Konfiguration des Frameworks -->
<persistence>

  <!-- Symbolischer Name der Persistenz-Unit und der zu verwendende
       Transaktionstyp (Standard ist 'JTA')  -->
  <persistence-unit name="masterclass"
                    transaction-type="RESOURCE_LOCAL">

    <!-- Zu verwendende Service Provider Implementierung -->
    <provider>org.hibernate.ejb.HibernatePersistence</provider>

    <!-- Persistierbare Klassen dieser Persistenz-Unit -->
    <class>de.akdabas.javaee.persistence.bean.Address</class>
    <class>de.akdabas.javaee.persistence.bean.SimpleAddress</class>

    <!-- Konfiguration der Service Provider Implementierung -->
    <properties>

      <!-- Name des intern verwendeten JDBC-Treibers -->
      <property name="hibernate.connection.driver_class"
                value="org.hsqldb.jdbcDriver"/>

      <!-- URL der zu verwendenden Datenbank -->
      <property name="hibernate.connection.url"
                value="jdbc:hsqldb:data/masterclass"/>

      <!-- SQL-Dialect, den Hibernate verwenden soll -->
      <property name="hibernate.dialect"
                value="org.hibernate.dialect.HSQLDialect"/>

      <!-- Benutzername und Passwort; Standardwerte der HSQLDB -->
      <property name="hibernate.connection.username" value="sa"/>
      <property name="hibernate.connection.password" value=""/>

      <!-- Flag, ob Tabellen automatisch erzeugt werden sollen -->
      <property name="hibernate.hbm2ddl.auto" value="create"/>

      <!-- Flag, ob SQL-Statements ausgegeben werden sollen -->
      <property name="hibernate.show_sql" value="true"/>
    </properties>
  </persistence-unit>
</persistence>
```

Allgemeine Konfiguration

Der in Listing 10.5 gezeigte Persistence Descriptor unterteilt sich in zwei Abschnitte, wobei der erste standardisiert ist und der zweite implementierungsspezifische Konfigurationseinstellungen enthält.

Tipp

Die nachfolgenden Beispiele verwenden eine In-Memory HSQLDB, d.h., die Datenbank wird bei jedem Start neu initialisiert.
Mehr zum Thema Referenzieren von Datensätzen erfahren Sie im Abschnitt 10.6.

Zunächst einmal werden alle zu persistierenden JavaBeans einer oder mehreren Persistenz-Units zugeordnet. Diese beschreiben eine abgeschlossene Einheit von Datenbanktabellen, die sich untereinander referenzieren können.

In obigem Beispiel definieren Sie hierfür die Persistenz-Unit `masterclass`, zu der die beiden Klassen `SimpleAddress` und `Address` gehören.

Außerdem müssen Sie über das Tag `<provider>` die zu verwendende Service Provider-Implementierung angegeben, über die die Daten mit der Datenbank synchronisiert werden sollen.

Als Letztes ist die Angabe des Transaktionstyps (`transaction-type`) wichtig. Dieses Attribut ist zwar optional, aber der Standardwert `JTA` ist für den Einsatz innerhalb eines Application Servers gedacht und funktioniert außerhalb eines solchen nicht. Wenn Sie das Persistenz-Framework also für Standard-Java-Anwendungen einsetzen möchten, müssen Sie den Transaktionstyp auf `RESOURCE_LOCAL` setzen.

```
...
  <!-- Symbolischer Name der Persistenz-Unit und der zu verwendende
       Transaktionstyp; Standard ist 'JTA'  -->
  <persistence-unit name="masterclass"
                    transaction-type="RESOURCE_LOCAL">
...
```

Listing 10.6
Setzen des Transaktionstyps für Standard-Java-Umgebungen

Konfiguration des Service Providers

Im zweiten Teil des Persistence Descriptor (*persistence.xml*) finden Sie schließlich innerhalb des Tags `<properties>` Parameter, die zur Konfiguration der verwendeten Implementierung dienen. Um eine möglichst große Anzahl von Anbietern zu unterstützen, sind diese Angaben nicht von der Spezifikation vorgegeben und können sich von Provider zu Provider und sogar zwischen verschiedenen Versionen einer Implementierung unterscheiden.

Welche Angaben hier tatsächlich benötigt werden, müssen Sie also der jeweiligen Dokumentation entnehmen. Eine Übersicht für die Konfiguration von Hibernate erhalten Sie z.B. unter der Adresse:

http://www.hibernate.org/hib_docs/entitymanager/reference/en/html/

10.4 Arbeiten mit der JavaBean

Nachdem Sie die `Address`-Bean nun erstellt und das Persistenz-Framework über die Datei *META-INF/persistence.xml* konfiguriert haben, können Sie sich auf die Beispiele stürzen und beginnen, Kontakte zu verarbeiten. Dabei werden Ihnen die folgenden Absätze Listing-Fragmente vorstellen, in denen die jeweils wichtigsten Anweisungen zusammengefasst sind.

Alle Beispiele sind anschließend Bestandteil der Java-Klasse `PersistenceClient`, welche sich auch auf der beiliegenden CD befindet und einfach gestartet werden kann.

10.4.1 Anlegen eines neuen Datensatzes

Da die Datenbank zu Beginn noch keine Datensätze enthält, ist es sinnvoll, mit der Erzeugung neuer Einträge zu beginnen. Das folgende Listing-Fragment zeigt die hierfür notwendigen Java-Anweisungen.

Listing 10.7 Anlegen eines neuen Datensatzes

```
package de.akdabas.javaee.persistence.client;
import javax.persistence.Persistence;
import javax.persistence.EntityManager;
import javax.persistence.EntityTransaction;
import javax.persistence.EntityManagerFactory;
import de.akdabas.javaee.persistence.bean.Address;

/**
 * Dieser Client arbeitet mit Address-Objekten
 */
public class PersistenceClient {

   public static void main(String[] args) {
      try {
         // Erzeugen der EntityManagerFactory
         EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("masterclass");

         // Anlegen zweier identischer neuer Datensätze
         createAddress(emf);
         createAddress(emf);


         // ...
         // an dieser Stelle werden die Methoden der folgenden
         // Beispiele aufgerufen
         // ...


         // Schließen der EntityManagerFactory und Freigeben der
         // belegten Ressourcen
         emf.close();
      } catch (Exception exc) {
         exc.printStackTrace();
      }
   }

   /**
    * Diese Methode legt einen neuen Datensatz an
    */
   private static void createAddress(EntityManagerFactory emf) {
```

Listing 10.7 (Forts.)
Anlegen eines neuen Datensatzes

```
      // Erzeugen eines Entity-Managers
      EntityManager manager = emf.createEntityManager();

      // Beginn einer neuen Transanktion
      EntityTransaction tx = manager.getTransaction();
      tx.begin();

      // Erzeugen einer neuen Adresse
      Address address = new Address("Stark", "Thomas");

      // Speichern der Adresse in der Datenbank
      manager.persist(address);

      // Beenden der Transaktion mit einen Commit
      tx.commit();

      // Freigabe der Ressourcen
      manager.close();

      // Ausgabe der ID
      System.out.println(address.toString());
   }
...
}
```

Die EntityManagerFactory

Wie Sie sehen, wird in diesem Beispiel zunächst in der Methode `main()` eine `EntityManagerFactory` erzeugt und anschließend der Methode `createAddress()` übergeben. Hintergrund ist, dass das Erzeugen der `EntityManagerFactory` ein relativ ressourcenintensiver Prozess ist, der innerhalb einer Anwendung nach Möglichkeit nur ein einziges Mal gerufen werden sollte.

Anschließend können Sie über die Factory beliebig viele `EntityManager` erstellen und mit diesen auf der Datenbank arbeiten. Bevor die Anwendung anschließend geschlossen wird, sollten Sie sicherstellen, dass die Ressourcen der `EntityManagerFactory` durch den Aufruf der Methode `close()` wieder freigegeben werden.

Ob die Methode `close()` einer `EntityManagerFactory` bereits gerufen wurde, können Sie über die Methode `isOpen()` überprüfen.

Listing 10.8
Arbeiten mit der EntityManagerFactory (schematisch)

```
...
   try {
      // Erzeugen der EntityManagerFactory
      EntityManagerFactory emf =
         Persistence.createEntityManagerFactory("NameDerUnit");

      // An dieser Stelle können beleibig viele EntityManager
      // erzeugt werden, mit denen auf die Datenbank zugegriffen
      // werden kann

      // Schließen der EntityManagerFactory und Freigeben der
      // belegten Ressourcen
      emf.close();
   } catch (Exception exc) {
      exc.printStackTrace();
   }
...
```

Um eine `EntityManagerFactory` zu erzeugen, übergeben Sie dieser einfach den symbolischen Namen der im Persistence Descriptor (*persistence.xml*) hinterlegten Konfiguration. Hierfür muss diese im Verzeichnis `META-INF` unterhalb des Ordners `classes` abgelegt werden.

Der EntityManager

Der über die `EntityManagerFactory` erzeugte `EntityManager` stellt Ihre lokale Schnittstelle zur Datenbank dar. Über dieses Objekt werden Sie in den nächsten Beispielen Datensätze anlegen, suchen, verändern und löschen. Dabei ist zu beachten, dass alle Operationen, die über das Lesen von Datensätzen hinausgehen, innerhalb einer Transaktion stattfinden müssen, die nach Abschluss der Operation über ein `commit()` bestätigt oder via `rollback()` verworfen werden muss.

Tipp

Die Datenbankoperationen Create, Retrieve, Update und Delete werden in der Literatur oftmals mit CRUD abgekürzt.

Anschließend ist es guter Programmierstil, wenn Sie die mit der Erzeugung des `EntityManager` belegten Ressourcen wieder freigeben, indem Sie die Methode `close()` rufen. Sie vermeiden außerdem Probleme mit dem Ressourcenmanagement.

Listing 10.9 Arbeit mit einem EntityManager (schematisch)

```
...
   // Erzeugen eines Entity-Managers
   EntityManager manager = emf.createEntityManager();

   // Beginn einer neuen Transanktion
   EntityTransaction tx = manager.getTransaction();
   tx.begin();

   // ... Hier können die verschiedenen Datenbank-Operationen
   //     ausgeführt werden ...

   // Beenden der Transaktion mit 'Commit' oder 'Rollback'
   tx.commit();

   // Freigabe der Ressourcen
   manager.close();
...
```

Anlegen eines neuen Datensatzes

Nachdem Sie die Funktionsweise des `EntityManager` und der `EntityManagerFactory` verstanden haben, ist das Erzeugen eines neuen `Address`-Datensatzes ein Kinderspiel:

- Erzeugen Sie die JavaBean über den gewünschten Konstruktor.
- Übergeben Sie das Objekt der Methode `persist()` des EntityManager.

Listing 10.10
Anlegen eines neuen Datensatzes (Create)

```
...
   // Erzeugen einer neuen Adresse
   Address address = new Address("Stark", "Thomas");

   // Speichern der Adresse in der Datenbank
   manager.persist(address);
...
```

Diese beiden Zeilen genügen, um einen neuen Datensatz in der Tabelle `Addresses` der Datenbank anzulegen. Wie Sie sehen, benötigen Sie hierfür keine einzige Zeile SQL und dank des Flag `hibernate.hbm2ddl.auto` im Persistence Descriptor (Listing 10.5) werden auch die benötigten Tabellen automatisch angepasst.

10.4.2 Suche anhand des Primärschlüssels

Ähnlich einfach wie das Anlegen neuer Datensätze gestaltet sich die Suche eines Datensatzes anhand seines Primärschlüssels (ID). Und da Sie hierbei nur lesend auf die Datenbank zugreifen, müssen Sie nicht einmal eine Transaktion öffnen.

Listing 10.11
Suche eines Datensatzes anhand seines Primärschlüssels

```
...
   /**
    * Sucht den Datensatz mit dem übergebenen Primary Key
    */
   private static void findByPrimaryKey(EntityManagerFactory emf,
                                        int primaryKey) {

      // Erzeugen eines EntityManagers
      EntityManager manager = emf.createEntityManager();

      // Suche einer Adresse
      Address address = manager.find(Address.class, primaryKey);

      // Überprüfen, ob der Datensatz gefunden wurde
      if (address != null) {
         System.out.println(address.toString());
      } else {
         System.out.println("Datensatz nicht vorhanden!");
      }

      // Freigabe der Ressourcen
      manager.close();
   }
...
```

Die Methode `find()` des EntityManager übernimmt die Klasse des zu suchenden Objekts und Ihren Primärschlüssel und gibt entweder das zugehörige Objekt oder `null` zurück. Die übergebene Klasse sollte dabei Bestandteil dieser Persistenz-Unit sein (Listing 10.5).

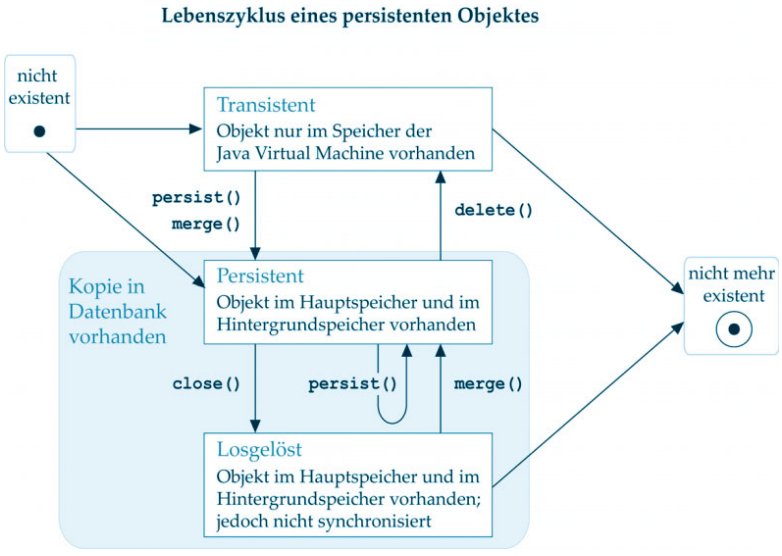
Info

Die Methode `find()` ist übrigens, wie auch andere Teile des Persistenz API, generisch programmiert und gibt Ihnen deshalb stets Objekte vom Typ des ersten Parameters oder `null` zurück. Sie müssen Ihren Code deshalb nicht durch einen zusätzlichen Cast des Rückgabewerts nach Address aufblähen.

10.4.3 Zustände einer JavaBean

Bevor Sie im nächsten Absatz sehen, wie Sie auch einen bestehenden Datensatz manipulieren und die Änderungen anschließend mit der Datenbank synchronisieren können, benötigen Sie noch einiges Zusatzwissen über die verschiedenen Zustände, in denen sich ein Datensatz befinden kann.

Abbildung 10.2
Lebenszyklus eines Datensatzes



Transienter Zustand

Wenn Sie ein neues Java-Objekt erzeugen, befindet es sich zunächst nur im Hauptspeicher der Java Virtual Machine (VM). Die Datenbank im Hintergrund kann schließlich noch nichts von dem neuen Datensatz ahnen. Das Objekt befindet sich jetzt im so genannten *transienten Zustand*. Wenn Sie vergessen, das Objekt zu speichern, und der Java Garbage Collector es aus dem Speicher entfernt, gehen alle Daten der JavaBean verloren.

Info

Als *transient* (engl. zeitlich vorübergehend) bezeichnet man Objekte, deren Daten nur im Hauptspeicher verfügbar sind und die verloren gehen, wenn das Programm terminiert.

Sobald Sie den Zustand eines transienten Objekts über einen `EntityManager` in die Datenbank übernehmen (Listing 10.10), geht dieses Objekt in den persistenten Zustand über.

Persistenter Zustand

Als persistent bezeichnet man hingegen alle Objekte, deren Zustand sich (auch) auf einem nichtflüchtigen Speicher wie der Datenbank befindet und die somit auch über einen Neustart des Programms erhalten bleiben.

Kopien persistenter Objekte können sich auch im Hauptspeicher befinden, etwa wenn Sie einen bereits zuvor angelegten Datensatz über die Methode `find()` laden. Persistente Objekte sind dabei über den `EntityManager`, über den sie geladen oder persistiert wurden, mit der Kopie in der Datenbank verknüpft.

Losgelöster Zustand

Doch was passiert, wenn Sie bei einem persistenten Objekt die Verbindung zur Datenbank kappen, etwa indem Sie die Methode `close()` der `EntityManagerFactory` aufrufen (Listing 10.8)?

In diesem Fall wird das Objekt natürlich nicht in der Datenbank gelöscht und auch die Referenz auf den Hauptspeicher bleibt erhalten. Der Unterschied zum persistenten Zustand ist, dass beide nun nicht mehr direkt miteinander verknüpft sind. Nun befindet sich das Objekt im so genannten Detached State (engl. losgelöster Zustand) und auch wenn der Garbage Collector das Objekt aus dem Hauptspeicher entfernt, können Sie es natürlich jederzeit erneut laden.

Änderungen an der JavaBean im Hauptspeicher können Sie nur in die Datenbank übernehmen, wenn sich ein Objekt im persistenten Zustand befindet und über den aktuellen `EntityManager` mit seiner Kopie verbunden ist. Das heißt, dass Sie ein losgelöstes Objekt zunächst erst mit dem `EntityManager` synchronisieren müssen. Hierzu dient die Methode `merge()`, die Ihnen schon im nächsten Beispiel wiederbegegnen wird.

10.4.4 Manipulieren einer JavaBean

Mit Ihrem jetzigen Wissen und der Methode `contains()` können Sie Objekte speichern, ohne zu wissen, ob diese bereits eine Kopie in der Datenbank besitzen.

Listing 10.12 Speichern eines (bestehenden) Datensatzes

```
...
   /**
    * Ändert Vor- und Nachnamen des übergebenen Datensatzes
    */
   private static void changeValues(EntityManagerFactory emf,
                                    Address addressToChange) {
      // Erzeugen eines EntityManagers
      EntityManager manager = emf.createEntityManager();

      // Begin einer neuen Transaktion
      manager.getTransaction().begin();

      addressToChange.setLastName("Altmann");
      addressToChange.setFirstName("Ulli");
```

Listing 10.12 (Forts.) Speichern eines (bestehenden) Datensatzes

```
        // Da Sie an dieser Stelle nicht wissen, ob es sich bei der
        // übergebenen Adresse um ein neue oder ein losgelöstes Objekt
        // handelt verwenden Sie die Methode merge() statt persist()
        manager.merge(addressToChange);

        // Abschluss der Transaktion
        manager.getTransaction().commit();

        // Freigabe der Ressourcen
        manager.close();
    }
...
```

Die Methode merge()

Die Methode `merge()` dient dazu, Objekte, die sich entweder im transienten oder losgelösten Zustand befinden, in den persistenten Zustand zu überführen. Dabei spielt es keine Rolle, ob das Objekt bereits eine Kopie in der Datenbank besitzt (detached) oder noch nie gespeichert wurde, denn während Erstere mit der Datenbank synchronisiert werden, legt die Methode im zweiten Fall einfach einen neuen Datensatz an.

Info

Für neue Objekte benötigt die Methode `merge()` etwas mehr Ressourcen als ihr Pendant `persist()`. Wenn Sie also sicher sein können, dass es sich um ein transientes Objekt handelt, ist es günstiger, Letztere zu verwenden.

10.4.5 Löschen eines vorhandenen Datensatzes

Die letzte Basisoperation, die Ihnen nun noch fehlt, ist das Löschen eines Datensatzes aus der Datenbank. Dabei geht ein persistentes Objekt wieder in den transienten Zustand über, was bedeutet, dass es zwar noch im Hauptspeicher existiert, aber kein Pendant in der Tabelle mehr besitzt.

Listing 10.13 Löschen eines persistenten Datensatzes

```
...
    /**
     * Löscht den übergebenen persistenten Datensatz
     */
    private static void removeEntity(EntityManagerFactory emf,
                                     Address addressToRemove) {
        // Erzeugen eines EntityManagers
        EntityManager manager = emf.createEntityManager();

        // Begin einer neuen Transaktion
        manager.getTransaction().begin();

        // Löschen eines Datensatzes
        manager.remove(addressToRemove);

        // Abschluss der Transaktion
        manager.getTransaction().commit();

        // Freigabe der Ressourcen
        manager.close();
    }
...
```

10.5 Das Query API

Der letzte Absatz hat Sie mit den Basisoperationen persistenter Objekte vertraut gemacht:

- Erzeugen eines Datensatzes (Create)
- Suche eines Datensatzes anhand seines Primärschlüssels (Retrieve)
- Manipulieren eines Datensatzes (Update)
- Löschen eines Datensatzes (Delete)

Dieser Abschnitt zeigt Ihnen nun, wie Sie über das leistungsfähige Query API des Persistenz-Framework auch komplexe Anfragen formulieren können.

10.5.1 Suche über den Nachnamen

Über Query-Objekte, die Sie natürlich ebenfalls über den `EntityManager` erzeugen, haben Sie die Möglichkeit, Anfragen an die Datenbank zu formulieren. Die verwendete Query Language orientiert sich dabei stark an SQL und ist um objektorientierte Eigenschaften wie die Navigation über JavaBean-Eigenschaften ergänzt worden.

Listing 10.14 zeigt Ihnen, wie Sie eine Anfrage formulieren, die Ihnen alle Datensätze mit einem bestimmten Nachnamen zurückgibt.

Listing 10.14
Suche nach Adressen anhand des Nachnamens

```
...
   /**
    * Sucht Datensätze anhand des Nachnamens
    */
   private static void findByLastName(EntityManagerFactory emf,
                                      String nameToFind) {
      // Erzeugen eines EntityManagers
      EntityManager manager = emf.createEntityManager();

      // Definition einer Query mit einem Parameter
      String queryString =
         "select adr from Address adr where lastName = :param";
      Query query = manager.createQuery(queryString);

      // Setzen der Parameter
      query.setParameter("param", nameToFind);

      // Iterieren über die Ergebnismenge
      List<Address> addresses = query.getResultList();
      for(Address address : addresses) {
         System.out.println(address.toString());
      }

      // Freigabe der Ressourcen
      manager.close();
   }
...
```

Definition der Anfrage

Als Erstes benötigen Sie einen String, der die tatsächliche Anfrage enthält. Diese besteht in der Regel aus einer Select-From- und eine Where-Klausel, kann aber auch beliebig komplex werden (siehe 10.5.3). In der Select-

From-Klausel teilen Sie dem `EntityManager` zunächst mit, welche Art von Objekten Sie suchen. In der optionalen Where-Klausel folgen anschließend die Eigenschaften, die die gesuchten Datensätze beschreiben.

Listing 10.15
Eine einfache Anfrage in der Query Language

```
...
   // Definition einer Query mit einem Parameter
   String queryString =
         "select adr from Address adr where lastName = Stark";
...
```

Um flexible Anfragen zu definieren, können Sie bei der Definition der Anfrage auch Platzhalter, so genannte Parameter, mit symbolischen Namen verwenden. Hierfür markieren Sie diese einfach mit dem Präfix `:`.

Listing 10.16
Anfrage mit einem Parameter

```
...
   // Definition einer Query mit einem Parameter
   String queryString =
         "select adr from Address adr where lastName = :param";
...
```

Den Parametern `param` können Sie später zur Laufzeit durch den tatsächlichen Wert ersetzen, um so dynamische Anfragen zu erzeugen.

Erzeugen einer Query und Setzen der Parameter

Mit dem Anfrage-String können Sie anschließend über den `EntityManager` ein `Query`-Objekt erzeugen und die zuvor definierten Platzhalter mit konkreten Werten befüllen. Hierfür verwenden Sie wieder deren symbolische Namen, nun allerdings ohne das Präfix `:`.

Listing 10.17
Erzeugen einer Anfrage und Setzen der Parameter

```
...
   Query query = manager.createQuery(queryString);
   query.setParameter("param", nameToFind);
...
```

Auslesen der Ergebnismenge

Anschließend brauchen Sie nichts weiter zu tun, als sich die Liste der Datensätze zurückgeben zu lassen, die den in der Where-Klausel beschriebenen Eigenschaften entspricht.

Listing 10.18
Auslesen der Ergebnismenge einer Query

```
...
   List<Address> addresses = query.getResultList();
   for(Address address : addresses) {
      // An dieser Stelle können Sie die einzelnen Datensätze
      // verarbeiten.
   }
...
```

Über die Methode `setFirstResult()` und `setMaxResult()` einer `Query` können Sie deren Ergebnismenge auch seitenweise durchlaufen, was gerade bei umfangreichen Tabellen einen großen Performance-Vorteil bringen kann.

Wenn Sie sich sicher sind, dass die Liste in jedem Fall nur ein Objekt enthalten kann, können Sie dieses über die Methode `getSingleResult()` auch direkt auslesen.

```
...
   // Rückgabe des ersten (und einzigen) Datensatzes der Query-
   // Ergebnismenge
   Address singleResult = (Address) query.getSingleResult();
...
```

Listing 10.19
Auslesen eines einfachen Ergebnisses

10.5.2 Benannte Anfragen hinterlegen

Neben den dynamisch erzeugten `Query`-Objekten können Sie auch schon bei der Definition einer persistenten Klasse Anfragen hinterlegen, die von anderen Entwicklern anschließend über einen symbolischen Namen verwendet werden können. Diese Technik ist dabei vor allem für Anfragen geeignet, die voraussichtlich von verschiedenen Clients benötigt werden.

Hinterlegen der Anfragen unter einem symbolischen Namen

Um eine vordefinierte Anfrage zu hinterlegen, verwenden Sie natürlich wiederum eine Annotation.

```
package de.akdabas.javaee.persistence.bean;
import javax.persistence.*;

@Entity()
@Table(name = "ADDRESSES")
@NamedQuery(name= "findAddressByLastName",
   query = "select adr from Address adr where lastName = :lastName")
public class Address {
...
   // Hier folgt die gleiche Klassendefinition wie in Listing 10.2
...
}
```

Listing 10.20
Hinterlegen einer vordefinierten Anfrage

Der Parameter `name` der Annotation `@NamedQuery` definiert einen symbolischen Namen, unter dem die Anfrage später referenziert werden kann. An der `query` hingegen ändert sich gegenüber Listing 10.14 – zwecks besserer Dokumentation – nur der Name des Parameters.

Verwenden einer benannten Anfrage

Nachdem die Anfrage erst einmal hinterlegt wurde, ist deren Verwendung für einen Client denkbar einfach. Er muss hierfür lediglich die symbolischen Namen der Query und ihrer Parameter kennen.

```
...
   // Referenzieren einer zuvor hinterlegten Anfrage
   Query query = manager.createNamedQuery("findAddressByLastName");

   // Setzen der Parameter der Query
   query.setParameter("lastName", "Stark");

   // Ausführen der Anfrage
   List<Address> addresses = query.getResultList();
...
```

Listing 10.21
Verwendung einer benannten Anfrage

Die großen Vorteile, die Sie mit dieser Technik gerade für größere Anwendungen bekommen, sind:

- Der Client muss die Query nicht selbst definieren und braucht kein Wissen über deren Aufbau.
- Alle Clients verwenden die gleiche Anfrage.
- Die Anfrage kann auch im Nachhinein an zentraler Stelle weiterentwickelt und optimiert werden, ohne dass die Clients angepasst werden müssen.

10.5.3 Anspruchsvolle Anfragen

Wenn Sie mit der Datenbankanfragesprache SQL sehr vertraut sind und sich an dieser Stelle fragen, wie weit Sie die Query Language des Persistenz API bei komplexen Anfragen unterstützt, soll Ihnen dieser Absatz einen kurzen Überblick über die Möglichkeiten geben.

Achtung

Leider können wir an dieser Stelle keine umfangreiche Einführung in die verschiedenen Anfragekonstrukte von SQL und deren Auswirkungen auf die Ergebnismenge liefern, weil diese den Umfang des Buchs einfach sprengen würde. Wenn Sie mit SQL weniger vertraut sind, überspringen Sie diesen Absatz einfach. Die hier vermittelten Möglichkeiten sind dafür gedacht, Datenbankanfragen zu optimieren. Um das API zu verwenden, ist dieses Wissen nicht zwingend erforderlich.

Um den Absatz kurz zu halten, gehen die folgenden Listings davon aus, dass Sie über eine JavaBean `Order` verfügen, die eine Liste von ebenfalls persistierbaren `Item`-Objekten hat. Ein `item` besitzt in den folgenden Beispielen außerdem ein Attribut `amout`, welches den Preis definiert.

In dieser Konstellation von Objekten unterstützt Sie die Query Language beispielsweise mit folgenden Erweiterungen.

Unterstützung von Joins

Listing 10.22 Beispiel für einen Join

```
"select o from Order o left join o.items i where i.amount > 100"
```

Unterstützung von Subselects

Listing 10.23 Subselects mit der Query Language

```
"select o from Order o where
     exists(select i from o.items i where i.amount > 100)"
```

Unterstützung von Aggregationen

Listing 10.24 Aggregationen

```
"select o.id, sum(i.amount)
     from Order o join o.items i group by o.id"
```

Update und Delete-Operationen

```
"delete from Customer cust where cust.id = 12345"
"update Order o set o.fulfilled = 'Y' where o.id = 9876543"
```

Listing 10.25
Update und Delete über eine Query

... und und und

Außerdem versteht sich die Query Language auf eine Reihe von Funktionen, die (fast) keine Wünsche mehr offenlassen.

```
trim(), locate(), concat(), substring(), lower(), upper(), length,
abs(), sqrt(), mod(), size()
```

Listing 10.26
Funktionen, die Sie in Querys ebenfalls verwenden können

10.6 Datensätze verknüpfen

Das neue Persistenz API ermöglicht es Ihnen jedoch nicht nur, einzelne JavaBeans mit der Datenbank zu synchronisieren, sondern auch, diese miteinander zu verknüpfen. Um dies zu demonstrieren, werden Sie die `Address`-Bean nun mit der Liste von E-Mail-Beans verknüpfen, die dabei in einer separaten Tabelle gespeichert werden.

10.6.1 Die E-Mail-Bean

Zuallererst müssen Sie die neue Bean natürlich implementieren und mit entsprechenden Annotationen versehen. Listing 10.27 beschränkt sich auch hier auf den Kern.

Listing 10.27
Eine einfache E-Mail-Bean

```
package de.akdabas.javaee.persistence.bean;
import javax.persistence.*;

@Entity
@Table(name = "EMAIL_ADDRESSES")
public class EmailAddress {

   /** ID dieser eMail-Adresse */
   private int eMailID;

   /** Die tatsächlich abgelegte Adresse */
   private String email;

   /**
    * Standard-Konstruktor; Dieser wird vom Persistenz-Framework
    * zur Verwaltung der Klasse mit dem Reflection-API benötigt;
    */
   protected EmailAddress() {}

   /**
    * Minimaler Konstruktor; Dieser kann von Clients zur Erzeugung
    * neuer Objekte verwendet werden und enthält alle Pflichtfelder
    */
   public EmailAddress (String email){
      this.email = email;
   }

   @Id
   @GeneratedValue(strategy=GenerationType.AUTO)
   @Column(name="emailID")
   public int getEmailID() {
```

Listing 10.27 (Forts.)
Eine einfache E-Mail-Bean

```
        return eMailID;
    }
    protected void setEmailID(int id) {
        this.eMailID = id;
    }

    @Column(name="value", nullable=false)
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
}
```

Diese Bean gleicht in ihrem Aufbau sehr ihrem Pendant aus Listing 10.2 und muss wie diese natürlich in den Persistence Descriptor (*persistence.xml*) eingebunden werden.

Listing 10.28
Konfiguration der E-Mail-Bean

```
<!-- Persistence Descriptor zur Konfiguration des Frameworks -->
<persistence>

  <!-- Symbolischer Name der Persistenz-Unit und der zu verwendende
       Transaktionstyp; Standard ist 'JTA'  -->
  <persistence-unit name="masterclass"
                    transaction-type="RESOURCE_LOCAL">

    <!-- Zu verwendende Service Provider Implementierung -->
    <provider>org.hibernate.ejb.HibernatePersistence</provider>

    <!-- Persistierbare Klassen dieser Persistenz-Unit -->
    <class>de.akdabas.javaee.persistence.bean.Address</class>
    <class>de.akdabas.javaee.persistence.bean.SimpleAddress</class>
    <class>de.akdabas.javaee.persistence.bean.EmailAddress</class>

    <!-- Konfiguration der Service Provider Implementierung -->
    <properties>

      <!-- Hier folgt die Konfiguration des Hibernate Service
           Providers, wie sie in Listing 10.5 zu sehen ist. -->

    </properties>
  </persistence-unit>
</persistence>
```

Nachdem Sie die Anwendung nun übersetzt haben, ist die neue JavaBean bereits vollkommen einsatzfähig. Sie können an dieser Stelle bereits neue E-Mail-Adressen erzeugen, verändern, suchen ...

10.6.2 Relationen zwischen Datensätzen

Bevor Sie nun jedoch die `EmailAddress`-Bean mit der Adresse verknüpfen, führt Sie dieser Abschnitt in die wichtigsten Aspekte der Relationen zwischen Datensätzen ein.

Relationen zwischen Datensätzen entstehen, wenn ein Datensatz von der Existenz eines oder mehrerer Datensätze »weiß« und mit diesen verknüpft ist. Diese Information über die Existenz eines zugeordneten Datensatzes kann entweder einseitig (*unidirektional*) oder beidseitig (*bidirektional*) vorliegen. Außerdem unterscheidet man die Relationen nach der *Kardinalität*, die die Komplexität der Relation beschreibt.

Unidirektionale vs. bidirektionale Relationen

Bei unidirektionalen Beziehungen kennt nur ein Partner der Relation die ihm zugeordneten Datensätze vom Typ des anderen Partners. Wenn Sie ein solches Objekt instanziiert haben, können Sie, z.B. über entsprechende Getter, direkt auf die von ihm referenzierten Datensätze zugreifen. Der umgekehrte Weg ist bei unidirektionalen Beziehungen nicht möglich, da die referenzierten Objekte nichts von der Relation wissen.

Info

Die miteinander verknüpften Objekte müssen nicht unterschiedlichen Typs sein. Es ist auch möglich, »auf sich selbst verweisende« Relationen zu deklarieren.

Um zwischen zwei miteinander verknüpften Objekten in beliebige Richtung navigieren zu können, muss eine *bidirektionale Relation* zwischen den beiden Datensätzen existieren. Hierbei besitzen beide Partner einen Getter, der das jeweils andere Objekt zurückgibt.

Im nachfolgenden Beispiel werden Sie eine bidirektionale Beziehung zwischen `Address`- und `EmailAddress`-Objekten erzeugen. Das heißt, dass jede `Address`-Bean über eine Liste aller ihr zugeordneten E-Mail-Adressen verfügt und gleichzeitig jede `EmailAddress`-Bean einen Getter besitzt, der die mit ihr verknüpfte Adresse zurückgibt.

Eins-zu-eins-Kardinalität (1:1)

Von *Eins-zu-eins-Beziehungen* (1:1) oder One-to-One-Relations spricht man, wenn *jedem Datensatz* des einen Typs *genau ein Datensatz* des anderen Typs (und umgekehrt) zugeordnet ist. Es gibt also genau eine Kombinationsmöglichkeit.

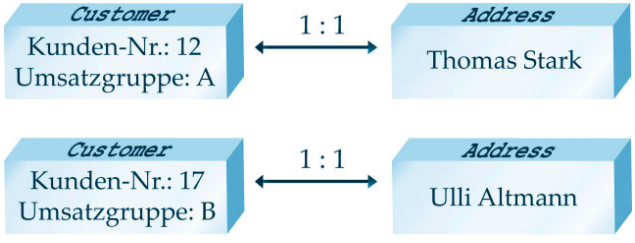


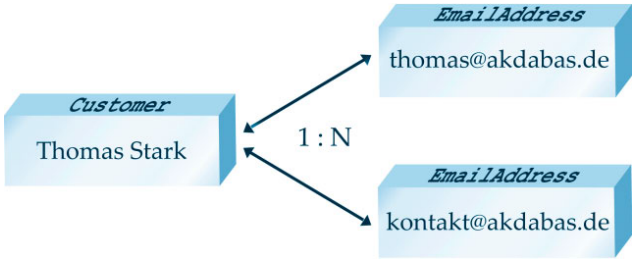
Abbildung 10.3
Eins-zu-eins-Beziehung (1:1) zwischen Kunden und Adressen

Ein typisches Beispiel könnte die Verbindung zwischen einem Kunden und seiner Adresse sein. Abbildung 10.3 zeigt eine solche 1:1-Beziehung, bei der jeweils eine `Address`-Bean mit einer hypothetischen Kunden-Bean (`Customer`) verknüpft ist.

Eins-zu-n-Kardinalität (1:n)

Um die Leistungsfähigkeit der Anwendung zu erhöhen und mehrere E-Mail-Kontakte zu einem Adressdatensatz zuzulassen, werden Sie im nachfolgenden Beispiel eine Eins-zu-n-Beziehung (1:n) bzw. Many-To-One-Relation implementieren. Hierbei können einem Objekt des einen Typs mehrere Objekte des anderen Typs zugeordnet werden.

Abbildung 10.4
Eins-zu-n-Beziehung (1:n) zwischen Adressen und E-Mail-Kontakten



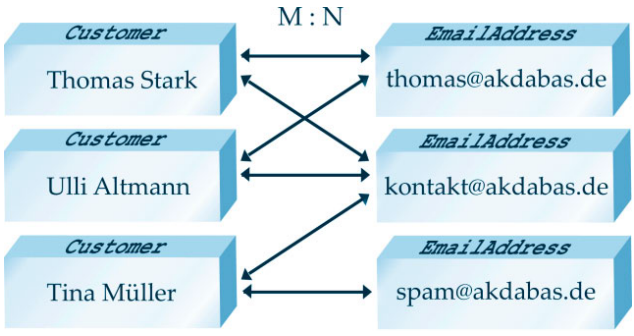
Zwar können bei dieser 1:n-Beziehung einer Adresse mehrere E-Mail-Adressen zugeordnet sein, umgekehrt muss die Relation jedoch eindeutig sein. Das heißt, dass eine `EmailAddress`-Bean zu genau einer spezifischen `Address`-Bean gehört. Soll es hingegen auch andersherum möglich sein, eine E-Mail-Adresse unterschiedlichen Adressen zuzuordnen, spricht man von einer m-zu-n-Beziehung.

m-zu-n-Kardinalität (m:n)

Von einer m-zu-n-Beziehung (m:n) oder auch Many-to-Many-Relation spricht man, wenn beide an der Relation beteiligten Objekte noch anderen Objekten dieses Typs zugeordnet sein können bzw. mehrere Partnerobjekte referenzieren.

In unserem hypothetischen Beispiel wäre das der Fall, wenn eine E-Mail-Adresse auch mehreren `Address`-Objekten zugeordnet sein könnte.

Abbildung 10.5
m-zu-n-Beziehung (m:n) zwischen Adressen und E-Mail-Kontakten



10.6.3 Verknüpfen zweier JavaBeans über Annotationen

Auch die Verknüpfung zweier bestehender Persistenz-Objekte lässt sich natürlich über Annotationen realisieren. Die in diesem Beispiel gezeigte 1:n-Relation ist dabei natürlich nur exemplarisch und es werden vom Framework auch alle übrigen Relationen 1:1 und m:n unterstützt.

Erweiterung der E-Mail-Bean um eine Adresse

Als Erstes werden Sie nun Ihre `EmailAddress`-Bean mit einer Adresse verknüpfen, zu der diese gehören soll. Hierfür erweitern Sie diese einfach um eine entsprechende Variable einschließlich Getter und Setter und verwenden eine neue Annotation.

Listing 10.29
Verknüpfen der E-Mail mit einer Adresse

```
package de.akdabas.javaee.persistence.bean;
import javax.persistence.*;

@Entity
@Table(name = "EMAIL_ADDRESSES")
public class EmailAddress {

   /** ID dieser eMail-Addresse */
   private int eMailID;

   /** Die tatsächlich abgelegte Addresse */
   private String email;

   /** Address-Bean, mit der dieser Datensatz verknüpft ist */
   private Address address;

   /**
    * Standard-Konstruktor; Dieser wird vom Persistenz-Framework
    * zur Verwaltung der Klasse mit dem Reflection-API benötigt;
    */
   protected EmailAddress() {}

   /**
    * Minimaler Konstruktor; Dieser kann von Clients zur Erzeugung
    * neuer Objekte verwendet werden und enthält alle Pflichtfelder
    */
   public EmailAddress (String email){
      this.email = email;
   }

   @Id
   @GeneratedValue(strategy=GenerationType.AUTO)
   @Column(name="emailID")
   public int getEmailID() {
      return eMailID;
   }
   protected void setEmailID(int id) {
      this.eMailID = id;
   }

   @Column(name="value", nullable=false)
   public String getEmail() {
      return email;
   }
   public void setEmail(String email) {
      this.email = email;
   }

   @ManyToOne(cascade = CascadeType.ALL)
   public Address getAddress() {
      return address;
   }
   public void setAddress(Address address) {
      this.address = address;
   }
}
```

Über die `@ManyToOne`-Annotation teilen Sie dem Framework mit, dass mehrere JavaBeans dieses Typs mit einer `Address`-Bean verknüpft sein können. Die zugehörige Klasse wird hierbei aus dem Rückgabewert der Methode `getAddress()` spezifiziert, kann aber auch als (optionales) Attribut der Annotation hinterlegt werden.

Kaskadierende Datenbankoperationen

Der optionale Parameter `cascade` der Many-To-One-Relation gibt dabei an, was mit verknüpften Objekten passieren soll, wenn dieses Objekt mit einer Datenbankoperation manipuliert wird. Der Aufruf der Datenbankoperation wird dabei gewissermaßen kaskadierend an das verknüpfte Objekt weitergereicht.

Tabelle 10.2 Die verschiedenen Cascade-Typen

Cascade-Type	Beschreibung
`MERGE`	Wenn dieses Objekt der Methode `merge()` übergeben wird, wird diese Methode implizit auch für die verknüpften Objekte aufgerufen.
`PERSIST`	Wenn dieses Objekt der Methode `persist()` übergeben wird, wird diese Methode implizit auch für die verknüpften Objekte aufgerufen.
`REFRESH`	Wenn dieses Objekt der Methode `refresh()` übergeben wird, wird diese Methode implizit auch für die verknüpften Objekte aufgerufen.
`REMOVE`	Wenn dieses Objekt der Methode `remove()` übergeben wird, wird diese Methode implizit auch für die verknüpften Objekte aufgerufen.
`ALL`	Alle Datenbankoperationen werden automatisch auch auf den angehängten Objekten ausgeführt. Dieses Einstellung ist gleichbedeutend mit `{MERGE, PERSIST, REFRESH, REMOVE}`.

Ist eine Relation mit einer oder mehreren Cascade-Optionen ausgestattet, wird die zugehörige Datenbankoperation automatisch für die damit verknüpften Elemente übernommen. So erreichen Sie über folgende Konfiguration beispielsweise, dass alle `EntityManager`-Aufrufe von `persist()` und `merge()` auf der `EmailAddress`-Bean auch auf der `Address`-Bean aufgerufen werden. Für die Operationen `remove()` und `refresh()` bleibt die verknüpfte Adresse hingegen unangetastet.

Listing 10.30 Weitere Cascade-Optionen für die E-Mail-Adresse (Beispiel)

```
...
  /**
   * Mit dieser Konfiguation werden die Datenbankoperationen
   * 'persist()' und 'merge()' an die verknüpften Address-
   * Objekte weitergereicht.
   */
  @ManyToOne(cascade = {CascadeType.MERGE, CascadeType.PERSIST})
```

```
   public Address getAddress() {
      return address;
   }
   public void setAddress(Address address) {
      this.address = address;
   }
...
```

Listing 10.30 (Forts.)
Weitere Cascade-Optionen für die E-Mail-Adresse (Beispiel)

Der Einfachheit halber werden wir in diesem Beispiel über die Option `CascadeType.ALL` alle Datenbankoperationen an die referenzierten Objekte »durchreichen« (Listing 10.29).

Erweitern der Address-Bean um eine Liste von E-Mail-Adressen

Auch die Erweiterung der `Address`-Bean in nicht aufwändig, nur dass Sie hier zusätzlich das Attribut der `EmailAddress`-Bean angeben müssen, über das diese mit der Adresse verknüpft werden soll. Da der Getter der E-Mail-Adresse in Listing 10.29 `getAddress()` lautet, geben wir als Wert für das Property `mappedBy` einfach `address` an.

Listing 10.31
Verknüpfen der Address-Bean mit einer Liste von E-Mails

```
package de.akdabas.javaee.persistence.bean;
import javax.persistence.*;
import java.util.List;
import java.util.LinkedList;

@Entity
@Table(name = "ADDRESSES")
@NamedQuery(name= "findAddressByLastName",
   query = "select adr from Address adr where lastName = :lastName")
public class Address {

   private int id;
   private String lastName;
   private String firstName;

   /** Liste mit verknüpften eMail-Addressen dieses Kontaktes */
   private List<EmailAddress> emailAddresses;

   /**
    * Standard-Konstruktor; Dieser wird vom Persistenz-Framework
    * zur Verwaltung der Klasse mit dem Reflection-API benötigt;
    */
   protected Address() {}

   /**
    * Minimaler Konstruktor; Dieser kann von Clients zur Erzeugung
    * neuer Objekte verwendet werden und enthält alle Pflichtfelder
    */
   public Address(String lastName){
      this.lastName = lastName;
      emailAddresses = new LinkedList<EmailAddress>();
   }

   /**
    * Vollständiger Kontruktor; Dieser enthält alle Attribute dieses
    * Objektes und erleichtert die Erzeugung von neuen Objekten
    */
   public Address(String lastName, String firstName){
      this.lastName = lastName;
      this.firstName = firstName;
      emailAddresses = new LinkedList<EmailAddress>();
   }
```

Listing 10.31 (Forts.)
Verknüpfen der Address-Bean mit einer Liste von E-Mails

```
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="id")
    public int getId() {
       return id;
    }
    protected void setId(int id) {
       this.id = id;
    }

    @Column(name="last_name", nullable=false, length=120)
    public String getLastName() {
       return lastName;
    }
    public void setLastName(String lastName) {
       this.lastName = lastName;
    }

    @Column(name="first_name", length=120)
    public String getFirstName() {
       return firstName;
    }
    public void setFirstName(String firstName) {
       this.firstName = firstName;
    }

    /**
     * Diese Addresse kann beliebig viele eMail-Addressen enthalten
     */
    @OneToMany(mappedBy="address", cascade = CascadeType.ALL)
    public List<EmailAddress> getEmailAddresses() {
       return emailAddresses;
    }
    public void setEmailAddresses(List<EmailAddress> emailAddresses){
       this.emailAddresses = emailAddresses;
    }

    /** Nur für Debug-Zwecke und für die nachfolgenden Beispiele */
    public String toString(){
       return "Address [" + id +"]: " + firstName + " " + lastName);
    }
}
```

Fertig! Nun können Sie jedes `Address`-Objekt mit einer Liste von zugehörigen `EmailAddress`-Beans ausstatten und sich gleichzeitig sicher sein, dass diese beim Laden der Adresse auch wiederhergestellt werden kann.

Erstellen eines Adressdatensatzes samt E-Mail-Adressen

Um eine Adresse nun mit einer oder mehreren E-Mail-Adressen auszustatten, müssen Sie nichts weiter tun, als beide Objekte zu erstellen, miteinander zu verknüpfen und anschließend über die Methode `persist()` mit der Datenbank zu synchronisieren.

Listing 10.32
Erstellen eines Adressdatensatzes mit zwei E-Mail-Adressen

```
...
   // Erzeugen der JavaBeans, ...
   Address address = new Address("Stark", "Thomas");
   EmailAddress email1 = new EmailAddress("thomas@akdabas.de");
   EmailAddress email2 = new EmailAddress("kontakt@akdabas.de");

   // ... Verknüpfen ...
   address.getEmailAddresses().add(email1);
   address.getEmailAddresses().add(email2);
```

```
   email1.setAddress(address);
   email2.setAddress(address);

   // ... und Speichern.
   manager.getTransaction().begin();
   manager.persist(address);
   manager.getTransaction().commit();
...
```

Listing 10.32 (Forts.)
Erstellen eines Adressdatensatzes mit zwei E-Mail-Adressen

Dank des zuvor eingestellten Cascade-Verhaltens muss der Methode `persist()` lediglich das `Address`-Objekt übergeben werden. Die beiden »angehängten« E-Mail-Datensätze werden automatisch mit erzeugt (siehe auch Tabelle 10.2).

10.6.4 Unterschiedliche Fetch-Typen

Wenn Sie sich jetzt fragen, wann die mit einer JavaBean verknüpften Datensätze eigentlich geladen werden, enthält dieser Absatz die Antwort. Es gibt prinzipiell zwei gewünschte Verhaltensweisen:

- Alle verknüpften Objekte werden gemeinsam mit dem Ursprungsobjekt geladen. Dieses Verhalten wird auch als Eager-Loading (engl., »Eifriges Laden«) bezeichnet.
- Die verknüpften Objekte werden erst bei Bedarf nachgeladen. Dieses Verhalten bezeichnet man als Lazy-Loading (engl., »Träges Laden«).

Während die erste Methode natürlich die komfortablere ist, benötigt sie auch mehr Ressourcen. Gerade wenn eine JavaBean mit vielen anderen Objekten verknüpft ist, kann das Laden dieser Bean eine kleine Ewigkeit dauern, weil alle anderen damit verknüpften Objekte ebenfalls in den Arbeitsspeicher geladen werden müssen. Dafür ist das Objekt anschließend vollständig geladen und kann auch ohne geöffneten `EntityManager` als Detached Object verwendet werden

Bei der zweiten Methode wird zunächst nur das ursprüngliche Objekt in den Speicher geladen. Erst wenn Sie anschließend über einen Getter auf eines der verknüpften Objekte zugreifen, werden auch diese dynamisch nachgeladen. Diese Methode ist zwar prinzipiell die ressourcen-schonendere, dafür funktioniert das Nachladen allerdings nur, wenn sich das Ursprungsobjekt im persistenten Zustand befindet.

Welche von beiden Varianten für die jeweilige Verknüpfung zum Einsatz kommt, können Sie über das Attribut `fetch` einer Relationsannotation steuern.

```
...
   /**
    * Diese Konfiguration lädt die verknüpften E-Mail-Adressen
    * sobald die Address-Bean geladen wird; Eager-Loading
    */
   @OneToMany(mappedBy="address",
              cascade = CascadeType.ALL,
              fetch = FetchType.EAGER)
   public List<EmailAddress> getEmailAddresses() {
      return emailAddresses;
```

Listing 10.33
Address-Bean mit Eager-Loading (Auszug)

Listing 10.33 (Forts.)
Address-Bean mit Eager-Loading (Auszug)

```
    }
    public void setEmailAddresses(List<EmailAddress> emailAddresses){
        this.emailAddresses = emailAddresses;
    }
...
```

Während mit der Konfiguration in Listing 10.33 die E-Mail-Adressen also zusammen mit der `Address`-Bean instanziiert werden (Eager-Loading), lädt der `EntityManager` die Liste mit `EmailAddress`-Beans mit der Konfiguration aus Listing 10.34 erst dann nach, wenn auf diese zugegriffen wird (Lazy-Loading). Als Standardverhalten definiert die Spezifikation Eager-Loading.

Listing 10.34
Address-Bean mit Lazy-Loading (Auszug)

```
...
    /**
     * Diese Konfiguration lädt die verknüpften E-Mail-Adressen
     * zusammen mit der Address-Bean; Lazy-Loading
     */
    @OneToMany(mappedBy="address",
               cascade = CascadeType.ALL,
               fetch = FetchType.LAZY)
    public List<EmailAddress> getEmailAddresses() {
        return emailAddresses;
    }
    public void setEmailAddresses(List<EmailAddress> emailAddresses){
        this.emailAddresses = emailAddresses;
    }
...
```

Wenn Ihre Anwendung meistens mit Lazy-Loading zurechtkommt und manchmal zwingend auf das Eager-Loading angewiesen ist, definieren Sie die Relation als lazy und greifen Sie im zweiten Fall, bevor Sie den `EntityManager` schließen, einmal auf das benötigte Attribut zu. Sobald die verknüpften Objekte einmal in den Arbeitsspeicher geladen sind, können Sie nicht mehr unterscheiden, ob diese lazy oder eager geladen worden sind.

10.7 Persistenz im EJB-Container

Die vorangegangenen Abschnitte machten Sie mit der Funktionsweise des neuen Persistenz API in einem Standard-Java-Umfeld, außerhalb eines Application Servers, vertraut. Dieser Absatz zeigt Ihnen nun, wie Sie das Persistenz API auch als Nachfolger der auf Entity-Beans basierenden Persistenzschicht innerhalb eines Application Servers einsetzen können.

10.7.1 Grundlagen

Wie Sie bereits in Kapitel 8 über EJBs gesehen haben, wird die Geschäftslogik innerhalb eines Application Servers in Form von Session-Beans gekapselt und über Service-Methoden nach außen verfügbar gemacht. Für Session-Beans, die ihrerseits auf Datenbankobjekte zugreifen müssen, sah die EJB-2.x-Spezifikation eine darunter liegende Schicht aus so genannten Entity Beans vor.

Diese EJB Entity Beans repräsentieren einen Datensatz in der Datenbanktabelle und werden vom EJB-Container mit diesem synchronisiert. Hierfür besitzen EJB Entity Beans einen relativen, umfangreichen Lebenszyklus und müssen aufwändig implementiert sowie konfiguriert werden. Außerdem sind EJB Entity Beans fest an die Laufzeitumgebung des EJB-Containers gebunden und können nicht ohne weiteres an den Client gesendet oder von diesem verwendet werden, weil die EJB-Spezifikation 2.x keinen losgelösten Zustand kennt.

Tipp

Um die Abwärtskompatibilität zu gewährleisten, ist es natürlich weiterhin möglich, Entity Beans über die verschiedenen Deployment Descriptors zu verwalten. Sie werden jedoch sehen, dass es mit dem neuen Persistenz API auch viel leichter geht.

Mit dem neuen Persistenz API ist Sun jedoch ein großer Wurf und die Vereinigung der verschiedenen Anforderungen an eine flexible Persistenzschicht gelungen. Dieses API ersetzt die »alten« EJB Entity Beans vollständig und ermöglicht es Ihnen, das zuvor in diesem Kapitel Gelernte nahtlos auf einen Application Server zu übertragen, mit dem Unterschied, dass dieser Ihnen nun einige lästigen Verwaltungsaufgaben abnehmen möchte.

Folgende Dinge erledigt der Application Server nun für Sie:

1. Bereitstellung der Service Provider-Implementierung
2. Verwalten der Datenquellen und Transaktionen
3. Bereitstellung des `EntityManager` über einen `PersistenceContext`

Sie sehen also, das einzige, worum Sie sich selbst noch kümmern müssen, sind die Basis-Datenbankoperationen.

10.7.2 Eine einfache JavaBean

Um die Arbeitsweise des Persistenz API innerhalb eines EJB-Containers zu demonstrieren, werden Sie in diesem Abschnitt eine `UserBean` implementieren und mit dieser die Stateful Session Bean aus Kapitel 8 erweitern, um die registrierten Benutzer zukünftig in einer Datenbank zu speichern.

Listing 10.35
Eine einfache UserBean

```
package de.akdabas.javaee.persistence;
import javax.persistence.*;

/** Diese Entity-Bean speichert Benutzerdaten einer Session-Bean */

@Entity
@Table(name="SESSION_BEAN_USER")
public class UserBean {

   private int id;
   private String name;

   /** Standard-Konstruktor */
   protected UserBean() {}
```

Listing 10.35 (Forts.)
Eine einfache UserBean

```
    /** Minimaler Konstruktor */
    public UserBean(String aName){ this.name = aName; }

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="id")
    public int getId() { return id; }
    protected void setId(int id) { this.id = id; }

    @Column(name="name", nullable=false, length=120)
    public String getName() { return name; }
    public void setName(String aName) { this.name = aName; }
}
```

Diese `UserBean` enthält in komprimierter Form verschiedene Elemente des Persistenz API, die Sie in den vorangegangenen Abschnitten kennen gelernt haben, und stellt eine gute Wiederholung dar. Wie Sie sehen, sind Entity-Beans, die innerhalb eines Application Servers zum Einsatz kommen, nicht von denen der Standard-Java-Umgebung zu unterscheiden.

10.7.3 Konfiguration der Persistenz-Unit

Da viele Teile der Konfiguration vom EJB-Container übernommen werden, reduziert sich auch der Persistence Descriptor auf ein Minimum. Sie müssen lediglich einen symbolischen Namen für die Persistenz-Unit und die zu verwendende JTA-Datenquelle angeben, welche ebenfalls vom Application Server verwaltet wird.

Listing 10.36
Persistence Descriptor im EJB-Umfeld

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence>
  <persistence-unit name="masterclassEjbPersistence">
    <jta-data-source>java:/DefaultDS</jta-data-source>
    <properties>
      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
    </properties>
  </persistence-unit>
</persistence>
```

In obigem Beispiel greifen wir dabei auf die innerhalb des JBoss standardmäßig laufende HSQLDB zu, welche unter dem Namen `DefaultDS` erreichbar ist. Und da ich Sie nicht mit tristen SQL-Statements zur Erstellung der notwendigen Tabellen langweilen möchte, verwenden wir auch hier wieder das bereits bekannte Flag `hibernate.hbm2ddl.auto`, welches Hibernate anweist, die erforderlichen Tabellen selbst zu erstellen.

10.7.4 Anpassen der Session Bean

Nun müssen Sie lediglich die Stateless Session Bean `HelloWorldBean` aus Kapitel 8 anpassen und diese auf die soeben erstellte `UserBean` umstellen. Hierbei kommt Ihnen wiederum der EJB-Container zu Hilfe, indem er Ihnen einen geöffneten `EntityManager` zur Verfügung stellt, über den Sie sofort auf die Persistenz-Unit zugreifen können. Alles, was Sie hierfür benötigen, ist eine Instanzvariable und eine neue Annotation:

```
package de.akdabas.javaee.ejb;
import javax.ejb.*;
import javax.persistence.*;
import java.io.Serializable;
import de.akdabas.javaee.persistence.UserBean;

/**
 * Diese Stateful Session Bean (EJB) speichert die Benutzerdaten
 * über eine Entity-Bean.
 */

@Stateful
@Remote(de.akdabas.javaee.ejb.HelloWorld.class)
public class HelloWorldBean implements HelloWorld, Serializable {

   @PersistenceContext(unitName = "masterclassEjbPersistence")
   private EntityManager manager;

   /** Diese Variable speichert die ID der UserBean */
   private int userId = -1;

   /** Dies Methode speichert die Benutzerdaten in der Datenbank */
   public void setName(String name) {

      // Erzeugen der JavaBean
      UserBean userBean = new UserBean(name);

      // Speichern der JavaBean
      manager.persist(userBean);

      // Speichern der Id in der lokalen Instanzvariablen
      this.userId = userBean.getId();
   }

   /**
    * Diese Methode ließt die Benutzerdaten aus der Datenbank aus
    * und gibt die Grußbotschaft zurück.
    */
   public String getGreeting() {

      // Definition der Anfrage über die Query Language
      String queryString =
         "select ub from UserBean ub where id = " + userId;

      // Ausführen der Anfrage an die Datenbank
      Query query = manager.createQuery(queryString);

      // Auslesen des ersten gefundenen Datensatzes
      UserBean userBean = (UserBean) query.getSingleResult();

      // Test, ob der Benutzer bekannt ist
      if (userBean != null) {
         return "Hello " + userBean.getName() + "!";
      } else {
         return "Hello unbekannter Benutzer!";
      }
   }
}
```

Listing 10.37
Angepasste Stateless Session Bean aus Kapitel 8

Listing 10.37 zeigt Ihnen, wie Sie über die Methode `getSingleResult()` schnell den ersten gefundenen Datensatz einer Ergebnismenge auslesen können. Diese eignet sich vor allem für Anfragen, deren Ergebnismengen null oder einen Datensatz enthalten können.

Wenn Sie sich Listing 10.37 genau ansehen, werden Ihnen zwei Dinge auffallen:

1. Die private (!) Instanzvariable `manager` wird scheinbar nie zugewiesen, kann jedoch einfach verwendet werden.
2. Beim Anlegen eines neuen Datensatzes (`persist()`) wurde scheinbar keine Transaktion verwendet.

Da es sich bei der `userId` in obigem Listing um den Primärschlüssel des zu suchenden Datensatzes handelt, hätten Sie statt der `Query` auch die Methode `find()` des `EntityManager` verwenden können. Auf diese Weise zeigt Ihnen das Beispiel allerdings noch einmal die Verwendung der Methode `getSingleResult()`.

Der EntityManager

Um das Erstellen und Schließen der `EntityManagerFactory` und der darüber erzeugten `EntityManager` brauchen Sie sich innerhalb des Application Servers keine Gedanken zu machen. Diese Standardaufgaben nimmt Ihnen der EJB-Container gerne ab.

Das einzige, was Sie tun müssen, um in einer Session Bean auf die Persistenz-Unit zugreifen zu können, ist eine Instanzvariable vom Typ `EntityManager` zu deklarieren und diese mit der Annotation `@PersistenceContext` zu kennzeichnen. Über den Parameter `unitName` geben Sie hierbei den Namen der Persitenz-Unit für diesen `EntityManager` an.

Listing 10.38 Zugriff auf den EntityManager innerhalb der Session Bean

```
...
   @PersistenceContext(unitName = "masterclassEjbPersistence")
   private EntityManager manager;
...
```

Transaktion im EJB-Container

Neben der Bereitstellung des `EntityManager` übernimmt der EJB-Container auch das Verwalten der Transaktionen, so dass Sie sich um `commit()` und `rollback()` keine Gedanken machen müssen. Die Transaktion der Persistenzschicht wird einfach an die laufende Transaktion der EJB-Schicht gekoppelt und gemeinsam mit dieser abgeschlossen oder eben zurückgesetzt.

Bereitstellung, Remote-Interface und Client

Das Erzeugen einer Session Bean umfasst in Kapitel 8 natürlich einige weitere Punkte. So wurden unter anderem ein Remote-Interface definiert und ein Client entwickelt, der über dieses Remote-Interface und die JNDI-`Lookup`-Klasse aus Kapitel 7 auf diese Session Bean zugreifen konnte.

Da die interne Speicherung der Werte innerhalb der Session Bean für den Client jedoch vollkommen transparent ist, bleiben alle anderen Klassen hiervon unberührt. Sie müssen lediglich die `UserBean` und den Persistence Descriptor (*persistence.xml*) in die *jar*-Datei für Ihre EJBs aufnehmen und diese wie in Kapitel 8 beschrieben bereitstellen.

10.7.5 Abschließende Bemerkungen

Die in diesem Abschnitt eingeführte Speicherung der Benutzerdaten in der `UserBean` ist natürlich nur exemplarisch zu verstehen und soll die Funktionsweise des Persistenz API innerhalb des Application Servers anhand eines bekannten Beispiels demonstrieren.

Stateful vs. Stateless Session Bean

Da die Session Bean in Listing 10.37 nun statt des Benutzernamens die `userId` speichert, sind Sie immer noch auf Stateful Session Beans angewiesen und können nicht auf die wesentlich ressourcenschonenderen Stateless Session Beans zurückgreifen. Dies könnten Sie allerdings leicht erreichen, indem Sie die Service-Schnittstelle der EJB erweitern und den Client nun mit der Speicherung der ID beauftragen.

Erweiterung der zu speichernden Benutzerdaten

Durch die Verwendung der `UserBean` können Sie die zu speichernden Daten nun leicht erweitern, ohne die Session Bean mit immer mehr Instanzvariablen versehen zu müssen. Datenbanken sind auf die Speicherung großer Datenmengen optimiert und über das Persistenz API können Sie komfortabel und objektorientiert darauf zugreifen.

Bereitstellung von Datenbankobjekten über einen Persistenz-Service

Da die Entity-Objekte des neuen Persistenz API im Gegensatz zu ihren EJB-2.x-Pendants nicht auf den Einsatz innerhalb eines Application Servers beschränkt sind, können diese nun auch direkt an den Client gesendet und von diesem als losgelöste Objekte weiterverwendet werden. Es ist also nicht mehr zwingend notwendig, die Datensätze in Wertobjekte (Value Objects) zu überführen.

Tipp

Denken Sie daran, dass es sich bei der Verwendung von EJBs häufig um Remote Procedure Calls (RMI) handelt, bei denen die ausgetauschten Objekte serialisiert werden müssen. Übertragen Sie also möglichst kleine Objekte und verwenden Sie nach Möglichkeit das Lazy-Loading (siehe Abschnitt 10.6.4).
Achtung: Auf Seiten des Clients sind die Entity-Beans des Servers als losgelöste Objekte zu betrachten.

Vom Client zurück an der Server übertragene Objekte, deren Änderungen in die Datenbank übernommen werden sollen, müssen natürlich über die Methode `merge()` synchronisiert werden.

10.8 Zusammenfassung

Das neue Persistenz API gestattet es Ihnen, einfache JavaBeans ohne vorgegebenen Lebenszyklus dauerhaft in einer Datenbank zu speichern. Das einzige, was hierfür notwendig ist, sind einige Annotationen in der JavaBean, ein Persistence Descriptor (*persistence.xml*) und eine Implementierungsschicht wie beispielsweise Hibernate.

Neben den Basis-Datenbankoperationen Erzeugen, Suchen, Update und Löschen eines Datensatzes, für die Ihnen der Entity Manager entsprechende Methoden zur Verfügung stellt, bietet Ihnen die Spezifikation mit der an SQL angelehnten Query Language auch die Möglichkeit, komplexe Anfragen zu formulieren.

Der letzte Abschnitt zeigte Ihnen außerdem, wie Sie nunmehr drei verschiedene Java-EE-Technologien – JNDI, EJB und Persistenz – gemeinsam einsetzen können, um flexible und leistungsstarke Anwendungen zu entwickeln.

11

eXtensible Markup Language (XML)

eXtensible Markup Language oder kurz XML, so heißt der große Hype zu Beginn dieses Jahrtausends. Kaum sieben Jahre später ist diese Technologie aus keinem Computerbuch mehr wegzudenken. Es gibt ganze Zeitschriften, die sich nur mit diesem Thema beschäftigen, und eine nicht enden wollende Anzahl von Diplomarbeiten eröffnet immer neue Möglichkeiten der Nutzung von XML.

Ob Sie nun ein wiederverwendbares XML-Dokument oder ein konfigurierbares XSL-Stylesheet benötigen, eine SVG-Grafik definieren oder eine Datenbank aufbauen: XML scheint eine Wunderwaffe für jedes Problem und es vergeht kein Tag, an dem nicht irgendjemand eine neue Idee, samt eigens dafür entwickelter Abkürzung, präsentiert, mit der dann wieder alles schneller, schöner, kommunikativer und trotzdem einfacher realisiert werden kann.

Informationen sind es, um die sich bei XML alles dreht, und der große Traum, diese über alle Plattformen und Programmiersprachen hinweg zu teilen – egal, ob von Computer zu Computer oder zwischen Maschine und Mensch. Die Sprache soll frei definiert und auf intuitive Weise verstanden werden können. Dieses Kapitel zeigt Ihnen, was sich hinter der eXtensible Markup Language (XML) verbirgt, was sie zu leisten vermag und wie Sie sich ihre Eigenschaften zunutze machen können.

11.1 Kurze Einführung in XML

XML ist, wie der Name schon sagt, eine Markup Language, zu Deutsch eine Auszeichnungssprache, und mit ihr schließt sich ein Kreis zu den ersten Kapiteln in diesem Buch, in denen Sie schon einmal mit einer solchen Sprache zu tun hatten: Die Rede ist von der Hypertext Markup Language (HTML), die Sie einsetzen, um Ihre Webseiten und JSPs zu strukturieren.

Info

Es gibt Entwickler, die den Vergleich zwischen XML und HTML kritisieren. Ausgangspunkt dieser Kritik ist der Standpunkt, dass XML eine Metasprache und HTML eine aus einer Metasprache generierte Anwendung seien und hier somit Äpfel mit Birnen verglichen würden.
Dies ist nach Auffassung des Autors jedoch eine für die praxisnahen Beispiele dieses Buchs zu akademische Auffassung und so werden Sie in diesem Kapitel hin und wieder Vergleiche zu HTML finden.

Auszeichnungssprachen dienen dazu, textbasierte Dokumente zu verfassen und Informationen zur Struktur zusammen mit dem Inhalt abzulegen. Die Strukturinformationen werden dazu in so genannte Tags verpackt, die dem Betrachter bei der Darstellung in der Regel nicht mehr angezeigt werden.

Was ist eigentlich ein Tag?

Ein Tag besteht aus einem Paar spitzer Klammern (<>), die die Information in Form bestimmter Zeichenketten einschließen. Zu jedem Tag mit einem frei wählbaren Namen (<tag>) gehört ein End-Tag mit folgender Struktur </tag>. Diese sind in ein Dokument eingefügt, wobei die Zeichen zwischen Start- und End-Tag als Rumpf des Tag bezeichnet werden:

Listing 11.1
Aufbau eines Tag

```
...
   <tag> ... Rumpf des Tags ... </tag>
...
```

Eine Sonderform bildet dabei das so genannte leere Tag <tag/>, welches Start- und End-Tag zusammenfasst und keinen Rumpf besitzt.

Achtung

Da ein Tag in XML-Dokumenten gleichsam die kleinste Informationsstruktur bildet, wird es bei der Verarbeitung von XML in der Regel als Element bezeichnet. In serialisierter Form, als Datei, werden die einzelnen Elemente zwar durch Tags symbolisiert, aber hierbei handelt es sich nur um eine mögliche Darstellungsform für die Strukturen, die im Arbeitsspeicher ganz anders repräsentiert werden kann.

Die Ähnlichkeit zwischen XML und HTML kommt nicht von ungefähr, obgleich es zwei wesentliche Unterschiede gibt:

- XML ist erweiterbar.

  Während HTML über einen festen Satz von Tags verfügt, der im Standard der Sprache festgeschrieben ist und abgesehen von Spracherweiterungen statisch ist, beschreibt die XML-Spezifikation lediglich, nach welchen Regeln Elemente definiert werden können. Den Rest der Arbeit müssen Sie erledigen. Es gibt also nicht die XML-Tags in dem Sinne, in dem Sie von den HTML-Tags sprechen können.

Achtung

Wir werden dieser Nomenklatur folgen. Wenn also in den folgenden Absätzen von Elementen die Rede ist, ist nichts anderes als ein verarbeitetes Tag der XML-Datei gemeint.

- XML ist strenger als HTML.

  Da die Elemente in XML nicht vorgegeben sind, kann jemand, der sie interpretiert (z.B. ein Browser), die Funktion dieser Elemente auch nicht aus dem Kontext erschließen. Deshalb ist die XML-Spezifikation formaler und lässt insbesondere *keine* Ausnahmen zu.

  Wenn Sie beispielsweise in HTML erst eine neue Tabellenzeile (`<tr>`) und anschließend eine Tabellenzelle (`<td>`) öffnen und anschließend nur die Zeile beenden (`</tr>`), ist dem Computer klar, dass Sie damit auch die Zelle abgeschlossen haben, da eine Zelle nur innerhalb einer Zeile existieren kann. Viele Browser stellen eine solche Seite korrekt dar, obwohl das Dokument fehlerhaft ist. In einem XML-Dokument kann ein solcher Kontext nicht erschlossen werden.

Info

Natürlich könnten Sie jetzt anmerken, dass Sie bereits in Kapitel 4 Ihre HTML-Seiten mit eigenen Tags erweitert haben, doch diese werden vor der Auslieferung des Dokuments stets entfernt bzw. durch Text und HTML ersetzt.
Dies muss auch geschehen, Ihr Browser versteht nämlich nichts anderes.

11.1.1 Kurze Geschichte der Markup-Sprachen

Nachdem Menschen dem Computer beigebracht hatten, Daten zu speichern und Ergebnisse zu berechnen, wollten sie diese natürlich auch ansprechend ausgeben. Da die Konsumenten in erster Linie wiederum Menschen waren, fassten die ersten elektronischen Formate vor allem die Darstellung ins Auge.

Diese meist binärkodierten Dokumente machten es Computern jedoch sehr schwierig, die Struktur der Dokumente zu erfassen oder ihren Inhalt gar zu verstehen. So sind farbige oder *kursiv gedruckte* Informationen, die sich vom Rest abheben und die Sie mit dem Auge sofort als wichtig erfassen, für den Computer nichts anderes als den Text unterbrechende Steueranweisungen.

Der erste große Durchbruch auf diesem Weg war die Ersetzung der Formatierungscodes durch *generische Tags*, die die wichtige Information »einklammerten.« Computer waren damit einen großen Schritt näher daran, `<code>Diese wichtige Information</code>` vom Rest des Textes unterscheiden zu können.

Seinen Ursprung hatte dieses Umdenken in einem IBM-Projekt der siebziger Jahre. Die geistigen Urväter Charles Goldfarb, Edward Mosher und Raymond Lorie schufen mit ihrer *Generalized Markup Language (GML)* die Grundlage der Anfang der achtziger Jahre entwickelten *Standard Generalized Markup Language (SGML)*.

Info

Für alle, die es interessiert: Ende 1997 veröffentlichte das W3C ein Dokument mit dem Titel *Comparison of SGML and XML*, welches unter *http://www.w3.org/TR/NOTE-sgml-xml.html* heruntergeladen werden kann.

SGML ist dabei ein allumfassendes und dennoch äußerst flexibles Kodierungsschema, welches die standardisierte Definition aller nur denkbaren Dokumente ermöglicht, und so wurde es bald überall dort eingesetzt, wo es um das Verfassen und den Austausch von Dokumenten ging: in Verlagen und Zeitungen, aber auch im US-Verteidigungsministerium und der US-amerikanischen Steuerbehörde, die für eine Institution damals viel Mut zur Innovation zeigte.

Doch gerade die Allmächtigkeit von SGML wurde ihr bald darauf zum Verhängnis. Der Standard war umfangreich und voller komplexer Parameter, die sich gegenseitig beeinflussten. So wurde die Implementierung von entsprechenden Anwendungen aufwändig und teuer und so suchte man bald nach Wegen, die Sprache zu vereinfachen, ohne die grundlegenden Ideen aufgeben zu müssen. Aus diesem Grund wird SGML oft auch als Mutter aller Auszeichnungssprachen bezeichnet.

Die erste große Vereinfachung gelang Tim Berners-Lee Anfang der neunziger Jahre. Er entwickelte eine Auszeichnungssprache, die es den am CERN arbeitenden Wissenschaftlern erleichtern sollte, ihre Ergebnisse zu publizieren und Informationen auszutauschen. Die von ihm und seiner Gruppe entwickelte Hypertext Markup Language (HTML) wird noch heute für die allermeisten Dokumente des Internets verwendet.

Nach dem großen Erfolg von HTML versuchten die unterschiedlichsten Arbeitskreise, diesen zu wiederholen und nun wiederum SGML webtauglich zu machen. Doch die allgemeine Spezifikation war zu unhandlich und umfangreich, um in kleine Browser-Applikationen integriert zu werden, und so musste ein Mittelweg her, der einerseits so flexibel und allgemein war wie SGML und sich anderseits so leicht erlernen und implementieren ließ wie HTML: Dies war die Geburtstunde für XML.

11.1.2 Die Geburtsstunde für XML

1996 rief Sun eine Arbeitsgruppe unter Leitung von Jon Bosak ins Leben, deren Ziel die Definition eines flexiblen und plattformunabhängigen Standards für die Darstellung von Dokumenten war – also quasi eine Art Java für Dokumente.

Info

Jon Bosak wird auch als Suns XML-Architekt bezeichnet. Er leitete die Arbeitsgruppe um den XML-Standard und ist eines der Gründungsmitglieder der *Organization for the Advancement of Structured Information Standards (OASIS)*. Bevor er zu *Sun Microsystems* wechselte und sich mit der Entwicklung von XML beschäftigte, entwickelte er ein SGML-basiertes Publishing-System, um Novells NetWare-Dokumentation online zur Verfügung zu stellen. Er war zudem maßgeblich an der Entwicklung des bei Linux- und Unix-Systemen zum Einsatz kommenden *DocBook*-Standards beteiligt. Daneben veröffentlichte er eine Reihe sehr populärer Artikel rund um XML, von dem sein 1997 verfasster Aufsatz mit dem Titel *XML, Java and the future of the Web* mit dem Leitbild *Giving Java something to do* zu den bekanntesten gehören dürfte. Mehr zu Jon Bosak und seinen Publikationen finden Sie unter *http://www.ibiblio.org/bosak/*.

Das Resultat war eine Metasprache, die auf den Ideen von SGML aufbaute, jedoch darin enthaltene komplexe und selten genutzte Eigenschaften außen vor ließ. So definiert XML keine eigenen Elemente, sondern beschränkt sich auf die Beschreibung, wie diese aussehen. Außerdem setzen die XML-Urväter auf bereits in Java etablierte Technologien, beispielsweise die Darstellung von Zeichen im Unicode-Format.

Mit diesem erweiterbaren Standard, auf dessen Grundlage nun wiederum eigene Sprachen definiert werden konnten, war man nun z.B. in der Lage, Informationen in einer sowohl von Maschinen wie auch von Menschen intuitiv zu erfassenden Form abzulegen. So sollten Sie, auch ohne weitere Einführung, keine Probleme haben, die folgende Information zu interpretieren:

Listing 11.2
Ein einfaches Element

```
...
   <publishing-date format="dd.MM.yyyy">03.07.2006</publishing-date>
...
```

Dabei wird eine komplexe Information in drei Teile gegliedert, die deren sichere Interpretation erlauben:

- Der Name des Elements (`<publishing-date>`)

  Da Sie Ihre Elementnamen in XML frei definieren, können Sie Ihre Informationen bereits an dieser Stelle katalogisieren. Dabei geben Sie jeder Information quasi eine persönliche Klammer aus Start-Tag und End-Tag, die diese einschließt und eindeutig begrenzt.

Diese Fähigkeit macht XML-Dokumente so intuitiv. Sie sehen auf den ersten Blick, welche Information vorliegt bzw. fehlt. Außerdem macht sie Ihre Applikationen robust gegen das Vertauschen zweier Werte.

Info

Wenn Sie jemals mit funktionalen Programmiersprachen zu tun hatten oder anderweitig mit stark geklammerten Ausdrücken arbeiten mussten, überlegen Sie mal, wie viel leichter Ihre Aufgabe mit eindeutig zu unterscheidenden »benannten« Klammern, z.B. in Form von Tags, gewesen wäre.

- Die Informationen im Rumpf des Elements (`03.07.2006`)

  Die Nutzinformationen eines Elements sind in der Regel im Rumpf enthalten. Dabei kann es sich sowohl um Text als auch um weitere Elemente handeln.
- Ergänzende Attribute (`format`)

  Über Attribute ist es Ihnen möglich, beliebig viele Zusatzinformationen zu Ihrem Element zu hinterlegen, welche z.B. die korrekte Interpretation sicherstellen.

Angenommen, das obige Element sei von einem Amerikaner erstellt worden und hätte nun die folgende Form:

Listing 11.3 Gleiche Information, andere Formatierung

```
...
   <publishing-date format="MM/dd/yyyy">07/03/2006</publishing-date>
...
```

Achtung

Als *Metainformationen* bezeichnet man Einträge, die die Ursprungsinformation ergänzen oder beschreiben. Diese Information über Informationen hilft, Missverständnisse und Fehler zu vermeiden.
Natürlich reicht es nicht allein, diese Informationen in ein XML-Dokument zu integrieren. Es bedarf auch hier einer Software, die Datum und Format ausliest, aber XML ermöglicht es Ihnen, diese Informationen strukturiert abzulegen und komfortabel zu verwalten.

Ein menschlicher Betrachter hätte, mit dem Wissen, dass es sich bei der Information um ein Datum handelt, auch bei dieser Form keine Probleme mit der Entschlüsselung des Tags. Ein Computer hingegen wäre ohne die ergänzende Information über das Format, in dem das Datum vorliegt, sicherlich hoffnungslos überfordert. Und hier liegt eine der großen Stärken von XML gegenüber anderen Datenaustauschformaten: Durch die Ablage von *Metainformationen* können Sie die eigentliche Information ergänzen und Fehlern vorbeugen. Auf der anderen Seite stören sie Sie auch nicht weiter, wenn sie nicht benötigt werden.

11.1.3 Element für Element zum Dokument – Regeln für XML

Nachdem Sie nun wissen, was ein Element ausmacht und dass diese wiederum Elemente enthalten können, sind Sie in der Lage, Basisinformationen zu immer größeren Einheiten zusammenzufassen, die immer noch relativ leicht interpretiert werden können.

Listing 11.4
Ein zusammengesetztes XML-Dokument

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<book book-number="2184">
   <title>Masterclass Java EE 5</title>
   <publishing-date format="dd.MM.yyyy">03.07.2006</publishing-date>
   <publishing-house>Addison-Wesley</publishing-house>
   <content>
      <chapter number="1">Kapitel 1</chapter>
      <chapter number="2">Kapitel 2</chapter>
   </content>
</book>
```

Listing 11.4 zeigt ein vollständiges XML-Dokument, an dem sich bereits alle wichtigen Eigenschaften und Regeln zeigen lassen.

Info

Das Einlesen eines XML-Dokuments bezeichnet man als *Parsen*.

1. Alle XML-Dokumente beginnen mit einem Prolog.

   Der Prolog weist dieses Dokument als XML-Dokument aus und enthält Angaben über die verwendete XML-Version. Um das Dokument korrekt einzulesen (parsen), können Sie optional den verwendeten Zeichensatz (`encoding`) angeben.

2. Alle Elemente mit Ausnahme des äußersten werden von einem Elternelement umschlossen.

   Alle in einem XML-Dokument auftretenden Elemente werden wiederum von einem Element umschlossen. Diese Reihe setzt sich bis zu einem einzelnen *Wurzelelement* (Root-Element) fort.

   Ein solches Wurzelelement ist beispielsweise das `<html>`-Tag in Ihren HTML-Dokumenten oder in obigem Beispiel das Element `<book>`.

3. Alle Start-Tags werden durch End-Tags geschlossen.

   Jedes geöffnete Tag wird durch sein Pendant geschlossen. Dieses Paar bildet eine Art Klammer um die von diesem Element eingeschlossene Information (Rumpf), bei der es sich um Text oder auch weitere Elemente handeln kann.

   Die einzige Ausnahme bildet hierbei das sich selbst schließende, leere Element, z.B. `<title/>`. Dieses Element ist Start- und End-Tag zugleich und besitzt keinen Rumpf. Es ist eine übliche Kurzschreibweise für `<title></title>`.

4. Es dürfen keine Verzahnungen von Elementen auftreten.

   Tags müssen in umgekehrter Reihenfolge geschlossen werden, in der sie geöffnet wurden, so dass keine Verschachtelungen von Elementen auftreten. Die folgende Kombination von Tags ist in XML-Dokumenten nicht gestattet und wird bereits vor dem Einlesen des Dokuments moniert.

   ```
   <content><chapter></content></chapter>
   ```

5. Attributwerte werden von doppelten Anführungszeichen eingeschlossen.

   Auch diese SGML-Regel wird in HTML sehr lax gehandhabt. In XML-Dokumenten sind die Anführungszeichen ("") ohne Ausnahme Pflicht. Zusätzlich muss allen Elementen auch ein expliziter Wert zugewiesen werden.

Erfüllt ein XML-Dokument diese Anforderungen, wird es als *wohlgeformt* bezeichnet.

Einrückungen und Blank-Zeichen

Die in Listing 11.4 dargestellten Einrückungen (engl. Indent) von eingeschlossenen Elementen sind übrigens optional und dienen nur der besseren Lesbarkeit. Auch ein fortlaufender Zeichenstrom von Elementen, ohne Einrückungen oder Zeilenumbrüche, wäre immer noch wohlgeformtes XML, solange das Dokument den oben angegebenen fünf Regeln gehorcht.

Listing 11.5
Ebenfalls wohlgeformtes XML-Dokument

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<book book-number="2184"><title>Masterclass Java EE 5</title>
<publishing-date format="dd.MM.yyyy">03.07.2006</publishing-date>
<publishing-house>Addison-Wesley</publishing-house><content>
<chapter number="1">Kapitel 1</chapter><chapter number="2">
Kapitel 2</chapter></content></book>
```

11.1.4 Dokumentzentriert vs. datenzentriert

XML ist vor allem eine Möglichkeit, Daten in eine bestimmte Form zu überführen und zu strukturieren. Da es inzwischen eine Vielzahl von Anwendungen für XML gibt, bei denen einmal die Daten selbst und einmal die Struktur der Daten im Vordergrund steht, prägen zwei Begriffe die Aufgabe des Dokuments.

- Dokumentzentriert

  Bei diesen Dokumenten liegt das Hauptaugenmerk auf der Struktur der Daten und der Reihenfolge, in der diese abgelegt sind. Typische Anwendungen sind Präsentationen und HTML-Seiten.

- Datenzentriert

  Diese Dokumente dienen in der Regel dazu, Informationen zu speichern oder zu übertragen, statt sie zu präsentieren. Dabei spielt die Reihenfolge, in der Geschwisterelemente angeordnet sind, in der Regel keine Rolle. Typische Anwendungen sind z.B. Inventursysteme oder auf XML basierende Datenbanken.

11.1.5 Wichtige auf XML basierende Standards

Keine neue Technologie ging mit so vielen neuen Abkürzungen einher wie XML. Dieser Absatz soll Sie in die wichtigsten Kürzel einführen, von denen Ihnen einige in den nächsten Kapiteln wiederbegegnen werden.

Info

Das World Wide Web Consortium (W3C) ist ein Gremium, das über die Einführung und Weiterentwicklung von Standards im Internet entscheidet, wobei ausschließlich patentfreie Technologien eingesetzt werden. Gründer und Vorsitzender des W3C ist übrigens niemand anderes als Tim Berners-Lee, der Erfinder von HTTP und HTML.

eXtensible Stylesheet Language (XSL)

Die XSL-Spezifikation war die erste Anwendung von XML und ist wohl mittlerweile die bekannteste auf XML basierende Technologie. Sie wurde bereits 1999 vom W3C vorgestellt und ermöglicht es Ihnen, Ihre Dokumente in verschiedene Formate (XSLT) oder für verschiedene Ausgabemedien (XSL-FO) zu transformieren, um diese anschließend beispielsweise auf unterschiedlichen Medien ausgeben zu können.

XPath

XPath ist eine eigenständige Spezifikation, die die Grundlage für viele weitere Technologien bildet. Sie ermöglicht es, Informationen und Elemente innerhalb eines XML-Dokuments aufzufinden oder herauszufiltern, und bildet damit eine Art Abfragesprache für XML.

Scalable Vector Graphics (SVG)

Diese ebenfalls vom W3C entwickelte Technologie ermöglicht die Definition von Vektorgrafiken auf Basis von XML. Diese Formate haben gegenüber pixelbasierten Formaten den Vorteil, dass Sie ohne Qualitätsverlust beliebig skaliert werden können.

Simple Object Access Protocol (SOAP)

Dieser aus dem *XML Remote Procedure Call (XML-RPC)* hervorgegangene Standard ermöglicht den Fernaufruf von Methoden und den Datenaustausch zwischen Applikationen auf Basis von XML, die die Grundlage der Web Services bilden.

Web Service Description Language (WDSL)

Nachdem die verschiedenen SOAP-Implementierungen der einzelnen Unternehmen zueinander inkompatibel waren, entwickelte eine Gruppe von Microsoft und IBM-Programmierern diesen Standard nach Vorbild der Interface Definition Language (IDL) zur WDSL weiter.

11.1.6 Vorteile von XML

XML ist der derzeit krönende Abschluss einer langen Erfolgsgeschichte von Auszeichnungssprachen. Obwohl dieser Standard noch sehr jung ist, hat er bereits Einzug in viele Bereiche des (Programmier-)Alltags gehalten und verdankt dies nicht zuletzt den folgenden Eigenschaften:

- XML ist allgemein und plattformunabhängig. Damit bildet es eine gute Basis für den Informationsaustausch sich immer mehr spezialisierender Anwendungen.
- XML ist textbasiert und leicht zu erstellen. Zwar gibt es inzwischen eine Unmenge verfügbarer IDEs, doch was Sie zum Erstellen eines XML-Dokuments eigentlich nur benötigen, ist ein einfacher Texteditor.
- XML ist strukturiert. Da die Informationen und Metainformationen zusammen abgelegt werden, können diese ohne großen Aufwand zusammengetragen werden. Komplexe Elemente lassen sich dabei aus einfacheren zusammensetzen.
- XML kann leicht durch Stylesheets formatiert werden. Dies ermöglicht es, über XML abgelegte Informationen leicht in verschiedene Formate zu transformieren.
- XML ist streng hierarchisch. Auf diese Weise kann jedes Element eines XML-Dokuments eindeutig einem Kontext zugeordnet werden.
- XML ist einfach zu validieren. Aufgrund der simplen Regeln, über die XML definiert wird und die keine Ausnahmen zulassen, können Sie die Syntax eines Dokuments leicht überprüfen.

11.2 Elemente eines XML-Dokuments

Um XML-Dokumente zu verarbeiten, bedienen Sie sich eines so genannten Parsers, einer Software, der Sie später in diesem Kapitel häufiger begegnen werden. Dieser analysiert die einzelnen Bestandteile des Dokuments und bereitet diese für die Weiterverarbeitung auf. Der Parser unterscheidet dabei zwischen den nachfolgenden Strukturen.

11.2.1 Der Prolog und XML-Anweisungen

Um ein XML-Dokument als solches zu identifizieren, muss es stets mit einer initialen Zeile, dem so genannten *Prolog*, beginnen. Dieser beschreibt die verwendete XML-Version sowie grundsätzliche Eigenschaften des Dokuments.

Listing 11.6 Prolog eines XML-Dokuments

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
... Hier folgt das eigentliche Dokument ...
```

Diese Zeile ist vergleichbar mit einer Page-Direktive für JSPs: Sie enthält Verarbeitungsanweisungen, ist jedoch *kein* Teil des resultierenden Dokuments. Jedes XML-Dokument beginnt mit dem Prolog `<?xml ... ?>`, dessen Attribute dem Parser folgende Informationen geben:

- `version` – die verwendete XML-Version

  Obwohl die Version 1.0 ohne Veränderung nun schon seit sieben Jahren zum Einsatz kommt, ist es gut möglich, dass auch dieser Standard eines Tages weiterentwickelt wird, um ihn neuen Gegebenheiten anzupassen.

Info

Wenn Sie nach der Anpassung Ihrer *Tomcat-* oder *JBoss-Konfiguration* (XML-Datei) plötzlich einen Fehler über ungültige Zeichen erhalten, überprüfen Sie, ob sich zum Beispiel deutsche Umlaute (ä, ö, ü usw.) eingeschlichen haben. Da diese kein Bestandteil von `UTF-8` sind, müssen wir in diesem Fall den Zeichensatz ändern oder ganz auf sie verzichten.

- `encoding` – der verwendete Zeichensatz (optional)

  Standardmäßig nimmt der Parser an, dass Ihr Dokument in UTF-8 kodiert ist. Verwenden Sie nun aber in einem Kommentar deutsche Umlaute, bricht der Parser die Verarbeitung des Dokuments mit einer entsprechenden Fehlermeldung ab, da er das empfangene Zeichen nicht versteht. Über das Attribut `encoding` können Sie den Zeichensatz des Parsers ändern.

XML Processing Instructions (PI)

Der obligatorische Prolog ist dabei eine Sonderform der *Processing Instructions (PI)*. Diese Verarbeitungsanweisungen geben der Applikation weitere Hinweise, wie das vorliegende Dokument zu behandeln ist.

Ein Processing Instruction (PI) hat dabei immer folgende Form:

```
...
   <?ZielDerProcessingInstruction Anweisungen ?>
...
```

Listing 11.7
Schematischer Aufbau einer Processing Instruction (PI)

Wobei das in Listing 11.6 definierte Ziel `xml` für den XML-Parser selbst gedacht ist. Eine weitere Processing Instruction könnte beispielsweise eine Transformation über XSL-Stylesheets beschreiben:

```
...
   <?xml-stylesheet href="stylesheets/myStyle.xsl" type="text/xsl"?>
...
```

Listing 11.8
Processing Instruction für eine Stylesheet Transformation

11.2.2 Das Element

Elemente sind neben Zeichenketten der Hauptbestandteil von XML-Dokumenten und dienen dazu, diese zu strukturieren und mit Meta-Informationen zu versehen. Dabei unterscheidet man die folgenden Varianten:

- Das gemeine/gewöhnliche Element
- Das leere Element
- Das Wurzelelement oder Root-Element

Das gemeine Element

Das gemeine oder gewöhnliche Element ist in der Regel das, was Sie von Ihren HTML-Seiten oder JSPs her kennen. Es besteht aus einem Paar spitzer Klammern (<>), die den Namen des Elements und gegebenenfalls zusätzliche Attribute einschließen.

Das Element teilt sich dabei in ein Start- und ein End-Tag auf, die beide denselben Namen haben müssen. Um ein End-Tag zu kennzeichnen, wird diesem ein Schrägstrich (/) vorangestellt.

Listing 11.9
Ein allgemeines Element (Beispiel)

```
...
   <NameDesElementes> ...Rumpf des Elementes... </NameDesElementes>
...
```

Alle Zeichenketten und Elemente, die zwischen Start- und End-Tag stehen, werden als Rumpf des Elements oder Elementinhalt bezeichnet, wobei dieser auch leer sein kann. Eine Kurzschreibweise dafür bildet das so genannte leere Element.

Listing 11.10
Allgemeines Element ohne Rumpf (Beispiel)

```
...
   <NameDesElementes></NameDesElementes>
...
```

Der Name eines Elements muss mit einem Buchstaben oder dem Unterstrich beginnen und darf beliebig viele Buchstaben, Zahlen, Punkte, Binde- und Unterstriche enthalten. Dabei wird zwischen Groß- und Kleinschreibung unterschieden.

Achtung

Die einzige Ausnahme bei der Bezeichnung Ihrer Elemente bildet die Zeichenkette `xml`. Sie stellt gewissermaßen ein reserviertes Wort dar. Somit darf keines Ihrer Elemente mit dem Buchstaben `xml` beginnen.

Das leere Element

Das leere Element ist eine Kurzschreibweise für ein Standardelement ohne Rumpf.

Listing 11.11
Ein leeres Element (Beispiel)

```
<NameDesElementes />
```

Diese Schreibweise hebt deutlich hervor, dass dieses Element keinen Rumpf besitzt (besitzen soll), und kann Sie so vor Tippfehlern bewahren.

Info

Auch das W3C empfiehlt die Verwendung von leeren Elementen explizit.

Das Wurzelelement

Sie wissen bereits, dass jedes in einem XML-Dokument enthaltene Element von einem anderen Element umschlossen werden muss, welches dann

auch als Elternelement bezeichnet wird. Diese Reihe setzt sich bis zu *einem* Wurzelelement (Root-Element) fort, in dessen Rumpf sich alle anderen Elemente des Dokuments befinden.

Das Root-Element kann auch ein leeres Elemente sein, wenn das Dokument keine weiteren Informationen enthält.

Achtung

Innerhalb eines Dokuments kann es stets nur *ein einziges* Wurzelelement geben. Wenn Sie mehrere gleichrangige Informationen speichern möchten, fassen Sie diese in einem neuen, gemeinsamen Elterelement zusammen, welches dann das Wurzelelement bildet.

Um also zwei Bücher (<book>) in einem XML-Dokument zusammenzufassen, könnten Sie diese in eine Bibliothek (<library>) integrieren.

Listing 11.12
Zusammenfassen mehrerer XML-Dokumente

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!-- Dieses Dokument demonstriert Ihnen, wie Sie durch Definition
     eines neuen Root-Elementes ('library') mehrere XML-Dokumente
     zusammenfassen können -->
<library>
   <book book-number="2184">
      <title>Masterclass Java EE 5</title>
      <publishing-date
         format="dd.MM.yyyy">03.07.2006</publishing-date>
      <publishing-house>Addison-Wesley</publishing-house>
      <content>
         <chapter number="1">Kapitel 1</chapter>
         <chapter number="2">Kapitel 2</chapter>
      </content>
   </book>
   <book book-number="6979">
      <title>Jetzt lerne ich J2EE</title>
      <publishing-date
         format="dd.MM.yyyy">12.10.2005</publishing-date>
      <publishing-house>Markt und Technik</publishing-house>
      <content>
         <chapter number="1">Kapitel 1</chapter>
         <chapter number="2">Kapitel 2</chapter>
      </content>
   </book>
</library>
```

Attribute eines Elements

Jedes Start-Tag kann, analog zu HTML, mit beliebig vielen Attributen versehen werden, um die verarbeitende Applikation mit Zusatzinformationen zu versorgen.

Für die Namen von Attributen gelten die gleichen Regeln wie für Namen des Elements, wobei zusätzlich die folgenden Wörter reserviert sind und nicht verwendet werden dürfen:

Tabelle 11.1 Reservierte Wörter für Attributnamen

Name	Mögliche Werte	Beschreibung
`xml:lang`	Sprachcode nach RFC 1766, also beispielsweise `fr`, `en-US` usw.	Dieses Attribut kann in jedem Element verwendet werden und zeigt die Sprache des Rumpfs an. Es ist dafür gedacht, in mehrsprachigen Dokumenten das vom Betrachter am leichtesten verstandene Element zu finden.
`xml:space`	`default`, `preserve`	Dieses Attribut gibt an, ob führende oder schließende Leerzeichen (Tab, Space, Zeilenumbruch) ein Bestandteil des Elements sind (`preserve`) oder entfernt werden können (`default`).
`xml:link`	`simple`, `document`, `extended`, `group`	Dieses Attribut verweist auf ein anderes XML-Element.

11.2.3 Zeichenketten

XML dient dazu, Informationen in Form von Zeichenketten abzulegen. Diese stehen im Rumpf eines Elternelements und können von weiteren Elementen unterbrochen sein. Grundsätzlich können Sie in Ihren Zeichenketten jedes im eingestellten Zeichensatz (`encoding`) vorkommende Zeichen, inklusive Unicode-Sequenzen, verwenden. Da der Parser jedoch Elemente und Attribute sicher unterscheiden können muss, müssen die folgenden fünf Sonderzeichen maskiert werden:

Tabelle 11.2 Reservierte Sonderzeichen in XML

Zeichen	Escape-Sequence	Beschreibung
<	`<`	Dieses Zeichen zeigt dem XML-Parser den Beginn eines neuen Elements an, weswegen Sie es innerhalb von Zeichenketten durch die Zeichenkette `<` (*Less Than*) ersetzen müssen.
>	`>`	Für das Element-Abschlusszeichen gilt das Gleiche wie für <, weshalb Sie auch hier eine Umschreibung `>` (*Greater Than*) verwenden müssen.
&	`&`	Das kaufmännische Und (&) wird als Referenz auf ein anderes Element interpretiert. Diese Ausnahme gilt nicht für *Processing Instructions (PI)*.
"	`&quot;`	Da Anführungszeichen (") Beginn und Ende eines Attributs anzeigen, können Sie diese nicht für Attributwerte verwenden.
'	`&apos;`	Hier gilt das Gleiche wie für Anführungszeichen.

Die in der Tabelle dargestellten Umschreibungen für reservierte Zeichen werden in XML auch als Entity bezeichnet. Neben den fünf oben beschriebenen Entities können Sie auch eigene Zeichen-Entities definieren.

Zusätzlich dienen Entities dazu, nicht vom Zeichensatz abgedeckte Elemente einzufügen, sofern diese über eine Unicode-Darstellung verfügen.

Info

Natürlich gelten die in Tabelle 11.2 beschriebenen Sonderzeichen auch für HTML-Dokumente.

```
&#Dezimalcode im Unicode-Zeichensatz;
&#xHexadezimalcode im Unicode-Zeichensatz;
```

Listing 11.13
Einsatz von Unicode in XML-Dokumenten

Um beispielsweise einen Zeilenumbruch (Code 13) einzufügen, können Sie folgende Entity verwenden:

```
...
   <satz>Dies ist ein &#13; umgebrochener Satz.</satz>
...
```

Listing 11.14
Verwendung einer Unicode-Entity

Zeichenkette vs. Attribut

Häufig stehen Sie vor der Entscheidung, eine atomare Information mit einer Zeichenkette im Rumpf eines Elements oder über ein Attribut zu hinterlegen. Beide Varianten haben Vor- und Nachteile, wobei Sie Folgendes berücksichtigen sollten:

- Datenrepräsentation

  Je nachdem, ob Sie XML-Dokumente mit dem DOM-Modell oder über SAX-Events (was das ist, erfahren Sie in den folgenden Absätzen) verarbeiten, eignen sich für erstere eher Elementinhalte und für SAX-Events Attribute zur Ablage von Informationen.

Info

Processing Instructions (PI), Elemente oder Zeichenketten kommen in so ziemlich jedem XML-Dokument vor, während die folgenden Elemente nicht zwingend benötigt werden.

- Kindelemente

  Besteht Ihre Information hingegen aus Text und weiteren Elementen, sind Attribute gänzlich ungeeignet, da Sie in diesen nur Zeichenketten ablegen können. Natürlich können auch die eingefügten Kindelemente wiederum Attribute und weitere Elemente enthalten.

11.2.4 Kommentare

Um Ihr Dokument oder Teile davon auch nach Jahren zu verstehen oder ihre Funktion für andere zu dokumentieren, ist es sinnvoll, Kommentare einzufügen. Außerdem können Sie über diese (analog zu Java) bestimmte Elemente zeitweilig auskommentieren, was besonders in der Testphase nützlich sein kann.

Kommentare haben in XML-Dokumenten die Form

Listing 11.15
Kommentarkasten in XML

```
...
   <!--
       ... Dies ist ein Kommentarkasten ...
   -->
...
```

XML kennt dabei nur Blockkommentare. Zeilenkommentare, die Sie bei Java mit `//` einleiten, gibt es nicht.

Achtung

Laut Spezifikation beginnen Kommentare mit der Zeichenkette `<!--` und werden mit `-->` abgeschlossen. Diese Definition verbietet es insbesondere, Kommentare ineinander zu verschachteln, da der Parser den zweiten öffnenden Kommentarkasten (`<!--`) übersieht und mit dem zweiten schließenden Kommentar (`-->`) nichts anzufangen weiß:

Listing 11.16
Verbotenes Schachteln von Kommentarkästen

```
<!--
     Hier beginnt der erste Kommentar-Kasten...

         <!-- Dies ist ein verbotener Kommentarkasten ! -->

                          ...hier endet der erste Kommentar-Kasten
-->
```

Verschachtelte Kommentare dieser Art werden vom Parser immer mit einer Fehlermeldung quittiert.

11.2.5 Namensräume

Da in XML jeder Programmierer seine eigenen Elemente definieren kann, kommt es häufig zu Überschneidungen, bei denen zwei Anwender Elemente mit gleichem Namen für unterschiedliche Aufgaben vorsehen. Solange alle Elemente nur in »ihrem« Dokument vorkommen und eine Anwendung nur Dokumente einer Quelle verarbeitet, treten keine Fehler auf. Kritisch wird es erst, wenn Sie beide Dokumente kombinieren und die Elemente mischen. Dann kann die Anwendung beide Versionen nicht mehr unterscheiden (beide Elemente besitzen den gleichen Namen!).

Bei der Entwicklung eigener Tag-Bibliotheken standen Sie vor einem ähnlichen Problem und wie dort liegt die Lösung in der Definition von Namensräumen. Diese bilden gewissermaßen den »Nachnamen« für Elemente und fassen diese zu so genannten Elementfamilien zusammen. Auf diese Weise können Sie die Elemente trotzdem unterscheiden.

Listing 11.17
XML-Namespace (Beispiel)

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<mc:book book-number="2184" xmlns:mc="http://www.akdabas.de/javaee">
   <mc:title>Masterclass Java EE 5</mc:title>
   <mc:publishing-date
         format="dd.MM.yyyy">03.07.2006</mc:publishing-date>
   <mc:publishing-house>Addison-Wesley</mc:publishing-house>
   <mc:content>
      <mc:chapter number="1">Kapitel 1</mc:chapter>
      <mc:chapter number="2">Kapitel 2</mc:chapter>
   </mc:content>
</mc:book>
```

Achtung

Laut Spezifikation ist das Präfix nur ein Platzhalter für die URL des Namensraums, die dann die eindeutige Zuordnung übernimmt. Zwei gleichnamige Elemente, die über unterschiedliche Präfixe die *gleiche* URL referenzieren, sind demnach also identisch, auch wenn nicht jede Applikation diese Unterscheidung vornimmt.
Des Weiteren können Sie natürlich nur Präfixe verwenden, die entweder in dem Element selbst oder zuvor in einem seiner Elternelemente deklariert wurden. Präfixe haben damit wie Variablen einen Gültigkeitsbereich, innerhalb dessen sie existieren, und können dort sogar überschrieben werden.

Über das Attribut `xmlns:mc` definieren Sie für das Element `<book>` und seinen Rumpf einen Namensraum für die URL `http://www.akdabas.de/javaee` und verknüpfen diesen mit dem Präfix `mc`. Jedes Element, dessen Name mit diesem Präfix ausgestattet wird, gehört danach diesem Namensraum an.

11.2.6 CDATA-Rohdaten

Alles textbasiert ablegen zu können, ist zwar schön und gut, doch es gibt auch Daten, die nicht vom XML-Parser interpretiert werden dürfen, etwa weil es sich hierbei um binäre Daten oder SQL-Scripte handelt, in denen Sie nicht alle enthaltenen Sonderzeichen (vgl. Tabelle 11.2) umkodieren wollen.

In diesem Fall können Sie den Inhalt (oder Teile des Inhalts) eines Elements als nicht zu interpretierende Zeichenkette (*Character Data*, `CData`) markieren, die wortwörtlich ins Dokument übernommen wird. Ein `CData`-Abschnitt beginnt stets mit der Zeichenkette `<![CDATA[` und endet mit `]]>`.

```
...
   <!-- Der im Rumpf des Tabg 'text' enthaltene Text wird nicht
        auf XML-Strukturen oder Sonderzeichen hin interpretiert -->
   <text>
      <![CDATA[ Dieser Text wird wortwörtlich übernommen
                und nicht interpretiert ]]>
   </text>
...
```

Listing 11.18
Verwendung von Rohdaten (CData)

Einige Parser unterstützen das Verarbeiten von Binärinformationen in CDATA-Sektionen nur unzureichend und stürzen ab, wenn der Datenstrom zufälligerweise Steuerzeichen ähnelt, die das Ende eines Blocks anzeigen.

Ein `CData`-Block kann sowohl *Sonderzeichen (Entities)* als auch binäre Daten wie Bilder enthalten. In ihm enthaltene Elemente werden als Text gelesen und nicht interpretiert.

11.3 Verarbeitungsmodelle für XML

Wie bereits eingangs beschrieben, handelt es sich bei XML lediglich um eine Spezifikation, wie Daten zu strukturieren sind. Wie diese anschließend von Programmen zu verarbeiten sind, lässt sie bewusst offen.

Derzeit existieren zwei gängige Modelle, um XML-Dokumente mit Java verarbeiten zu können.

- Der baumorientierte Ansatz *DOM*, den Sie als Nächstes kennen lernen werden, bildet das Dokument als Menge von Knoten (für Elemente, Zeichenketten etc.) über Listen und Maps ab. Das Dokument kann dabei in alle Richtungen durchlaufen (traversiert) werden.

  Da der Speicherbedarf bei diesem Modell wesentlich größer als beim SAX-Modell ist, eignet es sich vor allem für kleinere Dokumente.
- Das auf Ereignissen basierende *SAX-Modell*, das später in diesem Kapitel folgt, eignet sich aufgrund des erheblich geringeren Ressourcenbedarfs vor allem für umfangreiche Dokumente. Dafür können Sie diese nur vorwärts abarbeiten. Wenn Sie über eine Stelle einmal »hinweggelesen« haben, können Sie nicht mehr zu dieser zurückkehren.

Seit dem Java SDK 1.4 sind die Klassen zur Verarbeitung mit DOM und SAX sowie ein Parser zum Einlesen von XML-Dokumenten im API enthalten. Diese werden von Sun unter dem Oberbegriff *Java API for XML Processing (JAXP)* zusammengefasst. Daneben existieren viele andere Implementierungen, um ein Dokument in den Speicher zu überführen und dort zu traversieren.

Die Beispiele in diesem Kapitel verwenden dabei als Parser Apaches Referenzimplementierung *Xerces*, die von der Homepage *http://xml.apache.org/xerces2-j/* heruntergeladen werden kann und sich vor allem durch ihre Geschwindigkeit und hohe Standardtreue auszeichnet. Um diesen oder einen anderen Parser verwenden zu können, laden Sie einfach die entsprechende JAR-Datei herunter und binden diese in den Classpath ein.

11.3.1 DOM vs. JDOM vs. DOM4J

Auch bei der Verarbeitung von XML-Dokumenten im DOM-Modell werden Sie in diesem Kapitel ein API kennen lernen, dessen Komfort weit über die Standardimplementierung von Sun hinausgeht.

Das DOM-API

Das ursprüngliche Document Object Model (DOM) ist eine plattform- und programmiersprachenunabhängige Spezifikation für die Verarbeitung von strukturierten Elementen wie HTML, SGML oder XML. Sie wurde vom W3C standardisiert und basiert auf der *Interface Definition Language* der Open Management Group (OMG).

Damit bildet DOM gewissermaßen den kleinsten gemeinsamen Nenner aller Spezifikationen, in denen ein XML-Dokument sowohl in C++ wie auch in Java und vielen weiteren Programmiersprachen durch eine Menge miteinander verknüpfter Knoten repräsentiert wird. Das bisher verwendete Beispieldokument hat hier die folgende Form:

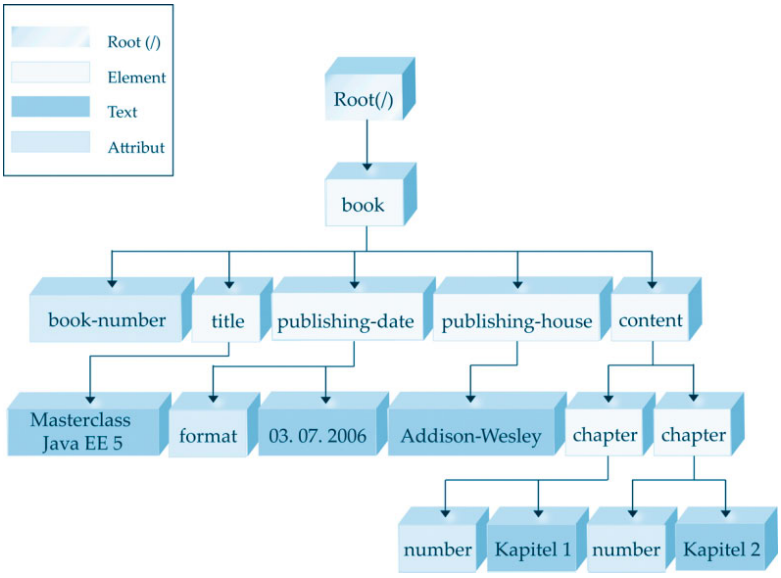


Abbildung 11.1
DOM-Repräsentation des Beispieldokuments

Wie Sie deutlich erkennen können, gibt es im DOM-Modell verschiedene Knoten für die Wurzel (Root, /), Elemente, Zeichenketten, Attribute etc. Damit ist dieses Modell auf der einen Seite sehr mächtig, auf der anderen aber auch sehr komplex. Da es außerdem für alle Programmiersprachen gleichermaßen gelten soll, kann es auch nicht auf komfortable Java-Funktionalitäten zurückgreifen.

Da die Arbeit mit diesem Modell deshalb oft schwerfällig ist, werden wir es hier nicht weiter betrachten. Wenn Sie dennoch mehr über den Umgang mit DOM lernen möchten, empfehlen wir Ihnen das Tutorial unter *http://www.w3schools.com/dom/*.

Das Java Document Object Model-API (JDOM)

Wenn Sie aufgrund des Namens nun einen erweiterten Standard von Sun erwarten, liegen Sie falsch. Hinter JDOM verbirgt sich ein weit verbreitetes, freies OpenSource API, welches mit dem Ziel entwickelt wurde, die Verarbeitung von XML-Dokumenten unter Java möglichst einfach und trotzdem ressourcenschonend zu ermöglichen.

Das API und seine Dokumentation finden Sie auf der Homepage des Projekts unter *http://www.jdom.org*. Dort erfahren Sie ebenfalls, dass JDOM mit dem *Java Service Request 102 (JSR-102)* einen Standardisierungsprozess durchläuft und bei erfolgreichem Abschluss später in die Standardbibliothek von Sun aufgenommen werden wird.

Das DOM4J-API

DOM4J (*http://www.dom4j.org/*) ist neben JDOM ein weiteres populäres OpenSource API, um XML mit Java zu verarbeiten. DOM4J versucht aber im Gegensatz zu JDOM, möglichst nah an der DOM-Spezifikation zu bleiben, während JDOM Effizienz und Komfort für Java-Entwickler in den Vordergrund stellt.

Für welches API Sie sich letztlich entscheiden, ist Geschmackssache, da alle drei die Kommunikation mit anderen Applikationen nach W3C-Standard unterstützen. Wir favorisieren in diesem Buch JDOM, wobei Sie sich mit dem Wissen dieses Kapitels und entsprechenden JavaDocs auch problemlos in die anderen Bibliotheken einarbeiten können sollten.

11.4 Arbeiten mit dem JDOM

Nach all der Theorie ist es nun Zeit, erste Erfahrungen im Umgang mit XML zu sammeln. In diesem Abschnitt finden Sie Beispiele zur Arbeit mit dem JDOM-API.

11.4.1 Erzeugen eines neuen XML-Dokuments

Das folgende Listing zeigt Ihnen, wie Sie ein neues XML-Dokument im Arbeitsspeicher erzeugen können.

Listing 11.19 Erzeugen eines neuen XML-Dokuments

```
package de.akdabas.javaee.jdom;

import org.jdom.Element;
import org.jdom.Document;

/** Demonstriert die Verwendung des JDOM-API */
public class CreateDocument {

   /** Erzeugt ein neues XML-Dokument */
   public static void main(String[] args) {

      // Erzeugen eines neuen Dokument-Containers
      Document doc = new Document();

      // Erzeugen des Elementes '<book>'
      Element root = new Element("book");

      // Setzen des Root-Elementes des Dokument-Containers
      doc.setRootElement(root);
   }
}
```

Zunächst erzeugen Sie ein `Document`-Objekt als Container für Ihr späteres Dokument. Anschließend erzeugen Sie das Wurzelelement `book` und fügen es in den `Document`-Container ein. Während ein (Wurzel-)`Element` mehrere weitere Elemente enthalten darf, kann ein `Document` stets nur *ein* Wurzelelement aufnehmen.

Wenn Sie Ihr Dokument an dieser Stelle ausgeben, würden Sie folgendes Resultat erhalten:

```
<?xml version="1.0" encoding="UTF-8" ?>
<book />
```

Listing 11.20
Form des ersten XML-Dokuments

Kurzform der Erzeugung eines XML-Dokuments

Einer der größten Vorteile von Java und des JDOM-API sind jedoch die Kurzformen, mit denen Sie Dokumente kurz und knapp definieren können. So können Sie die gleiche Funktionalität wie in Listing 11.19 auch mit folgendem »Einzeiler« formulieren:

```
...
   /** Erzeugt ein neues XML-Dokument (Kurzform) */
   public static void main(String[] args) {

      // Erzeugt ein neues Dokument mit dem Root-Element '<book>'
      Document doc = new Document(new Element("book"));
   }
...
```

Listing 11.21
Erzeugen eines neuen XML-Dokuments (Kurzform)

11.4.2 Hinzufügen von weiteren Elementen

Nachdem Sie einen `Document`-Container und ein Wurzelelement (`book`) erzeugt haben, können Sie weitere Elemente in dessen Rumpf einfügen. Dafür stellt Ihnen die Klasse `org.jdom.Element` zahlreiche `addContent()`-Methoden zur Verfügung. Das folgende Listing demonstriert den Aufbau des Beispieldokuments aus Listing 11.4.

```
package de.akdabas.javaee.jdom;
import org.jdom.Element;
import org.jdom.Document;

/** Demonstriert die Verwendung des JDOM-API */
public class CreateDocument {

   /** Erzeugt ein XML-Dokument von Grund auf */
   public static void main(String[] args) {

      // Erzeuge ein neues Document mit dem Root-Element <book>
      Document doc = new Document(new Element("book"));

      // Erzeuge das Element: <title>Masterclass Java EE 5</title>
      Element title = new Element("title");
      title.addContent("Masterclass Java EE 5");

      // Erzeuge das Element:
      // <publishing-date
      //    format="dd.MM.yyyy">03.07.2006</publishing-date>
      Element publishingDate = new Element("publishing-date");
      publishingDate.addContent("03.07.2006");
      publishingDate.setAttribute("format", "dd.MM.yyyy");

      // Erzeuge das Element:
      // <publishing-house>Addison-Wesley</publishing-house>
      Element publishingHouse = new Element("publishing-house");
      publishingHouse.addContent("Addison-Wesley");

      // Erzeuge das Element: <content>
      Element content = new Element("content");
```

Listing 11.22
Aufbau des Beispieldokuments (Listing 11.4)

Listing 11.22 (Forts.)
Aufbau des Beispieldokuments (Listing 11.4)

```
        // Erzeuge das Element:
        // <chapter number="1">Kapitel 1</chapter>
        Element chapter1 = new Element("chapter");
        chapter1.addContent("Kapitel 1");
        chapter1.setAttribute("number", "1");

        // Erzeuge das Element:
        // <chapter number="2">Kapitel 2</chapter>
        Element chapter2 = new Element("chapter");
        chapter2.addContent("Kapitel 2");
        chapter2.setAttribute("number", "2");

        /* Zusammensetzen der Elemente zur Struktur aus Listing 11.4*/

        // Referenzieren des Root-Elementes
        Element book = doc.getRootElement();

        // Setzen der ISBN-Nummer als Attribut des Root-Elementes
        book.setAttribute("book-number","2184");

        // Zusammenfügen der '<content>' - Elemente
        content.addContent(chapter1);
        content.addContent(chapter2);

        // Zusammenfügen des '<book>' - Elementes
        book.addContent(title);
        book.addContent(publishingDate);
        book.addContent(publishingHouse);
        book.addContent(content);
    }
}
```

Wie Sie hier am Beispiel des Elements `<publishing-date>` hervorgehoben sehen können, ist der Aufbau eines Dokuments aus verschiedenen Elementen ganz einfach. Zunächst erzeugen Sie das gewünschte Element und statten es mit dem Inhalt und benötigten Attributen aus und anschließend fügen Sie es einfach über die Methode `addContent()` in den Rumpf des Elternelements ein:

Listing 11.23
Einfügen eines Elements (Beispiel)

```
...
        // Erzeugen des Elementes '<publishing-date>'
        Element publishingDate = new Element("publishing-date");
        publishingDate.addContent("03.07.2006");
        publishingDate.setAttribute("format", "dd.MM.yyyy");
...
        // Einfügen in den Rumpf des Elter-Elementes '<book>'
        book.addContent(publishingDate);
...
```

11.4.3 Kurzform für die Definition des Beispieldokuments

Wie Sie sehen, kann das Erzeugen und Zusammenfügen der Elemente gerade bei großen Dokumenten sehr unübersichtlich werden. Aber JDOM hätte nicht den Ruf als kompaktes API, wenn sich nicht auch diese Form deutlich vereinfachen ließe.

```
package de.akdabas.javaee.jdom;
import org.jdom.Element;
import org.jdom.Document;

/** Demonstriert die Verwendung des JDOM-API */
public class CreateDocumentShort {

   /** Erzeugt ein neues XML-Dokument (Kurzform) */
   public static void main(String[] args) {

      // Erzeugt ein neues Dokument mit dem Root-Element <book>
      Document doc = new Document(new Element("book"));

      // Referenziert das Root-Element <book>
      Element root = doc.getRootElement();

      // Erzeugt die XML-Struktur aus Listing 11.4
      root.setAttribute("isbn", "2184")
         .addContent((Element) new Element("title")
            .setText("Masterclass Java EE 5"))
         .addContent((Element) new Element("publishing-date")
            .setText("03.07.2006")
            .setAttribute("format", "dd.MM.yyyy"))
         .addContent((Element) new Element("publishing-house")
            .setText("Addison-Wesley"))
         .addContent((Element) new Element("content")
            .addContent((Element) new Element("chapter")
               .setText("Kapitel 1"))
            .addContent((Element) new Element("chapter")
               .setText("Kapitel 2")));
   }
}
```

Listing 11.24
Aufbau des Beispieldokuments (Kurzform)

Wenn Sie zu einem über `new` erzeugten Objekt keinen Inhalt über die `addContent`-Methode hinzufügen, können Sie sich das zusätzliche *Casten* eigentlich sparen. Wenn Sie nach der Überarbeitung des Quellcodes jedoch trotzdem ein weiteres Element einfügen sollten, können Sie sich viel Sucherei sparen, wenn Sie den hier beschriebenen *Cast* bei jedem Objekt einbauen. Der *Overhead* durch die zusätzliche Anweisung hält sich in Grenzen und ist stets geringer, als bei der in Listing 11.22 beschriebenen Methode, da wir uns praktisch alle lokalen Referenzen auf die einzelnen Objekte sparen.

Info

Statt in Text über die Methode `setText()` in den Rumpf eines Elements einzufügen, können Sie auch die Methode `addContent(new Text())` verwenden, wobei diese bereits vorhandene Elemente nicht entfernt oder überschreibt.

Auf den ersten Blick werden Sie vielleicht etwas irritiert auf dieses Listing schauen und sich fragen, was das soll? Aber wenn Sie nur etwas länger hinsehen, werden Sie feststellen, wie sich die Struktur des Dokuments quasi von selbst erschließt. Wenn Sie mit dieser Kurzschreibweise arbeiten, können Sie auch später noch leicht neue Elemente hinzufügen oder ganze Zweige auskommentieren.

Diese Kurzform ist möglich, weil jede `addContent()`-Methode ein `Parent`-Objekt zurückgibt, hinter dem sich nichts anderes als das `Element` verbirgt, dessen `addContent()`-Methode Sie gerade aufgerufen haben. Sie müssen es nur noch zurückumwandeln (casten) und schon können Sie an dieser Stelle weitere Elemente einfügen. Das Casten ist deshalb nötig, da das `Parent`-Element des Root-Elements (`book`) beispielsweise der `Document`-Container und eben kein `Element` ist.

Achtung

Um Casting- und Verständnisprobleme zu vermeiden, empfehlen wir Ihnen beim Zusammenstecken der Dokumente in dieser Kurzform bei der Konstruktion eines Elements auf folgende Weise vorzugehen:

1. Erzeugen des Elements mit dem Konstruktor `new` und gleichzeitiges *Casten* auf den Typ.
2. Setzen des Textes über die Methode `setText()`.
3. Hinzufügen der erforderlichen Attribute.
4. Hinzufügen weiterer Kindelemente, so vorhanden, nach eben diesem Schema.

Listing 11.25 Aufbau eines neuen Elements (schematisch)

Das folgende Listing verdeutlicht diese Schritte schematisch:

```
...
   .addContent((Element)new Element("Neues-Element")
      .setText("Text")
      .setAttribute("Attribut 1", "Wert des Attributes")
      .setAttribute("Attribut 2", "...")
      .addContent(... weitere Kind-Elemente ...)
      .addContent(...))
...
```

Bei der Klammerung und dem korrekten Einrücken können Sie sich inzwischen in den allermeisten Fällen einfach zurücklehnen und sich von Ihrer IDE unterstützen lassen. Praktisch alle gängigen Entwicklungsumgebungen (inklusive des `vi` des Autors) verfügen über dieses Feature und so brauchen Sie am Ende einer Zeile nur so viele schließende Klammern einzufügen, bis das Element an der richtigen Stelle der Struktur steht.

Info

Die in Listing 11.22 und Listing 11.24 vorgestellten Techniken, XML-Dokumente von Grund auf zu erzeugen, führen zu exakt dem gleichen Ergebnis, wobei man anschließend nicht mehr feststellen kann, welche Variante gewählt wurde. Die zweite Technik erfordert etwas Übung und den Mut, die gewohnten Pfade des Programmieralltags einmal zu verlassen. Dabei werden Sie in der Regel gut durch Ihre IDE unterstützt. Gerade für große Dokumente ist diese Variante besser geeignet, da sich durch die regelmäßige Struktur zusätzliche Elemente und Attribute auch nachträglich leicht einfügen lassen.

11.4.4 Mischen von Text und Elementen

In den bisherigen Beispielen enthielt der Rumpf eines Elements entweder weitere Elemente oder eine Zeichenkette, jedoch nicht beides. Diese Struktur findet man häufig in datenzentrierten XML-Dokumenten (siehe Abschnitt 11.1.4). Ihre dokumentorientierten Pendants wie beispielsweise

XHTML-Dokumente enthalten hingegen viele Elemente, deren Inhalt aus Zeichenketten (Texten) besteht, die wiederum von Tags durchbrochen werden.

Info

Die *eXtensible Hypertext Markup Language (XHTML)* ist ein vom W3C entwickelter Standard, der HTML als Seitenbeschreibungssprache im Internet ablösen soll. Dazu enthält XHTML alle Elemente des HTML 4.01-Standards, wobei auf Konformität zum XML-Standard geachtet wurde. Damit lassen sich XML-Dokumente deutlich leichter verarbeiten und validieren.

Info

Einer der größten Unterschiede zwischen HTML und XHTML ist, dass bei Letzterem auch singuläre Elemente durch ein End-Tag abgeschlossen sein müssen. Damit wird `<br>` zu `<br />`, `<col>` zu `<col />` usw.

Listing 11.26
Tags und Zeichenketten gemischt

```
...
   <!-- Auszug aus einem (X)HTML-Dokument -->
   <td>Das ist ein <b>von Tags durchbrochener</b> Text.</td>
...
```

Da die bisher verwendete Methode `setText()` immer den bereits im Rumpf vorhandenen Inhalt überschreibt, müssen Sie hier mit dem Element `org.jdom.Text` arbeiten.

Listing 11.27
Elemente und Zeichenketten mit JDOM

```
import org.jdom.Text;
import org.jdom.Element;
...
   Element td = new Element("td");
   td.addContent((Text) new Text("Das ist ein ")
     .addContent((Element) new Element("b")
        .addContent((Text) new Text("von Tags durchbrochener")))
     .addContent((Text) new Text(" Text."));
...
```

Die Klasse `Text` ist das Pendant zum Textknoten `CharacterData` im DOM-Modell und ermöglicht es, Text und Elemente gemeinsam anzuordnen. Zu beachten sind hierbei lediglich zwei Dinge:

- Der mittlere Text steht im Rumpf des `<b>`-Tag und ist damit nur indirekt Bestandteil des Rumpfs von `<td>`.
- Die `Parent`-Elemente dieser `addContent()`-Methoden müssen dann natürlich in den Typ `Text` gecastet werden.

11.4.5 Einführen von Namensräumen

Sie haben bereits die Technik der Namensräume kennen gelernt, mit denen sich verschiedene Elemente zu einer Elementfamilie zusammenfassen lassen. Das folgende Listing zeigt Ihnen, wie Sie die Elemente des oben erzeugten Dokuments mit einem gemeinsamen Namensraum versehen (siehe Listing 11.17 und Listing 11.24).

Listing 11.28 Einsatz von Namensräumen

```
package de.akdabas.javaee.jdom;
import org.jdom.Element;
import org.jdom.Document;
import org.jdom.Namespace;

/** Demonstriert die Verwendung des JDOM-API */
public class CreateDocumentWithNamespace {

   /** Erzeugt ein neues XML-Dokument mit Namespace */
   public static void main(String[] args) {

      // Erzeugt ein Namespace Element für die angegebene URL
      Namespace ns =
        Namespace.getNamespace("mc","http://www.akdabas.de/javaee");

      // Erzeugt ein neues Dokument mit dem Root-Element <mc:book>
      Document doc = new Document(new Element("book", ns));

      // Referenziert das Root-Element <mc:book>
      Element root = doc.getRootElement();

      // Erzeugt die XML-Struktur aus Listing 11.4
      root.setAttribute("isbn", "2184")
         .addContent((Element) new Element("title", ns)
             .setText("Masterclass Java EE 5"))
         .addContent((Element) new Element("publishing-date", ns)
            .setText("03.07.2006")
            .setAttribute("format", "dd.MM.yyyy"))
         .addContent((Element) new Element("publishing-house", ns)
            .setText("Addison-Wesley"))
         .addContent((Element) new Element("content", ns)
            .addContent((Element) new Element("chapter", ns)
               .setText("Kapitel 1"))
            .addContent((Element) new Element("chapter", ns)
               .setText("Kapitel 2")));
   }
}
```

11.4.6 Einlesen eines vorhandenen XML-Dokuments

In der Praxis werden Sie aber, statt Ihre Dokumente immer von Grund auf neu zu erstellen, viel häufiger ein bereits in Dateiform vorliegendes XML-Dokument einlesen und anschließend mit dem JDOM-API weiterbearbeiten.

Info

Ein Adapter ist ein Entwurfsmuster der objektorientierten Programmierung und dient dazu, zwei Objekte, deren Schnittstellen nicht aufeinander passen, zu verknüpfen. Der Adapter übersetzt dabei die Schnittstelle der einen Klasse in die erwartete Schnittstelle der anderen. Mehr über Entwurfsmuster finden Sie im Standardwerk »Entwurfsmuster« von E. Gamma et. al., erschienen bei Addison-Wesley (ISBN 3827321999).

Um ein Dokument einzulesen, benötigen Sie einen so genannten Parser, der das Dokument auf syntaktische Korrektheit überprüft und anschließend in den Speicher überführt. Die oben genannten Bibliotheken zur Ver-

arbeitung können dabei mit einer beliebigen standardkonformen Parser-Implementierung verwendet werden, wobei wir uns in diesem und den nächsten Beispielen wieder auf Apaches Xerces stützen.

Die Adapter-Klasse, um ein JDOM-Dokument einzulesen, ist der `SAXBuilder` im Package `org.jdom.input`. Dieser kann das Dokument über verschiedene Eingabeströme wie `InputStream`, `Reader`, `URL` oder Objekte vom Typ `InputSource` parsen. Im folgenden Listing gehen wir dabei von einer Datei *book.xml* im Verzeichnis *xml* des aktuellen Projekts aus.

Listing 11.29
Einlesen eines vorhandenen Dokuments

Da das Validieren eines Dokuments während des Parsen oftmals größere Ressourcen benötigt, empfehle ich Ihnen, dieses Feature möglichst während der Implementierung zu nutzen und nach einer Testphase auszuschalten.

```
package de.akdabas.javaee.jdom;
import java.io.File;
import org.jdom.Document;
import org.jdom.input.SAXBuilder;

/** Parst eine XML-Datei in ein JDOM Document */
public class ParseDocument {

   /** Liest die Datei 'book.xml' ein und überführt Sie in JDOM */
   public static void main(String[] args) {

      // Gibt an, ob der Parser das Dokument validieren soll
      boolean validate = false;

      // Die zu verwendende Parser-Klasse, hier Apache Xerces
      String driverClass = "org.apache.xerces.parsers.SAXParser";

      // Die zu parsende Datei
      String pathToDocument = "xml/book.xml";

      // Erzeugen des SAXBuilders
      SAXBuilder builder = new SAXBuilder(driverClass, validate);

      Document doc;
      try {
         // Parsen des Dokumentes
         doc = builder.build(new File(pathToDocument));
      } catch (Exception exc) {
         exc.printStackTrace();
      }
   }
}
```

Achtung

Auch bei ausgeschalteter Validierung müssen Ihre Dokumente natürlich wohlgeformt (siehe Abschnitt 11.1.3) sein. An dieser Eigenschaft von XML führt kein Weg vorbei.

Über den Parameter `validate` legen Sie fest, ob der Parser das Dokument, z.B. gegen eine DTD oder eine Schema-Definition, auf semantische Korrektheit validieren soll, während die Zeichenkette `driverClass` den voll qualifizierenden Klassennamen der zu verwendenden Parser-Implementierung enthält. Anschließend wählen Sie aus den verschiedenen `build()`-Methoden der Klasse eine passende aus (hier `File`) und schon können Sie Ihr Dokument einlesen.

Verwendung einer beliebigen Parser-Implementierung

Im JAXP-Package von Sun ist ein Mechanismus implementiert, mit dem Sie einen beliebigen XML-Parser im Classpath auffinden können (`javax.xml.parsers.SAXParserFactory`). Wenn Ihnen die konkrete Implementierung also egal ist, lassen Sie den Parameter im Konstruktor des `SAXBuilder` einfach weg. Das API wird dann alle in den Classpath eingebundenen Bibliotheken nach einer geeigneten Klasse durchsuchen und die erste verwenden.

Listing 11.30 Erzeugen eines SAXBuilder ohne Angabe des Parsers

```
...
   // Verwendet die erste Parser-Implementierung im Classpath
   SAXBuilder builder = new SAXBuilder(validate);
...
```

Dieses Verfahren ist zu Testzwecken sicherlich sehr komfortabel, für Produktionsumgebungen jedoch ungeeignet, da Sie nicht wissen, welche Implementierung gerade verwendet wird, und sich diese gerade in der Performance und dem Ressourcenbedarf stark unterscheiden.

11.4.7 Traversieren eines Dokuments

Als Nächstes werden Sie das Dokument von oben nach unten traversieren. Dazu erweitern Sie das vorhergehende Listing und verwenden die Methode `getDescentants()` (dt. Nachkommen):

Listing 11.31 Traversieren des Dokuments

```
package de.akdabas.javaee.jdom;
import java.io.File;
import java.util.Iterator;
import org.jdom.Document;
import org.jdom.input.SAXBuilder;

/** Parst eine XML-Datei in ein JDOM Document */
public class ParseDocument {

   /** Liest die Datei 'book.xml' ein und überführt Sie in JDOM */
   public static void main(String[] args) {

      // Gibt an, ob der Parser das Dokument validieren soll
      boolean validate = false;

      // Die zu verwendende Parser-Klasse
      String driverClass = "org.apache.xerces.parsers.SAXParser";

      // Die zu parsende Datei
      String pathToDocument = "xml/book.xml";

      // Erzeugen des SAXBuilders
      SAXBuilder builder = new SAXBuilder(driverClass, validate);

      Document doc;
      try {
         // Parsen des Dokumentes
         doc = builder.build(new File(pathToDocument));

         // Iteriere über alle (Kind-)Elemente des Dokumentes
         Iterator iterator = doc.getDescendants();
         while (iterator.hasNext()){
            System.out.println(iterator.next());
         }
```

```
      } catch (Exception exc) {
         exc.printStackTrace();
      }
   }
}
```

Listing 11.31 (Forts.)
Traversieren des Dokuments

Dieses Listing durchläuft alle Elemente des Rumpfs in der Reihenfolge ihres Erscheinens und gibt sie anschließend auf der Kommandozeile aus. Zum Vergleich sei hier noch einmal das Originaldokument angegeben:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<book book-number="2184">
   <title>Masterclass Java EE 5</title>
   <publishing-date format="dd.MM.yyyy">03.07.2006</publishing-date>
   <publishing-house>Addison-Wesley</publishing-house>
   <content>
      <chapter number="1">Kapitel 1</chapter>
      <chapter number="2">Kapitel 2</chapter>
   </content>
</book>
```

Listing 11.32
Das Originaldokument (book.xml)

Dieses führt nach dem Parsen zu folgendem Resultat (die eventuell zwischen den Elementen vorhandenen Leerzeichen wurden der Übersichtlichkeit halber entfernt):

```
[Element: <book/>]
[Element: <title/>]
[Text: Masterclass Java EE 5]
[Element: <publishing-date/>]
[Text: 03.07.2006]
[Element: <publishing-house/>]
[Text: Addison-Wesley]
[Element: <content/>]
[Element: <chapter/>]
[Text: Kapitel 1]
[Element: <chapter/>]
[Text: Kapitel 2]
```

Listing 11.33
Ausgabe von Listing 11.31

Die Methode `getDescendants()` können Sie übrigens auch auf ein `Element`-Objekt anwenden, wobei Sie einen `Iterator` über dessen Kindelemente erhalten.

Unter direkten Kindelementen versteht man Elemente, deren Elternelement das aktuelle ist. Sie sind also die »direkten Nachfahren«. Enthalten die direkten Kindelemente wiederum Elemente, spricht man von indirekten Kindelementen, also in der Analogie von Enkeln, Urenkeln usw.

11.4.8 Ausgabe der direkten Kindelemente

Häufiger als das Traversieren *aller* im Rumpf vorkommenden Elemente ist die Ausgabe der *direkten* Nachfahren eines Elements, also nur die Elemente der nächsten Ebene, erforderlich. Um beispielsweise alle Kindelemente des Wurzelelements (<book>) auszugeben, schreiben Sie:

```
import java.util.List;
...
   // Ausgabe der direkten Kind-Element von <book>
   List<Element> children = doc.getRootElement().getContent();
   for (Element child : children) {
      System.out.println(child);
   }
...
```

Listing 11.34
Ausgabe der direkten Kindelemente des Wurzelelements

Die Methode `getContent()` liefert Ihnen eine Liste (`java.util.List`) der Kindelemente eines `Element`-Objekts zurück, die Sie anschließend ebenfalls mit einem `Iterator` durchlaufen können.

Listing 11.35
Ergebnis aus Listing 11.34

```
[Element: <title/>]
[Element: <publishing-date/>]
[Element: <publishing-house/>]
[Element: <content/>]
```

11.4.9 Löschen von Elementen

Um ein Element dauerhaft aus dem Dokument zu entfernen, stehen Ihnen drei Methoden der Klasse `Element` zur Verfügung:

- `public List removeContent()`
  Diese Methode löscht alle direkten Kindelemente des aktuellen Elements und gibt diese als Liste zurück.
- `public Element removeContent(int index)`
  Löscht das Element an der angegebenen Position (bei `0` beginnend) und gibt dieses zurück.
- `bublic boolean removeContent(Content element)`
  Löscht das übergebene Element, sofern es ein direktes Kindelement des aktuellen Elements ist und gibt den Erfolg der Löschoperation als `boolean` zurück.

Um also beispielsweise das erste Kindelement von `<book>` zu löschen (in diesem Fall `<title>`), schreiben Sie:

Listing 11.36
Entfernen des Elements <title> aus dem Dokument

```
...
   // Referenzieren des Root-Elementes '<book>'
   org.jdom.Element root = doc.getRootElement();

   // Entfernen des ersten Kindelementes von '<book>'
   org.jdom.Content removedElement = root.removeContent(0);

   // Ausgabe des entfernten Elementes, hier '<title>'
   System.out.println("Remove element: " + removedElement);
...
```

Nach dem Entfernen des Elements aus dem Dokument können Sie dieses, da zurückgegeben, entweder in ein anderes Dokument einfügen (`addContent()`) oder vom Garbage Collector entfernen lassen.

11.4.10 Herauslösen und Klonen eines Elements

Einer der häufigsten Fehler betrifft das Kopieren oder Klonen eines Elements. Um diesen nachvollziehen zu können, müssen Sie bedenken, dass ein Objekt der Klasse `Element` nicht einfach eine Zeichenkette mit dem Namen des Elements, sondern ein bestimmtes Objekt im Arbeitsspeicher Ihres Computers darstellt. Dieses ist dabei durch eine Referenz mit seinem Elternelement verknüpft, die Sie über die Methode `getParent()` auslesen können. Damit ist ein `Element` an *einer festen Stelle* innerhalb eines bestimmten Dokuments eingebunden und kann nicht einfach in ein neues Dokument oder eine andere Stelle eingefügt werden.

Um dies zu erreichen, müssen Sie sich einer der drei folgenden Methoden bedienen:

- Löschen des Elements per `removeContent()`

  Die eben beschriebene Methode eignet sich hervorragend, um Elemente samt Kindelementen von einem JDOM in einen anderen zu überführen.

- Herauslösen des Elements per `detach()`

  Wenn Sie nicht wissen, ob das aktuelle `Element` zu einem größeren Element gehört und Sie dieses aber woanders einfügen möchten, können Sie die Methode `detach()` verwenden. Diese meldet das Element bei seinem früheren Elternelement (so vorhanden) ab und macht es damit frei für das nächste Elternelement. Hat das `Element` kein Elternelement, passiert gar nichts.

- Klonen des Elements, samt Kindelementen über `clone()`

  Möchten Sie Ihr Element hingegen nicht von einem JDOM in einen anderen überführen, sondern explizit ein zweites Mal erstellen, verwenden Sie die Methode `clone()`. Diese erzeugt eine »tiefe Kopie« des `Element`-Objekts, indem sie alle direkten und indirekten Kindelemente mit kopiert. Der dabei entstandene Klon ist ungebunden und kann nach Belieben verwendet werden.

11.4.11 Einsatz von Filtern

Egal, ob Suchen, Löschen oder einfache Ausgabe: Bei jeder der bisher beschriebenen Methoden haben Sie zusätzlich die Möglichkeit, die Ergebnismenge durch den Einsatz von Filtern (Package `org.jdom.filter`) einzugrenzen. Dabei stehen Ihnen zwei Filter zur Auswahl:

- `ElementFilter`

  Dieser Filter dient dazu, Elementobjekte mit einem bestimmten Namen und/oder Namensraum herauszufiltern. Er gibt ausschließlich Elemente des Typs `org.jdom.Element` zurück.

- `ContentFilter`

  Bei diesem Filter können Sie eine Maske für Elemente erstellen, die den Filter passieren können. Alle anderen Elemente werden ausgefiltert.

Richtig flexibel wird diese Funktionalität allerdings erst durch die Möglichkeit, diese Filter über `and()`, `or()` und `negate()` zu komplexen Ausdrücken zu verknüpfen.

ElementFilter

Mit dem nächsten Listing können Sie beispielsweise einen `Filter` erstellen, der nur die Elemente `<publishing-date>` und `<publishing-house>` hindurchlässt.

Listing 11.37 Konstruktion eines zusammengesetzten Filters

```
...
   // Definition der Filter
   ElementFilter filter1 = new ElementFilter("publishing-house");
   ElementFilter filter2 = new ElementFilter("publishing-date");

   // Verknüpfen der Filter mit einem 'OR'
   Filter orFilter = filter1.or(filter2);

   // Filtern der Elemente eines Dokumentes
   Iterator iterator = doc.getDescendants(orFilter);
   while (iterator.hasNext()){
      System.out.println(iterator.next());
   }
...
```

Zunächst erzeugen Sie zwei `ElementFilter`-Objekte, die jeweils auf genau ein `Element` passen. Anschließend verknüpfen Sie beide über die Methode `or()` zu einem neuen `Filter`. Wenn Sie Listing 11.31 entsprechend abwandeln, erhalten Sie folgende Ausgabe (vgl. Abschnitt 11.4.8, Listing 11.35):

Listing 11.38 Ausgabe von Listing 11.37

```
[Element: <publishing-date/>]
[Element: <publishing-house/>]
```

ContentFilter

Der `ContentFilter` ist dazu gedacht, nur Elemente eines bestimmten Typs durchzulassen und alle anderen Typen herauszufiltern. Er kann mit den anderen Filtern kombiniert werden. Das nächste Listing konstruiert damit einen Filter, der nur auf Text- und Kommentarelemente passt.

Listing 11.39 Konstruktion eines Filters für Text und Kommentar

```
...
   // Grundeinstellung des Filters:
   // Sollen zunächst alle Elemente durchgelassen werden
   boolean allVisible = false;

   // Definition des Filters
   ContentFilter contentFilter = new ContentFilter(allVisible);

   // Durchzulassende Element-Typen konfigurieren
   contentFilter.setTextVisible(true);
   contentFilter.setCommentVisible(true);

   // Filtern der Elemente eines Dokumentes
   Iterator iterator = doc.getDescendants(contentFilter);
   while (iterator.hasNext()){
      System.out.println(iterator.next());
   }
...
```

Als Erstes entscheiden Sie über den Parameter im Konstruktor des Filters, ob zunächst alle Elemente gefiltert oder durchgelassen werden sollen. Anschließend konfigurieren Sie die gewünschte Funktionalität. Auch dieser Filter kann anschließend für alle `getDescendants()`- und `getContent()`-Methoden verwendet werden.

Kurzschreibweise der Definition eines ContentFilter

Zum Schluss noch ein Tipp für kompaktere Filterdefinitionen: Der `ContentFilter` speichert die gesetzten und gesperrten Elementtypen in einer *Bitmaske* (Typ `int`) ab. Dies ermöglicht es Ihnen, die vollständige Konfigu-

ration über *bitweises Oder* schon im Konstruktor zu erzeugen. Aus den vier Anweisungen wird dann ein eleganter Einzeiler:

```
...
   // Kompakte Definition des Filters aus Listing 11.39
   new ContentFilter(ContenFilter.TEXT | ContentFilter.COMMENT);
...
```

Listing 11.40
Kompakte Definition des Filters

11.4.12 Ausgabe des Dokuments

Nachdem Sie Ihren JDOM nun einlesen, traversieren und manipulieren können, möchten Sie das resultierende Dokument natürlich auch wieder in eine Datei oder einen anderen `Stream` zurückschreiben. Dabei helfen Ihnen die Klassen des Package `org.jdom.output`.

Info

Wir werden den Baum in den nächsten Beispielen der einfachen Dokumentation halber einfach auf die Konsole (`System.out`) ausgeben. Sie können die Beispiele natürlich auch adaptieren und die Ausgabe über einen anderen `java.io.OutputStream` z.B. in eine Datei umleiten.

Unformatierte Ausgabe des XML-Dokuments

Um ein Dokument als XML-Dokument auszugeben, verwenden Sie einfach eine Instanz der Klasse `XMLOutputter`, die ähnlich dem `SAXBuilder` mit verschiedenen Schnittstellen wie `Writer`, `OutputStream` etc. zusammenarbeiten kann.

```
import org.jdom.output.XMLOutputter;
...
   // Unformatierte Ausgabe eines zuvor erstellten Document 'doc'
   try {
      XMLOutputter out = new XMLOutputter();
      out.output(doc, System.out);
   } catch (java.io.IOException exc) {
      exc.printStackTrace();
   }
...
```

Listing 11.41
Unformatierte Ausgabe des Dokuments

Wenn Sie das Dokument neu erstellt und nicht von einer Datei geparst haben, liefert Ihnen dieser gut überschaubare Sechszeiler das folgende Resultat.

```
<?xml version="1.0" encoding="UTF-8?><book book-number="2184">
<title>Masterclass Java EE 5</title><publishing-date
format="dd.MM.yyyy">03.07.2006</publishing-date><publishing-
house>Addison-Wesley</publishing-house><content><chapter
number="1">Kapitel 1</chapter><chapter number="2">Kapitel2
</chapter></content></book>
```

Listing 11.42
Unformatierte Ausgabe

Dies liegt daran, dass das Dokument beim Erstellen ohne Einrückung (`indent`) erzeugt wird. Die Einrückungen eines geparsten Dokuments bleiben jedoch, wenn diese nicht explizit herausgefiltert werden, erhalten.

Tipp

Die »rohe« Ausgabe von XML in Listing 11.42 erzeugt den geringsten Overhead und eignet sich vor allem zur automatisierten Weiterverarbeitung, etwa wenn es sich bei dem Dokument um eine SOAP-Nachricht handelt. Diesen begegnen Sie in Kapitel 13 wieder.

Formatierte Ausgabe des XML-Dokuments

Wenn Sie das Aussehen Ihrer Dokumente nicht davon abhängig machen möchten, ob diese geparst oder neu erstellt wurden, können Sie dem `XMLOutputter` auch einen `Format` mitgeben, der die Ausgabe formatiert. Hierfür erweitern Sie Listing 11.31 nun ein letztes Mal, um die Ausgabe des Dokuments zu formatieren:

Listing 11.43 Formatierte Ausgabe eines Dokuments

```
package de.akdabas.javaee.jdom;
import java.io.File;
import java.util.Iterator;
import org.jdom.Document;
import org.jdom.input.SAXBuilder;
import org.jdom.output.XMLOutputter;
import org.jdom.output.Format;

/**
 * Parst eine XML-Datei in ein JDOM Document und gibt dieses
 * in verschiedenen Formen auf der Kommandozeile aus.
 */
public class ParseDocument {

   /** Liest die Datei 'xml/book.xml' ein */
   public static void main(String[] args) {

      // Gibt an, ob der Parser das Dokument validieren soll
      boolean validate = false;

      // Die zu verwendende Parser-Implementierung
      String driverClass = "org.apache.xerces.parsers.SAXParser";

      // Die zu parsende Datei
      String pathToDocument = "xml/book.xml";

      // Erzeugen des SAXBuilders
      SAXBuilder builder = new SAXBuilder(driverClass, validate);

      Document doc;
      try {
         // Parsen des Dokumentes
         doc = builder.build(new File(pathToDocument));

         // Iteration über alle (Kind-)Elemente des Dokumentes und
         // Ausgabe dieser auf der Kommandozeile
         Iterator iterator = doc.getDescendants();
         while (iterator.hasNext()){
            System.out.println(iterator.next());
         }

         // Erfeugen eines Ausgabe-Formats
         Format prettyFormat = Format.getPrettyFormat();

         // Erzeugen des XMLOutputter mit einem Format
         XMLOutputter out = new XMLOutputter(prettyFormat);
```

```
            // Formatierte Ausgabe des Dokumentes 'doc'
            out.output(doc, System.out);
         } catch (Exception exc) {
            exc.printStackTrace();
         }
      }
}
```

Listing 11.43 (Forts.)
Formatierte Ausgabe eines Dokuments

Neben den vordefinierten Formaten `RawFormat`, `PrettyFormat` und `CompactFormat` können Sie über die Klasse `Format` auch Ihren eigenen Stil wie beispielsweise die Einrückung der Kindelemente oder das Trimmen der Zeichenketten konfigurieren. Mit obigem Listing erhalten Sie nun auf jeden Fall folgende Ausgabe:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<book book-number="2184">
   <title>Masterclass Java EE 5</title>
   <publishing-date format="dd.MM.yyyy">03.07.2006</publishing-date>
   <publishing-house>Addison-Wesley</publishing-house>
   <content>
      <chapter number="1">Kapitel 1</chapter>
      <chapter number="2">Kapitel 2</chapter>
   </content>
</book>
```

Listing 11.44
Formatierte Ausgabe des Dokuments

11.4.13 Umwandeln von JDOM in DOM

Neben der Ausgabe des JDOM im XML-Format müssen Sie Ihren JDOM hin und wieder in einen spezifikationstreuen DOM vom Typ `org.w3c.dom.Document` umwandeln. Hier kommt das Package `org.jdom.adapters` ins Spiel.

Das folgende Listing wandelt Ihren JDOM mithilfe von Apaches Xerces-Parser in einen »gewöhnlichen« DOM um.

```
...
   // Wandelt einen JDOM in einen W3C-DOM um
   try {
      // Erzeugen des entsprechenden Adapters
      DOMOutputter domOutputter =
         new DOMOutputter("org.jdom.adapters.XercesDOMAdapter");

      // Umwandeln des Dokumentes: JDOM -> DOM
      org.w3c.dom.Document dom = domOutputter.output(doc);
   } catch (JDOMException jex) {
      jex.printStackTrace();
   }
...
```

Listing 11.45
Umwandeln eines JDOM in einen DOM

Achtung

Um diesen Code auszuführen, benötigen Sie neben dem JDOM-API die Xerces-Implementierung und die DOM-Spezifikation. Beide finden Sie z.B. in den Dateien *xercesImpl.jar* und *xml-apis.jar*, der Xerces-Distribution.

Die verwendete Adapter-Klasse wird dabei erst zur Laufzeit dynamisch per *Reflection* geladen und arbeitet anschließend mit den spezifischen APIs zusammen. Deswegen können Sie dieses Beispiel auch problemlos mit dem JDOM-API übersetzen und müssen die zusätzlichen Treiber nur zur Laufzeit zur Verfügung stellen.

Der anschließend erzeugte DOM wird dabei nach der Implementierung des *Apache Xerces* erstellt.

Info

Fragen Sie sich jetzt, warum so viele unterschiedliche Implementierungen für DOMs existieren? Nun, die W3C-Spezifikation ist eindeutig, jedoch nicht auf eine spezielle Plattform festgelegt. Sie beschreibt vielmehr, welche Methoden DOMs zur Verfügung stellen und wie sie funktionieren. Ob die Attribute unserer Elemente aber nun über *Listen*, *Maps* oder *Sets* abgebildet werden, bleibt dem Hersteller überlassen und so züchtet jeder seine eigene Baumsorte, die zwar kompatibel, aber eben nicht gleich sind.

11.5 Mit SAX-Events arbeiten

Eine weitere, besonders ressourcensparende Möglichkeit, XML-Dokumente zu verarbeiten, ermöglicht Ihnen das *Simple API for XML (SAX)*. Und dabei handelt es sich ausnahmsweise mal weder um eine Sun-Spezifikation noch um einen W3C-Standard. Die Technik hinter SAX wurde 1997 in der Mailing-Liste *XML-DEV* diskutiert und von der OpenSource-Gemeinde bis zur heutigen Version 2.0.2 weiterentwickelt.

SAX hat sich gerade bei der Verarbeitung von umfangreichen Dokumenten zu einem Quasi-Standard etabliert, bei dem Sie allerdings immer nur auf das aktuelle Element zugreifen und anschließend zum nächsten weitergehen können.

11.5.1 Arbeitsweise von SAX

Wenn Sie das kompakte XML-Dokument aus Listing 11.5 ohne Kenntnis seiner Struktur durchlesen, entwickelt Ihr Gehirn vielleicht folgende Gedankengänge:

- Element `book` mit Attribut `book-number` beginnt
- Öffnendes Tag von Element `title`
- Dann folgt die Zeichenkette »`Masterclass Java EE 5`«.
- Das Element `title` wird geschlossen.
- ...

Das sind im Wesentlichen auch die Ereignisse, die der XML-Parser beim Parsen des Dokuments registriert und zum Aufbau der internen DOM-Struktur verwendet. Beim SAX-Modell werden diese Ereignisse (engl. Event) nun nicht zum Aufbau einer Struktur im Speicher verwendet, sondern über Callback-Methoden direkt an die verarbeitende Klasse weitergegeben.

11.5.2 Callback-Methoden

Das Interface `ContentHandler`, das Ihre Klassen implementieren müssen, um SAX-Events verarbeiten zu können, definiert eine Reihe von Callback-Methoden, die Sie über die verschiedenen Ereignisse beim Parsen eines Dokuments informieren.

- `startDocument()`
  Mit dieser Methode zeigt Ihnen der Parser den Beginn eines Dokuments an. Sie wird nur ein einziges Mal gerufen und häufig zum Initialisieren der Klasse verwendet.
- `startElement()`
  Diese Methode zeigt den Beginn eines Elements an. Dabei werden Ihnen neben dem Namen und dem Namespace des Elements auch die vorhandenen Attribute übergeben.
- `characters()`
  Diese Methode enthält die in einem Element enthaltenen Zeichen. Sie kann auch mehrmals hintereinander gerufen werden und muss den Inhalt eines Elements nicht vollständig beschreiben.
- `endElement()`
  Das Pendant zu `startElement()`, zeigt Ihnen das Ende eines solchen an. Jetzt befinden Sie sich wieder im Rumpf des Elternelements.
- `endDocument()`
  Diese Methode wird als Letzte gerufen, um den Abschluss des Dokuments zu signalisieren.

Mit diesen fünf Methoden müssen Sie auf jeden Fall vertraut sein, da sie der Verarbeitung von Grundelementen jedes XML-Dokuments dienen. Seltener genutzt werden hingegen die folgenden sechs zusätzlichen Methoden des Interface `ContentHandler`.

- `startPrefixMapping()`
  Zeigt den Beginn eines Namensraums (z.B. `mc`) an. Diese Methode teilt Ihnen mit, dass die folgenden Elemente mit dem Präfix `mc` dem ebenfalls übermittelten Namensraum angehören. Da diese Information aber auch von der Methode `startElement()` übergeben wird, benötigen Sie diese nur sehr selten.

  `startPrefeixMapping()` wird unmittelbar *vor* der `startElement()`-Methode gerufen, in der dieser Namensraum definiert wird.
- `endPrefixMapping()`
  Diese Methode zeigt das Ende eines Namensraums an. Sie wird unmittelbar nach dem den Namensraum schließenden Tag gerufen.

Info

Enthält ein Element mehr als eine Namensraumdefinition, so werden die Methoden `startPrefixMapping()` bzw. `endPrefixMapping()` mehrmals hintereinander gerufen. Die Reihenfolge, in der die einzelnen Namensräume geöffnet und geschlossen werden, kann variieren. Obwohl ein Element mehrere Namensraumdefinitionen besitzen kann, ist es jedoch immer nur einem Namensraum zugehörig. Mehrere Definitionen sind dann nötig, wenn beispielsweise die Kindelemente zu unterschiedlichen Namensräumen gehören.

- `ignorableWhitespace()`
  Weiter vorn in diesem Kapitel haben Sie bereits gelernt, dass die Elemente eines XML-Dokuments nicht zwingend eingerückt (`indent`) werden müssen. Diese ignorierbaren Leerzeichen (engl. Whitespace), zu denen auch Zeilenumbrüche und Tabulatoren gerechnet werden, werden über die Methode `ignorableWhitespace()` übermittelt.
- `processingInstruction()`
  Mit dieser Methode teilt Ihnen der Parser das Auftreten von *Processing Instructions (PI)* mit. Hier können Sie auf selbst definierte Anweisungen warten und anschließend entsprechend reagieren.

  Auch die Standard-PIs, etwa von XSL Stylesheets, werden Ihnen mit dieser Callback-Methode signalisiert.
- `skippedEntity()`
  Mit dieser Methode übermittelt Ihnen der Parser gefundene Entities (siehe auch Abschnitt 11.2.3).
- `setDocumentLocator()`
  Der `DocumentLocator` ist kein Bestandteil des XML-Dokuments, sondern eine nützliche Hilfsklasse. Mit ihr können Sie die aktuelle Position (Zeile und Spalte) im XML-Dokument bestimmen. Dies ist besonders beim Debuggen nützlich, wenn Sie beispielsweise ein verbotenes Zeichen verwenden und dieses nicht per Hand suchen möchten. Die Methode wird nur ein einziges Mal gerufen, um die Klasse mit dem `DocumentLocator` für dieses Dokument auszustatten.

11.5.3 Ein Dokument via SAX parsen

Das nächste Listing zeigt Ihnen einen einfachen `ContentHandler`, der die registrierten Ereignisse einfach ausgibt:

Listing 11.46 Ein einfacher ContentHandler

```
package de.akdabas.javaee.sax;

import org.xml.sax.Locator;
import org.xml.sax.XMLReader;
import org.xml.sax.Attributes;
import org.xml.sax.ContentHandler;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.XMLReaderFactory;
```

Listing 11.46 (Forts.)
Ein einfacher ContentHandler

```
/** Ein einfacher ContentHandler zur Verarbeitung von SAX-Events */
public class SimpleContentHandler implements ContentHandler {

   /* Der 'DocumentLocator' für dieses Dokument */
   private Locator documentLocator;

   /** Die Methode 'main()' zum Testen der Klasse */
   public static void main(String[] args) {
      try {
         // Zu verwendende Parser-Klasse
         String parserClass = "org.apache.xerces.parsers.SAXParser";

         // Erzeugen des XML-Readers
         XMLReader reader =
             XMLReaderFactory.createXMLReader(parserClass);

         // Erzeugen eines SimpleContentHandlers
         SimpleContentHandler simpleContentHandler =
            new SimpleContentHandler();

         // Instanz ist Ereignis-Empfänger (Listener) für SAX-Events
         reader.setContentHandler(simpleContentHandler);

         // Parsen des Dokumentes, SAX-Ereignisse werden erzeugt
         reader.parse("xml/book.xml");
      } catch (Exception exc) {
         exc.printStackTrace();
      }
   }

   /* Methoden des Interface 'ContentHandler' */

   /** Zeigt den Beginn der Dokumentes an */
   public void startDocument() throws SAXException {
      System.out.println("Call: startDocument()");
   }

   /** Zeigt ein öffnendes Element an */
   public void startElement(String namespaceURI, String localName,
                            String qName, Attributes atts)
   throws SAXException {
      System.out.println("Call: startElement(), element: " + qName);
   }

   /** Geparste Zeichenketten zwischen Elementen */
   public void characters(char[] ch, int start, int length)
   throws SAXException {
      String text = new String(ch, start, length);
      System.out.println("Call: characters(), text: " + text);
   }

   /** Zeigt das Ende eines Elementes an */
   public void endElement(String namespaceURI, String localName,
                          String qName)
   throws SAXException {
      System.out.println("Call: endElement(), element: " + qName);
   }

   /** Zeigt den Abschluss eines Dokumentes an */
   public void endDocument() throws SAXException {
      System.out.println("Call: endDocument()");
   }

   /** Zeigt den Beginn eines Namensraumes an */
   public void startPrefixMapping(String prefix, String uri)
   throws SAXException {
```

Listing 11.46 (Forts.) Ein einfacher ContentHandler

```
        System.out.println("Call: startPrefixMapping(), prefix:" +
                           prefix);
    }

    /** Zeigt das Ende eines Namensraumes an */
    public void endPrefixMapping(String prefix) throws SAXException {
        System.out.println("Call: endPrefixMapping(), prefix: " +
                           prefix);
    }

    /** Nicht zwingend erforderliche Whitespace-Zeichen */
    public void ignorableWhitespace(char[] ch, int start, int length)
    throws SAXException {
        System.out.println("Call: ignorableWhitespace()");
    }

    /** Übermittelt eine Processing Instruction (PI) */
    public void processingInstruction(String target, String data)
    throws SAXException {
        System.out.println("Call: processingInstruction(), target: " +
                           target);
    }

    /** Übermittelt eine Entity */
    public void skippedEntity(String name) throws SAXException {
        System.out.println("Call: skippedEntity(), name " + name);
    }

    /** Setzt den DocumentLocator für dieses Dokument */
    public void setDocumentLocator(Locator locator) {
        System.out.println("Call: setDocumentLocator()");
        documentLocator = locator;
    }
}
```

Zunächst benötigen Sie natürlich eine Klasse, die die angegebene Datei (*book.xml*) parst und die SAX-Events an Ihre Klasse weiterreicht, damit diese sie ausgeben kann. Dazu erzeugen Sie zunächst mithilfe der `XMLReaderFactory` und einer Parser-Klasse (Xerces) einen neuen `XMLReader` und registrieren Ihre Klasse als `ContentHandler`. Anschließend müssen Sie lediglich beginnen, das Dokument zu parsen.

Info

Wenn Ihnen die konkrete Implementierung egal ist, können Sie den `XMLReader` auch erzeugen, ohne eine Parser-Klasse anzugeben. In diesem Fall müssen Sie nur sicherstellen, dass sich wenigstens eine Implementierungsklasse in Ihrem Classpath findet.

Listing 11.47
Erzeugen eines Parser-Objekts

```
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;
...
   try {
      // Zu verwendende Parser-Klasse
      String parserClass = "org.apache.xerces.parsers.SAXParser";

      // Erzeugen des XML-Readers
      XMLReader reader =
          XMLReaderFactory.createXMLReader(parserClass);

      // Instanz dieser Klasse erzeugen
      SimpleContentHandler contentHandler =
                              new SimpleContentHandler();

      // Instanz ist Ereignis-Empfänger (Listener) für SAX-Events
      reader.setContentHandler(contentHandler);

      // Parsen des Dokumentes, SAX-Ereignisse werden erzeugt
      reader.parse("xml/book.xml");
   } catch (Exception exc) {
      exc.printStackTrace();
   }
...
```

Da die Callback-Methoden eingangs schon erklärt wurden, sollte der Rest der Klasse kein (Interpretations-)Problem darstellen und wir werden uns auf die folgenden beiden Feinheiten beschränken:

characters()

Die übergebene Zeichenkette ist Teil des Character-Arrays. Sie beginnt beim Index `start` und ist `length` Zeichen lang, wobei `start` nicht immer `0` sein muss. Am besten erzeugen Sie vor der Verarbeitung der Zeichenkette zunächst einen `String`, dessen Konstruktor die Parameter übernimmt.

startElement() und endElement()

Da die Informationen über den Namensraum auch den Methoden `startElement()` und `endElement()` übergeben werden, erhalten diese folgende drei Parameter:

- String `namespaceURI`

  Dieser Parameter enthält die mit dem Präfix verknüpfte URL. Er muss nicht übermittelt werden und kann `null` sein.

- String `localName`

  Hinter diesem Parameter verbirgt sich der tatsächliche Name des Elements, ohne Präfix. Auch dieser Parameter ist laut Spezifikation optional, er wird aber von den meisten Parsern übermittelt.

- String Qualified Name (`qName`)

  Der qualifizierende Name (`qName`) besteht aus der Zeichenkette `Präfix:ElementName`. Existiert kein Namensraum, ist der `qName` gleich dem Attribut `name`. Der Qualified Name ist der einzige Wert, der immer übermittelt wird. Er kann nie `null` sein.

Das Ergebnis

Wenn Sie Ihren `SimpleContentHandler` mit der formatierten Beispieldatei aus diesem Kapitel testen, erhalten Sie das folgende Ergebnis:

Listing 11.48 Ausgabe des SimpleContentHandler

```
Call: setDocumentLocator()
Call: startDocument()
Call: startElement(), start element 'book'
Call: startElement(), start element 'title'
Call: characters(), text 'Masterclass Java EE 5'
Call: endElement(), end element 'title'
Call: startElement(), start element 'publishing-date'
Call: characters(), text '03.07.2006'
Call: endElement(), end element 'publishing-date'
Call: startElement(), start element 'publishing-house'
Call: characters(), text 'Addison-Wesley'
Call: endElement(), end element 'publishing-house'
Call: startElement(), start element 'content'
Call: startElement(), start element 'chapter'
Call: characters(), text 'Kapitel 1'
Call: endElement(), end element 'chapter'
Call: startElement(), start element 'chapter'
Call: characters(), text 'Kapitel 2'
Call: endElement(), end element 'chapter'
Call: endElement(), end element 'content'
Call: endElement(), end element 'book'
Call: endDocument()
```

Vergleichen Sie diese Ausgabe doch einmal mit der in Absatz 11.5.1 beschriebenen »Denkstruktur«.

Tipp

Wie Sie in Listing 11.48 sehen, wird die Methode `setDocumentLocator()` nur ein einziges Mal – vor allen SAX-Events – gerufen.

11.5.4 SAX-Events in der Pipeline

Das reine Traversieren eines XML-Dokuments mittels SAX-Events mag zwar die Arbeitsweise dieser Technologie veranschaulichen, richtig leistungsfähig ist diese Technologie allerdings erst durch die Verwendung von so genannten SAX-Pipelines. Hierbei werden die SAX-Events nicht nur von einem, sondern von einer beliebig großen Anzahl `ContentHandler` verarbeitet.

Arbeitsweise einer SAX-Pipeline

Als Erstes wird das Dokument natürlich weiterhin von einem `XMLReader` bzw. dessen Parser-Implementierung geparst. Der `XMLReader` selbst leitet die XML-Events anschließend an den registrierten Listener, der für deren Verarbeitung verantwortlich ist.

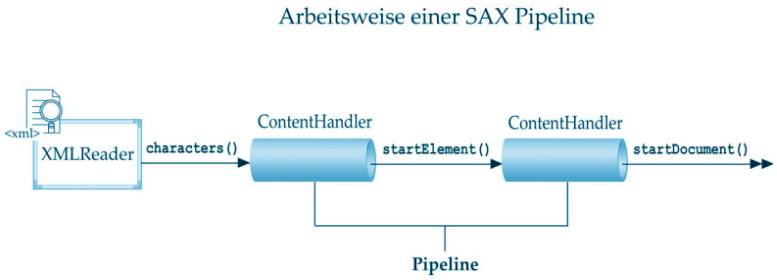
Listing 11.49
Auszug aus Listing 11.46

```
...
   // Erzeugen eines SimpleContentHandler
   SimpleContentHandler simpleContentHandler =
      new SimpleContentHandler();

   // Instanz ist Ereignis-Empfänger (Listener) für SAX-Events
   reader.setContentHandler(contentHandler);
...
```

Im nachfolgenden Beispiel werden Sie den `SimpleContentHandler` zunächst durch einen `ContentHandler` ersetzen, der die Aufgabe hat, den Wert des Attributs `book-number` in ein eigenes Element umzukopieren.

Außerdem implementieren Sie den `PipelineContentHandler` so, dass er alle verarbeiteten SAX-Events ebenfalls an einen weiteren `ContentHandler` weiterleiten kann, wofür wir in diesem Beispiel der Einfachheit halber wieder auf den `SimpleContentHandler` zurückgreifen werden.

Abbildung 11.2
Eine Pipeline aus ContentHandlern (schematisch)



Sie besitzen nun also eine Pipeline aus `ContentHandlern`, wobei der jeweils nachfolgende das Ergebnis seines Vorgängers weiterverarbeitet. Da kein `ContentHandler` weiß, von wem seine Callback-Methoden aufgerufen werden, können Sie so beliebig große Ketten aufbauen, ohne die Klassen selbst anpassen zu müssen.

Ein PipelineContentHandler

Um einen `ContentHandler` in eine solche Pipeline einfügen zu müssen, muss er einerseits alle SAX-Events empfangen können und andererseits über einen Listener verfügen, an den er diese weiterleiten kann. Zwischen diesen beiden Aufgaben können Sie anschließend den eigentlichen Code dieses `ContentHandler` platzieren.

Listing 11.50
Ein PipelineContentHandler

```
package de.akdabas.javaee.sax;
import org.xml.sax.Locator;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.ContentHandler;
import org.xml.sax.helpers.AttributesImpl;

/**
 * Dieser Content-Handler demonstriert die serielle Verarbeitung
 * von SAX-Events in Pipelines indem er das Attribute 'book-number'
 * eines '<book>'-Elementes in ein neues Child-Element kopiert
 */
public class PipelineContentHandler implements ContentHandler {
```

Listing 11.50 (Forts.)
Ein PipelineContent Handler

```
/* Nachfolgender Content-Handler dieser Pipeline */
private ContentHandler listener;

public ContentHandler getListener() {
   return listener;
}

public void setListener(ContentHandler listener) {
   this.listener = listener;
}

/* Methode des Interface 'ContentHandler' */

/** Leitet den SAX-Event an den Listener weiter */
public void startDocument() throws SAXException {
   listener.startDocument();
}

/**
 * - Überprüft ob es sich beim aktuellen Event um das gesuchte
 *   Element '<book>' handelt.
 * - Einfügen eines neuen Kind-Elementes von '<book>'
 */
public void startElement(String namespaceURI, String localName,
                         String qName, Attributes atts)
   throws SAXException {

   // Da das neue Element ggf. ein Child sein soll muss der Event
   // als erstes an den listender weitergeleitet werden
   listener.startElement(namespaceURI, localName, qName, atts);

   // handelt es sich um ein gesuchtes '<book>'-Element
   if (localName.equals("book")) {

      // Auslesen des Attributes 'book-number'
      String value = atts.getValue("book-number");

      // Einfügen einen neuen Elementes '<book-number>', bei
      // diesem sind 'localName' und 'qName' identisch
      listener.startElement
        ("", "book-number", "book-number", new AttributesImpl());

     listener.characters(value.toCharArray(),0, value.length());

      listener.endElement("", "book-number", "book-number");
   }
}

/** Leitet den SAX-Event an den Listener weiter */
public void characters(char[] ch, int start, int length)
   throws SAXException {
   listener.characters(ch, start,  length);
}

/** Leitet den SAX-Event an den Listener weiter */
public void endElement(String namespaceURI, String localName,
                       String qName)
   throws SAXException {
   listener.endElement(namespaceURI, localName, qName);
}

/** Leitet den SAX-Event an den Listener weiter */
public void endDocument() throws SAXException {
   listener.endDocument();
}
```

Listing 11.50 (Forts.)
Ein PipelineContent Handler

```
    /** Leitet den SAX-Event an den Listener weiter */
    public void startPrefixMapping(String prefix, String uri)
        throws SAXException {
        listener.startPrefixMapping(prefix, uri);
    }

    /** Leitet den SAX-Event an den Listener weiter */
    public void endPrefixMapping(String prefix) throws SAXException {
        listener.endPrefixMapping(prefix);
    }

    /** Leitet den SAX-Event an den Listener weiter */
    public void ignorableWhitespace(char[] ch, int start, int length)
        throws SAXException {
        listener.ignorableWhitespace(ch, start, length);
    }

    /** Leitet den SAX-Event an den Listener weiter */
    public void processingInstruction(String target, String data)
        throws SAXException {
        listener.processingInstruction(target, data);
    }

    /** Leitet den SAX-Event an den Listener weiter */
    public void skippedEntity(String name) throws SAXException {
        listener.skippedEntity(name);
    }

    /** Leitet den DocumentLocator an den Listener weiter */
    public void setDocumentLocator(Locator locator) {
        listener.setDocumentLocator(locator);
    }
}
```

Dieser `ContentHandler` verfügt über einen `listener` vom gleichen Typ, an den er alle eingehenden SAX-Events weiterleitet. Würde er dies nicht tun, würden diese einfach ausgefiltert werden.

Info

Bei dem neuen Element `<book-number>` handelt es sich um ein Element ohne speziellen *Namespace*. In diesem Fall ist der *lokale Name* (`localName`) des Elements identisch mit dem *Qualified Name* (`qName`).

Die eigentliche Geschäftslogik ist in der Methode `startElement()` untergebracht. Hier wird zunächst überprüft, ob der soeben weitergeleitete SAX-Event zum gesuchten öffnenden Tag `<book>` gehört und sich die serielle Verarbeitung nun somit innerhalb dieses Elements befindet. In diesem Fall wird der Wert des Attributs `book-number` ausgelesen und ein neues Element erzeugt. Der nachfolgende `ContentHandler` kann nicht entscheiden, ob die SAX-Events dieses neuen Elements zusätzlich hinzugefügt wurden oder nicht.

Ein Client zum Testen

Um die Funktionsweise des `PipelineContentHandler` zu testen, müssen Sie lediglich die Events des `XMLReader` an diesen schicken und den zuvor entworfenen `SimpleContentHandler` als dessen `listener` registrieren.

Listing 11.51
Ein Client zum Test der SAX-Pipeline

```
package de.akdabas.javaee.sax;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;

/**
 * Diese Klasse demonstriert die Funktionsweise von SAX-Pipelines
 * indem Sie die Events des XML-Readers durch den PipelineContent-
 * Hander schickt, der diese verarbeitet und an den SimpleContent-
 * Handler weiterleitet
 */
public class PipelineTest {

   /** Die Methode 'main()' zum Testen der Pipeline */
   public static void main(String[] args) {
      try {
         // Zu verwendende Parser-Klasse
         String parserClass = "org.apache.xerces.parsers.SAXParser";

         // Erzeugen eines XMLReaders
         XMLReader reader =
            XMLReaderFactory.createXMLReader(parserClass);

         // Erzeugen eines SimpleContentHandlers
         SimpleContentHandler simpleContentHandler =
            new SimpleContentHandler();

         // Erzeugen eines PipelineContentHandler
         PipelineContentHandler pipelineContentHandler =
            new PipelineContentHandler();

         // Registrieren des PipelineContentHandler als Empfänger
         // der SAX-Events des XMLReader
         reader.setContentHandler(pipelineContentHandler);

         // Registrieren des SimpleContentHandler als Empfänger
         // der SAX-Events des PipelineContentHandler
         pipelineContentHandler.setListener(simpleContentHandler);

         // Start des Parsens und Beginn der Verarbeitung:
         // Reader -> PipelineContentHandler -> SimpleContentHandler
         reader.parse("xml/book.xml");
      } catch (Exception exc) {
         exc.printStackTrace();
      }
   }
}
```

Das Resultat

Wenn Sie die Klasse aus Listing 11.51 zur Ausführung bringen und die Ausgaben des `SimpleContentHandler` mit denen aus Listing 11.48 vergleichen, machen Sie nun folgende Entdeckung:

```
Call: setDocumentLocator()
Call: startDocument()
Call: startElement(), start element 'book'
Call: startElement(), start element 'book-number'
Call: characters(), text '2184'
Call: endElement(), end element 'book-number'
Call: startElement(), start element 'title'
Call: characters(), text 'Masterclass Java EE 5'
Call: endElement(), end element 'title'
Call: startElement(), start element 'publishing-date'
Call: characters(), text '03.07.2006'
Call: endElement(), end element 'publishing-date'
Call: startElement(), start element 'publishing-house'
Call: characters(), text 'Addison-Wesley'
Call: endElement(), end element 'publishing-house'
Call: startElement(), start element 'content'
Call: startElement(), start element 'chapter'
Call: characters(), text 'Kapitel 1'
Call: endElement(), end element 'chapter'
Call: startElement(), start element 'chapter'
Call: characters(), text 'Kapitel 2'
Call: endElement(), end element 'chapter'
Call: endElement(), end element 'content'
Call: endElement(), end element 'book'
Call: endDocument()
```

Listing 11.52
Ausgabe der SAX-Events, die durch den PipelineContentHandler manipuliert wurden

Wenn Sie nun auf den Geschmack gekommen sind und mittlere bis große Applikationen mit Pipelines planen, schauen Sie sich zuvor auf jeden Fall das *Apache Cocoon Framework* (*http://cocoon.apache.org/*) an, welches diesen Gedanken aufgreift und uns mit einer großen Anzahl von Hilfsklassen, von der Datenbankabfrage bis zur PDF-Erzeugung, unterstützt.

Der große Clou des Framework besteht jedoch darin, dass Sie den Aufbau der Pipelines nicht wie in Listing 11.51 über eine Java-Klasse programmieren, sondern wiederum durch XML definieren und damit auch zur Laufzeit ohne Neukompilieren verändern können.

11.5.5 Einer für alles

Bei den bisherigen SAX-Handlern waren Sie stets gezwungen, alle vom Interface `ContentHandler` deklarierten Methoden selbst zu deklarieren. Um sich doppelten Code zu ersparen, empfehle ich Ihnen bei größeren Projekten mit vielen »pipeline-fähigen« `ContentHandler` die Implementierung einer Basisklasse, die standardmäßig alle Events an einen Listener weiterleitet. Sie müssen bei den tatsächlichen Pipeline-Handlern anschließend nur noch die tatsächlich benötigten Methoden überschreiben.

Wenn Sie hingegen keinen Listener benötigen, können Sie auch auf die Klasse `org.xml.sax.helpers.DefaultHandler` zurückgreifen, welche als Standardverhalten aller Methoden schlicht nichts tut.

11.5.6 Wofür eignet sich die Verarbeitung mit SAX-Events?

Gut, nun können Sie also XML-Dokumente analog zum Parser über SAX-Events verarbeiten. Da Sie das Dokument dabei nur vorwärts durchlaufen können, müssen Sie alle später benötigten Informationen selbst speichern und auch das Einfügen neuer Elemente bereitet deutlich Mehraufwand. Warum sollten Sie also auf SAX zurückgreifen, wenn Ihnen auch ein (J)DOM zur Verfügung steht?

Das in diesem Kapitel verwendete Beispieldokument besteht aus sechs Elementen, die Informationen zu einem Buch speichern. Stellen Sie sich nun einmal vor, Sie wollten die Bücher einer Bibliothek oder Bücherei über ein XML-Dokument verwalten, das ebensolche `<book>`-Elemente enthält. Eine solche Datei kann leicht mehrere hundert Megabyte groß werden, wobei

DOM und JDOM aufgrund der zusätzlichen Referenzen stets etwas mehr Speicherplatz benötigen. Da alle DOM-basierenden Frameworks immer die vollständige XML-Struktur im Speicher abbilden müssen, ist diese Art der XML-Verarbeitung nur für kleinere Dokumente praktikabel.

Auf SAX-Events operierende `ContentHandler` benötigen hingegen unabhängig von der Größe des zu verarbeitenden Dokuments stets die gleichen Ressourcen und arbeiten in der Regel schneller. Die Performance- und Ressourcenvorteile erkaufen Sie sich jedoch nicht ganz ohne Einschränkungen. Die wichtigsten Unterschiede zum DOM-/JDOM-Modell sind:

- Sie können das XML-Dokument nur sequentiell, von oben nach unten, durchlaufen. Ein einmal gesendeter SAX-Event kann nicht mehr rückgängig gemacht werden.
- Beim Hinzufügen von Elementen müssen Sie selbst auf die Einhaltung einer wohlgeformten Struktur achten. Sie können über SAX-Events auch nicht wohlgeformtes XML erstellen, welches beim nächsten Parsen zu Fehlern führen wird. Dies kann Ihnen mit dem (J)DOM-API nicht passieren.

Tipp

Wenn Sie häufig sehr große Datenmengen verarbeiten, in denen viel *ignorable Whitespace* enthalten ist, und auf Einrückungen bei der Speicherung der Dokumente verzichten können, erzielen Sie mitunter einen großen Performance-Gewinn, wenn Sie einen Kompressor-ContentHandler an den Beginn der Pipeline stellen. Dieser leitet alle SAX-Events, mit Ausnahme von `ignorableWhitespace`, an die nachfolgenden `ContentHandler` weiter und reduziert so die erforderlichen Methodenaufrufe.

11.5.7 Zusätzliche Handler

Das *ContentHandler-Interface* stellt Ihnen bereits alle Methoden zur Verfügung, die Sie benötigen, um ein XML-Dokument mit Namensraumunterstützung und Verarbeitungsanweisungen (PIs) zu verarbeiten. Daneben definiert die SAX-2.0-Spezifikation eine Reihe zusätzlicher Interfaces, deren Callback-Methoden Ihnen beispielsweise auch das Verarbeiten einer DTD, die Behandlung von Fehlern und das Auflösen von *Entities* ermöglichen. Hierzu gehören

- `org.xml.sax.DTDHandler`
  Dieses Interface definiert Methoden, die der Parser beim Verarbeiten einer DTD ruft. Sofern Sie keinen eigenen Parser oder Validator schreiben möchten, benötigen Sie diesen in der Regel nicht.
- `org.xml.sax.ext.LexicalHandler`
  Dieser optionale Handler bietet Ihnen erweiterte Möglichkeiten zum Verarbeiten einer internen DTD. Außerdem können Sie über ihn CDATA-Sektionen ermitteln.

- `org.xml.sax.ErrorHandler`
  Wenn Sie bereits ein wenig mit Java und XML herumexperimentiert haben, sind Sie sicher schon über den einen oder anderen Fehler gestolpert, bei deren Auftreten der `XMLReader` seine Arbeit einstellte.

  Durch Registrieren eines eigenen *ErrorHandler* (`setErrorHandler()`) können Sie die Fehlerbehandlung selbst in die Hand nehmen, wobei die Spezifikation zwischen den drei Fehlergraden `warning`, `error` und `fatalError` unterscheidet.
- `EntityResolver`
  Dieses Interface verwendet der XML-Parser, wenn er auf externe Quellen des XML-Dokuments, wie ausgelagerte DTDs oder Entity-Definitionen, zurückgreifen will. Per Default lädt er diese eigenständig nach.

  Über dieses Interface haben Sie auch hier die Möglichkeit, in den Standardprozess einzugreifen und diesen nach Belieben zu steuern.

Die hier beschriebenen zusätzlichen Interfaces erlauben eine sehr anspruchsvolle Verarbeitung von XML-Dokumenten über *SAX-Events* und lassen praktisch keine Wünsche offen. Für den Programmieralltag spielen sie jedoch häufig eine eher untergeordnete Rolle.

11.6 Zusammenfassung

Java und XML scheinen wie füreinander geschaffen: Beide sind plattformunabhängig und während Java für portablen Code steht, liefert XML portable Daten.

Seit der Einführung von XML befindet sich das Programmieren von Software in einem starken Umbruch: Während die klassische Programmierung und Anpassung immer stärker in der Hintergrund rückt, kommen viele neue APIs mit einer umfangreichen XML-basierten Konfigurationsschnittstelle daher, durch die sie sich an nahezu alle Umgebungen und Aufgabenstellungen anpassen lassen.

Bei der Verarbeitung von XML-Strukturen können Sie auf zwei unterschiedliche Techniken zurückgreifen:

- DOM/JDOM/DOM4J

  Bildet die XML-Dokumentstruktur im Speicher ab und gestattet komfortables Navigieren. Benötigt aber gerade bei großen Dokumenten erhebliche Ressourcen.
- SAX-Events

  Die Verarbeitung basiert auf Ereignissen, die beim Parsen des Dokuments registriert werden. Das hält den Speicherverbrauch gering, lässt Sie aber auf der anderen Seite nur von »oben« nach »unten« navigieren.

Als Mittel zur Strukturierung von Daten, um diese anschließend abzulegen oder auszutauschen, ist XML hingegen nicht mehr aus unserer Informationsgesellschaft wegzudenken. Egal, ob Börsennachrichten aufs Handy,

Haltbarkeitsdaten in Supermärkten oder Bonitätsauskünfte bei der Bank: Ohne die Möglichkeit, das Format leicht an neue Anforderungen anzupassen, wären viele alltägliche Dienste nicht oder nur mit erheblichem Aufwand realisierbar. Das nächste Kapitel macht Sie nun abschließend mit einigen wichtigen Anwendungsgebieten für XML, wie die Verwendung von XSL-Stylesheets und die Erzeugung von weiteren Datenformaten wie PostScript oder PDF, vertraut.

12

XSL, XPath und Co.

Das vorangegangene Kapitel beschäftigte sich mit der Einführung in XML und da dies eine Schlüsseltechnologie für viele weitere Anwendungen darstellt, wollen wir es wagen, dieses Buch mit einem Kapitel abzuschließen, das keine spezielle Java-EE-Spezifikation beschreibt, aber wichtige Standards vorstellt, die keinem ernsthaften Java-EE-Entwickler fremd sein sollten: *XPath*, die *eXtensible Stylesheet Language (XSL)* und *Formatting Object (FO)*.

12.1 XPath – eine Einführung

Achtung

Da dieses Kapitel Anwendungen für XML zeigt und damit eine logische Fortsetzung des vorangegangenen Kapitels darstellt, werden wir auch die dort beschriebene XML-Beispielstruktur <book> bzw. eine angepasste Version weiterverwenden.

Was SQL für die Datenbank ist, ist XPath für XML-Dokumente. Unter dem Kürzel XPath spezifiziert das W3C eine Sprache, mit der Sie in der Lage sind, bestimmte (Teil-)Strukturen von XML-Dokumenten mittels Patterns zu beschreiben.

Angenommen, Sie wollten das Erscheinungsdatum eines Buchs aus dem Beispieldokument (<book>) auslesen und würden dazu im einfachsten Fall den Computer anweisen, den Inhalt des dritten Elements unterhalb von <book> auszugeben. Nun legen die meisten datenzentrierten XML-Dokumente keinen Wert auf die Reihenfolge der enthalten Elemente und gänzlich aufgeschmissen sind Sie, wenn Sie im Laufe der Zeit 15 weitere Kindelemente hinzunehmen. *XPath* liefert Ihnen eine einfache und elegante Lösung für dieses Problem:

```
/book/publishing-date
```

Listing 12.1
Ein einfacher XPath-Ausdruck

Wie in einer Ordnerstruktur oder einem Verzeichnisdienst ermöglicht es Ihnen XPath, von einem Element auf ein anderes zu verweisen. Dabei sind die Regeln ähnlich den Verzeichnisregeln unter Unix/Linux:

- »/« bezeichnet die oberste Ebene unseres Dokuments, also den Dokumentknoten an sich. Von diesem können Sie über das Root-Element und seine Kinder einen beliebigen Pfad zum gewünschten Ziel definieren.
- ».« steht für das aktuelle Element.
- »..« bezeichnet das Elternelement des aktuellen Elements, »../..« das Elternelement des Elternelements usw.

Info

XPath war eine der ersten Spezifikationen, die das W3C in Zusammenhang mit XML veröffentlichte. So wurde die erste Version für XPath (XPath 1.0) kaum ein Jahr nach XML veröffentlicht.

Für XPath-Ausdrücke allein existiert noch kein sinnvolles Anwendungsgebiet, so wie *SQL-ResultSet*s ohne eine Anwendung keinen Sinn haben. Erst wenn Sie die Datensätze ausgeben oder weiterverarbeiten, kommt Leben in die Sache. Und so liegt ein Hauptanwendungsgebiet für XPath in der XSL-Transformation, die Sie im nächsten Abschnitt kennen lernen werden. Doch um diese erfolgreich anwenden zu können, werden wir uns zunächst mit dem Auslesen von XML-Elementen beschäftigen.

12.2 Arbeiten mit XPath

Um die Mächtigkeit und Eleganz von XPath-Ausdrücken zu demonstrieren, müssen Sie zunächst noch einmal das XML-Dokument erweitern. Dazu erweitern Sie in diesem Abschnitt die <book>-Struktur und fügen die verschiedenen Elemente zu einer Sammlung (<library>) zusammen.

Listing 12.2
Erweitertes Beispiel eines XML-Dokuments

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<library>
   <book book-number="2184">
      <title>Masterclass Java EE 5</title>
      <author author-number="815">
         <name>Stark</name>
         <first-name>Thomas</first-name>
      </author>
      <publishing-date
         format="dd.MM.yyyy">03.07.2006</publishing-date>
      <publishing-house>Addison-Wesley</publishing-house>
      <price>29.90</price>
   </book>
   <book book-number="6979">
      <title>Jetzt lerne ich J2EE</title>
      <author author-number="815">
         <name>Stark</name>
         <first-name>Thomas</first-name>
      </author>
```

```
      <publishing-date
         format="dd.MM.yyyy">12.10.2005</publishing-date>
      <publishing-house>Markt und Technik</publishing-house>
      <price>24.95</price>
   </book>
   <book book-number="2176">
      <title>Codebook Java EE</title>
      <author author-number="816">
         <name>Samaschke</name>
         <first-name>Karsten</first-name>
      </author>
      <author author-number="815">
         <name>Stark</name>
         <first-name>Thomas</first-name>
      </author>
      <publishing-house>Addison-Wesley</publishing-house>
      <price>99.90</price>
   </book>
</library>
```

Listing 12.2 (Forts.)
Erweitertes Beispiel eines XML-Dokuments

12.2.1 Einfache Ausdrücke

Wie Sie bereits wissen, können Sie über den folgenden Ausdruck alle <book>-Elemente beschreiben.

```
/library/book
```

Listing 12.3
XPath-Ausdruck für <book>-Elemente

Diese Ergebnismenge können Sie nun durch Angabe von bestimmten Eigenschaften eingrenzen. Wenn Sie sich beispielsweise auf Bücher konzentrieren wollen, die weniger als 40 EUR kosten, definieren Sie:

```
/library/book[price<40.00]
```

Listing 12.4
XPath-Ausdruck für Bücher, die weniger als 40 EUR kosten

Und auch innerhalb der eingeschränkten Ergebnismenge können Sie über XPath weiter navigieren. Der Ausdruck:

```
/library/book[price<40.00]/title
```

Listing 12.5
Navigation auf einer bereits eingeschränkten Ergebnismenge

liefert beispielsweise nur die Titel der Bücher zurück, die weniger als 40 EUR kosten.

12.2.2 Suche nach Elementen

Manchmal kennen Sie den genauen Pfad zu den gesuchten Elementen nicht, weil sich diese beispielsweise an verschiedenen Stellen im Dokument befinden können oder von unterschiedlichen Wrapper-Elementen umschlossen werden oder weil Sie einfach eine komfortable Möglichkeit suchen, sich einen langen *XPath* vom Wurzelelement (Root) zu sparen.

Um alle Titel (<title>) der Bibliothek herauszusuchen, können Sie entweder den voll qualifizierenden Pfad

```
/library/book/title
```

Listing 12.6
Beschreibt alle <title>-Elemente in der Beispielstruktur

verwenden oder einfach kurz

Listing 12.7 Andere Variante für Listing 12.6

```
//title
```

schreiben. Dabei steht der doppelte Schrägstrich (//) für einen beliebigen Pfad im XML-Dokument. Diesen erweitern Sie anschließend um die Bedingung `title`, die auf alle `<title>`-Elemente passt, egal an welcher Stelle sie sich befinden.

Eine weitere Möglichkeit, unbekannte Pfade zu definieren, ist der Stern bzw. die Wildcard (*). Dieser steht für ein *einzelnes*, aber beliebiges Element. Während also `//title` alle `<title>`-Elemente im Baum findet, adressiert:

Listing 12.8 Beschreibt alle <title>-Elemente direkt unterhalb von Root

```
/*/title
```

nur die Titel-Elemente, die sich direkt unterhalb des Root-Elements befinden, in diesem Fall (Listing 12.2) also keines. Um alle Titel in der Beispielbibliothek zu finden, könnten Sie aber auch folgenden Pfad verwenden:

Listing 12.9 Weitere, korrekte Möglichkeit zur Beschreibung von <title>

```
/*/*/title
```

12.2.3 Navigation über »Arrays« von Elementen

Natürlich können Sie die Ergebnismenge nicht nur durch die Angabe von bestimmten Eigenschaften eingrenzen, sondern Sie können in dieser auch, wie in einem Feld (Array) von Elementen, navigieren. Um also das zweite `<book>`-Element der Bibliothek zu beschreiben, verwenden Sie

Listing 12.10 Zweites <book>-Element in <library>

```
/library/book[2]
```

Achtung

Bitte beachten Sie, dass die Feldbezeichnungen in XML mit `1` beginnen, statt mit der in Java üblichen `0`.

Und das letzte Element? Da das Array kein Feld namens `length` besitzt, müssen Sie sich in diesem Fall mit der Funktion `last()` auseinander setzen, auf die wir später noch genauer eingehen werden. Das letzte Buchelement beschreiben Sie also mit:

Listing 12.11 XPath-Ausdruck für das letzte <book>

```
/library/book[last()]
```

Und um nur die Bücher herauszusuchen, für die bereits ein Erscheinungstermin (Publishing Date) existiert, schreiben Sie einfach:

Listing 12.12 Passt auf <book>-Elemente mit einem <publishing-date>

```
/library/book[publishing-date]
```

12.2.4 Verwenden des Operators or

Wenn die gesuchten Elemente sich nun an zwei verschiedenen Stellen im Dokument befinden, stehen Sie jetzt noch vor der Wahl: Entweder lassen Sie durch die Verwendung von // *jeden beliebigen Pfad* zu Listing 12.7, was zu Fehlern führen kann, oder Sie splitten die Verarbeitung in zwei Schritte und schreiben alles doppelt.

Da beide Alternativen nicht besonders elegant erscheinen, kommt Ihnen an dieser Stelle der *Oder-Operator* | zu Hilfe, mit dem Sie gleichwertige, alternative Pfade definieren können. Um also alle Namen und Vornamen der Autoren zu selektieren, definieren Sie einfach:

```
/library/book/author/name | /library/book/author/first-name
```

Listing 12.13
Passt auf <name> und <first-name>

Oder auch kurz:

```
//name | //first-name
```

Listing 12.14
Andere Variante für Listing 12.13

12.2.5 Referenzieren von Attributen

Natürlich können Sie auch auf die Attribute und deren Werte zugreifen. Diese kennzeichnen Sie durch ein dem Namen vorangestelltes @. So liefert Ihnen der Ausdruck:

```
/library/book/@book-number
```

Listing 12.15
Liste aller Buchnummern

eine Liste aller Buchnummern der enthaltenen Bücher. Dabei verhalten sich Attribute prinzipiell wie Elemente und können wie diese, z.B. zur Eingrenzung von Mengen, verwendet werden:

```
/library/book[@book-number='2184']
```

Listing 12.16
Passt nur auf ein bestimmtes Buch

Info

Eine Liste aller Attribute des aktuellen Elements erhalten Sie übrigens mit dem XPath-Ausdruck: `./@*`

12.2.6 Komplexe Pfade definieren

Sie haben bereits am Anfang dieses Abschnitts gelernt, dass es neben den globalen, mit einem Schrägstich (/) beginnenden Pfaden auch lokale Pfade gibt. So finden Sie über den XPath

```
author/name
```

Listing 12.17
Ein lokaler Pfad

die Nachnamen aller Autoren des aktuellen Elements (in unserem Fall, mit einem <book>-Element als aktuellem Element, gäbe es natürlich nur einen Autor und folglich auch nur einen Nachnamen). Doch was macht der Parser beim Auswerten eines XPath-Ausdrucks? Er durchläuft den XPath Knoten für Knoten von links nach rechts und überprüft dabei für jedes Element, ob dieses einen entsprechenden Kindknoten besitzt und damit dem XPath weiter genügt. Dann wandert er zu diesem weiter und setzt die Überprüfung fort. Bei einem Knoten kann es sich dabei, wie beim DOM-Modell, um ein Element, ein Attribut oder ein Textelement handeln.

Ist das Ende des XPath erreicht, fügt der Parser das aktuelle Element der Ergebnismenge hinzu und widmet sich dem nächsten möglichen Pfad, bis er den ganzen Baum durchlaufen hat. Und in eben diesen Test von Knoten können Sie auch eingreifen, wobei jeder Test folgende Form hat:

Listing 12.18 Allgemeine Form von XPath-Tests

```
KnotenTyp::TestKnoten[Prädikate]
```

Um also alle <name>-Knoten des aktuellen Elements auszugeben, deren Inhalt der Zeichenkette `Stark` entspricht, verwenden Sie:

Listing 12.19 Knoten mit Element <author>Stark</author>

```
child::author[author='Stark']
```

Übersicht und Anwendung der Knotentypen im XPath

Die folgende Tabelle listet die verschiedenen Knotentypen auf.

Tabelle 12.1 Knotentypen in XPath-Ausdrücken

Knotentyp	Beschreibung
`parent`	Der Elternknoten des aktuellen Elements
`child`	Ein direktes Kindelement
`ancestor`	Zu Deutsch Vorfahr, also alle darüber liegenden Elternelemente bis zur Wurzel (Root)
`self`	Das aktuelle Element
`descendant`	Zu Deutsch Nachfahren, also alle Kindelemente des aktuellen Knotens, deren Kindelemente usw.
`anchestor-or-self`	Verbindung aus `anchestor` und `self`
`descendant-or-self`	Verbindung aus `descendant` und `self`
`following`	Alle Elemente nach dem schließenden Tag des aktuellen Elements
`following-sibling`	Zu Deutsch nachfolgende Geschwister, also alle nachfolgenden Elemente, die das gleiche Elternelement besitzen
`preceding`	Alle Elemente vor dem aktuellen Element
`preceding-sibling`	Alle vorhergehenden Elemente mit gleichem Elternelement
`attribute`	Enthält alle Attribute des aktuellen Elements

Beispiele für die Anwendung der oben beschriebenen Pfadelemente:

Tabelle 12.2 Beispiele für die Anwendung von XPath-Ausdrücken

Beschreibung	Code
Alle <author>-Elemente des aktuellen Elements	`child::author`
Das erste <author>-Element des aktuellen Knotens	`child::author[position()=1]`
Das letzte <author>-Element des aktuellen Knotens	`child::author[position()=last()]`
Alle Elemente des aktuellen Knotens	`child::*`

Beschreibung	Code
Liste der Attribute des aktuellen Elements	`attributes::*`
Liste aller Bücher des aktuellen Elements mit der Buchnummer 2184	`child:book[attribute::book-number=2184]`

Tabelle 12.2 (Forts.) Beispiele für die Anwendung von XPath-Ausdrücken

12.2.7 Operationen und Relationen

Neben dem reinen Selektieren von Elementen mit bestimmten Eigenschaften oder festgelegten Positionen im Dokument ermöglicht XPath auch grundlegende *mathematische Operationen* und *Vergleiche* sowie boolesche Ausdrücke.

Mathematische Operationen

Folgende mathematische Operationen werden von XPath unterstützt.

Beschreibung	Beispiel	Ergebnis
Addition zweier Zahlen	`3 + 4`	`7`
Subtraktion	`6 - 5`	`1`
Multiplikation	`3 * 3`	`9`
Ganzzahlige Division	`8 div 4`	`2`
Rest einer ganzzahligen Division	`7 mod 2`	`1`

Tabelle 12.3 Mathematische Operationen mit XPath

Vergleiche

Tabelle 12.4 zeigt die in der XPath-Spezifikation vorgesehenen Vergleiche:

Beschreibung	Beispiel	Ergebnis
Gleichheit (Equality)	`book-number=2184`	`true` oder `false`
Ungleichheit	`book-number!=2184`	`true` oder `false`
Größer/Kleiner	`price>40.00` oder `price<40.00`	`true` oder `false`
Größer als/Kleiner als	`price>=40.00` oder `price<=40.00`	`true` oder `false`

Tabelle 12.4 Von XPath unterstützte Vergleiche

Boolesche Operatoren

Und schließlich werden die beiden booleschen Operatoren unterstützt.

Tabelle 12.5 Boolesche Operatoren und XPath

Beschreibung	Beispiel	Ergebnis
Und	`name='Stark' and first-name='Thomas'`	`true` oder `false`
Oder	`book-number=6970 or book-number= 2180`	`true` oder `false`

12.2.8 XPath-Funktionen

Ihre große Flexibilität verdanken XPath-Ausdrücke aber vor allem den vordefinierten Funktionen, die Sie schon an der einen oder anderen Stelle ganz intuitiv eingesetzt haben. Diese teilen sich in vier Gruppen und geben je nach Typ einen bestimmten Datentyp zurück, den Sie z.B. mit unseren mathematischen Ausdrücken weiterbearbeiten können.

Operationen auf Knoten

Diese Operationen lassen sich auf alle Elemente anwenden und ermöglichen v.a. die Ermittlung der aktuellen Position.

Tabelle 12.6 Operationen auf Knoten

Name der Funktion	Beschreibung	Syntax
`last()`	Diese Funktion ermittelt die Position des letzten Elements einer Liste von Knoten und entspricht damit der Java-Collection-Methode `size()`.	`number=last()`
`position()`	Gibt die Position des aktuellen Elements in der eben verarbeiteten Liste zurück.	`number=position()`
`count()`	Ermittelt die Anzahl der Knoten in einer *übergebenen* Knotenmenge.	`number=count(node-set)`
`id()`	Gibt das Element mit der übergebenen ID zurück.	`node-set=id()`
`name()`	Gibt den qualifizierenden Namen (`qName`), also inklusive Namensraum, zurück.	`string=name(node)`
`local-name()`	Gibt den lokalen Namen des übergebenen Elements zurück.	`string=local-name(node)`
`namespace-uri()`	Gibt den Namensraum des übermittelten Elements zurück.	`uri=namespace-uri(node)`

Da in der obigen Tabelle kein Platz für Beispiele war, folgen diese auf dem Fuße. Um beispielsweise die Anzahl der Bücher in der Bibliothek zu ermitteln, könnten Sie folgenden XPath-Ausdruck verwenden:

```
count(/library/book)
```

Listing 12.20
Anzahl der <book>-Elemente

Dessen Wert ist natürlich vollkommen identisch zur Position des letzten <book>-Elements.

```
/libary/book/last()
```

Listing 12.21
Anzahl mal anders berechnet

Info

Es gibt übrigens kein `first()`-Pendant zu `last()`, da der Wert dieser Funktion konstant `1` wäre.

Operationen auf Zeichenketten

Neben den reinen Knotenoperationen definiert XPath eine Reihe von Funktionen, mit denen Sie Zeichenketten, die Grundlage praktisch aller textbasierten Formate, manipulieren können. Tabelle 12.7 zeigt die wichtigsten, wobei Ihnen eine gewisse Ähnlichkeit zu den `String`-Operationen in Java nicht entgehen sollte.

Tabelle 12.7
Ausgesuchte Operationen auf Zeichenketten

Name der Funktion	Beschreibung	Syntax
`concat()`	Verknüpfung zweier Zeichenketten	`string=concat (wert1, wert2,...)`
`contains()`	Überprüft, ob die übergebene Zeichenkette im String enthalten ist	`boolean=contains (string, substring)`
`string()`	Entspricht der `toString()`-Methode für Objekte. Diese Methode wandelt Zahltypen, Knoten und boolesche Werte in Zeichenketten um.	`string=string(zahlwert)`
`substring()`	Extrahiert einen Teil der übergebenen Zeichenkette.	`string=substring (string, start, length)`
`translate()`	Eine der interessantesten Funktionen. Sie ersetzt in der Zeichenkette `value` alle in `org` aufgelisteten Zeichen durch die in `replacment` angegebenen Zeichen.	`string=translate (value,org,replacement)`

Natürlich sollen auch hier einige Beispiele die Anwendung der `String`-Operationen verdeutlichen. Um beispielsweise den Titel und den Namen des Autors (getrennt durch Leerzeichen) zu einer gemeinsamen Zeichenkette zusammenzufügen, könnten Sie sich des folgenden Ausdrucks bedienen:

Listing 12.22 Erstellen einer neuen Zeichenkette

```
concat ("/library/book/title/text()", " ", "/library/book/autor/
first-name/text()", " ", "/library/book/author/name/text()")
```

Und folgender Aufruf überprüft, ob die Zeichenkette »Java EE« im obigen Beispiel vorhanden ist.

Listing 12.23 Testet, ob eine dynamisch definierte Zeichenkette enthalten ist

```
contains (concat ("/library/book/title/text()", " ",
                  "/library/book/autor/first-name/text()", " ",
                  "/library/book/author/name/text()"), "Java EE")
```

Das Beste kommt natürlich zum Schluss: Mit der Funktion `translate` sind Sie in der Lage, einzelne Zeichen einer Zeichenkette umzukodieren. Dabei wird jedes Zeichen des Originalpattern durch das neue Zeichen an der gleichen Stelle im `replacement` ersetzt. So können Sie beispielsweise das Datum 12.10.2005 auf folgende Weise umformen:

Listing 12.24 »Übersetzen« einer Zeichenkette

```
translate ('03.07.2006', '0123', 'ABCD')
```

Hierdurch geht das Datum in die Zeichenkette `AD.A7.CAA6` über.

Numerische Operationen

Natürlich kennt XPath auch die wichtigsten mathematischen Operationen, womit auch dynamischen Berechnungen, z.B. von Spaltenbreiten, gelingen. Diese sind:

Tabelle 12.8 Numerische Operationen

Name der Funktion	Beschreibung	Syntax
`number()`	Überführt eine Zeichenkette in eine Zahl.	`number=number('123')`
`sum()`	Mit dieser Funktion können Sie Summen über Knotenelemente bilden.	`number=sum(node-set)`
`round()`	Macht aus einem Float/Double einen Integer-Typ.	`number=round(9.87)`
`floor()`	Schneidet die Nachkommastellen weg.	`number=floor(9.87)`
`ceiling()`	Gibt den nächst größten Integer-Wert zurück.	`number=ceiling(9.87)`

Die letzten drei Operationen kennen Sie bestimmt aus den `java.math`-Packages. So ist das Ergebnis von `round(9.87)` ganz klar `10`, von `floor(9.87)` `9` und `ceiling(9.87)` ergibt schließlich wieder `10`.

Interessant ist hingegen wieder die Summenfunktion, über die Sie beispielsweise den Wert aller vorhandenen Bücher unserer Bibliothek ermitteln können.

```
sum(/library/book/price)
```

Listing 12.25
Gesamtwert der <library>

Boolesche Funktionen

Den Abschluss sollen schließlich die *Booleschen Operationen* bilden, mit denen Sie die oben angegebenen Funktionen zu beliebig komplexen Aussagen zusammenfügen können.

Tabelle 12.9
Boolesche Operationen

Name der Funktion	Beschreibung	Syntax
`boolean()`	Wandelt eine Zeichenkette in einen booleschen Wert um.	`boolean=boolean('true')`
`true()`	Diese Konstantwert-Funktion liefert immer `true` zurück.	`boolean=true()`
`false()`	Das Gegenstück zu `true()`	`boolean=false()`
`not()`	Diese Funktion negiert das boolesche Argument.	`boolean=not(true())`

Um also beispielsweise alle Bücher zu erhalten, deren Titel nicht die Zeichenkette »Java EE« ziert, schreiben Sie:

```
not (contains ("/library/book/author/name/text()", "Java EE"))
```

Listing 12.26
Ein komplexer XPath-Ausdruck

12.3 Zusammenfassung XPath

XPath-Ausdrücke an sich bringen so viel wie ein Buch über Sterne und Sternzeichen bei helllichtem Tage: Sie können viel Theorie erfahren und die beeindruckende Welt dahinter erahnen, aber alle Bücher über Astronomie zusammen können doch nicht den gleichen Eindruck von der Unendlichkeit des Universums vermitteln, den Sie in einer sternklaren Nacht mit einem guten Teleskop erhalten.

Der nächste Abschnitt, in dem Sie XSL-Transformationen kennen lernen, gibt Ihnen hoffentlich einen guten Eindruck davon, wie mächtig XPath-Ausdrücke sind und wie elegant XML-Strukturen durch sie beschrieben werden können. Die folgenden Kerninformationen sollten Sie jedoch mitnehmen:

- XPath-Ausdrücke werden nach einem einfachen Satz aus Regeln aufgebaut, deren Struktur sich an Verzeichnissen orientiert.
- Sie können jede nur denkbare XML-Struktur mit einem XPath-Ausdruck beschreiben und in den enthaltenen Knoten beliebig navigieren.

- Durch die vordefinierten Funktionen können Sie XPath-Ausdrücke um logische Strukturen erweitern und unerwünschte Elemente herausfiltern.

XPath und Namensräume

Listing 12.27
Path und Namensräume

Zum Schluss sei noch eine wichtige Eigenschaft im Umgang mit Namensräumen in XPath-Ausdrücken genannt:

Achtung

Um ein Element über XPath-Ausdrücke zu beschreiben, verwenden Sie immer den *voll qualifizierenden* Elementnamen. Wenn Sie Ihre Bibliothek also mit dem Präfix `mc` ausstatten, verwenden Sie folgenden XPath-Ausdruck für die Titel:

```
/mc:library/mc:book/mc:title
```

... und ist da sonst gar nichts mehr?

Na gut – allen, die tiefer in die Materie einsteigen wollen oder eine Online-Referenz benötigen, seien die Tutorials unter *http://www.w3schools.com/xpath/* und die vom Übersetzer kommentierte deutsche Übersetzung der Spezifikation unter *http://www.edition-w3c.de/TR/xpath* wärmstens ans Herz gelegt.

Und wenn Sie sich kurz testen möchten, wie sicher Sie schon mit XPath-Ausdrücken sind, dann versuchen Sie doch einmal, das folgende Listing zu interpretieren:

Listing 12.28
Ein Test

```
*[not(self::mc:author or self::mc:title)]
```

Info

Ok, ok, ok, zur Überprüfung kommt hier die Auflösung: Es handelt sich um einen *lokalen Pfad* (kein einleitender /), der *alle Kindelemente* des aktuellen Knotens (*) auswählt, wobei `<mc:author>` und `<mc:title>` übersprungen werden./
Meinen herzlichsten Glückwunsch zum Erlernen Ihrer ersten pattern-orientierten Fremdsprache.

12.4 Die eXtensible Stylesheet Language

Die *eXtensible Stylesheet Language (XSL)* dient dazu, abstrakte XML-Dokumente in eine andere Struktur umzuformatieren, häufig um diese für die grafische Darstellung oder die Umwandlung in andere Datenformate aufzubereiten.

Ähnlich wie Sie vielleicht Cascading Stylesheets (CSS) verwenden, um das Aussehen abstrakter HTML-Elemente genau zu formulieren, spezifizieren XSL-Elemente die Struktur von XML-Elementen und werden häufig verwendet, um datenzentrierte Dokumente in dokumentzentrierte (siehe Kapitel 10) zu überführen. So können Sie <book>-Dokumente über XSL beispielsweise in XHTML-Seiten überführen, die dann von einem Browser dargestellt werden können.

Info

Cascading Stylesheets (CSS) ermöglichen die Definition von Regeln oder Stilinformationen für *Markup-Sprachen* wie HTML oder XML. Durch die Trennung von Darstellung(-sstil) und Inhalt können beide unabhängig voneinander weiterentwickelt und der gleiche Stil für unterschiedliche Dokumente definiert werden.

12.4.1 Anwendungen für die eXtensible Stylesheet Language

Die eXtensible Stylesheet Language überführt also strukturierte (XML-) Dokumente in andere strukturierte Dokumente. Dazu definieren Sie mithilfe von XPath-Ausdrücken für jedes XML-Element ein *Template*, anhand dessen dieses überführt werden kann. Dabei sind vor allem folgende Anwendungsszenarien denkbar:

- Datenaustausch.

  Firma A und Firma B speichern ihre Daten in unterschiedlichen XML-Strukturen. Um diese miteinander austauschen zu können, entwickeln beide ein gemeinsames Datenaustauschformat, in das sie ihre Daten vor der Übertragung transformieren. Auf diese Weise bleiben die internen Applikationen unangetastet und können unabhängig voneinander weiterentwickelt werden, solange die Transformation jeweils mit angepasst wird.

- Unterstützung verschiedener Medien und Formate

  Um nicht alle vorhandenen Dokumente für jedes verfügbare Medium und Format doppelt und dreifach erstellen zu müssen, werden diese in einem abstrakten Format abgelegt und erst zur Laufzeit in das entsprechende Format überführt. Auf diese Weise kann ein und dasselbe Dokument sowohl für Browser (XHTML) als auch für mobile Endgeräte (WML) oder zum Drucken (PDF) aufbereitet werden, wobei für Letzteres zusätzlich ein passender Prozessor wie FOP (Abschnitt 12.8) benötigt wird.

  Änderungen müssen dann nur noch an einer einzigen Stelle (im Stylesheet) eingepflegt werden und um ein neues Format zu unterstützen, genügt es fortan, ein weiteres Stylesheet zu erstellen.

12.4.2 Ein einfaches Template

Unter einem *Template* verstehen wir eine Schablone oder ein Muster, das den Aufbau eines Elements beschreibt, jedoch keine Aussagen über den konkreten Inhalt macht. Beispielsweise definieren wir, dass das <book>-Element einen Erscheinungstermin und einen Verlag enthält und diese in einem HTML-Dokument als Liste aufgeführt werden sollen.

Listing 12.29 Ein einfaches XSL-Template

```
<xsl:template match="book">
   <html>
      <body>
         <h3><xsl:value-of select="./title"/></h3>
         <ul>
            <li><xsl:value-of select="./publishing-house"/></li>
            <li><xsl:value-of select="./publishing-date"/></li>
         </ul>
      </body>
   </html>
</xsl:template>
```

Wir verwenden die XPath-Ausdrücke sowohl zum Spezifizieren der Elemente im *Template-Kopf* als auch um innerhalb des *Template* auf Kind- und Elternelemente zuzugreifen. Im Rumpf des <xsl:template>-Elements können Sie dabei eine beliebige wohlgeformte XML-Struktur angeben, in die das <book> überführt werden soll. Die Elemente <xsl:value-of> werden dabei, analog zu *JSP-Ausdrücken*, durch den Wert des angegebenen Elements ersetzt.

Um das Beispieldokument nun mit einem XML-fähigen Browser darzustellen, müssen Sie zunächst eine XSL-Stylesheet-Datei erstellen. Listing 12.30 zeigt eine solche.

Listing 12.30 Einfügen unseres Template in eine XSL-Stylesheet-Datei (simpleStyle.xsl)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

   <xsl:template match="book">
      <html>
         <body>
            <h3><xsl:value-of select="./title"/></h3>
            <ul>
              <li><xsl:value-of select="./publishing-house"/></li>
              <li><xsl:value-of select="./publishing-date"/></li>
            </ul>
         </body>
      </html>
   </xsl:template>

</xsl:stylesheet>
```

Da es sich bei XSL-Dateien auch um XML handelt, beginnt natürlich auch das Stylesheet mit einem entsprechenden Prolog (siehe Kapitel 11).

Listing 12.31 XML-Prolog des Stylesheet

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

Anschließend folgt das Wurzelelement (<xsl:stylesheet>), in dessen Rumpf Sie alle Template-Definitionen integrieren.

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

   <!-- ... Hier stehen die Template Definitionen ... -->

</xsl:stylesheet>
```

Wichtig für XSL-Stylesheets ist dabei die Angabe der unterstützten XSL-Version. Dies ist nicht zu verwechseln mit der XML-Spezifikation, die ebenfalls im Augenblick die Nummer 1.0 trägt, und der Definition des Namensraums `xls`.

Info

Die derzeit aktuelle XSL-Version ist immer noch die Spezifikation 1.0 vom 16. November 1999. Die angedachte Erweiterung 1.1 ist nie über den Stand eines Arbeitsdokuments hinausgekommen und wird von kaum einem Anbieter unterstützt. Derzeit entwickelt die Arbeitsgruppe des W3C die Version 2.0, deren dritte Candidate Recommendation vom 03. November 2005 unter *www.w3.org/TR/xslt20/* zu finden ist.
Version 2.0 wird zusammen mit der ebenfalls in Bearbeitung befindlichen *XPath 2.0*-Spezifikation vor allem einige neue Navigationsmethoden und Verbesserungen beim Iterieren über Mengen bringen sowie *benutzerdefinierte Funktionen* ermöglichen. Diese Features sollten Sie sich, zu gegebenem Zeitpunkt, nach der Lektüre dieses Kapitels jedoch mit den angegebenen Online-Tutorials selbst erarbeiten können.

Jetzt müssen Sie noch im Beispieldokument (*book.xml*) auf das zugehörige Stylesheet verweisen. Dazu betten Sie einfach die folgende Verarbeitungsanweisung *(PI)* ein.

Listing 12.32
Verweis auf das Stylesheet in der Datei book.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<?xml-stylesheet type="text/xsl"
                 href="../xsl/simpleStyle.xsl" ?>

<book book-number="2184">
  <title>Masterclass Java EE 5</title>
  <author author-number="815">
    <name>Stark</name>
    <first-name>Thomas</first-name>
  </author>
  <publishing-date format="dd.MM.yyyy">03.07.2006</publishing-date>
  <publishing-house>Addison-Wesley</publishing-house>
</book>
```

Dabei geben Sie entweder den globalen oder den relativen URL des Stylesheet (`href`) sowie den MIME-Type (`text/xls`) an und öffnen anschließend die Datei *book.xml* mit einem XML-fähigen Browser Ihrer Wahl. Das Ergebnis kann sich sehen lassen.

Abbildung 12.1
Darstellung von book.xml mithilfe des XSL-Template

12.5 eXtensible Stylesheet Language Transformation (XSLT)

Achtung

Viele Quellen verwenden die Begriffe *XSL* und *XSLT* synonym. Lassen Sie sich von den verwirrenden Abkürzungen also nicht ins Bockshorn jagen. Wichtig ist lediglich zu wissen, dass sich hinter XSL die *Definition einer Umwandlung* verbirgt, deren Umsetzung als Transformation bezeichnet wird.

Obwohl viele Browser, wie *Microsofts Internet Explorer (IE)*, inzwischen in der Lage sind, XML-Dokumente mithilfe einer angegebenen XSL-Datei selbstständig zu transformieren, wird von dieser Technik eher selten Gebrauch gemacht, da hierfür sowohl das Dokument selbst als auch das Stylesheet übertragen werden müssen und nicht alle Systeme die Transformation vollständig unterstützen.

Stattdessen werden die Dokumente in der Regel häufig *vor* der Auslieferung durch den Server selbst in die vom Client erwartete Struktur gebracht. Diese Umwandlung von einer XML-Struktur in eine andere wird als *eXtensible Stylesheet Language Transformation (XSLT)* bezeichnet.

12.5.1 Die Qual der Wahl der Implementierung

Und genauso, wie es viele Parser-Implementierungen gibt, die alle mehr oder weniger standardkonform XML-Dokumente parsen und anschließend in eine digitale Form (DOM oder SAX) überführen, existieren auch verschiedene *XSL-Prozessoren*, um XML-Dokumente von einer Form in eine andere zu überführen.

Info

Als XSL-Prozessor beschreibt man eine Anwendung, die in der Lage ist, ein XML-Dokument über ein XSL-Stylesheet in ein anderes XML-Dokument zu überführen.

- Apache Xalan (*http://xml.apache.org/xalan-j/)*

  Apaches Referenzimplementierung des XSL-Standards wird wegen ihrer großen Standardgenauigkeit häufig zu Testzwecken herangezogen.
- Saxon (*http://saxon.sourceforge.net/*)

  Wieder ein sehr erfolgreiches SourceForge-Projekt. Autor Michael Kay war an der Entwicklung des XSLT-Standards beteiligt und wohl nicht nur deshalb gilt Saxon als einer der schnellsten XSLT-Prozessoren auf dem Markt.
- Microsofts MSXML (*http://msdn.microsoft.com/*)

  Auch in Microsofts XML-Paket darf ein XSLT-Prozessor natürlich nicht fehlen. Die Installation integriert (Microsoft-typisch) die XML-Unterstützung in verschiedene Produkte. Dafür muss man auf die Unterstützung neuerer Funktionen in der Regel länger warten.
- Oracles XDK (*http://www.oracle.com/technology/tech/xml*)

  Hinter dem Kürzel XDK verbirgt sich die Bezeichnung *XML Developer Kit*, welches vom Parser bis zum XSLT-Prozessor ebenfalls alles beinhaltet. Leider ist das XDK der langsamste der hier aufgeführten Vertreter.

Info

Michael Kay ist, trotzdem er eine *One-Man-Show* darstellt, in der Regel der erste, der neue Funktionen in seinen *Saxon* integriert. Wenn Sie also mit verschiedenen Features der Spezifikation 2.0 arbeiten wollen, legen ich Ihnen diesen Prozessor ans Herz.

Einige Anbieter versuchen, die Transformation so schnell wie irgend möglich zu vollziehen, andere wiederum legen den Fokus darauf, eine breite Palette an Erweiterungen anzubieten. Platzhirsch ist dabei *Apache Xalan*.

Diese Referenzimplementierung gehört zu den schnellsten, standardkonformen Transformatoren und lässt auch bei Erweiterungen (fast) keine Wünsche offen. Er steht wie alle Apache-Projekte unter der *Apache Software License (ASL)* und steht, unter Angabe der Referenzen, sowohl für den privaten wie auch den kommerziellen Gebrauch kostenlos zur Verfügung. Einmal von der oben angegebenen Seite heruntergeladen, genügt es, das Archiv in ein beliebiges Verzeichnis zu entpacken und die enthaltenen JAR-Files in den *Classpath* einzubinden.

12.5.2 Transformation über die Kommandozeile

Bevor Sie die Möglichkeiten von XSL-Stylesheets ausloten können, müssen Sie natürlich zunächst wissen, wie Sie diese auf eine XML-Datei anwenden. Der einfachste Weg, mit XSL-Stylesheets herumzuexperimentieren, ist sicherlich der Aufruf per Kommandozeile. Auch wenn dies in der späteren Praxis so gut wie keine Rolle mehr spielt, können Sie so auf einfache Art und Weise neue Templates ausprobieren, ohne diese erst in eine Applikation einbetten zu müssen.

Kommandozeilenbasierte Transformationen übernimmt beim *Apache Xalan* die Klasse `Process`, die sich über verschiedene Parameter steuern lässt. Der Aufruf, um die Datei *book.xml* mit dem oben angegebenen Stylesheet (*simpleStyle.xsl*) zu transformieren, lautet entsprechend:

Listing 12.33 Transformation per Kommandozeile

```
java -cp lib\xalan.jar;lib\serializer.jar;
   org.apache.xalan.xslt.Process -IN xml\book.xml
   -XSL xsl\simpleStyle.xsl -OUT result\result.html
```

Weitere Informationen und Optionen für die Verwendung des *Apache Xalan* aus der Kommandozeile finden Sie auf der Homepage des Projekts.

12.5.3 Transformation eines DOM

Sun beschreibt die Transformation von XML-Dokumenten im *Transformation API for XML (TrAX)*. Ähnlich wie das Parsen von Dokumenten durch Basisklassen des Package `javax.xml.parsers` beschrieben wird, finden Sie im Package `javax.xml.transform` Klassen, die eine XSL-Transformation von Dokumenten erlauben.

Für die Transformation benötigen Sie zunächst folgende fünf Objekte:

- `javax.xml.transform.TransformerFactory`
  Diese Basisklasse durchsucht, wie die `ParserFactory`, den *Classpath* nach einer geeigneten Implementierung und erzeugt anschließend eine Instanz dieser.
- `javax.xml.transform.Transformer`
  Diese konkrete Implementierung eines `Transformer`-Objekts wird von der jeweiligen `Factory` erzeugt und übernimmt die Umwandlung.
- `javax.xml.transform.Source`
  Für das Ausgangsdokument definiert das Package das Interface `Source`, welches von verschiedenen Quelltypen (`InputStream`, `File` etc.) implementiert wird.
- `javax.xml.transform.Result`
  Dieses Interface repräsentiert das Zieldokument, also das Resultat der Umwandlung. Ebenso wie für die `Source`-Schnittstelle existieren verschiedene `Result`-Implementierungen für unterschiedliche Szenarien.
- `Stylesheet`
  Schließlich benötigen Sie ein Stylesheet. Da es sich bei diesen jedoch ebenso um XML handelt wie bei den zu transformierenden Dokumenten, können Sie auch hier die `Source`-Schnittstelle verwenden.

Listing 12.34
Transformieren eines XML-Dokuments (SimpleTransformer.java)

```
package de.akdabas.javaee.xsl;

import java.io.File;
import javax.xml.transform.Source;
import javax.xml.transform.Result;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.TransformerException;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.stream.StreamResult;

/** Demonstriert eine javabasierte XSL-Transformation */
public class SimpleTransformer {

   /** Demonstriert die Funktionsweise des SimpleTransformer */
   public static void main(String[] args) {
      SimpleTransformer xslTransformer = new SimpleTransformer();
      xslTransformer.transform();
   }

   /** Transformiert ein XML-Dokument anhand eines Stylesheets */
   public void transform() {
     try {
         // Referenziert eine Transformer-Factory
         TransformerFactory tFactory =
            TransformerFactory.newInstance();

         // Erzeugen eines Stylesheet-Objekt
         Source stylesheet =
            new StreamSource(new File("xsl/simpleStyle.xsl"));

         // Erzeugen eines XSL Transformers
         Transformer transformer =
            tFactory.newTransformer(stylesheet);

         // Über die Source wird das Dokument eingelesen
         Source inputDoc =
            new StreamSource(new File("xml/book.xml"));

         // Über einen Result können Sie da Dokument ausgeben
         Result outputDoc =
            new StreamResult(new File("result/book.html"));

         // Start der Transformation
         transformer.transform(inputDoc, outputDoc);
      } catch (TransformerException tex) {
         tex.printStackTrace();
      }
   }
}
```

Zunächst instanziieren Sie eine geeignete `TransformerFactory` aus den vorhandenen Bibliotheken unseres Classpath. Anschließend erzeugen Sie eine neue `InputSource` für das Dokument und erzeugen mit dieser einen spezifischen `Transformer`. Dann erzeugen Sie auf die gleiche Weise eine `Source` für das Ausgangs- und ein `Result` für das Zieldokument und transformieren beide ineinander. Effektiv sechs Zeilen Javacode, mehr benötigen Sie nicht.

Info

Analog zum XML-Parser aus dem vorangegangenen Kapitel verwenden Sie bei dieser Art des Referenzierens einer `TransformerFactory` die erste im Classpath gefundene Implementierung. Sie können hier auch eine spezielle Implementierung wie beispielsweise die `org.apache.xalan.xsltc.trax.TransformerFactoryImpl` angeben.

12.5.4 Verschiedene Transformationsquellen und Ziele

In Listing 12.34 verwenden Sie `StreamSourcen` und `StreamResults`, um die XML-Dateien zu lesen und zu schreiben. Insgesamt stehen Ihnen dafür beim *Apache Xalan* vier Implementierungen zur Verfügung.

- `javax.xml.transform.stream.StreamSource` / `javax.xml.transform.stream.StreamResult`

  Diese Datenquelle dient dazu, auf `Reader`, `Writer` und `Streams` zuzugreifen und Dateien von diesen zu lesen bzw. dorthin zu schreiben.

- `javax.xml.transform.dom.DOMSource` / `javax.xml.transform.dom.DOMResult`

  Diese Datenquellen dienen, wie der Name schon sagt, dazu, *DOM-Bäume* im Sinne des W3C zu transformieren.

- `javax.xml.transform.sax.SAXSource` / `javax.xml.transform.sax.SAXResult`

  Bei diesen Datenquellen haben Sie es wieder mit SAX-Events, also im Wesentlichen `XMLReader` und `ContentHandler`, zu tun.

- `org.apache.xalan.xsltc.trax.XSLTCSource`

  Bei dieser Datenquelle handelt es sich um eine Besonderheit des *Xalan*, bei der eine der anderen `InputSourcen` über einen speziellen Compiler weiterverarbeitet wird. Dieser übersetzt die textbasierte Quelle (unser Stylesheet) in ein binäres Format, genannt Translet, um es anschließend mit einem speziellen Prozessor ausführen zu können. Hierdurch kann gerade beim Transformieren großer Quelldateien ein erheblicher Performancegewinn erreicht werden.

Tipp

Vielleicht liebäugeln Sie in diesem Moment auch mit der Aussicht, *SAX-* und *DOM-Model* mit einem »leeren« Stylesheet, über einen `Transformer` beliebig ineinander umwandeln zu können. Dies funktioniert in der Tat sehr gut und es kommt sogar noch besser: Die `org.apache.xalan.transformer.TransformerIdentityImpl` hat genau diese Aufgabe. Sie implementiert praktisch wichtige *SAX-Methoden*, mit Ausnahme des `ErrorHandler` und leitet die empfangenen `Events` einfach an den Result weiter.

Natürlich sind beim Verarbeiten und Transformieren von Objekten die unterschiedlichsten Kombinationen möglich, so können Sie auch eine `SAXSource` mit einem lokalen *Stylesheet* (`StreamSource`) zu einem DOM (`DOMResult`) verarbeiten oder eine `XSLTCSource` dazu verwenden, um reine SAX-Events zu verarbeiten – keine Kombination, die nicht möglich wäre.

12.5.5 SAX vs. DOM

Vielleicht ist es Ihnen ja aufgefallen, aber in Listing 12.34 haben Sie sich keine Gedanken darüber gemacht, auf welche Art und Weise unser `Transformer` arbeitet. Egal, ob er nun zunächst einen DOM aufbaut oder SAX-Events direkt transformieren kann, Sie haben keine Möglichkeit, darauf Einfluss zu nehmen – obwohl die meisten Implementierungen natürlich SAX-Events verwenden.

Nun gibt es neben der direkten Instanziierung einer bestimmten Klasse auch die Möglichkeit herauszufinden, ob es sich bei der augenblicklichen Implementierung um ein `TransformerFactory` oder eine `SAXTransformerFactory` handelt. Über die Methode `getFeature()` können Sie die verschiedenen Eigenschaften der gefundenen Implementierung ermitteln und gegebenenfalls einen sicheren Cast vornehmen. Listing 12.35 zeigt einen entsprechenden Code-Ausschnitt.

Listing 12.35
Ermitteln von Eigenschaften unserer TransformerFactory

```
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.sax.SAXTransformerFactory;
...
   //Referenzieren einer (bleibigen) TransformerFactory im Classpath
   TransformerFactory tFactory = TransformerFactory.newInstance();

   // Überprüfen, ob die SAX-Events unterstützt
   if (tFactory.getFeature(SAXTransformerFactory.FEATURE)) {

      SAXTransformerFactory saxFac =
         (SAXTransformerFactory) tFactory;

      // ... ab hier können Sie mit der SAXTransformerFactory
      //     weiterarbeiten ...
   }
...
```

12.5.6 Laden eines referenzierten Stylesheet

In Listing 12.32 wurde das Stylesheet über eine `<xml-stylesheet>`-*Processing Instruction (PI)* referenziert und auch für diese Fälle, in denen Sie gewöhnlich über keine direkte URL für die XSL-Datei verfügen, steht Ihnen das TrAX-API zur Seite, indem es Ihnen über die Methode `getAssociatedStylesheets()` ermöglicht, auch diese Quellen zu erschließen.

Listing 12.36
Laden eines referenzierten Stylesheet

```
...
   TransformerFactory tFactory = TransformerFactory.newInstance();
   Source inputDoc = new StreamSource(new File("xml/book.xml"));
   Source stylesheet = tFactory.getAssociatedStylesheet
                              (inputDoc, null, null, null);
...
```

Über die zusätzlichen Parameter (hier alle `null`), können Sie zusätzlich bestimmte Eigenschaften wie ein spezielles Zielmedium oder einen bestimmten Zeichensatz (`encoding`) angeben.

Tipp

Enthält das XML-Dokument mehrere *Stylesheet-PIs*, auf die die angegebenen Parameter zutreffen, so werden diese intern zu einem gemeinsamen Stylesheet zusammengefügt und als eine `Soure` zurückgegeben.

12.5.7 Template vs. Transformer

Bisher haben Sie eine `TransformerFactory` dazu verwendet, zusammen mit einer Stylesheet-Source einen `Transformer` zu erzeugen, der anschließend die Umwandlung (`transform`) vom Ausgangs- ins Zieldokument übernahm.

Dieses Vorgehen ist effizient für die einmalige oder nicht parallele Transformierung von XML-Dokumenten, da ein so erzeugter `Transformer` zwar für verschiedene Dokumente wiederverwendet werden kann, jedoch nicht *threadsafe* ist und sich damit nicht für die parallele Verarbeitung von `Sourcen` eignet. Eine mögliche Lösung ist das Erzeugen eines `Template`-Objekts, welches das Stylesheet ähnlich einer `XSLTCSource` vorkompiliert und uns anschließend beliebig viele `Transformer`-Objekte zur Verfügung stellt.

Info

Als *threadsafe* bezeichnet man Objekte oder Methoden, die auch dann sicher arbeiten, wenn sie von verschiedenen Threads (gleichzeitig) verwendet werden können. Ist eine Methode nicht threadsafe, muss in Multithread-Umgebungen der Zugriff auf diese synchronisiert werden.

Listing 12.37 Erzeugung von Transformern über threadsichere Templates

`Templates`-Transformer eignen sich für parallele Transformationen mit unterschiedlichen Threads.

```
package de.akdabas.javaee.xsl;
import java.io.File;
import javax.xml.transform.*;
import javax.xml.transform.stream.*;

/**
 * Diese Klasse demonstriert das Kompilieren eines XSL-Stylesheets
 * in eine In-Memory-Struktur (Templates)
 */
public class SimpleTemplates {

   /** Transformiert das '<book>' über XSL-Templates */
   public void transform() {
      try {
         // Referenziert eine Transformer-Factory
         TransformerFactory tFactory =
            TransformerFactory.newInstance();

         // Erzeugen eines Stylesheet-Objekt
         Source stylesheet =
            new StreamSource(new File("xsl/simpleStyle.xsl"));
```

```
            // Erzeuen eines Templates-Objektes durch Kompilieren des
            // Stylesheets in eine In-Memory-Struktur
            Templates templates = tFactory.newTemplates(stylesheet);

            // Erzeugen eines Transformers aus dem Templates-Objekt
            Transformer transformer = templates.newTransformer();

            // Über die Source wird das Dokument eingelesen
            Source inputDoc =
               new StreamSource(new File("xml/book.xml"));

            // Über einen Result können Sie da Dokument ausgeben
            Result outputDoc =
               new StreamResult(new File("result/book.html"));

            // Start der Transformation
            transformer.transform(inputDoc, outputDoc);
         } catch (TransformerException tex) {
            tex.printStackTrace();
         }
      }
   }
```

Listing 12.37 (Forts.)
Erzeugung von Transformern über threadsichere Templates

12.5.8 Transformieren eines JDOM

Nun haben wir uns im vorherigen Kapitel die ganze Zeit auf das JDOM-API eingeschworen und jetzt transformieren wir doch externe XML-Dokumente? Keineswegs, denn natürlich können Sie mit diesem API auch XSL-Transformationen durchführen. Praktischerweise kapselt die Klasse `XSLTransformer` nahezu die gesamte Funktionalität, wie es das folgende Listing-Fragment zeigt:

```
import org.jdom.Document;
import org.jdom.transform.XSLTransformer;
...
   Document input ...       // Einlesen oder Aufbau des Dokumentes
   Document stylesheet ...  // Einlesen oder Aufbau des Stylesheets

   // Erzeugen des Transformer-Objektes für das Stylesheet
   XSLTransformer transformer = new XSLTransformer(stylesheet);

   // Transformation des Dokumentes
   Document result = transformer.transform(input);
...
```

Listing 12.38
XSL-Transformation eines JDOM

Natürlich können Sie beim Transformieren eines JDOM oder für das Laden des *Stylesheet* auch auf `Files`, `Streams` und `Reader` zurückgreifen, dennoch sollten Sie auch die faszinierenden Möglichkeiten einer Transformationen über einen weiteren JDOM ins Auge fassen. Schließlich haben Sie hierdurch nicht nur die Möglichkeit, das Stylesheet vor der Transformation zu manipulieren, Sie können es sogar äußerst komfortabel zur Laufzeit erzeugen.

12.5.9 Zusammenfassung XSL-Transformationen

Durch XSLT-Prozessoren können Sie XML-Dokumente anhand von *Stylesheets* dauerhaft in neue Dokumente überführen, um diese beispielsweise anschließend an einen Browser weiterleiten zu können. Der Vorteil dieser Verfahrensweise ist, dass Sie das Stylesheet nicht bei jeder Übertragung mit übermitteln müssen, sondern stattdessen sogar in digitaler Form *cachen* können.

12.6 XSL Stylesheets

Wie Sie bereits wissen, ist ein XSL-Stylesheet ein spezielles XML-Dokument mit dem Root-Element `<xsl:stylesheet>`.

Listing 12.39 Root-Element eines Stylesheet

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

   <!-- ... hier stehen die Elemente des Stylesheets ... -->

</xsl:stylesheet>
```

In dieses umschließende Element können Sie verschiedene *Top-Level-Stylesheet-Elemente* integrieren, von denen Sie die wichtigsten in diesem Abschnitt kennen lernen werden. Beim Verarbeiten des Stylesheet werden die Elemente von oben nach unten durchsucht, wodurch oben stehende Elemente die höhere Priorität besitzen.

12.6.1 Ein einfaches Template

Templates (dt. Muster) bilden den Kern des Stylesheet. Sie enthalten eine abstrakte Beschreibung, wie die konkreten Elemente unseres Ausgangsdokuments in Elemente des Zieldokuments überführt werden sollen.

Listing 12.40 Ein einfaches Template

```
<!--
    Ein Beispiel-Template zur Verarbeitung des XML-Knotens 'author'
-->
<xsl:template match="author">
   <xsl:value-of select="first-name"/> <xsl:value-of select="name"/>
</xsl:template>
```

12.6.2 Importieren anderer Stylesheets

Ähnlich wie Sie Teile von JSPs in eigene Dateien auslagern, um diese anschließend in verschiedenen Seiten gemeinsam zu nutzen, können Sie einen Teil der Definitionen unseres Stylesheet auslagern, um es anschließend zu importieren.

Listing 12.41 Importieren eines externen Stylesheet

```
<!-- Import eines externen Stylesheets -->
<xsl:import href="part.xsl" />
```

Achtung

Die Reihenfolge aller anderen Top-Level-Elemente ist egal, doch `<import>`-Anweisungen müssen vor allen anderen, ganz oben im *Stylesheet* stehen. Des Weiteren darf sich ein XSL-Stylesheet natürlich nicht selbst importieren.

Das Attribut `href` des leeren `<import>`-Elements gibt dabei den URL des zu importierenden Stylesheet an. Dieser kann global oder relativ zum aktuellen Dokument sein.

12.6.3 Einfügen eines Stylesheet

Dieser Punkt macht besonders Anfängern zu schaffen, denn neben der Möglichkeit, ein anderes *Template* zu importieren, haben Sie über das Element:

```
<!-- Einfügen eines Stylesheets -->
<xsl:include href="part.xsl">
```

Listing 12.42
Einfügen externer Stylesheets

die Möglichkeit, dieses direkt einzufügen. Der Unterschied zwischen beiden Verfahren besteht darin, dass ein zu importierendes Stylesheet gewissermaßen unten an unsere XSL-Datei angehängt wird, wodurch alle Elemente automatisch eine niedrigere Priorität erhalten, während das `<include>`-Element direkt durch den Inhalt des referenzierten Stylesheet ersetzt wird.

Tipp

Wenn Sie nicht explizit auf die Verwendung von `<import>`-Anweisungen angewiesen sind, verwenden Sie stattdessen `<include>`.

12.6.4 Attribut-Sets

Durch Attribut-Sets können Sie bestimmte, immer wieder verwendete Attribute zusammenfassen und über einen gemeinsamen Namen verwalten.

```
<!-- Definition eines Attribut-Sätze -->
<xsl:attribute-set name="font-style">
   <xsl:attribute name="face">arial</xsl:attribute>
   <xsl:attribute name="color">green</xsl:attribute>
</xsl:attribute-set>

<!-- Verwenden des zuvor definierten Attribut-Satzes -->
<xsl:template match="author">
   <font xsl:use-attributes-sets="font-style">
      <xsl:value-of select="first-name"/>
      <xsl:value-of select="name"/>
   </font>
</xsl:template>
```

Listing 12.43
Verwendung von Attribut-Sets

Hierbei werden die im Attribut-Set definierten Attribute in das Zieldokument eingefügt, so dass anschließend folgende Struktur entsteht.

```
...
   <font face="arial" color="green">Thomas Stark</font>
...
```

Listing 12.44
Resultat des Attribut-Set

12.6.5 Variablen

Variablen ermöglichen es Ihnen, Platzhalter für verschiedene Elemente zu deklarieren. Um die Standardfarbe unseres Dokuments beispielsweise beliebig austauschen zu können, können Sie diese in einer Variablen speichern und dann in allen Attribut-Sets und Templates darauf zurückgreifen.

Listing 12.45
Verwendung von Variablen

```
<!-- Definition einer Variablen -->
<xsl:variable name="defaultColor">green</xsl:variable>

<!-- Verwenden der zuvor definierten Variablen -->
<xsl:attribute-set name="font-style">
   <xsl:attribute name="face">arial</xsl:attribute>
   <xsl:attribute name="color">
      <xsl:value-of select="$defaultColor" />
   </xsl:attribute>
</xsl:attribute-set>
```

Um innerhalb eines Elements oder Attribut-Set auf eine bestimmte Variable zu verweisen, stellen Sie dem Variablennamen ein Dollar-Zeichen ($) voran. Und es kommt sogar noch besser, denn schließlich können Sie Variablen auch zur Definition komplexer (XML-)Strukturen verwenden, wie das folgende Listing zeigt.

Listing 12.46
Verwendung von Variablen zur Definition fester Strukturen

```
<!-- Definition einer komplexen Variablen -->
<xsl:variable name="table-header">
   <tr>
      <th>Vorname</th>
      <th>Name</th>

   </tr>
</xsl:variable>

<!-- Einfügen der Variablen in das resultierende Dokument -->
<xsl:template name="list-authors">
   <table>
      <xsl:copy-of select="$table-header"/>

      <!-- ... Inhalt der Tabelle ... -->

   </table>
</xsl:template>
```

Achtung

Eine bessere Beschreibung für XSL-Variablen wäre XSL-Konstanten: Nachdem Sie einer Variablen einmal einen Wert zugewiesen haben, können Sie diesen nicht mehr überschreiben. XSL-Variablen dienen demzufolge nur als Platzhalter für komplexe Strukturen.

12.6.6 Weiterführende Literatur

Natürlich kennt ein XLS-Stylesheet noch eine ganze Reihe weiterer Top-Level-Elemente, doch deren ermüdende Aufzählung überlassen wir an dieser Stelle der Spezifikation, die Sie unter *http://www.w3.org/TR/xslt* finden.

Tipp

Auch die XSL-Spezifikation ist von editionW3C inzwischen ins Deutsche übersetzt worden. Sie enthält ebenfalls zahlreiche Beispiele und ist unter http://www.edition-w3c.de/TR/xslt erreichbar.

Zu diesem Zeitpunkt kennen Sie alle Elemente, die Sie für die kommenden Beispiele – und auch die meisten anderen Fälle – benötigen. Und falls Sie nun auf den Geschmack gekommen sind, bietet Ihnen W3Schools auch für diese Spezifikation ein umfassendes Tutorial unter *http://www.w.3schools.com/xsl/* an.

12.7 Ein Stylesheet für die Bibliothek

Nun wollen wir endlich zur Tat schreiten und ein komplexeres Beispiel erstellen. Weil sich die Bücherbibliothek (`<library>`) aus dem letzten Abschnitt so gut bewährt hat, wollen wir sie hier gleich weiterverwenden. Dabei werden Sie dieses Mal nicht wie im vorherigen Beispiel ein einziges globales *Template* definieren, sondern das Dokument von oben nach unten durchlaufen und die verschiedenen Teilstrukturen einzeln transformieren.

12.7.1 Das Root-Element

Beginnen wir bei der Dokumentwurzel (/). Diese dient als Einstiegspunkt in das Dokument und hier beginnen alle Transformationen.

Listing 12.47
Transformation des Dokumentknotens

```
...
   <!-- Transformation der Dokumentwurzel -->
   <xsl:template match="/">

      <!-- Einfügen von umschließenden Elementen -->
      <html>
         <body>
            <!-- Fortsetzen der Transformation für untergeordnete
                 Elemente -->
            <xsl:apply-templates />
         </body>
      </html>
   </xsl:template>
...
```

Im Dokumentkopf können Sie Elemente definieren, die das resultierende Dokument umschließen sollen. Und durch den Aufruf von:

Listing 12.48
Fortsetzen der Transformation

```
...
   <!--Fortsetzen der Transformation für untergeordnete Elemente-->
   <xsl:apply-templates />
...
```

weisen Sie den XSLT-Prozessor an, die Transformation für untergeordnete Elemente fortzusetzen. Die daraus resultierenden Elemente werden an dieser Stelle eingefügt.

Achtung

Der Unterschied zwischen der Dokumentwurzel und dem Wurzelelement (Root) besteht darin, dass die Dokumentwurzel den Dokumentknoten selbst referenziert, während Root auf das umschließende Element zeigt. Auf das JDOM-Modell gemünzt, bedeutet dies, dass »/« für das `Document` und `library` für `Document.getRootElement()` steht.
Fehlt in einem Template die *Apply-Templates*-Anweisung, wird das Resultat der Kindelemente nicht eingefügt. Das Template enthält dann nur den definierten statischen Inhalt.

12.7.2 Das Library-Element und eine konstante Kopfzeile

Das nächste Element nach der Dokumentwurzel (/) ist das eigentliche *Root-Element* des Dokuments. Dies ist in unserem Fall natürlich das einzige Kindelement der Dokumentwurzel, welches von `<xsl:apply-templates/>` aufgerufen wird.

Listing 12.49
Transformation des Root-Elements

```
...
   <!-- Template für das Root-Element 'library' -->
   <xsl:template match="library">
      <h3>Die folgenden Bücher können sie bei uns bestellen</h3>
      <table>
         <!-- Einfügen eines statischen Templates -->
         <xsl:call-template name="table-head" />

         <!-- Fortsetzung der Transformation für Kind-Elemente vom
              Typ 'book', andere Kind-Elemente werden ignoriert -->
         <xsl:apply-templates select="book"/>
      </table>
   </xsl:template>

   <!-- Definition des statischen Templates 'table-head' -->
   <xsl:template name="table-head">
      <tr>
         <th>Titel</th>
         <th>Verlag</th>
         <th>Erscheinungtermin</th>
      </tr>
   </xsl:template>
...
```

Zunächst definieren Sie eine Überschrift und anschließend eine Tabelle, die später die `<book>`-Elemente aufnehmen soll. Um die Überschrift der Tabelle auch in anderen Tabellen verwenden zu können, lagern Sie diese in ein eigenes *Template* aus. Dabei fällt auf, dass dieses *Template* nicht mit einem XPath-Ausdruck gekennzeichnet ist, mit dem es übereinstimmen soll (`match`), sondern über einen Namen verfügt, mit dem es aufgerufen wird.

Listing 12.50
Benannte Templates

```
...
   <!-- Aufruf eines 'benannten' Templates -->
   <xsl:call-template name="table-head" />
...
```

Die Call-Template-Anweisung ruft ein definiertes Template auf, das an dieser Stelle eingefügt wird. Die Verarbeitung wird anschließend im `<library>`-Template fortgesetzt.

Tipp

Vielleicht ist Ihnen schon aufgefallen, dass wir für die *XPath-Ausdrücke* in diesem Template lokale Pfade verwenden können, also beispielsweise `library` statt `/library`. Da wir die Verarbeitung an der Dokumentwurzel (/) begonnen haben und uns jetzt immer weiter vorhangeln, befinden wir uns nach einer `<apply-templates>`-Anweisung an genau dieser Stelle im Baum.

Als Nächstes werden Sie die eben erstellte Tabelle mit den Datensätzen, also unseren `<book>`-Elementen, füllen. Dazu verwenden wir abermals die `<apply-templates>`-Anweisung, um mit der Verarbeitung der Kindelemente fortfahren zu können, mit dem Unterschied, dass wir uns diesmal zur Sicherheit wirklich nur auf `<book>`-Elemente beschränken wollen.

```
<xsl:apply-templates select="book"/>
```

Hätte unsere `<library>` noch andere Kinder als `<book>`, würden trotzdem nur diese weiter transformiert und an dieser Stelle eingefügt.

12.7.3 Transformation von Book-Elementen

Jetzt kommt der interessanteste Teil der Transformation. Schließlich sollen nun die einzelnen Bücher in Form von Tabellenzeilen (`<tr>`) eingefügt werden und da Sie deren Aussehen auch im Nachhinein global anpassen wollen, definieren Sie zunächst ein entsprechendes Attribut-Set.

Listing 12.51
Definition eines Attribut-Set für unsere Tabellenzellen

```
...
   <!--
      Definition eines Attribut-Sets für Tabellen-Zeilen 'tr'
   -->
   <xsl:attribute-set name="data-style">
      <xsl:attribute name="cellpadding">3</xsl:attribute>
      <xsl:attribute name="cellspacing">5</xsl:attribute>
   </xsl:attribute-set>
...
```

Jetzt können Sie sich den Zeilen widmen und die Nutzdaten mit `<value-of>` in unser Zieldokument überführen.

Listing 12.52
Transformation eines Book-Elements

```
...
   <!-- Transformation eines 'book'-Elementes -->
   <xsl:template match="book">
      <tr>
         <td width="20%" xsl:use-attribute-sets="data-style">
            <xsl:value-of select="title"/>
         </td>
         <td width="30%" align="center"
             xsl:use-attribute-sets="data-style">
            <xsl:value-of select="publishing-house"/>
         </td>
         <td width="50%" xsl:use-attribute-sets="data-style">
```

Listing 12.52 (Forts.) Transformation eines Book-Elements

```
            <xsl:value-of select="publishing-date"/>
          </td>
        </tr>
    </xsl:template>
...
```

Das XSL-Element `<xsl:value-of select="XPath" />` nimmt den (Text-)Inhalt des beschriebenen XPath-Ausdrucks und fügt ihn in das Template ein. Ein ähnliches Element ist

Listing 12.53 Kopiert das Element

```
...
   <!-- Fügt das durch den XPath-Ausdruck beschriebene Element an
        dieser Stelle ein -->
   <xsl:copy-of select="XPathAusdruck" />
...
```

Achtung

Der korrekte *XPath-Ausdruck*, z.B. für den Textinhalt des `<title>`-Elements, lautet natürlich `./title/text()`. Aber da die *Value-Of*-Anweisung nur Textinhalt übernimmt, genügt die Angabe `./title` bzw. `title`.

welches auch die Knoten (Element) des Zielelements einfügt.

Es steht uns natürlich frei, neben den Attributen des Attribut-Set weitere spezifische Attribute zu definieren.

Listing 12.54 Mischen von Attributen und Attribut-Sets

```
...
   <!-- Verwendet die Attribute des Attribut-Satzes und 'width' -->
   <td width="20%" xsl:use-attribute-sets="data-style">
...
```

Der Attribut-Set steht für eine vordefinierte Menge von Attributen, die Sie nach Belieben erweitern können.

Tipp

Existiert ein Element nicht, auf das wir mit einem `<xsl-value-of select="XPath" />` oder `<xsl:copy-of select="XPath" />`-Element verweisen, so ist das Ergebnis einfach eine leere Zeichenkette.

12.7.4 Entscheidungen

Eigentlich ist das Stylesheet an dieser Stelle fertig. Doch im Augenblick wird jeder `<book>`-Datensatz gleich behandelt, egal, welchen Inhalt er hat, was zu unschönen Formatierungsfehlern führen kann. Beispielsweise hatten wir vorhin festgestellt, dass noch nicht erschienene Bücher in diesem Beispiel kein `<publishing-date>`-Element besitzen, auf das Sie im *Block-Template* jedoch verweisen. Da also kein lokales Element auf diesen *XPath-Ausdruck* passt, bleibt die entsprechende Zelle leer. Hier wollen wir stattdessen einen Hinweis platzieren.

Listing 12.55
Erweitertes Block-Template mit Hinweisen

```
...
   <xsl:template match="book">
      <tr>
         <td width="20%" xsl:use-attribute-sets="data-style">
            <xsl:value-of select="title"/>
         </td>
         <td width="30%" align="center"
             xsl:use-attribute-sets="data-style">
            <xsl:value-of select="publishing-house"/>
         </td>
         <td width="50%" xsl:use-attribute-sets="data-style">

            <!-- Test ob ein 'publishing-date' Element existiert -->
            <xsl:if
                test="string-length(publishing-date/text()) &gt; 0">
                  <xsl:value-of select="publishing-date"/>
            </xsl:if>
            <xsl:if
               test="string-length(publishing-date/text()) = 0">
                  Erscheinungstermin leider noch unbekannt.
            </xsl:if>
         </td>
      </tr>
   </xsl:template>
...
```

Hier sehen Sie *XPath-Ausdrücke* und *-Funktionen* nun endlich einmal live und in Farbe. Über das Element `<xsl:if test="`*XPathAusdruck*`">` können wir das Aussehen der Elemente abhängig von deren Inhalt oder Kontext definieren. Mit dem folgenden Ausdruck überprüfen Sie dabei, ob die Menge der angegebenen Knoten größer (*greater than*) als `0` ist.

Listing 12.56
Ein XPath-Ausdruck

```
...
   string-length(publishing-date/text()) &gt; 0
...
```

Achtung

In Listing 12.55 und Listing 12.56 wurde das Größer-Zeichen (>) durch die im vorangegangenen Kapitel beschriebene Entität `>` maskiert, da dieses Zeichen in XML-Dokumenten für den Abschluss eines Elements reserviert ist.

12.7.5 Das vollständige XSL Stylesheet

Jetzt sind Sie aber endgültig fertig. Das folgende Listing zeigt das Stylesheet noch einmal in seiner ganzen Schönheit.

Listing 12.57
Das gesamte XSL-Stylesheet (libraryStyle.xsl)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<!-- Definition eines Attribut-Sets für Tabellen-Zeilen 'tr' -->
<xsl:attribute-set name="data-style">
   <xsl:attribute name="cellpadding">3</xsl:attribute>
   <xsl:attribute name="cellspacing">5</xsl:attribute>
</xsl:attribute-set>

<!-- Transformation der Dokumentwurzel -->
<xsl:template match="/">
```

Listing 12.57 (Forts.)
Das gesamte XSL-Stylesheet (libraryStyle.xsl)

```
   <!-- Einfügen von umschließenden Elementen -->
   <html>
      <body>
         <!-- Fortsetzen der Transformation für untergeordnete
              Elemente -->
         <xsl:apply-templates />
      </body>
   </html>
</xsl:template>

<!-- Template für das Root-Element 'library' -->
<xsl:template match="library">
   <h3>Die folgenden Bücher können sie direkt bei uns bestellen</h3>
   <table>
      <!-- Einfügen eines statischen Templates -->
      <xsl:call-template name="table-head" />

      <!-- Fortsetzung der Transformation für Kind-Elemente vom
           Typ 'book', andere Kind-Elemente werden ignoriert -->
      <xsl:apply-templates select="book"/>
   </table>
</xsl:template>

<xsl:template match="book">
   <tr>
      <td width="20%" xsl:use-attribute-sets="data-style">
         <xsl:value-of select="title"/>
      </td>
      <td width="30%" align="center"
          xsl:use-attribute-sets="data-style">
         <xsl:value-of select="publishing-house"/>
      </td>
      <td width="50%" xsl:use-attribute-sets="data-style">

         <!-- Test ob ein 'publishing-date' Element existiert -->
         <xsl:if
         test="string-length(publishing-date/text()) &gt; 0">
               <xsl:value-of select="publishing-date"/>
         </xsl:if>
         <xsl:if
            test="string-length(publishing-date/text()) = 0">
               Erscheinungstermin leider noch unbekannt.
         </xsl:if>
      </td>
   </tr>
</xsl:template>

<!-- Definition des statischen Templates 'table-head' -->
<xsl:template name="table-head">
   <tr>
      <th>Titel</th>
      <th>Verlag</th>
      <th>Erscheinungtermin</th>
   </tr>
</xsl:template>

</xsl:stylesheet>
```

Egal, ob Sie das Beispieldokument nun mit einem XSLT-Prozessor in ein HTML-Dokument übersetzen oder die Transformation mithilfe einer *Stylesheet-PI* im Browser direkt erledigen, das Ergebnis unserer Transformation kann sich sehen lassen.

Abbildung 12.2
Die Library nach der Transformation

12.7.6 Zusammenfassung XSL Stylesheet

Die *eXtensible Stylesheet Language (XSL)* gehört neben der XML-Spezifikation und XPath zu den wichtigsten XML-basierten Technologien. Dabei macht XSL viel Gebrauch von XPath und ist praktisch nur durch dessen Möglichkeiten beschränkt. Nicht umsonst fällt die XSL-Spezifikation 2.0 zusammen mit dem überarbeiteten XPath 2.0-Standard.

Über XSL-basierte Stylesheets können Sie Ihre abstrakten XML-Dokumente in eine andere Struktur überführen, um diese beispielsweise in einem HTML-Browser darstellen zu können oder zueinander inkompatible Datenformate zwecks Informationsaustauschs in ein gemeinsames Format zu überführen. Die Umwandlung unserer XML-Dokumente anhand eines Stylesheet übernimmt ein so genannter XSLT-Prozessor, wobei Sie sich zwischen verschiedenen Implementierungen mit unterschiedlicher Standardgenauigkeit und spezifischen Erweiterungen entscheiden können.

Ein Stylesheet besteht aus verschiedenen Elementen, wobei das Hauptaugenmerk auf den *Templates* liegt. Diese werden für jedes Element des XML-Dokuments von oben nach unten durchsucht, bis ein Template mit passendem XPath-Ausdruck gefunden ist, anhand dessen das Element anschließend formatiert wird.

12.8 XSL Formatting Objects

Die *eXtensible Stylesheet Language Formatting Objects (XSL-FO)* sind eine weitere XML-Spezifikation, die der rein grafischen Aufbereitung von XML-Dokumenten dienen soll. Während bei XSLT die Transformation in ein beliebiges anderes XML-Format im Vordergrund stand, geht es bei den Formatting Objects um die Spezifikation eines bestimmten XML-Formats, welches von entsprechenden Renderern in verschiedene druckbar Formate, wie Adobes Portable Document Format (PDF), PostScript (PS) oder Microsofts Rich Text Format (RTF) umgewandelt werden kann.

Während sich XSL-Transformationen also mit der Restrukturierung von XML-Dokumenten beschäftigen, befasst sich XSL Formatting Object mit der Definition des Layouts. XML-FO soll es ermöglichen, XML-basierte Daten in beliebiger Form auf Druckern auszugeben. Dazu definiert es eine Reihe von Objekten, wie Tabellen und Fußnoten oder auch Textcontainer, die mit verschiedenen Attributen ausgestattet werden können.

12.8.1 Geschichte

Um das Erscheinungsbild von HTML-Seiten genau festzulegen, bedienen Sie sich vermutlich der *Cascading Stylesheets (CSS)*. Diese enthalten eine Reihe von Attributen, mit denen Sie das Aussehen der verschiedenen Elemente beschreiben können. So legt das Attribut `font-size` beispielsweise die Schriftgröße des enthaltenen Texts fest.

CSS bildeten auch den Ausgangspunkt für die Entwicklung von XSL-FO, so dass sich zwar die Syntax der Sprache änderte (sie wurde XML-konform gemacht), andererseits aber auch viele Attributbezeichnungen übernommen wurden. Beispielsweise legt die folgende Anweisung die Schriftgröße in XSL-FO fest

Listing 12.58
Definition der Schriftgröße in XSL-FO

```
...
   <!-- Definition eines Blockes samt Schriftgröße -->
   <fo:block font-size="12pt">Hier steht Text.</fo:block>
...
```

12.8.2 Grundlagen und Installation

Info

Da viele Attributbezeichnungen in XSL-FO identisch mit denen in CSS sind, erleichtert ein gewisses Basiswissen in CSS das Erlernen von XSL-FO erheblich.

Die XSL-FO-Spezifikation beschreibt Eigenschaften, wie sie für die Erzeugung von Printmedien wichtig sind. Dazu gehören unter anderem:

- Seitenaufbau (Kopf- und Fußzeilen, Body) sowie die Größe der Seite
- Größe, Position und Farbe von Rahmen und Tabellen
- Zu verwendende Schriftarten und Zeilenabstände

Um Ihre abstrakten XML-Daten in eine druckbare Form zu bringen, müssen Sie diese in ein entsprechendes FO-Dokument umwandeln und es anschließend von einem XSL-FO-Renderer in das gewünschte Format überführen lassen. Für die Transformation in XSL-FO bedienen wir uns dabei natürlich der XSLT, die gewissermaßen als Druckvorstufe fungiert.

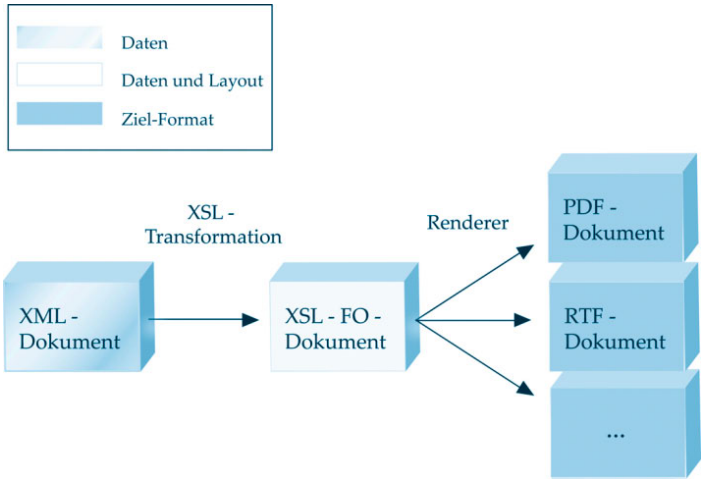


Abbildung 12.3
Umwandlung eines XML-Dokuments

Bei der Wahl des Formatierers stehen Ihnen wieder einmal verschiedene Implementierungen zur Verfügung:

- Apaches *Formatting Object Processor (FOP)*

  Apaches FOP (*http://xml.apache.org/fop/*) soll, wie viele andere Projekte, als Referenzimplementierung für die XSL-FO-Spezifikation dienen und die Transformationen in verschiedene Druckformate wie PDF, PostScript und RTF ermöglichen.

- *XEP* von RenderX

  Auch RenderX (*http://www.renderx.com/*) bietet mit XEP einen FO-Prozessor an, der Transformationen in die unterschiedlichsten Formate erlaubt. Da hinter dieser Software ein kommerzielles Unternehmen steht, ist der Entwicklungsstand von XEP deutlich weiter als der des Apache FOP, wobei alle Elemente des Standards unterstützt werden.

- *XSL Formatter*, der Antenna House Inc.

  Dieses kommerzielle Produkt der in Japan ansässigen Firma besticht vor allem durch die hohe Geschwindigkeit, mit der die Dokumente erzeugt werden. Auch dieses Produkt kann zum Testen kostenlos von der Homepage (*http://www.antennahouse.com/*) heruntergeladen werden. Die Testversion hinterlässt, wie der XEP, einen Hinweis auf jeder Seite.

Tipp

Die offizielle W3C-Spezifikation für *XSL-FO* ist, wie sollte es anders sein, in verschiedenen Formaten erhältlich. Sie finden die verlinkte HTML-Version unter *http://www.w3.org/TR/xsl/*, von wo aus Sie sich ebenfalls die 400 Seiten starke PDF-Version fürs Offline-Arbeiten herunterladen können.

Die Installation des in diesem Kapitel verwendeten FOP beschränkt sich, wie bei fast aller Java-Software auf den Download des Archivs, das Entpacken und das Einbinden der enthaltenen JAR-Bibliotheken in den Classpath.

12.9 Aufbau eines FO-Dokuments

Der Aufbau eines FO-Dokuments ist immer gleich und orientiert sich an einem fest vorgegebenen Muster:

- Zunächst werden die einzelnen Seiten und ihre Maße (Größe, Ränder etc.) definiert. Mit XSL-FO können nämlich beliebige Papierformate erzeugt werden.
- Optional folgt dann die Definition eines oder mehrerer zu verwendender Farb- und Font-Profile.
- Danach folgt der eigentliche Inhalt der Seiten, wobei Text und Tabellen, die nicht auf einer einzelnen Seite Platz finden, an den entsprechenden Stellen umgebrochen werden.

Listing 12.59 zeigt den grundlegenden Aufbau eines XSL-FO Dokuments schematisch.

Listing 12.59 Grundlegender Aufbau eines XLS-FO-Dokuments (aus simpleDocument.fo)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">

   <!-- Definition der Seiten-Layouts -->
   <fo:layout-master-set>
      ...
   </fo:layout-master-set>

   <!-- Optionale Deklaration von Profilen -->
   <fo:declarations>
      ...
   </fo:declarations>

   <!-- Inhalt des Dokumentes -->
   <fo:page-sequence>
      ...
   </fo:page-sequence>

</fo:root>
```

Während das Wurzelelement `<fo:root>` beliebig viele `<page-sequence>`-Elemente enthalten darf, erlaubt die Spezifikation nur ein `<fo:layout-master-set>`, welches die Definition aller Seiten enthalten muss. Da die optionalen Deklarationen nur für den weit fortgeschrittenen Benutzer interessant sind und Anfänger häufig verwirren, werden wir diese im Folgenden nicht weiter betrachten.

Tipp

Dem interessierten Leser sei an dieser Stelle das Tutorial unter *http://www.w3schools.com/xslfo/* ans Herz gelegt.

12.9.1 Die Definition einer A4-Seite

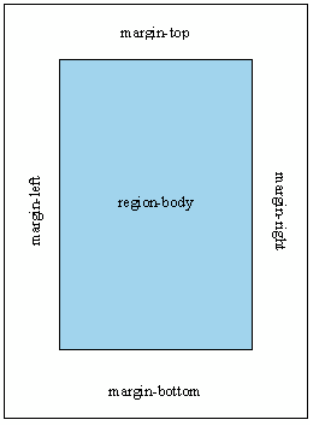
Da die FO-Spezifikation zur Erzeugung von Printmedien gedacht ist, muss ein Dokument zumindest die Definition *einer Seite* enthalten. Das folgende Listing beschreibt dabei eine in Europa übliche DIN-A4-Seite.

Listing 12.60
Definition einer einfachen DIN-A4-Seite (simpleDocument.fo)

```
<!-- Ein einfaches Layout -->
<fo:layout-master-set>

   <!-- Definition einer DIN A4 Seite -->
   <fo:simple-page-master master-name="simpleA4"
       page-height="29.7cm" page-width="21cm"
       margin-top="2cm" margin-bottom="2cm"
       margin-left="2.5cm" margin-right="2.5cm">

      <fo:region-body backgroud-color="grey" />
   </fo:simple-page-master>
</fo:layout-master-set>
```

Hiermit geben wir an, dass unser Dokument Seiten der Größe 21cm x 29,7cm enthält, die innerhalb der angegebenen Ränder beschrieben werden können. Mit `<fo:region-body>` beschreiben Sie dabei die eigentliche Schreibfläche des Dokuments, die sich in unserem Fall bis zum Beginn der Seitenränder erstreckt und die Hintergrundfarbe grau (`grey`) haben soll:

Abbildung 12.4
Aufbau einer einfachen DIN-A4-Seite



Neben der `<region-body>`, die in jeder Seite vorhanden ist und den Inhalt unseres Dokuments (`flow`, siehe unten) aufnehmen soll, können Sie vier weitere Regionen, etwa für Kopf- und Fußzeilen, definieren, welche für statischen Inhalt wie Seitenzahlen oder Titel gedacht sind.

Listing 12.61
Definition einer DIN-A4-Seite mit Kopf- und Fußzeile (aus simpleDocument.fo)

```
<!-- Ein komplexes Layout -->
<fo:layout-master-set>

   <!-- Definition einer DIN A4 Seite -->
   <fo:simple-page-master master-name="defaultA4"
       page-height="29.7cm" page-width="21cm"
       margin-top="2cm" margin-bottom="2cm"
```

Listing 12.61 (Forts.)
Definition einer DIN-A4-Seite mit Kopf- und Fußzeile (aus simpleDocument.fo)

```
        margin-left="2.5cm" margin-right="2.5cm">

      <!-- Beschreibung des Inhaltsbereiches -->
      <fo:region-body
          margin-top="1.5cm" margin-bottom="1.8cm"
          margin-left="2cm" margin-right="2.5cm" />

      <!-- Beschreibung der statischen Regionen -->
      <fo:region-before region-name="header" extent="1.3cm"/>
      <fo:region-after region-name="footer" extent="1.5cm" />
      <fo:region-start region-name="left" extent="1cm"/>
      <fo:region-end region-name="right" extent="2cm" />

   </fo:simple-page-master>
</fo:layout-master-set>
```

Damit bekommt unsere DIN-A4-Seite schließlich folgendes Aussehen:

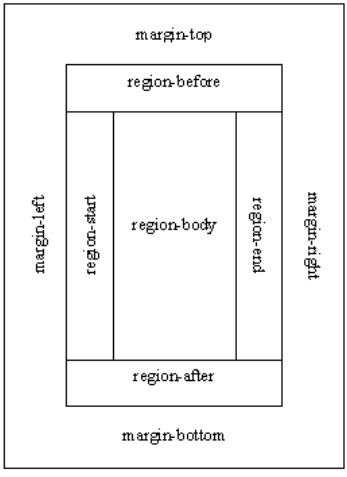


Abbildung 12.5
Aufbau einer DIN-A4-Seite mit Kopf- und Fußzeile

Wie Sie sehen, ist zwischen den statischen Regionen und dem Region-Body ein gewisser Rand. Diesen steuern Sie über die `<fo:region-body>`-Attribute `margin-xxx` und `extend`. Hierdurch können Sie die Kopfzeile beispielsweise auch in den Dokument-Body hineinragen lassen.

12.9.2 Das Auffüllen mit Inhalt

Nachdem Sie das Aussehen der Seiten im `<fo:layout-master-set>` definiert haben, können Sie sich dem Inhalt dieser Seiten in der `<fo:page-sequence>` widmen. Alle Sequenzen von Seiten werden durch ein bestimmtes, zuvor festgelegtes Layout beschrieben.

Grundlage für die Beschreibung der Seiten bilden hier, wie bei HTML-Dokumenten, vor allem Blöcke und Tabellen, die auch ineinander geschachtelt werden können. Um eine Seite zu erzeugen, verwenden Sie beispielsweise folgende Definitionen:

Listing 12.62
Erstellen einer Seite (aus simpleDocument.fo)

```
<!-- Beschreibung einer Seite -->
<fo:page-sequence master-reference="defaultA4">

   <!-- Beschreibung der Kopfzeile -->
   <fo:static-content flow-name="header" text-align="center">
      <fo:block font-size="18pt">Dies ist die Kopfzeile</fo:block>
   </fo:static-content>
```

```
    <!-- Beschreibung der Fußzeile -->
    <fo:static-content flow-name="footer">
       <fo:block font-size="12pt">Der Text steht im Fuß</fo:block>
    </fo:static-content>

    <!-- Ein Rand fuer Marginalien rechts -->
    <fo:static-content flow-name="right">
       <fo:block font-size="12pt">Platz für Marginalien</fo:block>
    </fo:static-content>

    <!-- Die Region fuer den Fließtext -->
    <fo:flow flow-name="xsl-region-body">
       <fo:block>Das ist Ihre erste mit XSL-FO erzeugte Seite.
                 Dieser Text steht im &lt;region-body&gt;
       </fo:block>
    </fo:flow>

</fo:page-sequence>
```

Listing 12.62 (Forts.)
Erstellen einer Seite (aus simpleDocument.fo)

Achtung

Um reservierte Sonderzeichen in die resultierenden Dokumente zu integrieren, verwenden wir auch hier die dafür vorgesehenen *Entities* `<` für <, `>` für > usw.

Achtung

Da der Renderer zuerst alle statischen Elemente auf einer Seite platzieren muss, bevor er mit der Umsetzung des dynamischen Inhalts beginnen kann, müssen alle `<fo:static-content>`-Elemente einer `<page-sequence>` vor dem Flusselement (`<fo:flow>`) definiert werden.

Wie Sie sehen, besteht eine `<fo:page-sequence>` aus verschiedenen statischen Elementen, die Sie zuvor definiert haben, und *einem Flusselement* (`flow`), welches den Fließtext der Seite aufnimmt. Geht dieser über die definierte Seite hinaus, wird er an der entsprechenden Stelle umgebrochen und auf der nächsten Seite fortgesetzt. Die statischen Elemente werden auch auf die folgende Seite übernommen.

12.9.3 Rendern mit dem Formatting Objects Processor (FOP)

Um ein einfaches FO-Dokument zu rendern, bringt *Apaches FOP* ein Shell- bzw. Batchfile mit, mit dem Sie das Dokument per Kommandozeile rendern können. Das folgende Listing zeigt die Anweisung, um die XML-FO-Datei (Listing 12.56 bis 12.59) in ein PDF-Dokument umzuwandeln. Dazu führen Sie von der Kommandozeile folgenden Aufruf im FOP-Installationsverzeichnis aus:

```
// Rendern eines Dokumentes via Kommandozeile
fop simpleDocument.fo simpleDocument.pdf
```

Listing 12.63
Ansteuern des FOP über die Kommandozeile

Tipp

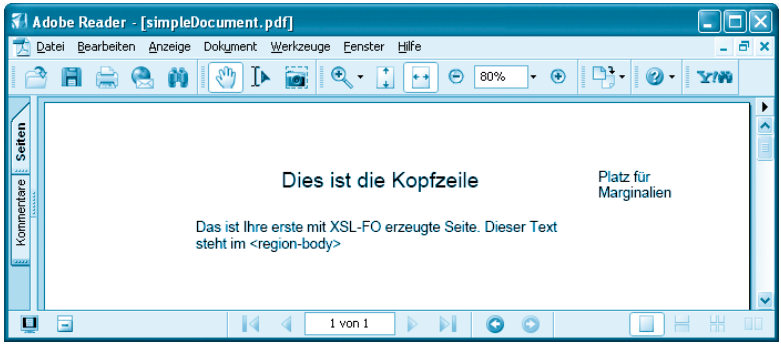
Sie können das Beispiel natürlich auch direkt aus dem Verzeichnis *Beispiele/fo* aufrufen. In diesem Fall müssen Sie vor fop nur den Pfad zum Installationsverzeichnis des FOP angeben. Also beispielsweise:

```
C:\Programme\FOP\fop simpleDocument.fo simpleDocument.pdf
```

12.9.4 Resultat

Hier kommt es, ohne große Umschweife: das Resultat Ihrer Mühen.

Abbildung 12.6
Das Resultat



12.10 Von XML zum PDF – ein Beispiel

Zu guter Letzt erfahren Sie schließlich noch, wie Sie die eben kennen gelernten Technologien XML, XSLT, XPath und XSL-FO gemeinsam verwenden können, um zum Beispiel dynamisch erstellte PDF-Dokumente erzeugen zu können. Dazu werden Sie das zuvor besprochene Bibliotheksbeispiel (`<library>`), welches wir weiter oben schon über ein XSL-Stylesheet im Browser dargestellt haben, nun in ein druckbares Dokument überführen.

12.10.1 Das Stylesheet

Den Kern der Umwandlung bildet natürlich auch hier ein XSL-Stylesheet, anhand dessen das abstrakte Dokument transformiert wird – mit dem Unterschied, dass das Zielformat diesmal nicht HTML, sondern eben XSL-FO ist.

Listing 12.64
Unser FO-Stylesheet (foStyle.xsl)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:fo="http://www.w3.org/1999/XSL/Format">

   <!-- Definition von Attribut-Sets -->
   <xsl:attribute-set name="cell-style">
      <xsl:attribute name="border-width">0.5pt</xsl:attribute>
      <xsl:attribute name="border-style">solid</xsl:attribute>
```

Listing 12.64 (Forts.)
Unser FO-Stylesheet (foStyle.xsl)

```
   <xsl:attribute name="border-color">black</xsl:attribute>
</xsl:attribute-set>

<xsl:attribute-set name="block-style">
   <xsl:attribute name="font-size">10pt</xsl:attribute>
   <xsl:attribute name="line-height">15pt</xsl:attribute>
   <xsl:attribute name="start-indent">1mm</xsl:attribute>
   <xsl:attribute name="end-indent">1mm</xsl:attribute>
</xsl:attribute-set>

<!-- Transformation der Dokumentwurzel -->
<xsl:template match="/">
   <fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
      <fo:layout-master-set>

         <fo:simple-page-master master-name="defaultA4"
             page-height="29.7cm" page-width="21cm"
             margin-top="2cm" margin-bottom="2cm"
             margin-left="2.5cm" margin-right="2.5cm">

            <fo:region-body
                margin-top="1.5cm" margin-bottom="1.8cm"
                margin-left="2cm" margin-right="2.5cm" />

          <fo:region-before region-name="header" extent="1.3cm"/>
          <fo:region-after region-name="footer" extent="1.5cm" />
           <fo:region-start region-name="left" extent="1cm"/>
           <fo:region-end region-name="right" extent="2cm" />
         </fo:simple-page-master>
      </fo:layout-master-set>

      <fo:page-sequence master-reference="defaultA4">

         <fo:static-content flow-name="header">
            <fo:block font-size="14pt" text-align="center">
               Die folgenden Bücher können sie direkt bei
               uns bestellen
            </fo:block>
         </fo:static-content>

        <fo:static-content flow-name="footer" >
           <fo:block text-align="center">Seite <fo:page-number/>
               von <fo:page-number-citation ref-id="LastPage"/>
           </fo:block>
         </fo:static-content>

         <fo:flow flow-name="xsl-region-body">
            <xsl:apply-templates />
            <fo:block id="LastPage" />
         </fo:flow>

      </fo:page-sequence>
   </fo:root>
</xsl:template>

<!-- Template des Root-Elementes 'library '-->
<xsl:template match="library">
   <fo:table border-style="solid" table-layout="fixed">
      <fo:table-column column-width="3cm" />
      <fo:table-column column-width="3cm" />
      <fo:table-column column-width="5cm" />

      <fo:table-header>
         <xsl:call-template name="table-head" />
      </fo:table-header>
      <fo:table-body>
```

Listing 12.64 (Forts.)
Unser FO-Stylesheet (foStyle.xsl)

```
            <xsl:apply-templates select="book"/>
         </fo:table-body>
      </fo:table>
   </xsl:template>

   <!-- Template der 'book'-Elemente -->
   <xsl:template match="book">
      <fo:table-row>
         <fo:table-cell xsl:use-attribute-sets="cell-style">
            <fo:block xsl:use-attribute-sets="block-style">
               <xsl:value-of select="title"/>
            </fo:block>
         </fo:table-cell>
         <fo:table-cell xsl:use-attribute-sets="cell-style">
            <fo:block xsl:use-attribute-sets="block-style">
               <xsl:value-of select="publishing-house"/>
            </fo:block>
         </fo:table-cell>
         <fo:table-cell xsl:use-attribute-sets="cell-style">
            <fo:block xsl:use-attribute-sets="block-style">
               <xsl:if
                test="string-length(publishing-date/text()) &gt; 0">
                  <xsl:value-of select="publishing-date"/>
               </xsl:if>
               <xsl:if
                test="string-length(publishing-date/text()) = 0">
                  Erscheinungstermin leider noch unbekannt.
               </xsl:if>
            </fo:block>
         </fo:table-cell>
      </fo:table-row>
   </xsl:template>

   <!-- Ein statisches Template -->
   <xsl:template name="table-head">
      <fo:table-row>
         <fo:table-cell xsl:use-attribute-sets="cell-style">
            <fo:block xsl:use-attribute-sets="block-style"
                      text-align="center">Titel</fo:block>
         </fo:table-cell>
         <fo:table-cell xsl:use-attribute-sets="cell-style">
            <fo:block xsl:use-attribute-sets="block-style"
                      text-align="center">Verlag</fo:block>
         </fo:table-cell>
         <fo:table-cell xsl:use-attribute-sets="cell-style">
            <fo:block xsl:use-attribute-sets="block-style"
                      text-align="center">
               Erscheinungstermin</fo:block>
         </fo:table-cell>
      </fo:table-row>
   </xsl:template>

</xsl:stylesheet>
```

Was Ihnen zunächst sicher auffallen wird: Dieses Stylesheet fällt im Vergleich mit seinem HTML-Pendant wesentlich umfangreicher aus. Der Grund dafür ist, dass sich HTML auf die reine Darstellung von Inhalten in Browsern beschränkt, während mit XSL-FO beliebige, druckbare Formate erstellt werden können. Daher lassen sich Aussehen und Form sehr differenziert einstellen. Wenn Sie das Stylesheet jedoch in Ruhe von oben nach unten durcharbeiten, werden Sie feststellen, dass es die gleichen Elemente wie sein Vorgänger enthält.

12.10.2 Attribut-Sets

Zunächst definieren Sie zwei Attribut-Sets, mit denen Sie später unsere Zellen und Texte formatieren werden.

Listing 12.65
Definition von Attribut-Sets

```
...
   <xsl:attribute-set name="cell-style">
      <xsl:attribute name="border-width">0.5pt</xsl:attribute>
      <xsl:attribute name="border-style">solid</xsl:attribute>
      <xsl:attribute name="border-color">black</xsl:attribute>
   </xsl:attribute-set>

   <xsl:attribute-set name="block-style">
      <xsl:attribute name="font-size">10pt</xsl:attribute>
      <xsl:attribute name="line-height">15pt</xsl:attribute>
      <xsl:attribute name="start-indent">1mm</xsl:attribute>
      <xsl:attribute name="end-indent">1mm</xsl:attribute>
   </xsl:attribute-set>
...
```

Wie bereits erwähnt, sieht die XSL-FO-Spezifikation eine Fülle von möglichen Attributen vor, mit denen Sie jedes Element bis ins kleinste Detail anpassen können. (Nicht umsonst ist die Referenz 400 Seiten stark.) Wir beschränken uns hier auf die nötigsten und gebräuchlichsten.

12.10.3 Das Root-Element

Im Dokument-Root (/) definieren Sie schließlich zunächst das Dokument und die statischen Inhalte und setzen die Verarbeitung anschließend im Flusselement Region-Body mit `<xsl:apply-templates>` fort.

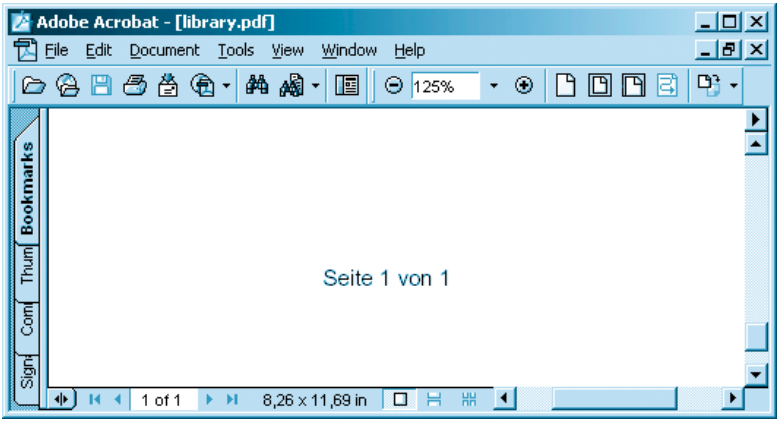
Listing 12.66
Ausschnitt aus der Dokumentdefinition

```
...
   <fo:static-content flow-name="footer" >
       <fo:block text-align="center">Seite <fo:page-number/>
           von <fo:page-number-citation ref-id="LastPage"/>
       </fo:block>
   </fo:static-content>

   <fo:flow flow-name="xsl-region-body">
       <xsl:apply-templates />
       <fo:block id="LastPage" />
   </fo:flow>
...
```

Besondere Aufmerksamkeit sollten Sie dabei dem `footer` und der Ausgabe der Seitenzahl schenken:

- Zunächst definieren Sie ein leeres Blockelement am Ende der `xsl-region-body` und versehen dieses mit einer ID.
- Außerdem können Sie über das Tag `<fo:page-number>` die aktuelle Seitenzahl ermitteln.
- Da Sie außerdem wissen, dass Ihr leerer Block auf der letzten Seite des Dokuments steht und sie dessen Seitenzahl über das Tag `<fo:page-number-citation ref-id="LastPage"/>` ermitteln können, sind Sie nun in der Lage, auch anspruchsvolle Seitenzahlen zu definieren (Abbildung 12.7).

Abbildung 12.7
Der PDF-Footer



Der Prozessor kann eine Seite natürlich erst fertig stellen, wenn alle Elemente der Seite bekannt sind. Das bedeutet, dass er bei diesen Vorwärtsverweisen auf ein Element der letzten Seite natürlich auch die erste Seite erst nach dem Rendern der letzten Seite herausschreiben kann und diese bis dahin im Speicher cachen muss.

Gerade bei großen Dokumenten mit mehr als 100 Seiten macht sich dies in einem deutlichen Ressourcenhunger und Performance-Einbruch bemerkbar, weshalb diese Technik nur für kleinere und mittelgroße Dokumente verwendet werden sollte.

12.10.4 Library und Book

Die Templates für die `<library>`- und `<book>`-Elemente unterscheiden sich nur unwesentlich vom HTML-Pendant. Es ist lediglich zu beachten, dass sich Textelemente innerhalb der FO-Spezifikation immer innerhalb eines Blockelements befinden müssen. Sie können verschiedenen Blockelemente auch ineinander schachteln, wobei das innere Element die Eigenschaften des äußeren erben bzw. überschreiben kann.

Listing 12.67
Block im Block

```
<fo:block font-size="15pt" font-style="italic">
          Dieser Text steht außen ...
    <fo:block font-size="13pt"> ... und dieser drinnen.</fo:block>
</fo:block>
```

Dadurch wird folgende Formatierung erreicht:

Abbildung 12.8
Block in Block

Dieser Text steht Außen...
... und dieser Innen.

Neben dieser Schachtelungstechnik von Blöcken und Tabellen existieren eine Unmenge weiterer formatierender Elemente, mit denen wir unseren Text gestalten können, für die ich Sie allerdings ein weiteres Mal auf die Referenz verweisen möchte.

12.10.5 Das Rendering

Nun müssen Sie die Ausgangsdatei und das Stylesheet nur noch zusammenführen und schon erhalten Sie das gewünschtes PDF-Dokument.

Per Kommandozeile via Script

Um eine Änderung am Stylesheet mal eben zu testen oder eine neue Formatierung auszuprobieren, schreiben Sie auf der Kommandozeile im Verzeichnis *Beispiele/fo*:

Listing 12.68
Rendern unseres Dokuments per Kommandozeile

```
// Rendern mit Stylesheet
Pfad/zum/FOP/Installationsverzeichnis/fop -xsl xsl/foStyle.xsl -xml
xml/library.xml -pdf output/library.pdf
```

Info

Damit der Aufruf aus Listing 12.68 funktioniert, müssen Sie die Zeichenkette *Pfad/zum/FOP/Installationsverzeichnis/* natürlich durch Ihr Installationsverzeichnis für den FOP (z.B. *C:\Programme\FOP*) ersetzen.

Über eine Java-Klasse

Natürlich können Sie auch eine eigene Java-Klasse schreiben, um etwa ein dynamisch erzeugtes Dokument mit einem Servlet auszuliefern:

Listing 12.69
Rendern in Java

```
package de.akdabas.javaee.fop;
import java.io.*;
import javax.xml.transform.*;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.sax.SAXResult;

import org.apache.fop.apps.Fop;
import org.apache.fop.apps.FopFactory;
import org.apache.fop.apps.MimeConstants;

/** Rendert ein XSL-FO Dokument mit Hilfe Apache's FOP */
public class RenderTest {

   /** Demonstriert die Funktionsweise des FOP Renderes */
   public static void main(String[] args) {
      RenderTest renderTest = new RenderTest();
      renderTest.render();
   }

   /** Rendert das Dokument mit dem oben beschriebenen Stylesheet */
   public void render() {
      try {
         // Erzeugen des Ausgabe-Stroms
         OutputStream out = new FileOutputStream("fo/library.pdf");

         // Referenzieren der FOP-Factory
         FopFactory fopFactory = FopFactory.newInstance();

         // Erzeugen eines Formatting Objects Processors (FOP)
         Fop fop = fopFactory.newFop(MimeConstants.MIME_PDF, out);

         // Einlesen von Quelldokument und Stylesheet
         Source src = new StreamSource("xml/library.xml");
         Source xsltSrc = new StreamSource("xsl/foStyle.xsl");
         Result res = new SAXResult(fop.getDefaultHandler());

         // Referenzieren einer Transformer-Factory
         TransformerFactory tFactory =
            TransformerFactory.newInstance();
```

Listing 12.69 (Forts.)
Rendern in Java

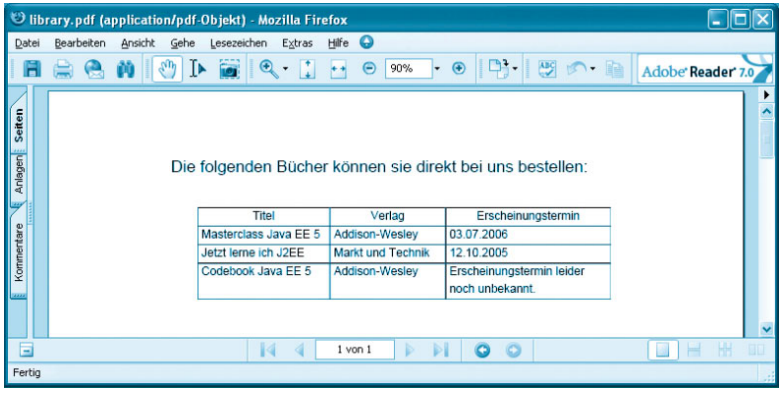
```
            // Erzeugen eines neuen Transformer-Objektes
            Transformer transformer = tFactory.newTransformer(xsltSrc);

            // ... und Action
            transformer.transform(src, res);
        } catch (Exception any) {
            any.printStackTrace();
        }
    }
}
```

Kern der Transformation ist hierbei die Klasse `Fop`, die abhängig vom verwendeten MIME-Format eingehende SAX-Events in einen gerenderten Zeichenstrom transformiert. Da auch der FOP-Renderer über einen SAX-Eventhandler verfügt, ist der Rest der Klasse einfach nur eine Wiederholung der Java-basierten XSL-Transformation:

- Zunächst erzeugen Sie zwei `Source`-Objekte für das Ausgangsdokument und das Stylesheet.
- Anschließend erzeugen Sie ein `SAXResult`-Objekt, das die *Events* an den `ContentHandler` des `Drivers` weiterreicht.
- Über die `TransformerFactory` und die *Stylesheet*-`Source` erzeugen Sie einen neuen `Transformer` ...
- ... und transformieren anschließend das Ausgangsdokument nach FO, von wo es gerendert wird.

Damit leben Sie die bereits in Abbildung 12.3 beschriebene Umwandlung eines XML-Dokuments live.

Abbildung 12.9
Resultat der »doppelten« Transformation durch XSL Stylesheet und FOP



12.11 Zusammenfassung

XPath, XSLT und XSL-FO bilden ein nahezu unschlagbares Dreigespann. Während Sie über XPath Muster für beliebige Elemente eines XML-Dokuments beschreiben können, verwendet die XSLT diese zur Transformation der Elemente in andere Formate, die neben den Nutzdaten dann auch Dar-

stellungsinformationen beinhalten. Eines dieser Formate ist das vom W3C standardisierte XSL-FO, das eine Beschreibung beliebiger Druckformate ermöglicht.

Auf Basis dieser Technologien existieren heute bereits ganze Frameworks wie *Apache Cocoon*, die abstrakte Basisdokumente über verschiedene Transformationen mit Daten füllen und je nach Anforderung in verschiedene Ausgabeformate wie HTML oder WML umwandeln. Damit bilden diese Technologien auch eine interessante Alternative zu *JSPs* und *Servlets*. Gerade wenn sehr generische Dokumente in unterschiedlichen Formaten dargestellt werden sollen, sind *Cross Media Publishing Frameworks* gegenüber klassischen Ansätzen meist klar im Vorteil, denn solange ein entsprechender Renderer existiert, ist jedes Format realisierbar.

13

Webservices

In der Informationsgesellschaft leben Unternehmen davon, wie gut sie mit anderen kommunizieren. Sich rasant entwickelnde Technologien und das Internet haben die weltweite Kommunikation vereinfacht und ermöglichen rund um den Globus verteilte Arbeitsgruppen. Ein wichtiger Erfolgsfaktor stellt dabei die Verschmelzung der unterschiedlichen Systeme und Applikationen zu einem virtuellen Gesamtsystem dar und das ist der Punkt, an dem Webservices ins Spiel kommen.

Tipp

Bei den Webservices begegnen Ihnen viele der in diesem Buch vorgestellten Techniken wieder. Der größte Unterschied zwischen Webservices und den zuvor behandelten Technologien ist, dass Webservices stärker als alle anderen hier vorgestellten Technologien versuchen, die Lücke zwischen den unterschiedlichen Plattformen und Programmiersprachen zu schließen. Sie eignen sich damit vor allem für heterogene Systemlandschaften.

In diesem Kapitel erfahren Sie, was sich hinter den vielfältigen Kürzeln XML-RPC, SOAP, WSDL und UDDI verbirgt und wie Sie einen einfachen Webservice in Java aufsetzen können. Das Kapitel hat dabei nicht den Anspruch, Webservices und alle damit verbundenen Möglichkeiten erschöpfend zu behandeln. Stattdessen soll es Ihnen diese Technologie sowie die dahinter stehenden Konzepte näher bringen und einen Einstieg in diese scheinbar nur aus Kürzeln bestehende Welt des Programmierens vermitteln. Das Studium dieses Kapitels soll Sie in die Lage versetzen, sich z.B. mit den vielen online verfügbaren Tutorials zu beschäftigen oder weiterführende Fachartikel zu diesem Thema zu verfolgen.

13.1 Die Idee hinter Webservices

Die Idee, welche hinter Webservices steckt, ist sehr einfach: Man überführe die zu übertragenden Informationen in ein allgemeingültiges Format wie XML und übermittle die serialisierten Informationen anschließend über ein etabliertes Protokoll, welches sich leicht durch Firewalls und Router tunneln lässt. Schon hat man eine allgemeingültige Schnittstelle, über die Client und Server plattform- und programmiersprachenunabhängig miteinander kommunizieren können.

Kerntechnologie ist dabei ein weiteres Mal die eXtensible Markup Language (XML), deren Eigenschaften und Manipulation Sie in den Kapiteln 11 und 12 kennen gelernt haben. Sie wissen nun bereits, wie Sie Daten in abstrakten und erweiterbaren XML-Dokumenten ablegen und diese über Transformationen z.B. in ein anderes Format überführen können. Was Ihnen jedoch noch fehlt, ist die Möglichkeit, diese Daten mit anderen zu teilen. Und genau mit diesem fehlenden Baustein befasst sich dieses Kapitel.

Bezug und Installation des API

Bevor Sie sich jedoch in die Implementierung von Webservices stürzen, benötigen Sie ein weiteres Mal ein ergänzendes API, welches nicht Bestandteil des Standard SDK ist. Die Installation ist genauso einfach wie in den vorangegangenen Kapiteln: Laden Sie sich die neueste Version des *Java Web Service Developer Pack (JWSDP)* von der Produkt-Homepage *http://java.sun.com/webservices/jwsdp/* herunter, entpacken Sie es in ein Verzeichnis Ihrer Wahl und binden Sie die benötigten Bibliotheken in den Classpath ein. Neben diesen enthält das Paket auch eine umfassende Dokumentation der Bibliotheken und Beispiele für die Verwendung des API.

13.2 XML – Remote Procedure Call (XML-RPC)

Kaum war die erste XML-Spezifikation veröffentlicht, entstand die Idee, auf Basis dieser Technologie weltumspannende Computernetzwerke zu realisieren. Einer der ersten Ansätze waren damals die XML-basierten *Remote Procedure Calls (XML-RPC)*. RPC gehört zum Sprachumfang von Java und beschreibt die Möglichkeit, Methoden (die in anderen Programmiersprachen häufig Prozeduren, *Procedures*, genannt werden) auf einem entfernten Rechner ausführen zu lassen und anschließend mit dem zurückgegebenen Ergebnis weiterarbeiten zu können.

Info

Was sich hinter den Kürzeln *Stub* und *Skeleton* verbirgt, haben Sie bereits in Kapitel 8 in Zusammenhang mit *Enterprise JavaBeans (EJB)* erfahren.

Beim XML-RPC werden die aufzurufende Methode und ihre Parameter einfach in ein abstraktes XML-Dokument verpackt und über das HTTP-Protokoll an den Empfänger verschickt. Dieser extrahiert die Daten, startet die entsprechende Prozedur und schickt Ihnen das Ergebnis der Berechnung auf dem gleichen Weg, natürlich wieder in Form eines XML-Dokuments, zurück.

Hierbei machen Sie sich das einfache Frage-Antwort-Prinzip des HTTP-Protokolls zunutze und eben aufgrund dieser einfachen Struktur hat es dieser sehr rudimentäre Standard zu einer stattlichen Anzahl von Implementierungen geschafft und ist auf einer großen Anzahl von Betriebssystemen und für viele Programmiersprachen verfügbar.

Mit der Zeit zeigte sich jedoch, dass in XML-RPC wichtige Sprachelemente fehlten, was zur Entwicklung des anschließend behandelten SOAP-Protokolls führte. So werden *XML Remote Procedure Calls* heute immer seltener in dieser »nackten« Form eingesetzt und zunehmend von SOAP und JAX-RPC (auch dazu später mehr) verdrängt. Da XML-RPC jedoch als Basis für die folgenden Protokolle dient, wollen wir ihm trotzdem einen eigenen Absatz widmen.

13.2.1 Grundlagen

Info

Bei den in Kapitel 2 besprochenen Requests, bei denen der Client lediglich ein Dokument vom Server abruft, war der Request-Body häufig leer. Dies wird sich in diesem Kapitel ändern. Es gibt jedoch auch für HTML-Seiten so genannte PUT-Requests, die dazu dienen, Dokumente auf einem Server abzulegen.

Wie Sie bereits wissen, besteht ein HTTP-Request aus drei Teilen (vgl. Kapitel 2):

1. Einer initialen Zeile, die unter anderem die verwendete Anfragemethode, die angeforderte Ressource und die unterstützte HTTP-Version enthält.
2. Einer beliebigen Anzahl von Request-Header, welche Meta-Informationen über den Client enthalten können
3. Gefolgt vom eigentlichen Inhalt des Request, in Form eines Markup-Dokuments

In Fall von Webservices handelt es sich nun statt einer HTML-Seite einfach um ein XML-Dokument. Da der Client bei XML-RPC typischerweise mit einem Servlet kommuniziert, welches den Dienst anschließend erbringt, dient die Definition einer Methodensignatur nur zu Dokumentationszwecken.

Info

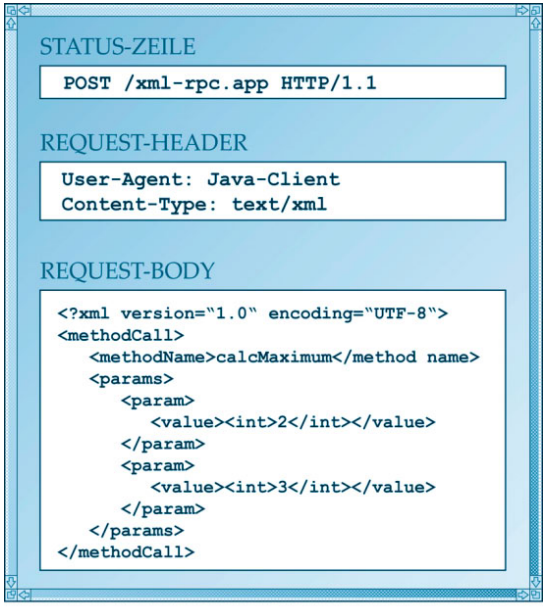
Über den `ServletInputStream` des Interface `javax.servlet.ServletRequest` können Sie das im Request enthaltene Dokument auslesen und beispielsweise über den Xerces-Parser in einen (J)DOM überführen.

Listing 13.1 Methodensignatur des angebotenen XML-RPC-Service

```
public static int calcMaximum(int value1, int value2);
```

Um diese »Methode« über XML-RPC mit den Parametern aufrufen zu können, könnte der XML-RPC Request beispielsweise folgende Form haben.

Abbildung 13.1 Aufbau eines XML-RPC Request



13.2.2 Aufbau eines XML-RPC-Dokuments

Wie Sie in Abbildung 13.1 sehen, ist der Aufbau eines XML-RPC-Dokuments sehr einfach gehalten. Das im Request-Body übertragene Dokument hat dabei immer folgenden schematischen Aufbau:

Listing 13.2 Aufbau eines Anfragedokuments (schematisch)

```
<methodCall>
   <methodName><!-- Name der aufgerufenen Methode --></methodName>
   <params>
      <param><!-- Erster Parameter der Methode --></param>
             <!-- ... weitere Parameter ... -->
   </params>
</methodCall>
```

Wenn die in diesem Beispiel verwendete Methode `calcMaximum`, wie es der Name schon suggeriert, einfach das Maximum der beiden Zahlen zurückliefert, würde das vom Server übermittelte Antwortdokument anschließend folgende Form haben:

Listing 13.3 XML-RPC-Antwortdokument

```
<methodResponse>
   <params>
      <param><value><int>3</int></value></param>
   </params>
</methodResponse>
```

Der einzige Unterschied zum Aufbau des Request-Dokuments besteht im Wurzelelement (`<methodResponse>`) und im Fehlen des Elements `<methodName>`.

13.2.3 Übergabe der Parameter

Wie Sie sehen, erfolgt die Übergabe von Objekten wie z.B. Parametern in der einen oder anderen Richtung immer nach dem gleichen Schema: So wird der Wert stets von einem `<value>`-Element umschlossen, welches dann wiederum ein Element für den Typ des Parameters enthält.

Listing 13.4 Aufbau eines Wertelements

```
...
   <!-- Zeigt an, dass es sich um einen Wert handelt -->
   <value>
      <int>3</int>
   </value>
...
```

Tabelle 13.1 zeigt die möglichen Basistypen des Protokolls.

Tabelle 13.1 Mögliche Basistypen für Werte

Tag	Beschreibung	Beispiel
`<int>` oder `<i4>`	Entspricht `Integer`	`<int>1234</int>`
`<double>`	Fließkommazahl	`<double>5.67</double>`
`<string>`	Zeichenkette	`<string>Java EE</string>`
`<boolean>`	Boolescher Wert als Zahl (0 = `false`, 1 = `true`)	`<boolean>1</boolean>`
`<base64>`	Eine nach base64 kodierte Bytefolge; eignet sich zum Übertragen serialisierter Objekte	`<base64>KT2d8=sl</base64>`
`<dateTime.iso8601>`	Datum und Uhrzeit im ISO-8601-Format	`<dateTime.iso8601>20060827T 10:10:53</dateTime.iso8601>`

Arrays

Natürlich können Sie auch Listen von Basistypen übergeben. Das Element `<array>` kann hierfür eine beliebige Folge von `<value>`-Elementen aufnehmen, die auch unterschiedlichen Typs sein können.

Komplexe Typen

Und selbstverständlich können wir unsere einfachen Basistypen auch zu komplexen Typen zusammensetzen, die wir benötigen, um etwa eine *JavaBean* zu übertragen. Die enthaltenen Properties unserer Bean werden dabei als `<member>` eines `<struct>`-Elements bezeichnet.

Listing 13.5
Definition einer JavaBean mit den Properties street und streetNumber

```
<struct>
   <member>
      <name>name</name>
      <value><string>Thomas Stark</string></value>
   </member>
   <member>
      <name>email</name>
      <value><string>thomas@masterclass.de</string></value>
   </member>
</struct>
```

13.2.4 Fehlerbehandlung

Natürlich kann es wie bei lokalen Methodenaufrufen auch bei Remote Procedure Calls zu Ausnahmen kommen, z.B. wenn ein übergebener Parameter nicht dem gültigen Wertebereich entspricht. In diesem Fall kann das Antwortdokument statt der Rückgabeparameter auch das Element `<fault>` (engl. Fehler) enthalten, welches ebenfalls einen Wert enthalten kann.

Listing 13.6
Rückgabe einer NumberFormatException

```
<methodResponse>
   <fault>
      <value>
         <struct>
            <member>
               <name>faultCode</name>
               <value><int>0815</int></value>
            </member>
            <member>
               <name>faultString</name>
               <value><string>EmailFormatException</string></value>
            </member>
         </struct>
      </value>
   </fault>
</methodResponse>
```

13.2.5 Zusammenfassung XML-RPC

XML-RPC ermöglicht die Bereitstellung von rudimentären Webservices, deren Implementierung aus Servlets besteht, die den Request parsen und eine entsprechende Antwort erzeugen. XML-RPC ist dabei sehr einfach gehalten und kann in vielen Programmiersprachen mit einfachen Bordmitteln realisiert werden. Das führte zur schnellen Verbreitung und großen Popularität dieses Standards.

Tipp

Ob Sie es nun glauben oder nicht: Mit dem jetzigen Wissen aus diesem Buch sind auch Sie in der Lage, einen XML-RPC-Server zu implementieren. Ein einfaches Servlet mit der Methode `doPost()` (vgl. Kapitel 3) und das Parsen des im `ServletInputStream` enthaltenen Dokuments (vgl. Kapitel 11) genügen. Schließlich würzen wir das Ganze mit etwas Basiswissen *Reflection* und schon steht der Service.

Da das verwendete HTTP-Protokoll von vielen Systemen unterstützt wird, lässt es sich gut durch vorhandene Schutzvorrichtungen wie *Firewalls* und *Proxies* tunneln und bildet mit seinem Frage-Antwort-Schema das Wesen eines Methodenaufrufs gut ab.

Dennoch lässt der XML-RPC-Standard gerade beim Einsatz in komplexen Systemen einige Wünsche offen, die zur Weiterentwicklung und damit zum SOAP-Protokoll führten. Die drei Hauptkritikpunkte sind dabei:

- Ungenaue Kodierung der Datentypen

  Die von XML-RPC unterstützten Datentypen haben mitunter starke Einschränkungen. So lassen sich Zeichenketten z.B. nur im 7-Bit-ASCII-Format übertragen, wodurch Sonderzeichen aufwändig umkodiert werden müssen. Außerdem legt das ISO-8601-Format für Datum und Uhrzeit keine Zeitzone fest.

- Aufwändiges Kodieren von binären Daten

  Da alle Daten im XML-RPC mit base64-Format übertragen werden müssen, werden diese auf der Senderseite erst aufwändig verpackt und müssen vom Empfänger dann wieder zurücktransformiert werden. Als unschöner Nebeneffekt blähen sich die Dokumente dabei um rund 30 Prozent auf.

- Fehlende Meta-Beschreibung

  Schließlich fehlte der Spezifikation die Angabe von Meta-Informationen. Das heißt, ein Programmierer, der auf einen entfernten Dienst zugreifen will, muss ganz genau wissen, wie die entsprechende Methode heißt und welche Parameter sie erwartet. Hierzu sind natürlichsprachige Beispieldokumente oder Dokumentationen nötig, die eine manuelle Kommunikation zwischen beiden Programmierern (dem des Clients und dem des Servers) voraussetzen.

Damit ist das XML-RPC-Protokoll zwar einfach und leicht zu implementieren, aber eben doch unvollständig und nicht für alle Formen von entfernten Prozeduraufrufen geeignet.

Info

XML-RPC wurde vor der Veröffentlichung intern bereits unter dem Namen SOAP entwickelt und da der Autor der *XML Remote Procedure Calls* (Dave Winer) auch an der Entwicklung der W3C-Spezifikation beteiligt war, lag diese Bezeichnung natürlich sehr nahe.

13.3 Simple Object Access Protocol (SOAP)

Nach dem durchschlagenden Erfolg und der Definition der oben beschriebenen Beschränkungen wurde über eine Erweiterung des XML-RPC-Protokolls nachgedacht, dessen Spezifikation schließlich dem W3C übertragen wurde. Das daraus resultierende *Simple Object Access Protocol (SOAP)* sollte den allgemeinen Austausch von Daten nach dem Vorbild von XML-RPC ermöglichen. Dies schließt die ausschließliche Übertragung in eine Richtung ohne zurückgegebene Antwort mit ein.

Diese Erweiterungen sollten zudem beliebige Transportmedien und -protokolle ermöglichen und die Beschränkung auf HTTP aufheben, obwohl dieses natürlich auch weiterhin die Grundlage für die meisten Webservices darstellt.

13.3.1 Die Idee hinter SOAP

Um die Beschränkung der Metadaten in XML-RPC zu umgehen, begannen einige Implementierungen, diese in Form von *Request-* bzw. *Response-Headern* zu übertragen. Da SOAP aber unabhängig von HTTP definiert werden sollte, wurde die Möglichkeit von Headern für Metadaten aufgegriffen und kurzer Hand einfach in das Protokoll integriert. Daraus entstand der so genannte *SOAP Envelope* (dt. Umschlag), der Träger beider Informationen sein sollte. Ein SOAP-Dokument hat damit immer folgende Form:

Listing 13.7
Schematischer Aufbau eines SOAP-Dokuments

```
<soap-env:Envelope
      xmlns:soap-env="http://schemas/xmlsoap.org/soap/envelope/">
   <soap-env:Header>
      <!-- Hier stehen die Header-Informationen -->
   </soap-env:Header>
   <soap-env:Body>
      <!-- Hier stehen die Nutz-Informationen -->
   </soap-env:Body>
</soap-env:Envelope>
```

13.3.2 SOAP und Java

Das nachfolgende Beispiel erzeugt eine SOAP-Nachricht, die von einem Webservice verwendet werden könnte, um Adressdaten an einen Server zu schicken oder von diesem zu erhalten.

Listing 13.8
Zu versendende Nutzdaten

```
<soap-env:Envelope
      xmlns:soap-env="http://schemas/xmlsoap.org/soap/envelope/">
   <soap-env:Header/>
   <soap-env:Body>
      <!-- Hier stehen die Nutz-Informationen -->
      <contact status="new">
         <name>Thomas Stark</name>
         <email>thomas@masterclass.de</email>
      </contact>
   </soap-env:Body>
</soap-env:Envelope>
```

Da es sich bei SOAP-Dokumenten um XML handelt, können Sie diese natürlich mit den in Kapitel 11 gezeigten Techniken aus einer Datei parsen oder z.B. mit einer Bibliothek wie JDOM von Hand erstellen. Sie können allerdings auch die von Sun bereitgestellten Basisklassen verwenden und lediglich den SOAP-Body füllen. Listing 13.9 zeigt Ihnen, wie Sie die in diesem Beispiel verwendete SOAP-Nachricht mithilfe des `javax.xml.soap`-API erstellen.

Listing 13.9
Ein einfaches SOAP-Dokument

```
package de.akdabas.javaee.ws;
import javax.xml.soap.*;

/**
 * Diese Klasse demonstriert das Erzeugen einer SOAP-Nachricht mit
 * Hilfe des API 'javax.xml.soap'
 */
public class SoapMessageBuilder {

   /**
    * Diese Methode dient zum Testen der Methode 'createMessage()'
    */
   public static void main(String[] args) throws Exception {
      SoapMessageBuilder smb = new SoapMessageBuilder();
      SOAPMessage message = smb.createMessage();
      message.writeTo(System.out);
   }

   /** Erzeugt eine SOAP-Message */
   public SOAPMessage createMessage() throws SOAPException {

      // Erzeugen der SOAP Message-Factory
      MessageFactory mf = MessageFactory.newInstance();

      // Erzeugen einer neuen SOAP Nachricht
      SOAPMessage message = mf.createMessage();

      // Zugriff auf den SOAP Body, der später die Nuetzdaten trägt
      SOAPPart part = message.getSOAPPart();
      SOAPEnvelope envelope = part.getEnvelope();
      SOAPBody body = envelope.getBody();

      // Erzeugen des Wurzelelementes '<contact>' der Nutzdaten
      Name root = envelope.createName("contact");

      // Einfügen des Elementes in den SOAP Body
      SOAPBodyElement rootElement = body.addBodyElement(root);

      // Hinzufügen des Attributes 'status="new"'
      Name status = envelope.createName("status");
      rootElement.addAttribute(status, "new");

      // Aufbau und Einfügen des Elementes '<name>'
      Name contactName = envelope.createName("name");
```

Listing 13.9 (Forts.) Ein einfaches SOAP-Dokument

```
        SOAPElement soapElementName =
           rootElement.addChildElement(contactName);
        soapElementName.addTextNode("Thomas Stark");

        // Aufbau und Einfügen des Elementes '<email>'
        Name contactEmail = envelope.createName("email");
        SOAPElement soapElementEmail =
           rootElement.addChildElement(contactEmail);
        soapElementEmail.addTextNode("thomas@masterclass.de");

        return message;
     }
}
```

13.4 Erstellen eines einfachen Webservice

Es existieren viele Möglichkeiten, einen Webservice bereitzustellen, über die *Web Service Description Language* (WSDL) zu beschreiben und z.B. über UDDI-Verzeichnisse verfügbar zu machen. Da die hierfür benötigten Grundlagen jedoch den Rahmen dieses Einführungskapitels sprengen würden, beschränken wir uns in diesem Beispiel auf die einfachstmögliche Konfiguration, während Ihnen die Absätze 13.6 und 13.7 die Technologien WSDL und UDDI näher bringen.

Um sich zusätzliche Konfigurationsdateien und Deployment Descriptors zu sparen, können Sie sich auch bei den Webservices verschiedener Annotationen bedienen, die dazu dienen, eine Klasse und die von ihr bereitgestellten Methoden zu kennzeichnen. Listing 13.10 zeigt einen solchen Service, der eine über die Methode `invoke()` empfangene SOAP-Nachricht auf der Kommandozeile ausgibt. Natürlich könnte sich die Methode auf das Empfangen von SOAP-Nachrichten beschränken, um aber zu demonstrieren, dass diese auch in der anderen Richtung verschickt werden können, sendet die Methode auch eine SOAP-Nachricht an den Client zurück, der Einfachheit halber einfach das gleiche Dokument.

Listing 13.10 Ein einfacher SOAP-Service

```
package de.akdabas.javaee.ws;
import javax.ejb.Stateless;
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.xml.soap.SOAPMessage;

@Stateless
@WebService( name="TestService" )

/**
 * Dieser Service nimmt eine SOAP-Nachricht entgegen, gibt sie aus
 * und sendet die Nachricht anschließend wieder zurück
 */
public class SoapService {

   @WebMethod
   public SOAPMessage invoke(SOAPMessage message) {
      try {
         // Ausgabe der empfangen Nachricht auf der Kommandozeile
         message.writeTo(System.out);
      } catch (Exception exc) {
         exc.printStackTrace();
```

```
        }
        // Zurücksenden der Nachricht als Bestätigung
        return message;
    }
}
```

Listing 13.10 (Forts.)
Ein einfacher SOAP-Service

Um eine Klasse über Annotationen als Webservice zu deklarieren, markieren Sie diese einfach mit den Annotationen `@Stateless` und `@WebService`. Der JBoss erkennt diese Annotationen während des Startvorgangs und wird den Webservice über eine Standard-URL bereitstellen.

Info

Über die später in diesem Kapitel vorgestellten Technologien WSDL und UDDI sind sehr feingranulare Konfigurationen für Webservices möglich. Diese können über weitere Annotationen oder zusätzliche XML-Dateien beschrieben werden. Die in diesem Kapitel vorgestellten Beispiele beschränken sich auf das absolut notwendige Minimum.

Anschließend sind alle mit der Annotation `@WebMethod` gekennzeichneten Methoden für den Client sichtbar und können über diese ausgeführt werden.

Webservices vs. Enterprise JavaBeans

Listing 13.10 erinnert Sie vielleicht an die Enterprise JavaBeans (EJB), die Sie in Kapitel 8 kennen gelernt haben, und tatsächlich haben diese beiden Technologien viel gemein: Sie stellen einen Dienst in Form einer Java-Klasse bereit, der anschließend von verschiedenen Clients verwendet werden kann. Lediglich die zum Einsatz kommende Übertragungstechnologie unterscheidet sich. Während EJBs mit *Remote Method Invocation (RMI)* arbeiten und damit im Prinzip auf Java-Clients beschränkt sind, kommt bei Webservices XML zum Einsatz, welches in der Regel via HTTP übertragen wird.

Es gibt sogar eine Entsprechung für die asynchronen Aufrufe der Message Driven Beans aus Kapitel 9.

```
...
    @Oneway
    @WebMethod
    public void asynchonWebserviceMethod(SOAPMessage message) {...}
...
```

Listing 13.11
Definieren eines asynchronen Methodenaufrufs

Über die Annotation `@Oneway` definieren Sie, dass diese Webservice-Methode lediglich Nachrichten konsumiert und der Client deshalb auch nicht auf eine Antwort warten muss.

Tipp

Wenn Ihre SOAP-Nachrichten über HTTP versendet werden, könnten Sie diese natürlich auch mit einem einfachen Servlet verarbeiten. Das *Java API for XML Messaging (JAXM)* stellt Ihnen hierfür sogar eine Basisklasse bereit.

13.5 Einen Webservice ansprechen

Auch das Ansprechen eines Webservice ist mit Java und Annotationen kein Problem. Der folgende Client erzeugt mithilfe des in Listing 13.9 vorgestellten `SoapMessageBuilder` ein SOAP-Dokument, sendet dieses an den Service und gibt die zurückgesandte Antwort aus.

Listing 13.12 Ein Webservice-Client

```
package de.akdabas.javaee.ws;
import javax.xml.ws.Service;
import javax.xml.ws.WebServiceRef;
import javax.xml.soap.SOAPMessage;

/** Ein einfacher Client für eine SOAP Nachricht */
public class SoapClient {

   // Referenz auf den Webservice
   @WebServiceRef(
      wsdlLocation = "http://java:8080/kap13/SoapService?wsdl"
   )
   static Service service;

   /**
    * Diese Methode dient zum Testen der Methode 'doInvokeService()'
    */
   public static void main(String[] args) {
      SoapClient client = new SoapClient();
      client.doInvokeService();
   }

   /**
    * Erzeugt mit Hilfe des 'SoapMessageBuilder' eine neue SOAP
    * Nachricht, sendet diese an den 'SoapService' und gibt die
    * empfangene Antwort aus.
    */
   public void doInvokeService() {
      try {
         // Erstellen eines neuen Message Builders und Erzeugen
         // einer SOAP Nachricht
         SoapMessageBuilder msgBuilder = new SoapMessageBuilder();
         SOAPMessage message = msgBuilder.createMessage();

         // Referenzieren des Services
         SoapService ws = service.getPort(SoapService.class);

         // Sender der Nachricht und Empfang der Antwort
         SOAPMessage response = ws.invoke(message);

         // Ausgabe der Antwort-Nachricht
         response.writeTo(System.out);
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

Über die Annotation `@WebServiceRef` verknüpfen Sie eine Variable vom Typ `Service` mit einem Webservice, wobei das hier angegebene und im nächsten Absatz besprochene WSDL-Dokument in obigem Beispiel vom JBoss automatisch erzeugt wurde. Der Service verbindet Ihren Client über die Methode `getPort()` mit einer entfernten Instanz der Service-Klasse `SoapService` und ermöglicht es dem Client, auf diese zuzugreifen.

Webservices und SOAP-Nachrichten

Ihnen ist vielleicht aufgefallen, dass Client und Service in den vorangegangen Listings unabhängig von den zuvor besprochenen SOAP-Nachrichten waren und diese lediglich als (Rückgabe-)Parameter definierten. In der Tat ist es so, dass Webservices nicht auf den Austausch von SOAP-Nachrichten beschränkt sind, sondern jede Methode mit beliebigen Parametern und Rückgabewerten über Webservices angesprochen werden kann. Sie muss lediglich in einer `@WebService`-Klasse definiert sein und über eine `@WebMethod`-Annotation verfügen.

Dieser Abschnitt sollte Sie mit der Technologie der Webservices vertraut machen und Ihnen die Entwicklung eines rudimentären Service und Clients zeigen. Die nächsten beiden Abschnitte stellen Ihnen zwei weitere essenzielle Technologien für Webservices vor, die die Beschreibung und Registrierung von Webservices ermöglichen.

13.6 Webservice Description Language (WSDL)

Der Client in Listing 13.12 konnte den entfernten Service zwar auf dem Server ausführen, allerdings benötigt er dafür ebenfalls die Klasse `SoapService`. Analog zu den Session Beans könnten Sie sich hier zwar mit einem `Interface` behelfen, von echter Interoperabilität zwischen verschiedenen Programmiersprachen kann allerdings auch dann noch keine Rede sein, weil Sie auch mit einem Java-Interface auf Java-Clients beschränkt wären.

Was Ihnen fehlt, ist eine allgemeingültige Definition, die es Ihnen gestattet, Ihren Service und die bereitgestellten Methoden sowie ihre Parameter und Rückgabewerte zu beschreiben. Diese Lücke versucht die *Webservice Description Language (WSDL)* zu schließen.

Die Webservice Description Language ist dabei eine aus einem gemeinsamen Projekt zwischen IBM und Microsoft hervorgegangene Spezifikation, die anschließend schnell von anderen Anbietern übernommen worden ist. Sie kann als Pendant zur CORBA Interface Definition Language (IDL) verstanden werden und ermöglicht es Ihnen, alle für den Aufruf des Webservice benötigten Informationen abzulegen.

Die WSDL-Spezifikation umfasst dabei folgende vier Bereiche:

- Schnittstellendefinition aller zur Verfügung gestellten Funktionen
- Datentypdefinition aller ausgetauschten Dokumente
- Informationen über die Suche nach bestimmten Diensten
- Informationen über die zu verwendenden Transportprotokolle

Eine solche Beschreibung besteht dabei aus einem XML-Dokument, welches z.B. über verschiedene Kommandozeilenwerkzeuge erzeugt werden kann.

13.7 Universal Description and Discovery Interface (UDDI)

Auch dieser Industriestandard geht auf eine Initiative zwischen IBM und Microsoft zurück. Doch während WSDL beschreibt, »wie« Sie die verschiedenen Webservices verwenden können, geht es beim *Universal Description and Discovery Interface (UDDI)* um die Beschreibung, »wo« die verschiedenen Dienste angeboten werden.

Tim Berners-Lee und die anderen »Erfinder des Internets« haben zwar Hyperlinks vorgesehen, um verschiedene Dokumente miteinander zu verknüpfen, doch fehlt die Möglichkeit einer zentralen Registrierung. Als Folge dessen lassen wir das Web jeden Tag von riesigen Suchmaschinen durchkämmen, die mit unterschiedlichen Ansätzen versuchen, die gefundenen Seiten zu indizieren.

Info

Auch das Anlegen von Datensätzen sowie die Suche nach einem Dienst in einer UDDI Registry erfolgen über Webservices.

UDDI soll nun der zentralen Registrierung von Webservices dienen und eine effiziente Suche in dieser ermöglichen. Eine UDDI-Registry ist damit vergleichbar mit einem Namens- oder Verzeichnisdienst für Webservices. Dabei registrieren Anbieter von Webservices ihren Dienst an einer UDDI-Registry, die anschließend von Clients genutzt werden kann, um diese Dienste z. B. anhand von Schlüsselwörtern zu finden.

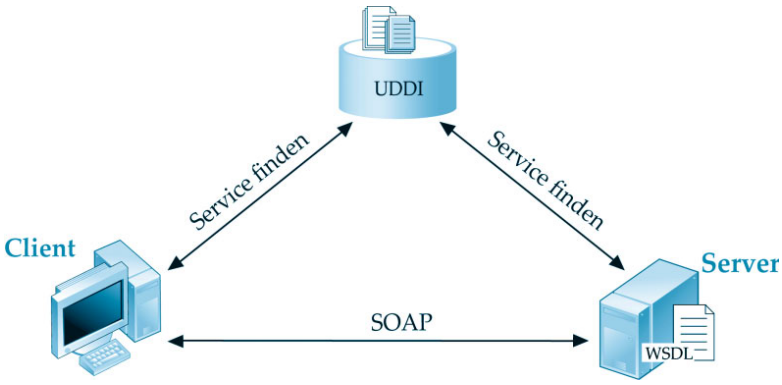


Abbildung 13.2
Zusammenspiel von SOAP, UDDI und WSDL

UDDI-Registries können sowohl auf ein Firmennetzwerk beschränkt als auch öffentlich zugänglich sein. Öffentliche UDDI-Datenbanken werden z. B. betrieben von IBM, HP, Microsoft und SAP.

13.8 Zusammenfassung

Webservices ermöglichen es Ihnen, viele der in den vorangegangenen Kapiteln vorgestellten Techniken, wie verteilte Applikationen und eine lose Kopplung der beteiligten Systeme, auch auf andere Plattformen und Programmiersprachen auszudehnen und die Kommunikation mit diesen zu vereinfachen.

Je nach Bedarf können Sie dabei dokumentzentriert arbeiten und Informationen in Form von SOAP-Nachrichten austauschen oder sich auch an klassischen Remote Procedure Calls orientieren und von SOAP unabhängige Methoden bereitstellen. Um Ihren Service anschließend allgemein verfügbar zu machen, können Sie sich der Web Service Description Language bedienen, deren XML-Dokumente alle benötigten Meta-Informationen enthalten.

UDDI ist wiederum ein weiterer, allgemeiner Webservice, der keine andere Aufgabe als das Registrieren von anderen Webservices hat. Dieser Dienst kann anschließend von Clients verwendet werden, um beispielsweise einen bestimmten Webservice und dessen WSDL-Descriptor zu lokalisieren.

Die Vorteile von Webservices liegen vor allem in der hohen Kompatibilität und der losen Kopplung zwischen Client und Server. Webservices sind sprach- und plattformunabhängig und lassen sich dank der verwendeten Protokolle (HTTP) auch durch Router und Firewalls verwenden. Außerdem erlaubt diese Technologie die Mischung von synchroner und asynchroner Kommunikation.

Auf der anderen Seite ist diese Technologie im Ganzen noch sehr jung und in verschiedenen Bereichen, wie Security, Transaktionsunterstützung oder Ausfallsicherheit, noch etwas unausgereift. Auch ihre Verbreitung lässt noch etwas auf sich warten, was unter anderem daran liegt, dass Webservices aufgrund der Serialisierung der Informationen in XML-Dokumente einen Overhead haben und in der Regel langsamer sind als ihre Pendants wie RMI oder CORBA.

Nichtsdestotrotz sind Webservices auf dem Vormarsch und werden früher oder später einen festen Platz in der Riege der Kommunikationstechnologien einnehmen.

13.8.1 Allgemeine Abschlussbemerkungen

Viele der in diesem Buch vorgestellten Technologien haben sich mit der Bereitstellung von Diensten beschäftigt, die anschließend von anderen Komponenten der Anwendung oder Clients verwendet werden können. Um dieses Buch abzuschließen, möchte ich den Service-Gedanken noch einmal aufgreifen und einige Punkte ansprechen, die bei der Entwicklung eigener Dienste bedacht werden sollten.

Entwurf einer serviceorientierten Architektur

Die folgenden Punkte behandeln Aspekte des Designs einer Service-Architektur

- Definieren Sie einen klaren Gültigkeitsbereich und Zweck für Ihre Services.

  Um einen Dienst verwenden zu können, müssen dessen Verwendung, Wert und Zweck klar und verständlich sein. Außerdem können Sie an dieser Stelle bereits Schwachstellen und Beschränkungen Ihres Service identifizieren.

- Verwenden Sie existierende Services wieder.

  Vermeiden Sie es, durch redundante Dienste das Rad immer wieder neu zu erfinden, und versuchen Sie, Ihre Dienste so zu entwickeln, dass diese von vielen Clients gemeinsam verwendet werden können.

  Entwerfen Sie die Services orthogonal voneinander und achten Sie auf funktionale Überschneidungen.

- »Small is beautiful« – beginnen Sie in kleinen Schritten.

  Gerade wenn es darum geht, gewachsene Systemlandschaften durch den Einsatz von wiederverwendbaren Diensten flexibler zu gestalten, ist die Versuchung oft groß, alles umstellen und ersetzen zu wollen.

  Eine große Infrastruktur kann jedoch in den meisten Fällen nicht von heute auf morgen umgestellt werden. Beginnen Sie mit einfachen und möglichst allgemeinen Diensten, wie dem Logging oder der Benutzerauthentifizierung. Auf diese Weise können Sie Erfahrungen mit der Technologie sammeln, ohne von fachlichen Anforderungen erschlagen zu werden.

- Verwenden Sie die Technologie passend zur Aufgabe.

  Nicht jede Technologie ist für jedes Szenario gleichermaßen geeignet. Wägen Sie deshalb unterschiedliche Ansätze gegeneinander ab und ziehen Sie auch eine Mischung verschiedener Technologien in Betracht.

Entwurf eines eigenen Dienstes

Beim konkreten Design eines Service sollten Sie folgende Aspekte berücksichtigen:

- Definieren Sie Ihre Services offen und gemeinsam nutzbar.

  Wiederverwendbarkeit erhöht den Wert Ihres Service und durch eine generische Gestaltung des Dienstes schließt dieser auch potenzielle Nutzungsszenarien ein.

- Kapseln Sie darunter liegende Schichten und verwenden Sie Standards wo möglich.

  Definieren Sie Ihren Dienst unabhängig von der verwendeten Infrastruktur (JBoss), dem zugrunde liegenden Komponentenmodell

(CORBA, EJB, JMS) oder den verwendeten Protokollen (HTTP), da sonst möglicherweise auch der Client auf die Schnittstellen dieser Dienste angewiesen ist.

Vermeiden Sie Abhängigkeiten zu anderen Applikationen, die die Verwendung Ihres Dienstes einschränken.

- Kapseln Sie Services voneinander.

  Auch wenn zwei Ihrer Dienste zur gleichen Teilanwendung gehören und aufeinander angewiesen sind, sollten diese nur über öffentliche Schnittstellen miteinander kommunizieren und keine Details der darunter liegenden Implementierung verwenden.

  Dabei sollten zusammenarbeitende Dienste dasselbe Typsystem verwenden, da Transformationen zur Laufzeit unnötige Ressourcen kosten.

- Beachten Sie die Performance und den Overhead der verwendeten Technologie

  Auch wenn ein Client durch den verwendeten Stub scheinbar auf einem lokalen Objekt arbeitet, dürfen Sie nie vergessen, dass es sich bei den meisten Diensten um entfernte Methodenaufrufe handelt und diese durch das Netzwerk, die Synchronisation, die Serialisierung etc. verzögert werden.

  Wenn ein Workflow viele kleine Einzelschritte erfordert, kapseln Sie diesen z.B. mit einer auf dem Server arbeitenden Session Bean.

- Entwerfen Sie Ihre Services möglichst zustandslos und ohne Seiteneffekte.

  Verwaltung kostet wertvolle Ressourcen des Frameworks. Versuchen Sie deshalb, Ihre Dienste zustandslos zu halten und jeden Client anonym zu lassen.

- Finden Sie einen Kompromiss hinsichtlich der Granularität Ihrer Services.

  Wenn zu viele Funktionalitäten von einem einzigen Service abgedeckt werden, sinkt dessen Wiederverwendbarkeit. Eine zu feine Unterteilung führt hingegen zu einem hohen Kommunikations- und Verwaltungsaufwand. Einen guten Kompromiss stellt häufig die fachliche Trennung dar.

Was auch immer Sie aus diesem Buch mitnehmen – ich hoffe, es ist mir gelungen, Ihnen zu zeigen, dass es häufig mehr als einen Weg der Realisierung gibt und dass wir uns stets den Blick über den Tellerrand hinaus bewahren sollten.

Ich hoffe, die Lektüre dieses Buchs war lehrreich und hat Ihnen Freude bereitet. Ich wünsche Ihnen alles Gute für Ihre zukünftigen Projekte und viel Spaß beim Einsatz der hier vorgestellten Technologien.

J2EE Software License

Sun Microsystems, Inc. ("Sun") ENTITLEMENT for SOFTWARE

Licensee/Company: Entity receiving Software.

Effective Date: Date of delivery of the Software to You.

Software: Java Platform, Enterprise Edition 5 SDK, which includes the following: (i) Java 2 Platform,

Standard Edition Development Kit 5.0, (ii) Java Platform, Enterprise Edition 5 Samples, (iii) Sun Java System Application Server Platform Edition 9 and (iv) Java BluePrints Solutions Catalog for Java EE 5.

License Term: Perpetual (subject to termination under the SLA).

Licensed Unit: Software Copy.

Licensed unit Count: Unlimited.

Permitted Uses:

1. You may reproduce and use the Software for Individual, Commercial, Service Provider and Research and Instructional Use only for the purposes of designing, developing, testing, and running Your applets and application ("Programs").

2. Subject to the terms and conditions of this Agreement and restrictions and exceptions set forth in the Software's documentation, You may reproduce and distribute portions of Software identified as a redistributable in the documentation ("Redistributable"), provided that:

(a) you distribute Redistributable complete and unmodified and only bundled as part of Your Programs,

(b) your Programs add significant and primary functionality to the Redistributable,

(c) you distribute Redistributable for the sole purpose of running your Programs,

(d) you do not distribute additional software intended to replace any component(s) of the Redistributable,

(e) you do not remove or alter any proprietary legends or notices contained in or on the Redistributable.

(f) you only distribute the Redistributable subject to a license agreement that protects Sun's interests consistent with the terms contained in this Agreement, and

(g) you agree to defend and indemnify Sun and its licensors from and against any damages, costs, liabilities, settlement amounts and/or expenses (including attorneys' fees) incurred in connection with any claim, lawsuit or action by any third party that arises or results from the use or distribution of any and all Programs and/or Redistributable.

3. Java Technology Restrictions. You may not create, modify, or change the behavior of, or authorize your licensees to create, modify, or change the behavior of, classes, interfaces, or subpackages that are in any way identified as "java", "javax", "sun" or similar convention as specified by Sun in any naming convention designation.

Sun Microsystems, Inc. ("Sun")

SOFTWARE LICENSE AGREEMENT

READ THE TERMS OF THIS AGREEMENT ("AGREEMENT") CAREFULLY BEFORE OPENING SOFTWARE MEDIA PACKAGE. BY OPENING SOFTWARE MEDIA PACKAGE, YOU AGREE TO THE TERMS OF THIS AGREEMENT. IF YOU ARE ACCESSING SOFTWARE ELECTRONICALLY, INDICATE YOUR ACCEPTANCE OF THESE TERMS BY SELECTING THE "ACCEPT" BUTTON AT THE END OF THIS AGREEMENT. IF YOU DO NOT AGREE TO ALL OF THE TERMS, PROMPTLY RETURN THE UNUSED SOFTWARE TO YOUR PLACE OF PURCHASE FOR A REFUND OR, IF SOFTWARE IS ACCESSED ELECTRONICALLY, SELECT THE "DECLINE" (OR "EXIT") BUTTON AT THE END OF THIS AGREEMENT. IF YOU HAVE SEPARATELY AGREED TO LICENSE TERMS ("MASTER TERMS") FOR YOUR LICENSE TO THIS SOFTWARE, THEN SECTIONS 1-5 OF THIS AGREEMENT ("SUPPLEMENTAL LICENSE TERMS") SHALL SUPPLEMENT AND SUPERSEDE THE MASTER TERMS IN RELATION TO THIS SOFTWARE.

1. Definitions.

(a) "Entitlement" means the collective set of applicable documents authorized by Sun evidencing your obligation to pay associated fees (if any) for the license, associated Services, and the authorized scope of use of Software under this Agreement.

(b) "Licensed Unit" means the unit of measure by which your use of Software and/or Service is licensed, as described in your Entitlement.

(c) "Permitted Use" means the licensed Software use(s) authorized in this Agreement as specified in your Entitlement. The Permitted Use for any bundled Sun software not specified in your Entitlement will be evaluation use as provided in Section 3.

(d) "Service" means the service(s) that Sun or its delegate will provide, if any, as selected in your Entitlement and as further described in the applicable service listings at www.sun.com/service/servicelist.

(e) "Software" means the Sun software described in your Entitlement. Also, certain software may be included for evaluation use under Section 3.

(f) "You" and "Your" means the individual or legal entity specified in the Entitlement, or for evaluation purposes, the entity performing the evaluation.

2. License Grant and Entitlement.

Subject to the terms of your Entitlement, Sun grants you a nonexclusive, nontransferable limited license to use Software for its Permitted Use for the license term. Your Entitlement will specify (a) Software licensed, (b) the Permitted Use, (c) the license term, and (d) the Licensed Units.

Additionally, if your Entitlement includes Services, then it will also specify the (e) Service and (f) service term.

If your rights to Software or Services are limited in duration and the date such rights begin is other than the purchase date, your Entitlement will provide that beginning date(s).

The Entitlement may be delivered to you in various ways depending on the manner in which you obtain Software and Services, for example, the Entitlement may be provided in your receipt, invoice or your contract with Sun or authorized Sun reseller. It may also be in electronic format if you download Software.

3. Permitted Use.

As selected in your Entitlement, one or more of the following Permitted Uses will apply to your use of Software. Unless you have an Entitlement that expressly permits it, you may not use Software for any of the other Permitted Uses. If you don't have an Entitlement, or if your Entitlement doesn't cover additional software delivered to you, then such software is for your Evaluation Use.

(a) Evaluation Use. You may evaluate Software internally for a period of 90 days from your first use.

(b) Research and Instructional Use. You may use Software internally to design, develop and test, and also to provide instruction on such uses.

(c) Individual Use. You may use Software internally for personal, individual use.

(d) Commercial Use. You may use Software internally for your own commercial purposes.

(e) Service Provider Use. You may make Software functionality accessible (but not by providing Software itself or through outsourcing services) to

your end users in an extranet deployment, but not to your affiliated companies or to government agencies.

4. Licensed Units.

Your Permitted Use is limited to the number of Licensed Units stated in your Entitlement. If you require additional Licensed Units, you will need additional Entitlement(s).

5. Restrictions.

(a) The copies of Software provided to you under this Agreement are licensed, not sold, to you by Sun. Sun reserves all rights not expressly granted. (b) You may make a single archival copy of Software, but otherwise may not copy, modify, or distribute Software. However if the Sun documentation accompanying Software lists specific portions of Software, such as header files, class libraries, reference source code, and/or redistributable files, that may be handled differently, you may do so only as provided in the Sun documentation. (c) You may not rent, lease, lend or encumber Software. (d) Unless enforcement is prohibited by applicable law, you may not decompile, or reverse engineer Software. (e) The terms and conditions of this Agreement will apply to any Software updates, provided to you at Sun's discretion, that replace and/or supplement the original Software, unless such update contains a separate license. (f) You may not publish or provide the results of any benchmark or comparison tests run on Software to any third party without the prior written consent of Sun. (g) Software is confidential and copyrighted. (h) Unless otherwise specified, if Software is delivered with embedded or bundled software that enables functionality of Software, you may not use such software on a stand-alone basis or use any portion of such software to interoperate with any program(s) other than Software. (i) Software may contain programs that perform automated collection of system data and/or automated software updating services. System data collected through such programs may be used by Sun, its subcontractors, and its service delivery partners for the purpose of providing you with remote system services and/or improving Sun's software and systems. (j) Software is not designed, licensed or intended for use in the design, construction, operation or maintenance of any nuclear facility and Sun and its licensors disclaim any express or implied warranty of fitness for such uses. (k) No right, title or interest in or to any trademark, service mark, logo or trade name of Sun or its licensors is granted under this Agreement.

6. Term and Termination.

The license and service term are set forth in your Entitlement(s). Your rights under this Agreement will terminate immediately without notice from Sun if you materially breach it or take any action in derogation of Sun's and/or its licensors' rights to Software. Sun may terminate this Agreement should any Software become, or in Sun's reasonable opinion likely to become, the subject of a claim of intellectual property infringement or trade secret misappropriation. Upon termination, you will cease

use of, and destroy, Software and confirm compliance in writing to Sun. Sections 1, 5, 6, 7, and 9-15 will survive termination of the Agreement.

7. Java Compatibility and Open Source.

Software may contain Java technology. You may not create additional classes to, or modifications of, the Java technology, except under compatibility requirements available under a separate agreement available at www.java.net.

Sun supports and benefits from the global community of open source developers, and thanks the community for its important contributions and open standards-based technology, which Sun has adopted into many of its products.

Please note that portions of Software may be provided with notices and open source licenses from such communities and third parties that govern the use of those portions, and any licenses granted hereunder do not alter any rights and obligations you may have under such open source licenses, however, the disclaimer of warranty and limitation of liability provisions in this Agreement will apply to all Software in this distribution.

8. Limited Warranty.

Sun warrants to you that for a period of 90 days from the date of purchase, as evidenced by a copy of the receipt, the media on which Software is furnished (if any) will be free of defects in materials and workmanship under normal use. Except for the foregoing, Software is provided "AS IS". Your exclusive remedy and Sun's entire liability under this limited warranty will be at Sun's option to replace Software media or refund the fee paid for Software. Some states do not allow limitations on certain implied warranties, so the above may not apply to you. This limited warranty gives you specific legal rights. You may have others, which vary from state to state.

9. Disclaimer of Warranty.

UNLESS SPECIFIED IN THIS AGREEMENT, ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT THESE DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

10. Limitation of Liability.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO THE USE OF OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN ADVISED OF THE POSSIBIL-

ITY OF SUCH DAMAGES. In no event will Sun's liability to you, whether in contract, tort (including negligence), or otherwise, exceed the amount paid by you for Software under this Agreement. The foregoing limitations will apply even if the above stated warranty fails of its essential purpose. Some states do not allow the exclusion of incidental or consequential damages, so some of the terms above may not be applicable to you.

11. Export Regulations.

All Software, documents, technical data, and any other materials delivered under this Agreement are subject to U.S. export control laws and may be subject to export or import regulations in other countries. You agree to comply strictly with these laws and regulations and acknowledge that you have the responsibility to obtain any licenses to export, re-export, or import as may be required after delivery to you.

12. U.S. Government Restricted Rights.

If Software is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in Software and accompanying documentation will be only as set forth in this Agreement; this is in accordance with 48 CFR 227.7201 through 227.7202-4 (for Department of Defense (DOD) acquisitions) and with 48 CFR 2.101 and 12.212 (for non-DOD acquisitions).

13. Governing Law.

Any action related to this Agreement will be governed by California law and controlling U.S. federal law. No choice of law rules of any jurisdiction will apply.

14. Severability.

If any provision of this Agreement is held to be unenforceable, this Agreement will remain in effect with the provision omitted, unless omission would frustrate the intent of the parties, in which case this Agreement will immediately terminate.

15. Integration.

This Agreement, including any terms contained in your Entitlement, is the entire agreement between you and Sun relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification of this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

Please contact Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, California 95054 if you have questions.

Stichwortverzeichnis

K

L

M

N

O

P

Q

R

S

T

Z