1 Einleitung

1.1 JDBC

Java DataBase Connectivity bzw. *JDBC*™ bezeichnet eine *Programmierschnittstelle* (Application Programming Interface oder API), über die man in Java-Programmen Informationen aus Datenbanksystemen verarbeiten, d.h. suchen, anzeigen, erzeugen und verändern kann. Für die Formulierung entsprechender Anweisungen an das Datenbanksystem wird gewöhnlich die standardisierte Datenbanksprache SQL verwendet. JDBC setzt damit auf einer recht tiefen sprachlichen Ebene an und hat dementsprechend schlichte Fähigkeiten im Vergleich zu den meist sehr anspruchsvollen Datenbankentwicklungswerkzeugen. Deshalb wird JDBC auch als „low-level" oder „call level" SQL-Schnittstelle für die Java-Plattform bezeichnet.
JDBC besteht aus folgenden Komponenten:

- Datenbanktreibern, die den Anschluss von Java-Anwendungen an Datenbanksysteme wie DB/2, Oracle, MS ACCESS oder MySQL ermöglichen, und
- das Paket `java.sql`, bestehend aus einem Manager für diese Treiber, Interfaces als Schnittstellen zu den Treibern sowie Hilfsklassen für Datum, Uhrzeit und gültige JDBC-Typen.

Weiters gibt es ein optionales Paket `javax.sql` („standard eXtensions") mit Techniken zur Unterhaltung von Verbindungs-Pools, für verteilte Transaktionen usw. Solche Techniken erfordern zusätzlich Laufzeitumgebungen etwa in Form von Applikations-Servern und liegen auch aus diesem Grunde außerhalb der Thematik des Buches, das auf die Darstellung der Grundlagen des Datenbankzugriffs mit Java-Techniken konzentriert ist.
JDBC-Treiber sind datenbank- und herstellerabhängige Java-Klassen, die sich dem Anwender aber als Implementierungen von JDBC-Interfaces des Pakets `java.sql` einheitlich präsentieren. Sie dienen dem Zweck, Java-Programmen herstellerneutrale, also von der speziellen Datenbank weitgehend unabhängige Programmierschnittstellen (APIs) anzubieten. Demgemäß sind alle JDBC-Treiber Implementierungen der Interfaces des Pakets `java.sql` mit Klassen wie `Connection`, `Statement`, `ResultSet` etc. Auch die

Treiber selbst sind zu Paketen geschnürt, beispielsweise der dem Java 2 SDK beiliegende ODBC-Treiber zum Paket (Package) `sun.jdbc.odbc` oder das Paket `oracle.jdbc` für Oracle-Datenbanken.
Neben den Schnittstellenfestlegungen verfügt `java.sql` noch über einen Treibermanager, mit dem Treiberobjekte verwaltet werden können. Objekte von JDBC-konformen Treiberklassen registrieren sich beim Treibermanager des Java-Programms stets selbst. Jede JDBC-Applikation hat genau einen solchen Treibermanager (auch dann, wenn der Manager umgangen und, was ohne weiteres möglich ist, direkt mit den Treibern gearbeitet wird). Der Treibermanager verwaltet lediglich die Treiberobjekte selbst, nicht deren Verbindungen zu Datenbanksystemen. Wird er verwendet, so werden über ihn zwar die Verbindungen zu den Datenbanken hergestellt, die darauf folgenden Datenbankanfragen werden aber direkt über die Treiber abgewickelt. (Dabei werden als Typen die JDBC-Interfaces und nicht die von Treiber zu Treiber variierenden Treiberklassen verwendet, also etwa `ResultSet` statt z.B. `OdbcJdbcResultSet` oder `MsqlResultSet`.)
Zusammenfassend ergibt sich daraus etwa eine Struktur wie in Abb. 1-1 veranschaulicht:

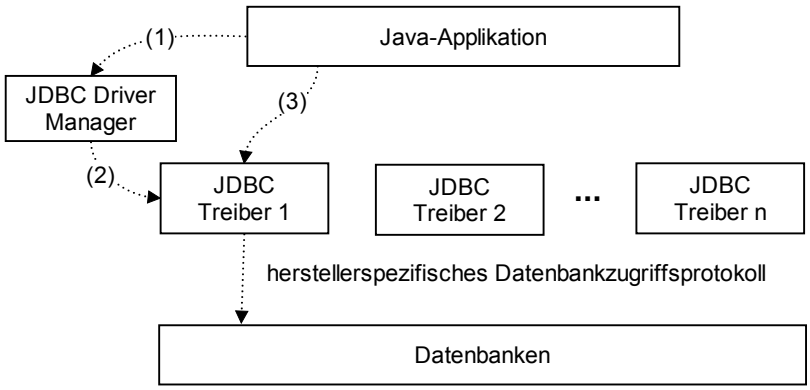

Abb. 1-1: JDBC-Schichten

Gezeigt sind beispielhaft (1) die Registrierung eines Treiberobjekts, (2) die Verbindungsaufnahme mit einer Datenbank über den Treibermanager und (3) die Abwicklung der Datenbankmanipulationen über den Treiber direkt, d.h. über das Objekt vom Typ `Connection`, das bei der Verbindungsaufnahme erzeugt wurde.

1.1.1 JDBC-Konformität

Treiber können sich dann als JDBC `COMPLIANT`™, d.h. JDBC-konform bezeichnen, wenn sie SUNs Konformitätstest bestehen und damit u.a. mindestens dem sog. *ANSI/ISO*[1] *SQL/92 Entry Level* genügen. JDBC-Konformität garantiert also einen kleinsten gemeinsamen funktionellen Nenner. JDBC-konforme Treiber in einer JDBC-Anwendung sollten

[1] ANSI: American National Standards Institute (`http://www.ansi.org`)
ISO: International Organization for Standardization (`http://www.iso.org`)

demnach ohne sonstige Code-Änderungen austauschbar sein, d.h. ein Wechsel von einem DBMS zu einem anderen sollte bei konformen Treibern allenfalls minimalen Aufwand verursachen.
Die Treiber selbst geben Auskunft darüber, ob sie JDBC-konform sind oder nicht. Wie diese Information in einem Java-Programm erfragt werden kann, wird im folgenden Beispiel für drei unterschiedliche Treiber gezeigt (zwei davon, der ODBC/Access- und der Oracle-Treiber, sind konform und antworten mit `true`, der dritte, für MySQL, ist es nicht und liefert entsprechend `false`).

Programm 1-1: JDBC-Konformität

```
// Programm 1-1: BSP_ROOT/meinewebapp/WEB-INF/classes/intro/JdbcKonform.java
public class JdbcKonform {
  public static void main(String[] args) throws Exception {
    System.out.println("ODBC    " +
                       new sun.jdbc.odbc.JdbcOdbcDriver().jdbcCompliant());
    System.out.println("MySQL " +
                      new org.gjt.mm.mysql.Driver().jdbcCompliant());
    System.out.println("Oracle9i " +
                       new oracle.jdbc.driver.OracleDriver().jdbcCompliant());
  }
}
```

Die Klasse `JdbcOdbcDriver` ist Bestandteil der Java-Distributionen, z.B. des Java 2 SDK 1.4, `Driver` ist der Treiber für den SQL-Server MySQL (Open Source), und der Treiber `OracleDriver` der Firma Oracle ist zuständig für Oracle-Datenbanken.
JDBC ist Grundlage und Rahmen sowohl für die Programmierung von Datenbankanwendungen als auch für die Entwicklung von Datenbanktreibern (zur Unterstützung von Letzterem stellt SUN zusätzlich zu den APIs noch den bereits erwähnten Konformitätstest bereit). Das Schwergewicht liegt in diesem Buch auf Ersterem, nämlich der Anwendungsprogrammierung mit JDBC.
JDBC-Konformität garantiert, dass ein Treiber eine definierte Mindestleistung anbietet. Darüber hinaus kann der Treiber aber nach Belieben zusätzliche Eigenschaften haben, die der Anwender über Objekte des Typs `DatabaseMetaData` bei dem Treiber erfragen kann. So kann beipielsweise festgestellt werden, in welcher Stufung er ANSI/ISO SQL/92 unterstützt, was im folgenden Programm 1-2 gezeigt wird.
Als Erstes wird die Klasse `sun.jdbc.odbc.JdbcOdbcDriver` geladen. Sie ist Bestandteil des Java 2 SDK und zusammen mit den Standard-Klassenbibliotheken in der Datei `rt.jar` enthalten (`classes.zip` in älteren Versionen). Beim Laden registriert sich der Treiber in einem Treibermanager, und mittels dieses Treibermanagers wird die Verbindung zur Datenbank `Kurse` hergestellt. Danach werden Metadaten der Datenbank erfragt.

Programm 1-2a: ANSI-Stufe von MS Access

```
// Programm 1-2a: BSP_ROOT/meinewebapp/WEB-INF/classes/intro/AnsiStufeAccess.java
import java.sql.*;
public class AnsiStufeAccess {
  public static void main(String[] args) throws Exception {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    Connection c = DriverManager.getConnection("jdbc:odbc:Kurse");
    DatabaseMetaData meta = c.getMetaData();
    System.out.println(meta.getDatabaseProductName() + " " +
                                    meta.getDatabaseProductVersion());
    System.out.println("Entry Level  " + meta.supportsANSI92EntryLevelSQL());
    System.out.println("Intermediate " + meta.supportsANSI92IntermediateSQL());
    System.out.println("Full Level   " + meta.supportsANSI92FullSQL());
  }
}
```

Die Antworten der dem Beispiel zugrunde liegenden MS Access-Datenbank sind

```
ACCESS 03.50.0000
Entry Level  true
Intermediate false
Full Level   false
```

d.h. der ANSI SQL/92 Entry Level wird unterstützt, die höheren Stufen dagegen nicht.
Ein ähnliches Programm folgt für die Überprüfung des Oracle9i-DBMS. An Stelle des ODBC-Treibers für die Access-Datenbank wird nun der passende Treiber für die Oracle-Datenbank geladen (`oracle.jdbc.OracleDriver`). Dazu muss der Klassenpfad `classpath` auf das Treiberpaket eingestellt sein (siehe auch Abschnitt 4.3.2). Mittels dieses Treibers wird über das Internet eine Verbindung zur Datenbank `orcl` auf dem Datenbankserver `localhost` hergestellt. `localhost` ist sozusagen das `this`-Objekt im Internet, d.h. der Server befindet sich auf dem gleichen Computer wie das Clientprogramm `AnsiStufeOracle`.

Programm 1-2b: ANSI-Stufe von Oracle9i

```
// Programm 1-2b: BSP_ROOT/meinewebapp/WEB-INF/classes/intro/AnsiStufeOracle.java
import java.sql.*;
public class AnsiStufeOracle {
  public static void main(String[] args) throws Exception {
    Class.forName("oracle.jdbc.driver.OracleDriver");
    Connection c = DriverManager.getConnection(
                  "jdbc:oracle:thin:@localhost:1521:Kurse", "system","manager");
    DatabaseMetaData meta = c.getMetaData();
    System.out.println(meta.getDatabaseProductName() + " "
                                      + meta.getDatabaseProductVersion() );
    System.out.println("Entry Level  " + meta.supportsANSI92EntryLevelSQL());
  }
}
```

Die Anwort ist ähnlich der von Access (höhere ANSI-Level würden sich als `false` erweisen):

```
Oracle Oracle9i Release 9.2.0.1.0 - Production
JServer Release 9.2.0.1.0 - Production
Entry Level  true
```

Anders als bei der Überprüfung der Konformität genügt es nicht, einen Treiber zu instanziieren und diesen direkt abzufragen. Denn hinter ODBC- bzw. JDBC-Treibern (vgl. Abschnitt 4.6) können sich die unterschiedlichsten Datenbanken verbergen. Konsequenterweise muss erst eine Verbindung zur konkreten Datenbank hergestellt sein, bevor dann über ein `DatabaseMetaData`-Objekt die so genannten Metadaten des Datenbanksystems, so z.B. die ANSI-Stufen, festgestellt werden können.

1.1.2 Grundstruktur von JDBC-Anwendungen

Um möglichst von Anfang an Sachverhalte mit Java/JDBC-Beispielprogrammen zu illustrieren, wird bereits hier eine knappe Einführung in JDBC gegeben. Das geschieht anhand des Programms `FuenfSchritte`, das zugleich auch Muster für die meisten Beispiele in den Folgekapiteln ist.
In jeder JDBC-Anwendung sind in der Regel folgende fünf Phasen erkennbar:

1. einen JDBC-Treiber registrieren und über den registrierten Treiber das Programm mit der Datenbank verbinden;
2. ein SQL-Anweisungsobjekt erzeugen;
3. eine Anweisung ausführen und
4. das Resultat der Anweisung verarbeiten, z.B. anzeigen;
5. die Verbindung zur Datenbank schließen (und gegebenenfalls deregistrieren).

Die Schritte 1 und 5 werden oft nur ein einziges Mal beim Programmstart bzw. -ende ausgeführt, während die folgenden Schritte 2 bis 4 sich so oft wiederholen, wie es das Anwendungsprogramm erfordert. Dazu ein einfaches Beispiel (die Schritte sind durch die vorangestellten Nummern gekennzeichnet):

Programm 1-3: Phasen der JDBC-Programmierung

```
     // Programm 1-3: BSP_ROOT/meinewebapp/WEB-INF/classes/intro/FuenfSchritte.java
     import java.sql.*;
     public class FuenfSchritte {
       public static void main(String[] args) throws Exception {

1      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
       Connection c = DriverManager.getConnection("jdbc:odbc:Kurse");

2      Statement s = c.createStatement();

3      s.execute("SELECT * FROM Personen WHERE nachname LIKE 'K%'");
       ResultSet r = s.getResultSet();

4      while(r.next())
        System.out.println(r.getString("vorname")+" "+r.getString("nachname"));

5      c.close();
     }
   }
```

Die Programmschritte im Einzelnen:

1. `Mit Class.forName()` wird die Klasse `JdbcOdbcDriver` aus dem Treiber-Paket `sun.jdbc.odbc` geladen, instanziiert und die Instanz im Treibermanager registriert. Jedes JDBC-Programm hat genau ein statisches Treiberregister (vgl. Abschnitt 3.2.1), das mit dem Laden des Treibers angelegt wird.
   Die Verbindung mit einer Datenbank wird mit `getConnection()` mittels des Treibermanagers über einen String hergestellt, der Ähnlichkeit mit der Syntax einer URL hat (`mailto:x@y.z`, `http://www.jugs.org` etc.) und deshalb auch JDBC-URL heißt: `jdbc:odbc:Kurse`. Die danach folgende Nutzerkennung `"gast"` und das Passwort `""` (leerer String) dienen der Anmeldung bei dem Datenbanksystem, das die Tabellen der Datenbank `Kurse` zur Bearbeitung freigeben soll. Das Resultat ist ein Objekt vom Typ `Connection`. Danach spielt der Treibermanager keine Rolle mehr. Da die Verbindungsaufnahme zu einer Datenbank ein sehr zeitintensiver Vorgang ist, sollte eine Verbindung in der Regel so lange aufrechterhalten werden, bis sie definitiv nicht mehr benötigt wird. Die bestehenden Verbindungen muss der Anwender selbst unter Kontrolle halten.
2. Durch Aufruf der Methode `createStatement()` wird ein Objekt vom Typ `Statement` zur Anwendung an der Datenbank `Kurse` erzeugt.
3. Sodann wird mit Hilfe dieses `Statement`-Objekts eine `SELECT`-SQL-Anweisung auf die Datenbank angewendet. Die Ausführung der SQL-Anweisung erfolgt mittels der Methode `execute()` im `Statement`-Objekt `s`. Dabei werden mit der SQL-Anweisung

   ```
   SELECT * FROM Personen WHERE nachname LIKE 'K%'
   ```

   in der Tabelle `Personen` alle Zeilen ausgewählt, in denen `nachname` mit einem `K` beginnt (`'K%'`, mit `%` als Jokerzeichen für den dem `K` evtl. noch folgenden Zeichenkettenrest). Die `execute()`-Methode bringt die `SELECT`-Anweisung zur Ausführung. Mittels der Methode `getResultSet()` erhält man aus dem `Statement`-Objekt `s` das Ergebnis in Form einer Tabelle, von der allerdings immer nur eine Zeile bearbeitet werden kann. Sie hat die gleichen Spaltennamen wie die Tabelle `Personen`, infolge der `WHERE`-Klausel meist mit geringerer Zeilenzahl gegenüber dem Original. (Die Spaltenzahl ließe sich dadurch reduzieren, dass an Stelle des Zeichens `*`, das stellvertretend für alle Spalten steht, z.B. die Spaltenliste `vorname`, `nachname` verwendet würde.)
4. In der `while`-Schleife werden mit der „Cursor"-Methode `next()` nacheinander die Zeilen der Ergebnistabelle (`ResultSet`) eingelesen und aus den Zeilen jeweils `vorname` und `nachname` extrahiert (z.B. mit `getString(vorname)`) und ausgedruckt. Mit dem `ResultSet`-Objekt (Ziffer 4) wird ein Cursor erzeugt, der nach dem ersten `next()`-Aufruf auf die erste Zeile der Ergebnistabelle weist, sofern die Tabelle nicht leer ist. Der Cursor wird mit jedem Aufruf von `next()` um eine Zeile weiterbewegt, bis das Tabellenende erreicht ist. Es gab in den frühen JDBC-Versionen keine Funktionen, um den Cursor eine Zeile zurück, drei Zeilen nach vorn oder ganz an den Anfang bzw. das Ende der Tabelle zu stellen. Ab der Version 2 ist das aber mit entsprechenden Treibern möglich.
5. Mit `close()` als abschließendem Schritt wird die Datenbankverbindung aufgehoben.

Dieser Schritt kann auch der virtuellen Maschine überlassen werden, die beim Beenden eines Programms automatisch die Ressourcenfreigabe bewirkt. Vorsicht ist aber bei Servlets und JSP angebracht.

Auch wenn in den neuen JDBC-Versionen zunehmend Unzulänglichkeiten bzw. Unbequemlichkeiten beseitigt wurden, können die folgenden Regeln zur Fehlervermeidung beitragen:

- Zu jedem `Statement`-Objekt gibt es im Normalfall *höchstens ein* gültiges `ResultSet`-Objekt.
- Auf ein Datenelement in einer Zeile eines `ResultSet` kann *höchstens einmal* zugegriffen werden (z.B. mit `getString("vorname")`). Außerdem kann es ratsam sein, eine `ResultSet`-Zeile grundsätzlich von links nach rechts einzulesen.
- In der Version 1 von JDBC sind die Zeilen eines `ResultSet`-Objekts grundsätzlich nur in einer Richtung durchschreitbar, und zwar vorwärts (`next()`).

1.2 Grundzüge einer JSP-Anwendung

Wie JDBC Treiber zur Verbindung mit Datenbanken auf „call-level“ oder über Internet-Ports benötigt, so sind vergleichbar JSP-Anwendungen auf eine Laufzeitumgebung, die ihre Verbindung mit dem World Wide Web sicherstellt, und auf APIs zu einer solchen als *Container* bezeichneten Umgebung angewiesen. Auf die Inhalte des Containers, nämlich JSP-Dokumente, kann aus Browsern mit URLs gezielt werden, mit denen sich die Inhalte aktivieren lassen. Das Verfahren ist weitläufig mit CGI verwandt, und die JSP-Dokumente, die diesem Verfahren unterworfen werden, sind im einfachsten Falle HTML-Dokumente, in die für die Server-seitige Verarbeitung Java-Code eingebettet ist. Letzteres Verfahren wird auch bei Vorläufern von JSP verwendet, z.B. bei ASP.

JSP kennt aber mindestens zwei weitere Verfahren der Programmierung, die beide auf XML-Tags basieren. Insgesamt bieten sich folgende Varianten an:

- Scripting, d.h. Einbettung von Java-Code in HTML bzw. ganz allgemein in Client-seitigen Code (HTML, CSS, JavaScript u.a.);
- Inanspruchname von JavaBeans zur Festlegung der Funktionalität von Standard-Tags (`jsp:useBean`, `jsp:setProperty`, `jsp:getProperty`), die in JSP Standard*aktionen* heißen;
- Zuhilfenahme von „Custom Tags“, deren Syntax in Tag-Bibliotheken mit XML als Metasprache festgelegt und deren Funktionalität in Java-Klassen, die als Tag-Handler bezeichnet werden, definiert ist. (Seit JSP Version 2 gibt es zusätzlich zwei Varianten, nämlich Tag-Dateien und vereinfachte Tag-Handler.)

Das folgende Beispiel, das der Erläuterung des Arbeitens mit JSP dienen soll, basiert auf der dritten Variante, d.h. Custom Tags. Vermutlich ist dies für umfangreichere Anwendungen die effizienteste Programmiermethode, mit Sicherheit aber die JSP-typischste und interessanteste.

Angewandt sieht ein Custom Tag z.B. so aus:

```
<my:sage text="Hallo JSP!"/>
```

Das Tag verursacht die Ausgabe des Textes im Attribut `text`. Es ist ein Tag ohne Körper, vergleichbar mit `<HR...>` und `<IMG...>` in HTML. Anhand dieses Beispiels werden nun alle Schritte durchlaufen, bis zuletzt die JSP-Seite ihren Inhalt in einem Browser präsentiert. Vorausgesetzt wird lediglich, dass ein Browser installiert und ein Java 2 SDK z.B. in der Version 1.3 verfügbar ist.

Schritt 1: Tomcat installieren

Voraussetzung ist, dass die JSP-Seiten eine Laufzeitumgebung vorfinden. Erster Schritt ist also die Installation eines adäquten Containers, als der sich Tomcat anbietet. Tomcat ist als einfacher Applikations-Server ein Open-Source-Produkt von Apache und beinhaltet die Referenzimplementierung der JSP- und Servlet-Packages gemäß Spezifikationen. Als Erstes wird Tomcat von einer Apache-Website geladen, z.B. die Version 4.1 mittels

```
http://jakarta.apache.org/builds/jakarta-tomcat-4.0/
                release/v4.1.24/bin/jakarta-tomcat-4.1.24.exe
```

für Windows-Betriebssysteme (oder mit einer ähnlichen URL als rpm-Datei für Linux). Es lässt sich aber auch über `http://jakarta.apache.org` und dort über den Anker „Binaries“-Downloads zur richtigen Stelle, nämlich zu den so genannten Release Builds navigieren.
Zur Installation von Tomcat wird die beschaffte Datei

```
jakarta-tomcat-4.1.24.exe
```

aktiviert, der Installationsvorgang beginnt unmittelbar:

- Als Erstes wird die Präsenz eines Java-SDK festgestellt. Fehlt ein solches oder ist die Version ungeeignet, wird die Installation abgebrochen; falls nicht, wird die Installation fortgesetzt.
- Nachdem die Lizenzierungsbedingungen anerkannt wurden, werden Tomcat-Komponenten zur Auswahl angeboten. Hier sollte unbedingt zusätzlich „NT Service“ angekreuzt werden, wenn das Betriebssystem auf NT beruht.
- Anschließend lässt sich das Wurzelverzeichnis umdefinieren. (Bei einem ersten Versuch kann man aber auch alles so belassen, wie es angeboten wird.)

Nach Start des Apache-Tomcat-Dienstes, per Hand oder durch Neustart des Betriebssystems, und nach Eintippen von

```
http://localhost:8080
```

in die Adresszeile des lokalen Browsers sollte sich Tomcat mit seiner Startseite melden.

Schritt 2: Die JSP-Seite erstellen

Das Wurzelverzeichnis, in dem Tomcat installiert ist, wird mit `TOMCAT_HOME` bezeichnet und steht z.B. für `C:\Tomcat 4.1`.
Nun begibt man sich in das Verzeichnis `TOMCAT_HOME/webapps/ROOT` und erzeugt die Datei `Sagwas.jsp` mit diesem Inhalt:

```
<%-- Programm Sagwas.jsp --%>
<%@ taglib uri="sagwas.tld" prefix="my" %>
<my:sage text="Hello World!"/>
```

Definiert wird dann des Weiteren eine Tag-Bibliotheksdatei `sagwas.tld`, in der seinerseits das Tag `sage` definiert ist. Als XML-Namenspräfix wird `my` vereinbart. Dem folgt die Definition eines Tags `<my:sage.../>`, in dem das Pflichtattribut `text` den Ausdruck von „Hello World!“ bewirkt.

Schritt 3: Das „Custom Tag“: Syntax und Funktion

Nächster Schritt ist die Erstellung einer Tag-Bibliothek im gleichen Verzeichnis. Sie wird rituell eröffnet, nämlich mit einem an erster Stelle stehenden XML-Prolog (`<?xml...?>`) und der Definition von `<taglib>`, auf dessen Syntax die URL im `!DOCTYPE`-Tag weist. Die drei darauf folgenden Tags (`tlib-version`, `jsp-version`, `short-name`) sind Pflicht und müssen angegeben werden. Danach folgt die Definition der Syntax des Custom-Tags `sage`, beginnend mit

- seinem Namen `sage`, gefolgt von
- dem Tag-Handler `meinehandler.SagwasTag`, der dem Tag Funktionalität verleiht;
- sodann wird das Attribut `text` definiert und außerdem festgelegt, dass
- es ein Pflichtattribut ist, was im `required`-Tag mit dem Inhalt `true` geschieht.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!-- meinehandler/sagwas.tld -->
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
                 "http://java.sun.com/j2ee/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>saghaltwas</short-name>
  <tag>
    <name>sage</name>
    <tag-class>meinehandler.SagwasTag</tag-class>
    <attribute>
      <name>text</name>
      <required>true</required>
    </attribute>
  </tag>
</taglib>
```

Schritt 4: Der Tag-Handler

Im letzten Schritt wird der Tag-Handler erstellt, und zwar mit den in der Tag-Bibliothek definierten Package- und Klassennamen. Er besteht aus einer Setter-Methode `setText()`, mittels derer der Text vorgegeben werden kann, wenn ein Custom-Tag „instanziiert“ wird.

```
/* WEB-INF/classes/meinehandler/SagWasTag.java */
package meinehandler;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class SagwasTag extends TagSupport {
  private String text;
```

```
  public void setText(String text) {
    this.text = text;
  }
  public int doStartTag() throws JspTagException {
    try {
      pageContext.getOut().println(text);
      return SKIP_BODY;
    }
    catch (Exception ex) { throw new JspTagException("Fehler " + ex); }
  }
}
```

Dem Aufruf der Setter-Methode folgt der Aufruf der als Lebenzyklus des Handlers bezeichneten Methode `doStartTag()`, in der der zuvor gespeicherte Text, im Beispiel „Hello World!“, ausgegeben wird.
Achtung! Sollte Tomcat den Ablauf der JSP-Seite verweigern, dann sollte der Tag-Handler wie im Beispiel einem Package zugeordnet werden. Ist dies bereits der Fall, sollte weiters überprüft werden, ob die in Abschnitt 5.4.3 unter dem Zwischentitel „Konfigurationen“ vorgeschriebenen Einstellungen bereits vorgenommen sind.
Außerdem kann ein Container-Neustart überraschend hilfreich sein.

1.3 Quellen

Programmbeispiele in diesem Buch	http://www.vsite.de/
JDBC-Kurzanleitung	http://www.vsite.de/kurz/jdbc.pdf
JSP-Tutorial	http://java.sun.com/webservices/docs/1.0/tutorial/doc/JSPIntro.html
JDBC-Tutorial	http://java.sun.com/docs/books/tutorial/jdbc/index.html
Web Developers Virtual Library	http://www.wdvl.com
internet.com	http://internet.com mit z.B. javascript.internet.com, java.internet.com etc.