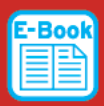
JAVA PERSISTENCE API 2

bernd MÜLLER
harald WEHR

HIBERNATE, ECLIPSELINK, OPENJPA UND ERWEITERUNGEN



EXTRA: Mit kostenlosem E-Book

HANSER

Müller/Wehr

Java Persistence API 2

Bleiben Sie einfach auf dem Laufenden:
www.hanser.de/newsletter
Sofort anmelden und Monat für Monat die neuesten Infos und Updates erhalten.

Bernd Müller
Harald Wehr

Java Persistence API 2

Hibernate, EclipseLink, OpenJPA und Erweiterungen

HANSER

Die Autoren:

Prof. Dr. Bernd Müller, Wolfenbüttel
Harald Wehr, Wernigerode

Alle in diesem Buch enthaltenen Informationen, Verfahren und Darstellungen wurden nach bestem Wissen zusammengestellt und mit Sorgfalt getestet. Dennoch sind Fehler nicht ganz auszuschließen. Aus diesem Grund sind die im vorliegenden Buch enthaltenen Informationen mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Autoren und Verlag übernehmen infolgedessen keine juristische Verantwortung und werden keine daraus folgende oder sonstige Haftung übernehmen, die auf irgendeine Art aus der Benutzung dieser Informationen – oder Teilen davon – entsteht. Ebenso übernehmen Autoren und Verlag keine Gewähr dafür, dass beschriebene Verfahren usw. frei von Schutzrechten Dritter sind. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Buch berechtigt deshalb auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Bibliografische Information der Deutschen Nationalbibliothek:

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Dieses Werk ist urheberrechtlich geschützt.
Alle Rechte, auch die der Übersetzung, des Nachdruckes und der Vervielfältigung des Buches, oder Teilen daraus, vorbehalten. Kein Teil des Werkes darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form (Fotokopie, Mikrofilm oder ein anderes Verfahren) – auch nicht für Zwecke der Unterrichtsgestaltung – reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

© 2012 Carl Hanser Verlag München, www.hanser-fachbuch.de
Lektorat: Margarete Metzger
Herstellung: Irene Weilhart
Copy editing: Manfred Sommer, München
Layout: die Autoren mit LaTeX
Umschlagdesign: Marc Müller-Bremer, www.rebranding.de, München
Umschlagrealisation: Stephan Rönigk
Datenbelichtung, Druck und Bindung: Kösel, Krugzell
Ausstattung patentrechtlich geschützt. Kösel FD 351, Patent-Nr. 0748702
Printed in Germany

print-ISBN: 978-3-446-42693-1
e-book-ISBN: 978-3-446-43129-4

Inhalt

1 Einführung ... 15

1.1 Motivation ... 15

1.1.1 De-facto-Standards ... 15

1.1.2 Zwei Welten treffen aufeinander ... 16

1.1.3 OR-Mapper etablieren sich ... 16

1.1.4 Aktualität und praktische Relevanz ... 17

1.2 JPA und JPA-Provider ... 17

1.2.1 JPA-Provider ... 17

1.2.2 Lizenzen ... 18

1.3 Über dieses Buch ... 18

1.3.1 Das Buch im Netz ... 18

1.3.2 Zur Sprache ... 19

1.3.3 Der Leser ... 19

1.3.4 Vollständigkeit ... 20

1.3.5 Beispielprogramme ... 21

1.3.6 Al, unser Strichmännchen ... 21

1.3.7 Die Struktur des Buches ... 22

2 Grundlegendes Mapping von Objekten ... 25

2.1 Primärschlüssel versus Objektidentität ... 25

2.1.1 Sequence-Generatoren ... 28

2.1.2 Table-Generatoren ... 30

2.2 POJO-Mapping ... 32

2.2.1 Was ist ein POJO? ... 33

2.2.2 Mapping-Optionen auf Entity-Ebene ... 34

2.2.3 Serialisierbarkeit von Entities ... 35

2.2.4 Der JPA-Zugriff auf Entity-Properties ... 36

2.2.5 Mapping-Optionen auf Property-Ebene ... 37
2.2.6 Die `@Column`-Annotation ... 42
2.3 Zusammengesetzte Schlüssel ... 45
2.3.1 Realisierung mit `@IdClass` ... 45
2.3.2 Realisierung mit `@EmbeddedId` ... 48
2.3.3 Realisierung mit `@Embeddable` ... 50
2.4 Mehrere Tabellen für ein Entity ... 51

3 Objektlebenszyklus ... 55
3.1 Der Zustand eines Entity ... 55
3.2 JPA in Java-SE-Anwendungen ... 56
3.2.1 Speichern von Entities ... 57
3.2.2 Laden von Entities ... 59
3.2.3 Aktualisieren von Entities ... 61
3.2.4 Löschen von Entities ... 63
3.2.5 Weitere Operationen auf Entities ... 63
3.2.6 Konfiguration ... 65
3.3 JPA in Java-EE-Anwendungen ... 67
3.3.1 Konfiguration ... 68
3.3.2 Speichern und Laden von Entities ... 71
3.3.3 Aktualisieren und Löschen von Entities ... 74
3.3.4 Der erweiterte Persistenzkontext ... 75
3.3.5 Analyse und Zusammenfassung ... 76
3.4 Callback-Methoden und Entity-Listener ... 77
3.4.1 Callback-Methoden ... 78
3.4.2 Entity-Listener ... 79
3.4.3 Default-Listener ... 80
3.4.4 Callbacks und Listener zur Validierung ... 82
3.5 Das neue Zustands-API ... 83
3.6 Das Packaging von Anwendungen ... 85
3.6.1 Packaging von Web-Anwendungen ... 85
3.6.2 Allgemeines Packaging ... 85
3.7 Mapping-Metadaten in XML ... 87

4 Objekte und Beziehungen ... 89
4.1 Standardbeziehungen ... 89
4.1.1 1:1-Beziehungen ... 90
4.1.1.1 Bidirektionale 1:1-Beziehungen ... 95

4.1.2 1:n- und n:1-Beziehungen ... 96
4.1.2.1 Bidirektionale 1:n-Beziehungen ... 98
4.1.3 n:m-Beziehungen ... 104
4.1.3.1 Bidirektionale n:m-Beziehungen ... 107
4.2 Andere Beziehungsarten ... 109
4.2.1 Assoziationsklassen ... 109
4.2.2 Rekursive Beziehungen ... 113
4.2.3 Geordnete Beziehungen ... 114
4.2.4 Beziehungen als Abbildungen ... 119
4.3 Wertemengen ... 120
4.3.1 Wertemengen ohne JPA ... 121
4.3.2 Einfache Datentypen ... 122
4.3.3 Einbettbare Datentypen ... 125
4.4 Beziehungen und Performanz ... 130

5 Vererbung ... 133
5.1 Eine Tabelle für eine Vererbungshierarchie ... 133
5.2 Eine Tabelle je Unterklasse ... 139
5.3 Eine Tabelle je konkreter Klasse ... 141
5.4 Vergleich der Vererbungsstrategien ... 143
5.4.1 Eine Tabelle für eine Vererbungshierarchie ... 143
5.4.2 Eine Tabelle je Unterklasse ... 144
5.4.3 Eine Tabelle je konkreter Klasse ... 144
5.5 Mapping von Oberklassen ... 144
5.6 Kombination von Vererbungsstrategien ... 146

6 Transaktionen ... 149
6.1 Probleme des Mehrbenutzerbetriebs ... 150
6.2 Isolation-Level ... 152
6.3 Transaktionen in der JPA ... 153
6.3.1 Konkurrierende Zugriffe ... 155
6.3.2 Optimistisches Locking ... 157
6.3.2.1 LockModeType.OPTIMISTIC ... 157
6.3.2.2 LockModeType.OPTIMISTIC_FORCE_INCREMENT ... 160
6.3.3 Pessimistisches Locking ... 160
6.3.3.1 LockModeType.PESSIMISTIC_READ ... 161
6.3.3.2 LockModeType.PESSIMISTIC_WRITE und LockModeType.PESSIMISTIC_FORCE_INCREMENT ... 161
6.3.3.3 Pessimistische Locks in der Praxis ... 161

6.3.4 Locks für Objekte erhalten ... 165
6.3.5 Caching ... 166
6.3.6 Vor- und Nachteile des Caching ... 171
6.4 Konversationen ... 172

7 JPQL (Java Persistence Query Language) ... 177
7.1 JPA-Query-Interfaces ... 178
7.1.1 Named Queries ... 181
7.1.2 Parameter für Abfragen ... 183
7.1.3 Direkte SQL-Abfragen ... 184
7.2 Elemente der Java Persistence Query Language (JPQL) ... 188
7.3 Die From-Klausel ... 188
7.3.1 Implizite Joins ... 189
7.3.2 Inner-Joins ... 189
7.3.3 Left-Joins ... 191
7.3.4 Fetch-Joins ... 192
7.3.5 Joins ohne definierte Mapping-Beziehungen ... 193
7.4 Selektionen mit Where ... 193
7.4.1 Unterabfragen (Subselects) ... 195
7.4.2 Funktionen ... 196
7.5 Projektionen mit Select ... 198
7.5.1 Aggregationsfunktionen ... 199
7.5.2 Gruppieren von Daten mit Group By und Having ... 200
7.5.3 Ergebnisse sortieren mit Order By ... 201
7.5.4 Konstruktoren im Select-Statement ... 202
7.6 Update- und Delete-Statements ... 202
7.7 Flush-Modi und Abfragen ... 204

8 Das Criteria-API ... 207
8.1 Der statische Zugriff ... 208
8.1.1 Grundsätze der Generierung ... 208
8.1.2 Manuelle und automatische Generierung ... 209
8.2 Abfragen mit der Criteria-API ... 212
8.2.1 Join-Bedingungen in der Criteria-API ... 212
8.2.2 Einschränkung der Ergebnismenge ... 215
8.2.3 Selektion der Ausgabe ... 216
8.2.4 Unterabfragen ... 220
8.2.5 Gruppierungen mit `Group By` und `Having` und Sortierungen mit `Order By` ... 222

8.2.6 Benutzung von datenbankspezifischen Funktionen ... 223
8.2.7 Modifikation von bestehenden Abfragen ... 223
8.3 Der dynamische Zugriff ... 224

9 Bean-Validierung mit JSR 303 ... 227
9.1 Einfache Validierungen ... 227
9.2 Beziehungen und eingebettete Objekte ... 231
9.3 Konfiguration ... 233
9.4 Explizite Validierung ... 234
9.5 Validierungsgruppen ... 236
9.6 Anwendungsdefinierte Validierungs-Constraints ... 239
9.7 Validierung mit XML ... 240

10 JPA-Konfiguration ... 243
10.1 Persistenzeinheiten ... 243
10.2 Mapping-Dateien und Annotationen ... 246
10.2.1 Das Element `<entity-mappings>` ... 247
10.2.2 Das Element `<persistence-unit-metadata>` ... 248
10.2.3 Das Element `<access>` ... 249
10.2.4 Das Element `<sequence-generator>` ... 249
10.2.5 Das Element `<table-generator>` ... 250
10.2.6 Das Element `<named-query>` ... 251
10.2.7 Das Element `<named-native-query>` ... 251
10.2.8 Das Element `<sql-result-set-mapping>` ... 252
10.2.9 Das Element `<entity>` ... 252
10.2.10 Das Element `<table>` ... 254
10.2.11 Das Element `<unique-constraint>` ... 255
10.2.12 Das Element `<secondary-table>` ... 255
10.2.13 Das Element `<primary-key-join-column>` ... 256
10.2.14 Das Element `<discriminator-column>` ... 256
10.2.15 Das Element `<attribute-override>` ... 257
10.2.16 Das Element `<association-override>` ... 257
10.2.17 Das Element `<attributes>` ... 258
10.2.18 Das Element `<column>` ... 259
10.2.19 Das Element `<element-collection>` ... 259
10.2.20 Das Element `<collection-table>` ... 260
10.2.21 Das Element `<id>` ... 261
10.2.22 Das Element `<embedded-id>` ... 262

10.2.23 Das Element <basic> ... 262
10.2.24 Das Element <version> ... 263
10.2.25 Das Element <join-column> ... 264
10.2.26 Das Element <cascade> ... 264
10.2.27 Das Element <one-to-one> ... 265
10.2.28 Die Elemente <many-to-one> und <one-to-many> ... 266
10.2.29 Das Element <many-to-many> ... 267
10.2.30 Das Element <order-by> ... 268
10.2.31 Das Element <embedded> ... 269
10.2.32 Das Element <join-table> ... 269
10.2.33 Das Element <mapped-superclass> ... 270
10.2.34 Das Element <embeddable> ... 270
10.2.35 Das Element <entity-listener> ... 271
10.2.36 Das Element <id-class> ... 271
10.2.37 Das Element <inheritance> ... 272
10.2.38 Das Element <order-column> ... 272
10.2.39 Das Element <map-key> ... 272
10.2.40 Die weiteren <map-key-...>-Elemente ... 273

11 Hibernate-Erweiterungen ... 275

11.1 Caching ... 276
11.1.1 Objekt-Cache ... 279
11.1.2 Query-Cache ... 281
11.2 Instrumentation ... 282
11.3 Hibernate-Search ... 284
11.3.1 Lucene ... 284
11.3.2 Konfiguration in Hibernate ... 285
11.3.3 Hibernate-Bridges ... 289
11.3.4 Abfragen mit der Lucene-Syntax ... 292
11.4 Envers ... 293
11.4.1 Einfache Auditierung ... 293
11.4.2 Auditieren von Beziehungen ... 296
11.4.3 Typische Anwendungsfälle ... 297
11.4.4 Die Auditierungsstrategie ... 299

12 EclipseLink-Erweiterungen ... 301

12.1 Caching ... 301
12.1.1 Objekt-Cache ... 303
12.1.2 Query-Cache ... 306
12.2 Weaving ... 307
12.3 Mandantenfähigkeit ... 309

13 OpenJPA-Erweiterungen 313
13.1 Caching 313
13.1.1 Object-Cache 314
13.1.2 Query-Cache 316
13.2 Enhancement 318
13.3 Slice 320
13.3.1 Konfiguration 320
13.3.2 Verteilungsstrategien 321

14 Ausblick auf JPA 2.1 325

Literaturverzeichnis 329

URL-Verzeichnis 331

Stichwortverzeichnis 333

Vorwort

JPA ist in der Entwickler-Community angekommen. Nachdem die Version 1.0 als Teil von Enterprise JavaBeans 3.0 im Jahr 2006 im Rahmen von Java-EE 6 veröffentlicht wurde, begann der Projekteinsatz. Zunächst zögerlich, mittlerweile sehr verbreitet, wurde JPA als das angenommen, was es ist: ein leichtgewichtiges und einfach einzusetzendes Framework für das Mapping von Java-Objekten auf relationale Datenbanksysteme, das zudem noch als Standard von allen großen, in diesem Bereich tätigen Firmen verabschiedet ist und unterstützt wird.

Hibernate, lange Jahre der De-facto-Standard in diesem Bereich, ist vom Marktführer zu einem von mehreren JPA-Providern geworden. Hibernate und andere JPA-Provider ermöglichten aber nach wie vor Funktionalitäten, die über den 1.0-Standard hinausgingen. Zu den Zielen von JPA 2.0 gehörte daher, diese Funktionalitäten, wie etwa ein objektbasiertes Criteria-API, die automatische Validierung von Entities sowie Caching und pessimistisches Locking in JPA aufzunehmen.

Mit JPA 2.0 ist ein Standard geschaffen worden, der die oben genannten Ziele erreicht. Der Sprung in der Versionsnummer macht deutlich, dass die Erweiterungen und Neuerungen nicht kosmetischer Natur sind, sondern einen echten Mehrwert bringen. JPA ist mit der Version 2.0 auch „erwachsen“ geworden: JPA 2.0 ist ein eigener Java-Specification-Request und nicht mehr Teil der EJB-Spezifikation. Der Umfang erhöhte sich von 256 auf 486 Seiten.

Die Entwicklung geht aber weiter, wenngleich in kleineren Schritten. Die Arbeiten an JPA 2.1 haben begonnen und werden voraussichtlich Ende 2012 abgeschlossen sein.

Wir beschreiben in diesem Buch JPA 2.0 und gehen auch auf Erweiterungen der Provider Hibernate, EclipseLink und OpenJPA ein. Das Buch schließt mit einen Ausblick auf JPA 2.1, der zwar nicht vollständig ist, aber zeigt, wohin die Reise geht.

März 2012, Bernd Müller und Harald Wehr

bernd.mueller@ostfalia.de, harald.wehr@gmail.com

1 Einführung

Dieses erste Kapitel führt JPA ein, indem der Bedarf für eine spezielle Art von System-Software, sogenannte OR-Mapper, sowie die speziellen Vorzüge von JPA als deren Realisierung motiviert werden. Wir verwenden im Buch und unseren Beispielen drei verschiedene Implementierungen von JPA, die kurz vorgestellt werden. Am Ende gehen wir auf das Buch selbst sowie auf mögliche Wege der Verwendung des Buches ein.

1.1 Motivation

1.1.1 De-facto-Standards

Seit vielen Jahren gehören objektorientierte Programmiersprachen zu den am häufigsten verwendeten Sprachen bei der Realisierung komplexer Anwendungen im betrieblichen Umfeld. Objektorientierte Konzepte und deren Umsetzung in modernen Sprachen wie Java und C# sind nach dem aktuellen Stand der Wissenschaft und der Erfahrungen aus vielen Projekten geeignet, die Anforderungen großer und komplexer Informationssysteme mit angemessenem Aufwand in lauffähige Anwendungen umzusetzen.

Relationale Datenbanksysteme haben sich für die Datenhaltung ebenfalls als Standard etabliert, lange bevor es Java und C# gab. Eine über 40 Jahre andauernde Erfahrung in der Implementierung dieser Systeme haben zu sehr sicheren und hochperformanten Datenverwaltungssystemen geführt, die weit verbreitet sind. Die Grundlagen relationaler Datenbanksysteme sind mathematisch fundiert, die Sprache zur Definition (Data Definition Language, DDL) und Manipulation (Data Manipulation Language, DML) von Datenbanken, die Structured Query Language, ist in mehreren Standards verabschiedet worden. Trotzdem sind die verschiedenen Implementierungen wie Oracle, DB/2 oder MS-SQL-Server keineswegs kompatibel. Die Migration einer Datenbank von einem System auf ein anderes ist in der Regel mit hohem Aufwand verbunden.

Mit dem Java-Database-Connectivity-API (JDBC) wurde ein Standard geschaffen, der einen Teil der Unterschiede zwischen verschiedenen Datenbanksystemen verbirgt, aber bei Weitem nicht vollständig beseitigt. Mit JDBC werden auf SQL-Ebene CRUD-Anweisungen (Create, Read, Update, Delete) an das Datenbanksystem geschickt und die Antwort als Ergebnis der Anfrage zur weiteren Verarbeitung bereitgestellt.

1.1.2 Zwei Welten treffen aufeinander

Die objektorientierte Welt und die relationale Welt sind auf den ersten Blick grundverschieden. *Vererbung, dynamisches Binden, Polymorphie, Fabriken* auf der einen, *Primär- und Fremdschlüssel, kaskadierendes Löschen, Normalformen* auf der anderen Seite sind Vokabeln, die in der jeweils anderen Welt keine Entsprechung besitzen. Die Unverträglichkeit der beiden Paradigmen ist offensichtlich und muss in der tagtäglichen Entwicklungsarbeit überbrückt werden, so dass sich hierfür ein eigener Begriff, der des *Paradigm Mismatch* oder auch *Impedance Mismatch* etabliert hat.

Auf den zweiten Blick sind ein Objekt der Klasse Kunde mit den Properties Vorname, Nachname und Geburtstag und eine Tabelle mit Namen Kunde und den Spalten Vorname, Nachname und Geburtstag einander durchaus in gewissem Sinne ähnlich. Ein erfahrener Entwickler könnte relativ schnell eine Lösung zur Abbildung von Kundenobjekten auf Zeilen der Kundentabelle entwickeln.

Da das Problem genereller Natur ist, tritt es in allen Projekten auf, die mit objektorientierten Programmiersprachen auf relationale Datenbanken zugreifen. Nachdem in vielen Projekten spezifische Lösungen entwickelt wurden, begannen die grundlagenorientierten Arbeiten zum sogenannten *Object-Relational-Mapping* oder kurz *OR-Mapping*, also der Abbildung von Objekten auf Relationen. Bereits Anfang der 90er Jahre wurden vor allem von Scott W. Ambler Arbeiten zu diesem Thema publiziert, die zum Teil auch heute noch im Web zur Verfügung stehen.

1.1.3 OR-Mapper etablieren sich

Die erste praktisch einsetzbare Java2-EE-Version war die Version 1.2, die im Dezember 1999 veröffentlicht wurde. Sie enthielt EJB 1.1 und damit bereits ein OR-Mapping, die *Container Managed Persistence* (CMP). Später wurde mit *Java Data Objects* (JDO) eine weitere offizielle Spezifikation für das Mapping von Java-Objekten auf Datenbanken veröffentlicht. JDO war nicht auf relationale Datenbanken beschränkt und unterstützte verschiedene Systeme zur Datenhaltung, neben relationalen Datenbanken etwa auch objektorientierte Datenbanken. Diese offiziellen Java-Spezifikationen für das OR-Mapping wurden aus verschiedensten Gründen von der Entwicklergemeinde nicht angenommen. Insbesondere wurde der Konfigurations- und Programmieraufwand für diese Lösungen als zu hoch kritisiert.

Die Open-Source-Community entwickelte als Antwort auf diese Kritik alternative OR-Mapper. Hier ist allen voran Hibernate zu nennen, das sich als Quasi-Standard etablierte. Andere Systeme waren z. B. Castor und iBatis, jetzt MyBatis genannt.

Als Reaktion auf die Kritik an den EJB-Versionen 1.x und 2.x und die Popularität von Hibernate wurde von Sun mit EJB 3.0 und der enthaltenen Version 1.0 von JPA ein Standard geschaffen, der sowohl funktional als auch programmiertechnisch mit Hibernate aufschloss und die Messlatte für OR-Mapper sehr hoch legte. Dies war unter anderem der Verdienst einer offenen Arbeit der Expertengruppe von JPA 1.0, der unter anderem Gavin King, der intellektuelle Kopf hinter Hibernate, angehörte.

1.1.4 Aktualität und praktische Relevanz

JPA 1.0 hat sich in sehr kurzer Zeit als *der* Standard für das OR-Mapping etabliert. Selbst Gavin King rät, für neue Projekte JPA und nicht natives Hibernate zu verwenden. JPA 2.0 ist unverzichtbarer Bestandteil der Java-Enterprise-Edition 6, wurde jedoch in seiner Unabhängigkeit unter anderem auch in der Namensgebung gestärkt. Der offizielle Name „*JSR 317: Java Persistence API, Version 2.0*“ enthält keinen Bezug mehr zu Enterprise JavaBeans. Die nach wie vor hohe Relevanz von JPA in Bezug zu Java-EE wird durch einen anderen Umstand deutlich: Selbst das neu definierte Web-Profile von Java-EE 6 enthält JPA 2.0, was für ein „Web“-Profile durchaus bemerkenswert ist.

Dass sich JPA als Standard für das OR-Mapping etabliert hat, kann durch seine öffentliche Sichtbarkeit und Popularität belegt werden. Keine Entwicklerkonferenz, keine Zeitschrift, kein Schulungsanbieter aus dem Java-Bereich kommt ohne ein entsprechendes JPA-Angebot aus. Selbst Stellenanzeigen enthalten mittlerweile neben Kentnissen in Hibernate auch Kenntnisse in JPA als geforderte Qualifikation.

1.2 JPA und JPA-Provider

JPA ist eine Spezifikation und damit ein „Papiertiger“, wenngleich in Form von PDF. Die JPA-Spezifikation 2.0 enthält auf 486 Seiten eine detaillierte Beschreibung der APIs, aber auch weiterer Verwendungsdetails wie etwa Vorgaben für das Packaging von Anwendungen oder Anforderungen an einen Java-EE-Container für das Deployment und Bootstrapping von Anwendungen. Dies wird durch umfangreiche API-Doc-Kommentare vervollständigt.

1.2.1 JPA-Provider

Eine Spezifikation benötigt (mindestens) eine Implementierung, bei JPA einen sogenannten JPA-Provider. Für JPA 2.0 existieren mehrere solcher Provider. Hibernate[URL-HIB], EclipseLink[URL-EL] und OpenJPA[URL-OJPA] sind nach unserer Einschätzung die derzeit populärsten und am weitesten verbreiteten Provider, so dass wir diese in unserem Buch und den Beispielen verwenden. EclipseLink besitzt eine Sonderrolle, da EclipseLink die Referenzimplementierung von JPA 2.0 ist. Ein weiterer JPA-Provider ist z. B. DataNucleus[URL-DN]. Dieser wird unter anderem von Google verwendet, um mit JPA auf den Datenspeicher der Google-App-Engine zugreifen zu können. Wir verwenden DataNucleus in diesem Buch nicht.

Die drei erstgenannten JPA-Provider sind zertifizierte Implementierungen. Im Rahmen des Java Community Process (JCP) wird für jede verabschiedete Spezifikation ein *Technology Compatibility Kit* (TCK) entwickelt, das eine Test-Suite, Werkzeuge und Dokumente enthält, die bestimmen, wann eine Implementierung den in der Spezifikation geforderten Anforderungen genügt oder nicht. Dies bedeutet nicht, dass sich alle zertifizierten Implementierungen *identisch* verhalten. Zum einen regelt die Spezifikation nicht alle Features bis ins letzte Detail und überlässt in manchen Punkten der Implementierung die letzte Entscheidung. Zum anderen ist es möglich, dass bestimmte Aspekte der Spezifikation von den Entwicklern

der jeweiligen Implementierungen unterschiedlich interpretiert und somit realisiert werden, dies aber vom TCK nicht überprüft bzw. erkannt wird.

Wir haben versucht, die aktuellste Version des jeweiligen Providers zu verwenden, die damit für ein gedrucktes Buch durch den Zeitpunkt der Manuskriptabgabe bestimmt wird. Wir verwenden Hibernate in der Version 4.0.0, EclipseLink in der Version 2.3.2 und OpenJPA in der Version 2.1.1.

Durch unsere Arbeit mit den drei Implementierungen im Rahmen der Beispiele für unser Buch haben wir auch Fehler in den Implementierungen gefunden. Diese wurden nach Einreichung der Fehler durch Mail, JIRA oder Bugzilla immer zeitnah behoben. Wir raten dem Leser in solchen Fällen ebenfalls zur Einreichung der Fehler und zur Aktualisierung der Version. Wenn Ihre Zielplattform Java-EE ist und Sie damit einen Application-Server als Ablaufumgebung verwenden, ist der JPA-Provider und damit auch dessen Version von der Version des Application-Servers abhängig. Prinzipiell ist der Austausch des JPA-Providers im Application-Server möglich. Insbesondere JPA-Anfängern raten wir hiervon jedoch ab. Stattdessen sollte die nächste Aktualisierung des Application-Servers abgewartet werden.

1.2.2 Lizenzen

Eine weitere charakteristische Eigenschaft der drei JPA-Provider ist ihre Entwicklungs- und Lizenz-Politik. Alle Provider werden als Open-Source-Systeme entwickelt und verwenden die folgenden Lizenzen: GNU Lesser General Public License 2.1 (Hibernate), Eclipse Public License 1.0 (EclipseLink), Apache License 2.0 (OpenJPA). Durch den Open-Source-Charakter kann prinzipiell jeder an der Weiterentwicklung der Provider aktiv mitarbeiten, falls man die entsprechenden Entscheider davon überzeugen kann, dass dies im Projektinteresse liegt und man den Zugang zu den Quellcode-Verwaltungssystemen bekommt. Man kann aber den öffentlich zugänglichen Bug-Tracking-Systemen und anderen Quellen entnehmen, dass mehrheitlich Firmenmitarbeiter die Entwicklungsarbeit leisten. Bei Hibernate sind dies Mitarbeiter von Red Hat bzw. JBoss, bei EclipseLink von Oracle und bei OpenJPA von IBM.

1.3 Über dieses Buch

1.3.1 Das Buch im Netz

Bücher zu Themen der Software-Entwicklung müssen möglichst aktuell sein. Gerade der Open-Source-Bereich ist durch hochfrequente Release-Zyklen gekennzeichnet. Wir erachten es daher als nicht sinnvoll, unserem Buch eine CD beizulegen. Vielmehr betreiben wir im Netz die Website

`www.jpainfo.de`

Sie finden dort den Quellcode für alle im Buch dargestellten Programme. Die für die Beispiele benötigten Bibliotheken finden Sie ebenfalls dort.

Falls Sie als Dozent tätig sind und unser Buch verwenden, berichten Sie uns bitte über Ihre Erfahrungen damit. Spezielle Themen oder Erweiterungsvorschläge für die nächste Aufla-

ge sind uns willkommen. Sind Sie als Entwickler tätig und setzen unser Buch im Tagesgeschäft ein, sind wir ebenfalls an Ihren Erfahrungen und Verbesserungsvorschlägen interessiert. Wenn Ihre Zuschriften eine kritische Masse erreichen, werden wir uns bemühen, einen Forumsbereich auf der Website einzurichten.

Zu guter Letzt werden Sie, wenn nötig, dort ein Druckfehlerverzeichnis und eventuell notwendige Aktualisierungen für das Buch finden.

1.3.2 Zur Sprache

Dieses Buch ist in Deutsch geschrieben. Die Sprache der Software-Entwicklung ist aber Englisch. Wir versuchen so weit wie möglich, englische Begriffe und Anglizismen zu vermeiden. Dies ist insbesondere bei Fachbegriffen, für die keine deutsche Entsprechung existiert, und bei Namen schlecht möglich. Das Wort „Entity“ mit „Entität“, „Bean“ mit „Bohne“ oder „Packaging“ mit „Paketierung“ oder „Konfektionierung“ zu übersetzen, erscheint uns nicht sinnvoll. Die genannten Begriffe sind mehr oder weniger Eigennamen mit feststehender Bedeutung und haben sich vor allem bei Praktikern eingebürgert.

Ein weiterer sprachlicher Stolperstein ist die Bildung zusammengesetzter Wörter. Im Englischen werden diese getrennt, im Deutschen zusammengeschrieben, aus Gründen der besseren Lesbarkeit auch mit Bindestrich. Aus „*Java Persistence API*“ im Englischen wird daher im Deutschen „*Java-Persistence-API*“. Die getrennt geschriebene Aneinanderreihung mehrerer Substantive ist im Deutschen nicht möglich. Wir sehen den Konflikt bei der Verwendung von Eigennamen, wie etwa „*Java Persistence API*“, ziehen die Bindestrichnotation aber vor, da sonst Problemfälle wie „*Java Persistence API-Implementierung*“ entstehen, deren Zusammenhang beim ersten Lesen oder bei Zeilenumbrüchen nicht mehr offensichtlich ist. Zum Teil verwenden wir in Überschriften das englische Original, im Fließtext die Bindestrichnotation.

Der letzte Punkt in Sachen Sprache ist die zweigeschlechtliche Ansprache. In Prüfungsordnungen von Hochschulen liest man in der Regel, dass Studentinnen und Studenten von Prüferinnen und Prüfern geprüft werden, die Professorinnen oder Professoren sein können. Wir wollen Ihnen und uns dies ersparen und sprechen Sie ausschließlich aus Gründen besserer Lesbarkeit mit grammatikalisch männlichem Geschlecht an. Unter einem Leser oder einem Entwickler verstehen wir eine Person, die liest oder entwickelt, völlig unabhängig vom Geschlecht.

1.3.3 Der Leser

Dieses Buch hat seine Wurzeln in Hochschullehrveranstaltungen, die wir in den Bereichen Software-Entwicklung, Datenbanken und Objektpersistenz durchgeführt haben. Es eignet sich daher besonders gut für die Unterstützung von Lehrveranstaltungen in Informatik-, Bindestrichinformatik- und Software-Technik-Studiengängen an Universitäten und Fachhochschulen. Wir führen aber auch Schulungen für Unternehmen in den Bereichen Java und Objektpersistenz durch, so dass wir die Ansprüche an solche Schulungen ebenfalls kennen. Auch als im Berufsleben stehender Praktiker werden Sie sicher viel Wissenswertes und Neues in diesem Buch erfahren können.

Java ist, was den Sprachkern und die Syntax angeht, eine einfache und gut konzipierte Sprache. Die Anzahl und der Umfang der APIs wächst jedoch zusehends und kann wahrscheinlich von einer einzelnen Person nicht mehr in vollem Umfang beherrscht werden. Wenn Sie mit dem Gedanken der Anschaffung dieses Buches spielen oder es bereits angeschafft haben, sollten Sie Java beherrschen bzw. es gerade lernen. Die Kenntnis der APIs für Reflection und JDBC sind nicht unbedingt nötig, wird Ihnen jedoch helfen, Dinge besser zu verstehen. Die provider-spezifischen Abschnitte über Caching und Instrumentierung setzen tiefergehende Java-Kenntnisse voraus.

Unser Schwerpunkt des Einsatzes von JPA ist Java-SE, da dies für den Anfänger einfacher ist. Die Abstecher in Java-EE erfordern nicht unbedingt Kenntnisse in diesem Bereich, letztere sind aber hilfreich. Prinzipiell ist JPA in Java-SE und Java-EE identisch einsetzbar, Unterschiede erwähnen wir explizit. Kenntnisse aus dem Bereich relationaler Datenbanken sind für das Verständnis der Buchinhalte ebenfalls sehr hilfreich.

Zu guter Letzt: Warum sollten Sie dieses Buch kaufen? Weil Sie sich für eines der folgenden Themen interessieren: Java-EE, JPA, OR-Mapper, die Produkte Hibernate, EclipseLink und OpenJPA oder ganz allgemein für die moderne Realisierung von Java-Systemen mit Persistenz.

1.3.4 Vollständigkeit

In der Version 2.0 umfasst die JPA-Spezifikation 486 Seiten, die Syntax und Semantik beschreiben, jedoch nicht in Probleme und Lösungen der Objektpersistenz einführen oder erläutern. Das Buch *Enterprise JavaBeans 3.1* von Rubinger und Burke [RB10], das sich in großen Teilen auch JPA 2.0 widmet, hat 738 Seiten. Die Handbücher der drei JPA-Provider, die JPA, aber auch provider-spezifische Erweiterungen beschreiben, sind im Seitenumfang schwer zu schätzen, da sie in der Regel als Web-Seiten verfügbar sind. Sie umfassen aber sicher mehrere Hundert Seiten.

Wir wollen mit dem vorliegenden Buch eine Einführung in JPA und einige interessante Erweiterungen der drei JPA-Provider geben, ohne vollständig zu sein. Eine Vollständigkeit in dem Sinne, dass JPA und die Erweiterungen komplett dargestellt werden, ist in unseren Augen weder sinnvoll noch möglich. Sowohl das API von JPA als auch die APIs der JPA-Provider sind im Java-Doc-Format auf den entsprechenden Internet-Seiten verfügbar. Die JPA-Spezifikation ist als PDF-Dokument ebenfalls unentgeltlich verfügbar [URL-JPA2].

Auch die Zahl von ca. 350 Seiten, die wir uns als Obergrenze gesetzt haben, impliziert, dass keine vollständige Darstellung erfolgen kann. Wir haben aber alle für die Anwendungsentwicklung relevanten Informationen zusammengetragen und sind uns sicher, dass Sie, lieber Leser, nach der Lektüre und aktivem Nachvollziehen der Beispiele dazu fähig sein werden, anspruchsvolle Lösungen aus dem Bereich der Persistenz in Informationssystemen auf der Basis von JPA zu erstellen. Außerdem werden Sie in der Lage sein, die innovativen Erweiterungen der JPA-Provider in Ihren Anwendungen zu verwenden.

Wir beschreiben in unserem Buch JPA 2.0 nahezu vollständig. Falls wir einigen Themen keine explizite Behandlung in einem gesonderten Abschnitt gewidmet haben, finden Sie im Kapitel 10 eine Aufzählung aller XML-Tags und damit implizit auch aller Annotationen mit einer kurzen Erläuterung.

1.3.5 Beispielprogramme

Alle Klassen und Code-Ausschnitte entstammen tatsächlich existierenden Programmen. Sie können, nein sollen diese Programme von den Internet-Seiten herunterladen, selbst ausprobieren und als Ausgangspunkt für eigene Entwicklungen verwenden.

Wir haben uns immer bemüht, zunächst Konzeptionelles zu erläutern und dann in Programmtext umzusetzen. Die Programmierung selbst ist allerdings untrennbar mit Konfigurieren, Testen und Debuggen verbunden. Die herunterladbaren Beispiele sind vollständig konfiguriert und sofort ausführbar. Eine kurze Anleitung hierzu finden Sie ebenfalls auf den Buchseiten im Web. Die Beispiele sind allesamt Eclipse-Projekte und als solche in Ihr Eclipse zu importieren. Sie sind aber auch über ein Ant-Build-File ausführbar und somit IDE-unabhängig. Auch hierzu finden Sie weitere Informationen auf den Buchseiten im Netz.

Allen Beispielprogrammen gemein ist, dass sie aus dem Bankenbereich stammen und damit in Teilen einander sehr ähnlich sind. Die Programme sind für die jeweiligen Themengebiete, in denen sie zum Einsatz kommen, angepasst und zum Teil minimalistisch. Es kann daher vorkommen, dass ein Kundenobjekt in einem Abschnitt mehr oder weniger Properties besitzt als in einem anderen Abschnitt, wenn dies der Kontext erfordert.

Bei der Darstellung von Programmtexten im Buch haben wir uns bemüht, nur die für die Erläuterung von Konzepten relevanten Ausschnitte darzustellen. Dies erfolgt zum einen im Hinblick auf die Herstellungskosten des Buches, zum anderen aber auch aus didaktischen Gründen. Wir wollen Sie auf die wirklich wichtigen Dinge hinweisen und nicht mit seitenweisen Code-Listings ermüden.

Sollten Sie, aus welchen Gründen auch immer, lieber den vollständigen Programmtext vorliegen haben, finden Sie diesen in den Beispielprojekten im Netz.

1.3.6 Al, unser Strichmännchen

An vielen Stellen des Buches wollen wir Ihre Aufmerksamkeit auf besonders hervorzuhebende Informationen lenken, Sie vor Fallstricken warnen oder zum aktiven Programmieren auffordern. Wir realisieren dies mit Hilfe von Al, einem netten Strichmännchen, das wir der Open-Clip-Art-Bibliothek [URL-OCA] entnommen haben. Die verwendeten Grafiken unterliegen der Creative-Commons-Lizenz, die ebenfalls unter [URL-OCA] eingesehen werden kann.

Wir verwenden Al in den folgenden vier Arten:

Hinweis:

Ein Hinweis ist eine allgemeine Aussage zu einem bestimmten Thema. Häufig ist die Aussage bereits im laufenden Text vorhanden. Sie erscheint uns aber so wichtig, dass wir explizit noch eimal darauf hinweisen.

Tipp:

Ein Tipp ähnelt einem Hinweis, rät aber zu einem bestimmten Vorgehen. Tipps sind damit so etwas wie „Best-Practice"-Regeln.

Achtung:

Mit *Achtung* wollen wir Sie warnen. Als unerfahrener Entwickler sollten Sie unsere Warnung befolgen und entsprechend verfahren. Als erfahrener Entwickler sollten Sie die Auswirkungen Ihres Tuns noch einmal überdenken.

Projekt:

Wenn Sie sich die Finger schmutzig machen und in den Ring steigen wollen, sollten Sie unsere Beispielprojekte herunterladen und ausprobieren. Wir raten dringend, dies zu tun.

1.3.7 Die Struktur des Buches

In den folgenden Kapiteln führen wir zunächst JPA und dann sinnvolle Erweiterungen ein. Je nachdem, ob Sie Ihre ersten Kontakte mit JPA machen wollen, bereits JPA 1.0 einsetzten oder andere Vorkentnisse oder Interessen haben, existieren verschiedene Wege durch dieses Buch. Wir geben daher zunächst einen Überblick über die einzelnen Kapitel.

- Kapitel 2:
  Grundlegendes Mapping von Objekten definiert die minimalen Anforderungen an die Peristenz von Java-Objekten und zeigt, wie man mit Annotationen die vom JPA-Provider benötigten Mapping-Informationen definiert.
- Kapitel 3:
  Objektlebenszyklus stellt den Lebenszyklus eines Entity dar. Da die zustandsbestimmenden Daten eines Entity in der Datenbank gehalten werden, unterscheidet sich der Lebenszyklus von Entities deutlich von normalen Java-Objekten. Das Kapitel wird durch Informationen zum Packaging von Anwendungen abgerundet.
- Kapitel 4:
  Objekte und Beziehungen widmet sich Beziehungen zwischen Objekten aber auch Beziehungen zwischen Objekten und Werten, im objektorientierten Sprachgebrauch also Assoziationen, Aggregationen und Kompositionen.
- Kapitel 5:
  Vererbung ist eine charakteristische Eigenschaft objektorientierter Sprachen. Dieses Kapitel beschreibt, wie Vererbungsbeziehungen auf relationale Datenbanksysteme abgebildet werden können, obwohl diese das Konzept der Vererbung nicht unterstützen.
- Kapitel 6:
  Transaktionen gruppieren mehrere Datenbankoperationen zu einer logisch einzelnen,

atomaren Operation. Das Kapitel führt in die Probleme des Mehrbenutzerbetriebs ein und zeigt auf, welche Möglichkeiten JPA bietet, um diese Probleme zu lösen. Auf JPAs Caching-Möglichkeiten wird ebenfalls eingegangen.

- Kapitel 7:
  Java-Persistence-Query-Language führt JPAs Abfragesprache ein. Mit einer SQL-ähnlichen Syntax ist es möglich, Abfragen zu definieren, auszuführen und deren Ergebnisse in Java weiterzuverarbeiten.
- Kapitel 8:
  Das *Criteria-API* beschreibt ein Meta-Modell für JPA-Artefakte und eine darauf aufbauende, Java-basierte Möglichkeit, Abfragen zu formulieren und auszuführen. Es ist somit eine Alternative zur Java-Persistence-Query-Language.
- Kapitel 9:
  Bean-Validierung mit JSR 303 ist ein unabhängig von JPA entwickelter Standard, um Validierungsbedingungen für POJO-Daten zu formulieren.
- Kapitel 10:
  JPA-Konfiguration stellt die Möglichkeiten zur Konfiguration von JPA mit den XML-Dateien `persistence.xml` und `orm.xml` sowie weiterer, anwendungsdefinierter XML-Dateien vor. Die Angaben von Mapping-Informationen sind in diesem Rahmen als Alternative, aber auch als Erweiterungs- oder Optimierungsoption zu sehen.
- Kapitel 11, 12 und 13:
  JPA ist eine Spezifikation. Jeder zertifizierte Provider implementiert die Spezifikation, versucht aber auch, sich durch Alleinstellungsmerkmale von den Mitbewerbern zu unterschieden. Jedes der drei provider-spezifischen Kapitel enthält zunächst zwei Abschnitte zu den Themen Caching und Instrumentierung und danach noch ein oder zwei Abschnitte zu verschiedenen Themen, die unserer Meinung nach zu den interessantesten Erweiterungen gehören.
- Kapitel 14:
  Ausblick auf JPA 2.1 soll einen kleinen Vorgeschmack auf die nächste JPA-Version geben. Da die Arbeiten an der Version 2.1 zum Zeitpunkt der Fertigstellung unseres Buches noch nicht abgeschlossen sind, ist dieses Kapitel als Momentaufnahme zu sehen. Mit der offiziellen Verabschiedung von JPA 2.1 werden wir die wichtigsten Neuerungen auf den Internet-Seiten publizieren.

Als Leser ohne bisherige JPA-Kenntnisse raten wir Ihnen, die Kapitel 2 bis 7 der Reihe nach durchzuarbeiten. Sie haben damit JPA praktisch vollständig kennengelernt. Das Kapitel 8 (Criteria-API) ist für den Anfänger sicher recht schwierig zu erarbeiten und kann zunächst entfallen. Sie können es später bei Bedarf nachholen. Auch das Kapitel 9 (Bean-Validierung) kann beim ersten Lesen des Buches zunächst entfallen. Die Datenqualität einer Anwendung wird durch den Einsatz der Bean-Validierung unzweifelhaft erhöht bzw. die Erhöhung erleichtert, für das tiefere Verständnis von JPA ist die Bean-Validierung jedoch nicht nötig.

Wenn Sie bereits Erfahrungen mit JPA 1.0 haben, sollten Sie sich zunächst mit den großen Neuerungen von JPA 2.0 beschäftigen. Diese sind das Criteria-API sowie die Integration der Bean-Validierung, also die Kapitel 8 und 9. Die weiteren Neuerungen sind in verschiedene Kapitel des Buches integriert. Für eine erste Recherche seien einige Einstiegspunkte genannt:

- Definition der Zugriffsart für Properties über `@Access`: Abschnitt 2.2.4

- Explizites Lösen eines Entity von der Persistenzeinheit durch die `detach()`-Methode des Entity-Managers und entsprechendes Kaskadieren (`DETACH`) bei Beziehungen: Abschnitte 3.2, 3.3, 4.1
- Wertemengen, sogenannte Element-Collections: Abschnitt 4.3
- Verbesserte Lock-Möglichkeiten beim parallelen Zugriff mehrerer Anwender, insbesondere das Sperren von Datenbanksätzen direkt auf der Datenbank: Abschnitt 6.3.3
- Zwischenspeichern von Objekten über den Persistenzkontext hinaus im so genannten Second-Level-Cache: Abschnitt 6.3.5

Falls Sie sich innerhalb einer Hochschulveranstaltung oder eines Projekts in JPA einarbeiten, wird wahrscheinlich Ihr Dozent oder Ihr Projekt bzw. Projektleiter den JPA-Provider bestimmen. Sie sollten dann einen Blick in das entsprechende Erweiterungskapitel des verwendeten Providers werfen. Falls der Provider nicht festgelegt ist, sollten Sie alle drei Erweiterungskapitel studieren. Die Provider stellen durchaus sinnvolle Erweiterungen bereit, die über den JPA-Standard hinausgehen und Ihnen eventuell viel Arbeit abnehmen.

2 Grundlegendes Mapping von Objekten

Dieses Kapitel beschreibt, wie Objekte auf Tabellen abgebildet werden. Ein solches persistentes Objekt wird *Entity* genannt, und wir zitieren dessen Beschreibung aus der JPA-Spezifikation Kapitel 2:

> „An entity is a lightweight persistent domain object."

Wir gehen in Abschnitt 2.2 detaillierter auf das Konzept eines leichtgewichtigen, persistenten Geschäftsobjekts ein, nachdem wir zunächst im Abschnitt 2.1 die grundsätzliche Problematik der verschiedenen Identitätskonzepte in objektorientierten Programmiersprachen und relationalen Datenbanken analysiert und Lösungen mit JPA vorgestellt haben. Das Kapitel schließt mit kürzeren Ausführungen zu angrenzenden Themen.

Beziehungen zwischen Objekten werden in diesem Kapitel nicht behandelt. Assoziationen als eine Art von Beziehungen zwischen Objekten werden in Kapitel 4, Vererbung als eine andere Art von Beziehung wird in Kapitel 5 behandelt.

2.1 Primärschlüssel versus Objektidentität

Java unterscheidet zwischen Werten (primitive Typen) und Objekten (Referenztypen). Für den Test auf Identität wird sowohl bei Werten als auch bei Objekten der Gleichheitsoperator, besser Identitätsoperator `==` verwendet. Bei Werten findet ein Vergleich der beiden Werte statt, bei Objekten ein Vergleich, ob beide Referenzen auf dasselbe Objekt verweisen. Dies entspricht intern ebenfalls einem Vergleich der Werte der Referenzen.

Um die Gleichheit von Objekten in Java zu testen, gibt es die `equals()`-Methode. Diese ist in der Klasse `Object` im Package `java.lang` definiert und wird daher von allen Klassen geerbt. Klassen können diese Methode überschreiben, falls die Anwendungslogik dies verlangt. Das Überschreiben der `equals()`-Methode erzwingt auch ein Überschreiben der `hashCode()`-Methode, da gleiche Objekte denselben Hash-Code besitzen müssen. Mit der `equals()`- und der `hashCode()`-Methode kann somit eine Objektgleichheit z. B. auf Basis der Gleichheit eines oder mehrerer Properties der beiden Objekte definiert werden.

Die Verwendung eines OR-Mappers macht die Dinge etwas komplexer. Ein persistentes Objekt repräsentiert eine bestimmte Zeile einer Datenbanktabelle, eventuell auch mehrere Zeilen in verschiedenen Tabellen, die ihrerseits durch je einen Primärschlüssel identifiziert wer-

den. JPA kennt und verwendet diese Datenbankidentität intern, und auch Sie als Anwendungsentwickler müssen diese eventuell verwenden. Wir haben es nun also mit drei Arten von Identität bzw. Gleichheit zu tun.

Die Datenbankidentität ist ein Realisierungsdetail der Objektpersistenz. Sie ist daher einerseits im Anwendungsmodell nicht enthalten, muss aber andererseits immer im Mapping verwendet werden. In JPA gibt es mehrere Alternativen, den Primärschlüssel zu erstellen. Er kann von der Anwendung oder von der Persistenzschicht in Zusammenarbeit mit dem Datenbanksystem erzeugt werden.

Die nachfolgend dargestelle Klasse `Kunde` realisiert die Alternative des anwendungserzeugten Primärschlüssels.

```
@Entity
public class Kunde {

  @Id
  private String id;
  private String vorname;
  private String nachname;

  public Kunde() { }

  // Getter und Setter
}
```

Die Annotation `@Entity` macht aus der gewöhnlichen, nicht persistenten Java-Klasse ein persistentes Entity. Die Instanzvariable `id` ist mit `@Id` annotiert und wird damit von JPA als Primärschlüsselvariable verwendet. Die beiden genannten Annotationen sowie alle weiteren JPA-Annotationen sind im Package `javax.persistence` enthalten. Einzige Ausnahme ist die Annotation `@StaticMetamodel`, die im Sub-Package `java.persistence.metamodel` definiert ist.

Ohne weitere Annotationen muss die Anwendung dafür Sorge tragen, dass das Entity vor dem Persistieren mit einem gültigen, also noch nicht vergebenen Primärschlüssel versorgt wird. Dies erfolgt sinnvollerweise in den Konstruktoren z. B. über einen *Universally Unique Identifier* (UUID) oder über einen von der Fachabteilung vorgegebenen Algorithmus. Die folgenden Konstruktoren realisieren die erste Alternative.

```
public Kunde() {
  this.id = UUID.randomUUID().toString();
}

public Kunde(String vorname, String nachname) {
  this();
  this.vorname = vorname;
  this.nachname = nachname;
}
```

Hinweis:

Die von uns verwendete Klasse `UUID` ist keine JPA-spezifische Klasse, sondern im Java-SDK im Package `java.util` enthalten. Hibernate und OpenJPA bieten alternativ eigene UUID-Generatoren an.

Die zentrale Einheit zur Verwaltung von Entities ist die Klasse `EntityManager`. Im Vorgriff auf Kapitel 3, wo wir den Entity-Manager detailliert vorstellen, wollen wir bereits jetzt ein Objekt persistieren und, falls wir den Primärschlüssel kennen, ein Objekt aus der Datenbank in die VM laden. Dazu verwenden wir die Methoden `persist()` und `find()` des Entity-Managers:

```
EntityManager em = ...
Kunde kunde = new Kunde("Max", "Mustermann");
em.getTransaction().begin();
em.persist(kunde);
...
kunde = em.find(Kunde.class, <primaerschluessel>);
em.getTransaction().commit();
```

Die Verwendung einer generierten UUID als Primärschlüssel kann durchaus sinnvoll sein. Wir raten von der Verwendung fachlicher oder sprechender Schlüssel ab und empfehlen dem Leser sogenannte Surrogatschlüssel, also Schlüssel ohne Geschäftssemantik. Dies ist mittlerweile Konsens in der modernen Software-Entwicklung [Amb03, Kar10].

Tipp:

In der Regel sollte Surrogatschlüsseln der Vorzug vor fachlichen Schlüsseln gegeben werden. Die nachträgliche Änderung von Schlüsseln in produktiven Systemen ist sehr aufwendig und bei der Verwendung von fachlichen Schlüsseln nicht auszuschließen.

Operationen des Entity-Managers, die zu Aktionen in der Datenbank führen sollen, müssen in einem Transaktionskontext ausgeführt werden. Dies ist im Beispiel durch eine Transaktionsklammer mit `begin()` und `end()` erkennbar:

```
em.getTransaction().begin();
...
em.getTransaction().commit();
```

Wir gehen auf Transaktionen hier nicht weiter ein, holen dies aber im Kapitel 3 (Objektlebenszyklus) nach und widmen diesem sehr wichtigen Thema das ganze Kapitel 6.

JPA stellt mehrere Alternativen zur Generierung von Primärschlüsseln zur Verfügung. Ausgangspunkt ist die Annotation `@GeneratedValue`, die zusätzlich zur `@Id`-Annotation verwendet wird. Im folgenden Beispiel sind wir dadurch vom Zwang der Generierung eines anwendungsgenerierten Schlüssels befreit.

```
@Entity
public class Kunde {

  @Id
  @GeneratedValue
  private String id;
  private String vorname;
  private String nachname;

  ...
}
```

JPA sieht vier alternative Generatorarten vor:

AUTO
Überlässt der JPA-Implementierung die Wahl des Verfahrens, in der Regel `IDENTITY` oder `SEQUENCE`. Die Wahl ist sowohl provider- als auch datenbankabhängig.

IDENTITY
Verwendet die Datenbankfunktionalität einer selbstinkrementierenden Spalte. Dies sind z. B. die Typ-Zusätze `IDENTITY` bei H2, `SERIAL` bei Postgres und `AUTO_INCREMENT` bei MySQL.

TABLE
Verwendet eine zusätzliche Tabelle zur Verwaltung des zuletzt generierten Primärschlüssels.

SEQUENCE
Verwendet die Datenbankfunktionalität einer *Sequence*. Sequences werden von H2, Oracle, Postgres und DB2 unterstützt.

Die vier genannten Generatorarten sind die Werte der Enum `GenerationType`, die als Werte des Attributs `strategy` der Annotation `@GeneratedValue` verwendet werden. `AUTO` ist der Default, so dass unser obiges Beispiel gleichbedeutend mit der folgenden Verwendung ist.

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private String id;
```

Um eine möglichst portable Anwendung zu erhalten, sollten die Generatorart `AUTO` oder `TABLE` verwendet werden. `AUTO` überlässt dem Provider die Auswahl der Realisierung, und `Table` wird von allen Providern für alle Datenbanken unterstützt. Allerdings sind die von den Providern generierten Tabellen nicht identisch.

Achtung:
Problematisch ist die Verwendung einer Generatorart, die vom Datenbanksystem nicht unterstützt wird. So unterstützt z. B. Oracle keine autoinkrementierenden Spalten. Die Verwendung von IDENTITY mit Hibernate und OpenJPA führt zu einem Fehler. EclipseLink umgeht das Problem durch Verwendung einer Sequence. MySQL kennt keine Sequence. Bei der Verwendung von `SEQUENCE` bricht Hibernate auch hier mit einem Fehler ab. EclipseLink verwendet MySQLs `AUTO_INCREMENT`, OpenJPA erzeugt eine extra Tabelle.

Die Verwendung von Sequences und Tabellen zur Generierung von Primärschlüsseln kann in weiten Teilen konfiguriert werden, so dass wir diese beiden Generatorarten in eigenen Abschnitten behandeln.

2.1.1 Sequence-Generatoren

JPA verwendet eine Default-Sequence, falls diese Generatorart ausgewählt wird. Falls die Schema-Generierung der JPA-Provider verwendet wird, wird neben der Entity-Tabelle auch die Sequence erzeugt. Diese ist global für die Persistenzeinheit und wird daher nur beim ersten derartigen Entity tatsächlich generiert. Bei jeder weiteren Klasse wird die existierende Sequence genutzt. Dieser Default ist sehr einfach zu verwenden:

```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE)
private String id;
```

EclipseLink verwendet die Sequence `SEQ_GEN_SEQUENCE`, Hibernate die Sequence `HIBERNATE_SEQUENCE`. OpenJPA verwendet keine Sequence, sondern erstellt eine Tabelle namens `OPENJPA_SEQUENCE_TABLE`.

Um einen bestimmten Sequence-Generator zu verwenden, dient die Annotation `@SequenceGenerator`, deren Attribute in Tabelle 2.1 dargestellt sind.

TABELLE 2.1 `@SequenceGenerator`-Attribute

`@SequenceGenerator(...)`			
Attribut	**Typ**	**Default**	**Beschreibung**
`name`*	`String`	Generator-name	Eindeutiger Name des Sequence-Generators
`sequenceName`	`String`	provider-spezifisch	Name der Datenbank-Sequence
`allocation-Size`	`int`	50	Der Wert, um den die Sequence inkrementiert wird, wenn Werte alloziert werden
`initialValue`	`int`	1	Der Anfangswert der Sequence
`catalog`	`String`	Default-Katalog	Der Katalog der Sequence
`schema`	`String`	Default-Schema	Das Schema der Sequence

Hinweis:

Annotationen können Attribute besitzen, deren Verwendung obligatorisch oder optional ist. Wir verwenden einen kleinen Stern, um ein Attribut als obligatorisch zu kennzeichnen. Ist ein Attribut nicht gekennzeichnet, ist seine Verwendung optional. Aus Tabelle 2.1 geht also hervor, dass bei Verwendung der Annotation `@SequenceGenerator` immer das Attribut `name` zu benutzen ist.

Das folgende Beispiel definiert und verwendet einen Sequence-Generator. Die Annotation `@SequenceGenerator` kann die Klasse oder die Instanzvariable annotieren. Wir empfehlen die Annotation der Variablen, da die entsprechenden Annotationen dann vereint vor der Primärschlüsselvariablen stehen.

```
@Id
@GeneratedValue(generator = "MySeq")
@SequenceGenerator(name = "MySeq", sequenceName = "MY_SEQ",
                   allocationSize = 100, initialValue = 1)
private Integer id;
```

Das Beispiel weist mit `@GeneratedValue(generator = "MySeq")` JPA an, den Generator namens `MySeq` zu verwenden. Dieser wird mit `@SequenceGenerator` als Sequence-Generator definiert. Der Datenbankname ist `MY_SEQ`. Das Attribut `allocationSize` legt das Inkrement fest, um das die Sequence bei Abruf von Primärschlüsseln erhöht wird. Damit

ist die Implementierung in der Lage, nicht bei jedem neu einzufügenden Datensatz zuerst den Generator nach einer neuen Id zu fragen, sondern kann aus einem eigens dafür angelegten Vorrat schöpfen.

In obigem Beispiel erhält der Entity-Manager die Werte 1 bis 100, um sie als neue Primärschlüssel verwenden zu können. In der Datenbank-Sequence wird 100 als letzter vergebener Wert vermerkt. Sind diese potenziellen Primärschlüssel verbraucht bzw. wird eine neue Persistenzeinheit erzeugt, erhält der Entity-Manager automatisch die Werte 101 bis 200 zur weiteren Verwendung.

Achtung:

Bei der Verwendung eines Sequence-Generators mit OpenJPA muss zusätzlich das `strategy`-Attribut von `@GeneratedValue` verwendet werden. Die Angabe von `@SequenceGenerator` alleine genügt nicht. Soll statt der zusätzlichen Tabelle tatsächlich eine Sequence verwendet werden, so ist das von OpenJPA definierte Literal `native` als Sequence-Name zu verwenden:

```
@Id
@GeneratedValue(generator = "MySeq",
                strategy = GenerationType.SEQUENCE)
@SequenceGenerator(name = "MySeq",
                   sequenceName="native")
private Integer id;
```

2.1.2 Table-Generatoren

Mit Hilfe von Table-Generatoren kann der JPA-Provider die Generierung von Primärschlüsseln auch ohne die Datenbankfunktionalität von Identitätspalten oder Sequences garantieren. Die Verwendung ähnelt der Verwendung von Sequences. Wird der Table-Generator nicht explizit genannt, wird ein Default-Generator verwendet. Bei unseren drei Providern sind dies die Tabellen `SEQUENCE`, `HIBERNATE_SEQUENCE` und `OPENJPA_SEQUENCE_TABLE`. Hierfür genügt auf Entity-Ebene die folgende Deklaration.

```
@Id
@GeneratedValue(strategy = GenerationType.TABLE)
private Integer id;
```

Soll dagegen eine spezielle Tabelle verwendet werden, so ist diese über die Annotation `@TableGenerator` anzugeben. Die Attribute der Annotation stellen wir in Tabelle 2.2 auf der nächsten Seite ausführlich dar und verwenden einige im folgenden Beispiel.

```
@Id
@GeneratedValue(strategy = GenerationType.TABLE,
                generator = "kundeGenerator")
@TableGenerator(name = "kundeGenerator",
                table = "ID_GEN",
                pkColumnName = "GEN_KEY",
                valueColumnName = "GEN_VALUE",
                pkColumnValue = "KUNDE_ID",
                allocationSize = 10)
private Integer id;
```

TABELLE 2.2 `@TableGenerator`-Attribute

`@TableGenerator(...)`			
Attribut	**Typ**	**Default**	**Beschreibung**
`name`*	`String`	—	Eindeutiger Name des Table-Generators
`allocationSize`	`int`	50	Der Wert, um den inkrementiert wird, wenn Werte alloziert werden
`catalog`	`String`	Default-Katalog	Der Katalog der Tabelle
`initialValue`	`int`	1	Der Anfangswert der Primärschlüsselgenerierung
`pkColumnName`	`String`	provider-spezifisch	Name der Spalte, mit der die Ids für das jeweilige Entity unterschieden werden
`pkColumnValue`	`String`	provider-spezifisch	Wert, der einen Primärschlüsselwert von einem anderen Wert unterscheidet. Wird verwendet, wenn für verschiedene Entities eine Tabelle zum Speichern des Primärschlüssels genutzt wird.
`schema`	`String`	Default-Schema	Das Schema der Tabelle
`table`	`String`	provider-spezifisch	Tabellenname
`uniqueConstraints`	`UniqueConstraint[]`	`{}`	Spaltenname(n) für Unique-Constraints
`valueColumnName`	`String`	provider-spezifisch	Name der Tabellenspalte, die den letzten generierten Wert je Entity aufnimmt

Die `@TableGenerator`-Annotation definiert einen Tabellen-Generator namens `kundeGenerator`. Dieser ist global für die Persistenzeinheit und wird im Beispiel im `generator`-Attribut der `@GeneratedValue`-Annotation referenziert. Die Tabelle heißt `ID_GEN` und besitzt die beiden Spalten `GEN_KEY` und `GEN_VALUE`, wobei die erstgenannte der Primärschlüssel der Tabelle ist. Weil die Tabelle als Generator für mehrere Entities verwendet werden kann, wird für das Entity Kunde der Primärschlüsselwert `KUNDE_ID` als String verwendet. Bei Verwendung dieser Tabellenzeile erhöht JPA den Wert um jeweils 10.

Achtung:

Die von uns als *provider-spezifisch* angegebenen Default-Werte der `@TableGenerator`-Attribute haben im API-Doc jeweils den leeren String als Wert. Es ist jedoch angemerkt, dass der Wert provider-spezifisch gewählt wird und somit nicht dem angegebenen leeren String entspricht. Falls etwa `pkColumnValue` nicht verwendet wird, verwendet EclipseLink `kundeGemerator`, Hibernate `KundeTable` und OpenJPA `DEFAULT` als entsprechenden Wert. Entsprechendes gilt für die Attribute `catalog` und `schema` und die entsprechenden Darstellungen in Tabelle 2.1 für die Annotation `@SequenceGenerator`. Für portable Anwendungen empfiehlt sich daher die Verwendung der entsprechenden Attribute.

Da der Generator in der ganzen Persistenzeinheit gültig ist, kann er auch bei anderen Entities z. B. einer Adresse verwendet werden:

```
@Entity
public class Adresse {

  @Id
  @GeneratedValue(strategy = GenerationType.TABLE,
                  generator = "kundeGenerator")
  private Integer id;
  ...
}
```

JPA wird nun für beide Entities den Generator `kundeGenerator` verwenden, was für die Adresse eventuell wegen der Namensgebung zu überdenken wäre.

Tipp:

In der Regel sollte jeder Entity-Typ eine eigene Zeile in der Generatortabelle verwenden. Wir empfehlen außerdem, positive Zahlen als Schlüsselwerte zu benutzen, was bei der Nichtverwendung des `initialValue`-Attributs der `@TableGenerator`-Annotation und somit dem initialen Default-Wert 1 automatisch der Fall ist.

Hinweis:

Bei Verwendung der Identitätsstrategie (`strategy = IDENTITY`) muss der Provider nach Einfügen des Entity in die Tabelle den vom Datenbanksystem erzeugten und genutzten Wert erfragen. Dies ist mit einem gewissen Aufwand verbunden. Bei der alternativen Verwendung von Sequences oder Tabellen kann jeweils über das Attribut `allocationSize` gesteuert werden, wie viele Primärschlüsselwerte ohne weiteren Datenbankzugriff vergeben werden können.

Projekt:

Die in diesem Abschnitt beschriebenen Code-Beispiele zum Thema Primärschlüsselverwendung und -generierung sind im Projekt *entity-id* enthalten.

2.2 POJO-Mapping

Zu Beginn dieses Kapitels haben wir ein Entity aus der JPA-Spezifikation als „*lightweight persistent domain object*“ zitiert. Was bedeutet aber *lightweight*? Damit ist ein einfaches Java-Objekt gemeint, das gemeinhin als POJO (Plain Old Java Object) bezeichnet wird. Da jedoch nicht alle Java-Konstrukte in JPA-Entities erlaubt sind, klären wir zunächst, welche Konstrukte von JPA unterstützt werden und erläutern dann, welche Mapping-Optionen zur individuellen Beschreibung der Persistenz einzelner Objekte existieren.

2.2.1 Was ist ein POJO?

Die JavaBeans-Spezifikation [URL-JB] wurde 1996 veröffentlicht und 1997 zur Version 1.01 überarbeitet. Diese Version ist die zuletzt veröffentlichte und somit die aktuell maßgebliche Definition einer *JavaBean*. Obwohl der Begriff *JavaBean* sogar als Warenmarke registriert ist, wird heute in der Regel eher der nicht näher definierte Begriff eines POJOs verwendet.

Unter einem POJO wird ein Objekt verstanden, das von keiner bestimmten Klasse erben und kein bestimmtes Interface implementieren muss. Damit werden auch keine Callback-Methoden vorausgesetzt. Für die Objektattribute gilt der erweiterte Begriff eines Property, in Anlehnung an die JavaBeans-Spezifikation. Ein Property ist die Gesamtheit eines Objektattributs mit je einem Getter und Setter, die nach dem üblichen Namensschema gebildet werden. Attribute haben meist die Sichtbarkeit `private` oder `protected`, Getter- und Setter-Methoden `public`. Zusätzlich besitzt das POJO einen Default-Konstruktor, also einen Konstruktor ohne Parameter.

Hinweis:

Die JavaBeans-Spezifikation versteht unter einem *Property* die Gesamtheit eines privaten Attributs und des zugehörigen Getter/Setter-Paares. Wir verwenden den Begriff *Property* häufig für das Attribut oder nennen es *Instanzvariable*, weil uns der Begriff eines *Attributs* zu allgemein erscheint. Die Bedeutung wird aber aus dem Verwendungskontext deutlich.

JPA verlangt von Entities einige zusätzliche Eigenschaften, schränkt also die Menge persistierbarer POJOs ein. Generell müssen Entities Klassen sein; Interfaces und Enums sind nicht erlaubt. Die Entity-Klasse sowie ihre Methoden und persistenten Instanzvariablen dürfen nicht als `final` deklariert sein. Als Datentypen der Instanzvariablen sind folgende Typen erlaubt:

- Primitive Datentypen und Strings
- Serialisierbare Datentypen, einschließlich Wrapper der primitiven Datentypen, und benutzerdefinierte serialisierbare Typen
- `BigInteger` und `BigDecimal` des Package `java.math`
- `Date` und `Calendar` des Package `java.util`
- `Date`, `Time` und `Timestamp` des Package `java.sql`
- `byte[]`, `Byte[]`, `char[]`, `Character[]`
- Enumerations
- Entity-Typen und Collections von Entity-Typen
- Eingebettete Klassen
- Collections von einfachen Typen und eingebetteten Klassen

Damit sind praktisch alle in diesem Zusammenhang sinnvollen Datentypen als persistente Properties erlaubt, da in der Regel anwendungsbezogene Klassen als serialisierbare Klassen oder, sehr viel sinnvoller, in Form der letzten drei genannten Datenarten definiert werden können. Das Kapitel 4 widmet sich ausschließlich diesen Themen.

2.2.2 Mapping-Optionen auf Entity-Ebene

Wir haben in Abschnitt 2.1 gesehen, dass für JPA die Angabe der `@Entity`- und `@Id`-Annotationen ausreicht, um aus einem normalen Java-Objekt ein persistentes Java-Objekt zu machen. Dies ist insgesamt relativ wenig Aufwand und vor allem relativ wenig Information, wie das Mapping von Objekten auf Tabellen vorgenommen werden soll. Es kann nur funktionieren, wenn von JPA eine ganze Reihe von Standardeinstellungen als gesetzt angenommen werden, z. B. dass ein String-Property auf eine VARCHAR-Spalte abgebildet wird oder dass alle Properties einer Klasse persistent sein sollen. Ausnahmen von diesen Standardeinstellungen müssen explizit angegeben werden, z. B. dass ein bestimmtes Property nicht persistent sein soll oder dass eine Tabellenspalte mit einem NOT-NULL-Constraint versehen werden soll. Die Annahme sinnvoller Standardwerte ohne ihre explizite Nennung ist unter dem Begriff *Convention over Configuration* bekannt und wurde erstmalig in der Spezifikation EJB 3.0, zu der auch JPA 1.0 gehört, praktiziert. Viele der im Folgenden dargestellten Optionen für das Mapping mit JPA gehören zu diesem Bereich, dienen also zur gezielten Abweichung vom Standardfall.

Wie wir bereits gesehen haben, wird die `@Entity`-Annotation verwendet, um ein POJO persistent zu machen. Die Option `name` ermöglicht die Angabe eines alternativen Namens zur Verwendung in der JPA-Abfragesprache. In Kapitel 7 werden wir die JPA-Abfragesprache JPQL kennenlernen, in denen Entities mit Namen referenziert werden. Ein Konstrukt dieser Abfragesprache ist etwa „`select k from Kunde k where k.nachname like ...`“. Ein Entity wird dabei standardmäßig mit nicht qualifiziertem Klassennamen verwendet. Bei identischen Klassennamen in verschiedenen Packages erzeugt dies einen Namenskonflikt. Mit der Option `name` lässt sich dies vermeiden, indem man einen alternativen Namen für das Entity vergibt. In größeren Projekten erfolgt die so motivierte Namensvergabe sinnvollerweise auf Basis der Packages der Entities. Tabelle 2.3 beschreibt die Verwendung der `@Entity`-Annotation mit dem einzigen Attribut `name`.

TABELLE 2.3 `@Entity`-Attribut

`@Entity(...)`			
Attribut	**Typ**	**Default**	**Beschreibung**
`name`	`String`	Entity-Name	Entity-Name in JPQL

Java und SQL(-Dialekte) haben verschiedene Regeln zur Bildung von Bezeichnern. Hinzu kommt, dass Datenbankadministratoren oder firmeninterne Konventionen meist die Einhaltung bestimmter Regeln fordern. Mit dem `name`-Attribut der `@Table`-Annotation kann der Tabellenname abweichend vom Klassennamen gewählt werden. Ein Katalog- und Schema-Name kann ebenfalls vergeben werden. Häufig sollen eine oder mehrere Spalten mit einem Unique-Constraint versehen werden. Auch dies ist mit der `@Table`-Annotation möglich. Im folgenden Beispiel lautet der Tabellenname `HV_KUNDE`, und die Kombination aus `vorname` und `nachname` ist in dieser Tabelle eindeutig. Das Präfix `HV` soll dabei eine nicht näher erläuterte Namenskonvention der Bank darstellen. Eine Übersicht über die Attribute enthält Tabelle 2.4 auf der nächsten Seite.

```
@Entity
@Table(
    name = "HV_KUNDE",
```

```
    uniqueConstraints = @UniqueConstraint(
        columnNames = {"vorname", "nachname"}))
public class Kunde {
    ...
```

Die Verwendung des `name`-Attributs der Annotation `@Entity` definiert einen vom Default abweichenden Entity-Namen. Da Entity- und Tabellenname im Normalfall übereinstimmen, bedeutet dies, dass sowohl mit der `@Entity`- als auch der `@Table`-Annotation der zu verwendende Tabellenname definiert werden kann. Verwenden Sie beide Annotationen gleichzeitig, wird die `@Table`-Version zur Benennung der Tabelle verwendet, und die `@Entity`-Version bestimmt den Entity-Namen zur Verwendung innerhalb von JPQL.

Tipp:

Die Verwendung von `@Entity(name = "...")` erfolgt in der Regel aus Gründen der Eindeutigkeit des nicht qualifizierten Entity-Namens und nicht, um den Tabellennamen zu definieren. Wir empfehlen daher immer die zusätzliche Verwendung von `@Table(name = "...")`, um den Tabellennamen explizit zu machen.

TABELLE 2.4 `@Table`-Attribute

`@Table(...)`			
Attribut	**Typ**	**Default**	**Beschreibung**
`name`	`String`	Entity-Name	Tabellenname
`catalog`	`String`	Default-Katalog	Katalogname
`schema`	`String`	Default-Schema	Schema-Name
`uniqueConstraints`	`UniqueConstraint[]`	`{}`	Spaltenname(n) für Unique-Constraint

2.2.3 Serialisierbarkeit von Entities

Unsere bisherigen Beispiele verwendeten nicht serialisierbare Klassen als Entities. Dies geschah jedoch lediglich aus Gründen der Lesbarkeit und Übersichtlichkeit. Serialisierbare Entities, also etwa

```
@Entity
public class Kunde implements Serializable {
  ...
```

besitzen eine Reihe von Vorteilen. Bei der Verwendung von persistenten Objekten in Java-EE wurden bis zur Version 1.4 sogenannte *Data Transfer Objects* (DTO) [ACM03] verwendet. Dies ermöglichte die Übertragung von Daten zwischen den einzelnen Schichten einer EE-Anwendung. Das DTO-Pattern ist mit JPA in fast allen Anwendungsfällen obsolet, zumindest aber optional, da JPA-Entities POJOs sind und daher ohne Einschränkung direkt zwischen den Schichten einer Anwendung ausgetauscht werden können. Die zu beachtende Vorgehensweise zur Verwendung sogenannter *detachter* Entities beschreiben wir in Kapitel 3.

Tipp:

Die Serialisierbarkeit eines Entity erleichtert seine Verwendung. Der nicht unwahrscheinliche Fall, dass das Entity mittels RMI übertragen, in einer HTTP-Session gespeichert oder aus anderen Gründen serialisiert werden soll, ist dann ohne nachträgliche Änderung möglich. Die im Buch dargestellten Code-Ausschnitte verzichten lediglich aus Gründen der Lesbarkeit auf die explizite Darstellung der Serialisierbarkeit der Entities.

2.2.4 Der JPA-Zugriff auf Entity-Properties

Beim Zugriff auf die Daten eines Entity muss zwischen dem Zugriff durch den JPA-Provider und dem Zugriff durch uns, den Anwendungsentwickler, unterschieden werden. Der JPA-Provider benötigt den Zugriff, um Daten in die Datenbank schreiben zu können. Analog benötigt er den Zugriff, um nach dem Lesen aus der Datenbank die Daten des Entity setzen zu können. Die *Zugriffsart* (access mode) wurde in JPA 1.0 ausschließlich durch die Position der `@Id`-Annotation definiert. Dazu kann alternativ die Instanzvariable oder der Getter mit dieser Annotation versehen werden, wie die folgende Gegenüberstellung verdeutlicht.

```
@Entity
public class Kunde {

  @Id
  private Integer id;
  ...
```

```
@Entity
public class Kunde {

  private Integer id;

  @Id
  public Integer getId() {
    return id;
  }

  public void setId(Integer id) {
    this.id = id;
  }
  ...
```

Hinweis:

Bei der Zugriffsart spricht die Spezifikation von *field* bzw. *property* based access. Die Übersetzung der ersten Alternative mit *Feld* erscheint uns sehr unglücklich, da sich Feld schon als deutsche Übersetzung des Begriffs Array etabliert hat. Wir sprechen daher immer von Instanzvariablen oder, wenn eine Verwechslung ausgeschlossen ist, kurz von Variablen oder Property. Bei der noch vorzustellenden Annotation `@Access` verzichten wir auf eine Übersetzung und unterscheiden *Field-Access* und *Property-Access*.

Um zu verhindern, dass der interne Zustand eines Entity durch die Anwendung direkt verändert werden kann, dürfen Instanzvariablen nicht `public` sein. Obwohl dies mit Reflection umgangen werden kann, garantiert es eine gewisse Sicherheit für den JPA-Provider. Soll der

Zugriff über die Properties erfolgen, müssen diese die Sichtbarkeit `public` oder `protected` haben.

In JPA 1.0 war die Position der Verwendung der `@Id`-Annotation und somit die Definition der Zugriffsart für eine gesamte Vererbungshierarchie bindend. Um in einer Unterklasse eine andere Zugriffsart zu ermöglichen, aber auch um die Zugriffsart explizit zu machen, führt JPA 2.0 die Annotation `@Access` ein, deren einziges Attribut `value` über die Werte `FIELD` und `PROPERTY` der Enumeration `AccessType` die Zugriffsart definiert. Tabelle 2.5 fasst dies zusammen.

TABELLE 2.5 `@Access`-Attribut

`@Access(...)`			
Attribut	**Typ**	**Default**	**Beschreibung**
`value`*	`AccessType`	—	Zugriffsart für Entity oder Property (`FIELD`, `PROPERTY`)

Das folgende Beispiel definiert die generelle Zugriffsart Field-Access für das Entity durch die Verwendung der `@Access`-Annotation auf Entity-Ebene. Die explizite Angabe der Zugriffsart überschreibt die implizite Zugriffsart durch die Positionierung von `@Id`. Die Annotation `@Id` hätte also auch for dem Getter stehen können. Für das Property `vorname` erfolgt der Zugriff durch den JPA-Provider durch das Getter/Setter-Paar.

```
@Entity
@Access(AccessType.FIELD)
public class KundeAccess {

  @Id @GeneratedValue
  private Integer id;
  private String vorname;
  private String nachname;
  ...

  @Access(AccessType.PROPERTY)
  public String getVorname() {
    return this.vorname;
  }
  public void setVorname(String vorname) {
    this.vorname = vorname;
  }
  ...
```

2.2.5 Mapping-Optionen auf Property-Ebene

Eine Reihe von Annotationen erlaubt die Anpassung des Mappings auf Property-Ebene. Wir beginnen mit der `@Transient`-Annotation, die verhindert, dass ein Property persistent ist, d. h. eine entsprechende Spalte in der Entity-Tabelle existiert. Dies kann alternativ auch mit dem Java-Schlüsselwort `transient` vor der Instanzvariablen erfolgen. Das folgende Beispiel verwendet durch die Postionierung der `@Id`-Annotation den Provider-Zugriff über die Getter und Setter.

```
@Entity
public class KundeTransient {

  private Integer id;
  private String vorname;
  private String nachname;

  public KundeTransient() { }

  @Id
  public Integer getId() { return this.id; }
  public void setId(Integer id) { this.id = id; }

  public String getVorname() { return this.vorname; }
  public void setVorname(String vorname) { this.vorname = vorname; }

  public String getNachname() { return this.nachname; }
  public void setNachname(String nachname) { this.nachname = nachname
      ; }

  @Transient
  public String getName() {
    return vorname + " " + nachname;
  }
  public void setName(String name) {
    String[] namen = name.split(" ");
    this.vorname = namen[0];
    this.nachname = namen[1];
  }
}
```

Die Methoden `getName()` und `setName()` realisieren ein vereinfachtes Arbeiten, wobei wir für das Beispiel voraussetzen, dass jeder Kunde nur einen einfachen Vor- und Nachnamen ohne Leerzeichen besitzt. Ohne die `@Transient`-Annotation geht JPA von vier Properties (`id`, `vorname`, `nachname`, `name`) aus, was in diesem Beispiel zu redundanten Daten in der Datenbank führen würde.

Falls ein einfaches Property, d. h. ein Property, das keine Assoziation repräsentiert, nicht annotiert ist, ist es implizit als

```
@Basic(fetch = FetchType.EAGER, optional = true)
```

konfiguriert. Die dargestellten Attribute sind auch die einzig möglichen. Eine Gesamtdarstellung gibt Tabelle 2.6 auf der nächsten Seite. Interessant ist die Ladeoption `LAZY`, die einen *Hinweis* an den JPA-Provider darstellt, das entsprechende Property erst zu laden, wenn darauf zugegriffen wird. Dies ist z. B. bei großen binären Daten sinnvoll. Auch das Attribut `optional` ist ein Hinweis, und zwar, ob `null` als Wert zugelassen ist. Ein JPA-Provider kann einen Hinweis befolgen, muss es aber nicht. Es kann daher zu Inkompatibilitäten bei einem Provider-Wechsel kommen. Bei der Generierung der Tabellen erzeugt z. B. Hibernate für `optional = false` auch den entsprechenden DDL-Code-Ausschnitt `NOT NULL`, EclipseLink und OpenJPA jedoch nicht.

Die Klassen `java.util.Date` und `java.util.Calendar` repräsentieren einen Zeitpunkt in Millisekundengenauigkeit. Sie werden in der Regel verwendet, um ein Datum, eine Zeit oder Datum und Zeit darzustellen. SQL kennt die Datentypen `date`, `time` und `timestamp`. Für den Geburtstag eines Kunden ist ein Zeitstempel in Millisekundengenauigkeit nicht sinn-

TABELLE 2.6 `@Basic`-Attribute

`@Basic(...)`			
Attribut	**Typ**	**Default**	**Beschreibung**
`fetch`	`FetchType`	`EAGER`	Ladeart (`EAGER`, `LAZY`)
`optional`	`boolean`	`true`	`null` erlaubt

voll, es sollte das alleinige Datum als korrekter SQL-Typ gewählt werden. Dies geschieht über die `@Temporal`-Annotation, wie das folgende Beispiel zeigt. Die Gesamtdarstellung der Annotation zeigt Tabelle 2.7.

```
@Temporal(TemporalType.DATE)
private Date geburtsdatum;
```

TABELLE 2.7 `@Temporal`-Attribute

`@Temporal(...)`			
Attribut	**Typ**	**Default**	**Beschreibung**
`value`*	`TemporalType`	—	Mapping-Optionen für `Date` und `Calendar`: `DATE`, `TIME`, `TIMESTAMP`

Die JPA-Spezifikation fordert die `@Temporal`-Annotation für die Typen `Date` und `Calendar`. Bei Nichtverwendung wirft EclipseLink einen Fehler, während Hibernate und OpenJPA ohne Fehlermeldung von einem `timestamp` ausgehen. Die Datumsvarianten im JDBC-Package, also `java.sql.Date`, `java.sql.Time` und `java.sql.Timestamp`, können ohne Annotation verwendet werden, da der JDBC-Treiber sie direkt unterstützt. Wir raten jedoch von der Verwendung ab, weil das Package `java.sql` als technische Komponente im Business-Modell nicht referenziert werden sollte.

Tipp:

Die Verwendung von `@Temporal` ist bei Properties vom Typ `Date` und `Calendar` verpflichtend. Dies wird lediglich von EclipseLink geprüft und bei Nichtbeachtung mit einer Exception geahndet. Sie sollten `@Temporal` auch bei Hibernate und OpenJPA verwenden, um die Portabilität zu erhöhen.

Bisher haben wir lediglich Zahlen, Zeichen(-ketten) und Datumsangaben behandelt. Datenbanksysteme unterstützen in der Regel aber auch binäre Datenformate, die meist unter dem Oberbegriff *Binary Large Object* (BLOB) bekannt sind. Für große Textdatentypen ist auch der Begriff *Character Large Object* (CLOB) gebräuchlich. JPA verwendet für diese Art des Mappings die Annotation `@Lob` (Large Object). Der tatsächlich im Datenbanksystem verwendete Datentyp ist in der Spezifikation nicht vorgegeben. Jede JPA-Implementierung kann daher unterschiedlich vorgehen. In Tabelle 2.8 auf der nächsten Seite haben wir die von den JPA-Providern verwendeten Datentypen bei der Tabellenerzeugung für einige Datenbanksysteme angegeben. Ein einzelner Datentyp in einem Tabellenfeld bedeutet, dass alle drei Provider diesen Datentyp verwenden. Sind mehrere Datentypen angegeben, so verwenden die Provider bei der Generierung der Tabelle verschiedene Datentypen. Diese sind in der Reihenfolge

EclipseLink, Hibernate, OpenJPA angegeben. Die folgende Entity-Definition wurde für die Bestimmung der Tabelleneinträge verwendet:

```
@Entity
public class AllLob implements Serializable {

    @Lob private byte[] bytes;
    @Lob private Byte[] bytesWrapper;
    @Lob private String string;
    @Lob private char[] chars;
    @Lob private Character[] charsWrapper;
    @Lob private SomeSerializable ss;
    @Lob private java.sql.Blob blob;  // Schluesselwort in MySQL
    @Lob private java.sql.Clob clob;
}
```

TABELLE 2.8 Datentypen bei Verwendung `@Lob`-Annotation

Typ	H2/HSQLDB	Postgres	Oracle	MySQL	DB2
`byte[]/Byte[]`	longvarbinary	bytea oid bytea	blob	longblob longblob blob	blob
`String`	longvarchar	text	clob	longtext longtext text	clob
`char[]/Character[]`	longvarchar	text	clob	longtext longtext text	clob
`Serializable`	longvarbinary	bytea oid bytea	blob	longblob longblob blob	blob
`Blob`	longvarbinary longvarbinary —	bytea oid —	blob blob —	longblob longblob —	blob blob —
`Clob`	longvarchar longvarchar —	text text —	clob clob —	longtext longtext —	clob clob —

Die leeren Einträge für OpenJPA bei `java.sql.Blob` und `java.sql.Clob` bedeuten, dass OpenJPA diese beiden Properties bei der Tabellengenerierung ignoriert, d. h. keine entsprechenden Tabellenspalten erzeugt werden.

Hinweis:

Das Beispiel des Entity `AllLob` zeigt, dass JPA *nicht* in jedem Fall davor schützt, einen gewissen Migrationsaufwand beim Wechsel des Datenbanksystems leisten zu müssen. Im Beispiel haben wir ein Property `blob` genannt, was mit H2/HSQLDB, Postgres, Oracle und DB2 problemlos funktioniert, aber mit MySQL nicht. In MySQL ist `blob` ein Schlüsselwort.

Die `@Lob`-Annotation ist nur für die in Tabelle 2.8 angegebenen Datentypen erlaubt. Ohne die Annotation verwendet z. B. Hibernate bei H2/HSQLDB den Typ `varbinary(255)` und bei Postgres `bytea` für ein Byte-Array.

Für größere binäre Daten bietet sich ein spätes Laden des einzelnen binären Property an. Wir haben dies bereits bei der `@Basic`-Annotation angesprochen. Weil die Verwendung von `LAZY` nur ein Hinweis an den JPA-Provider ist, verfahren die einzelnen Provider tatsächlich verschieden. Es kann sogar sein, dass ein und derselbe Provider den Hinweis unterschiedlich behandelt, abhängig davon, ob das Programm in Java-SE oder Java-EE ausgeführt wird. Wir gehen hierauf in den Abschnitten 11.2, 12.2 und 13.2 ein.

Die Version 5 von Java integrierte Aufzählungstypen, die *Enumerations*, in die Sprache. Das Sprachkonzept einer `enum` wird von JPA direkt unterstützt. Properties eines Enum-Typs werden ohne zusätzliche JPA-Konstrukte automatisch persistiert. Als Beispiel dient der Aufzählungstyp `Anrede` eines Kunden.

```
public enum Anrede {

  HERR, FRAU, DR, PROF, PROF_DR;

}
```

Dieser Typ wird wie gewohnt verwendet:

```
@Entity
public class Kunde {

  @Id @GeneratedValue
  private Integer id;
  ...
  private Anrede anrede;
```

Grundlage der Verwendung ist die Annotation `@Enumerated`, die im Default-Fall (die Nichtverwendung) den Enum-Typ auf eine Integer-Spalte abbildet und die Werte von 0 an hochzählt. Lässt man die Provider die Tabelle generieren, verwenden EclipseLink und Hibernate `Integer`, OpenJPA `Smallint`.

Das einzige Attribut der `Enumerated`-Annotation ist `value` vom Typ `EnumType`, selbst eine Enumeration. Diese besitzt die Werte `ORDINAL` und `STRING`. Die Annotation ist in Tabelle 2.9 in der Übersicht dargestellt.

TABELLE 2.9 `@Enumerated`-Attribut

`@Enumerated(...)`			
Attribut	**Typ**	**Default**	**Beschreibung**
`value`	`EnumType`	`ORDINAL`	Mapping für Aufzählungstypen (`ORDINAL`, `STRING`)

Bei der Verwendung von `STRING` findet ein Mapping auf eine VARCHAR-Spalte statt, wobei als Wert das Enum-Literal als String verwendet wird. Für die Deklaration

```
@Enumerated(EnumType.STRING)
private Anrede anrede;
```

also z. B. für `Anrede.FRAU` der String `"FRAU"`.

Hinweis:

Bei der Änderung einer Enumeration in einem produktiven System mit bestehenden Datenbankeinträgen verhält sich die Integer-Alternative gutartig, wenn neue Enum-Werte an das Ende der Liste angehängt werden, führt aber zu Problemen, wenn Enum-Werte in der Mitte der Liste eingefügt werden. Das Ändern des Enum-Literals führt Java-seitig zu keinen Problemen. Bei der String-Alternative verhält es sich gerade anders herum. Das Ändern führt zu Problemen, Anhängen oder Einfügen nicht.

2.2.6 Die `@Column`-Annotation

Wir behandeln die `@Column`-Annotation in einem eigenen Abschnitt, da sie sehr vielfältige Attribute zulässt. Die `@Column`-Annotation beschreibt detailliert, wie ein einzelnes Property auf eine Tabellenspalte abgebildet wird. Tabelle 2.10 führt die möglichen Attribute auf.

TABELLE 2.10 `@Column-Attribute`

`@Column(...)`			
Attribut	**Typ**	**Default**	**Beschreibung**
`name`	`String`	Property-Name	Spaltenname
`unique`	`boolean`	`false`	Unique-Constraint auf Spalte
`nullable`	`boolean`	`true`	NULL-Werte zugelassen
`insertable`	`boolean`	`true`	Spalte in Insert-Anweisung enthalten
`updatable`	`boolean`	`true`	Spalte in Update-Anweisung enthalten
`columnDefinition`	`String`	`""`	DDL-Anteil für Spaltendefinition
`table`	`String`	primäre Tabelle	Tabelle der Spalte, falls das Entity auf mehrere Tabellen aufgeteilt wird
`length`	`int`	255	Die maximale Länge von Zeichenketten
`precision`	`int`	0	Anzahl Ziffern zur Darstellung
`scale`	`int`	0	Anzahl Nachkommastellen

Häufige Verwendung finden die Attribute `name` und `nullable`, die einen alternativen Spaltennamen und das Zulassen von NULL-Werten beschreiben. Die Attribute `insertable` und `updatable` definieren, ob die entsprechende Spalte in Insert- und Update-Anweisungen verwendet wird. Die Attribute `precision` und `scale` dürfen nur für exakte Dezimalzahlen (`java.math.BigDecimal`) in der dort üblichen Bedeutung verwendet werden. Das Attribut `unique` beschreibt die Eindeutigkeit einer Spalte und bietet sich als Alternative zum `@Table`-Attribut `uniqueConstraints` (siehe Tabelle 2.4) an, wenn die Eindeutigkeit nur für eine und nicht für mehrere Spalten gelten soll.

Wir entwickeln im Folgenden für alle Attribute Beispiele, die auf den folgenden Kundendaten basieren. Dabei soll `eintritt` das Eintrittsdatum, also der Tag der ersten Geschäftsbe-

ziehung mit diesem Kunden und `kontostand` der Stand eines für den Kunden geführten Kontos sein.

```
public class Kunde {
  private String vorname;
  private String nachname;
  private Date eintritt;
  private BigDecimal kontostand;
  ...
```

Der Vorname wird mit den folgenden vier `@Column`-Attributen versehen:

```
@Column(name = "KD_VNAME", nullable = false,
        length = 30, unique = true)
private String vorname;
```

Das Attribut `name` definiert als Spaltennamen `KD_VNAME`. Durch `nullable = false` sind Nullwerte nicht zugelassen (`NOT NULL`). Strings werden standardmäßig in den SQL-Typ `CHARACTER VARYING` mit der Default-Länge 255 abgebildet. Durch `length = 30` wird die Länge auf 30 beschränkt. Die Deklaration `unique = true` verhindert, dass die Spalte mehrere identische Einträge besitzt oder, anders ausgedrückt, dass die Spalte die Schlüsseleigenschaft besitzt. In diesem Beispiel ist dies nicht sinnvoll, bitte stellen Sie sich alternativ eine eindeutige Kundennummer vor.

Mit dem Attribut `columnDefinition` kann der SQL-Code zur Definition der Spalte innerhalb des DDL-Statements der Tabelle angegeben werden. Soll für ein String-Property nicht `CHARACTER VARYING`, sondern `CHARACTER` als Spaltentyp verwendet werden, so kann dies durch die folgende Verwendung des Attributs `columnDefinition` realisiert werden. Zusätzlich definieren wir einen NOT-NULL-Constraint ohne Verwendung des Attributs `nullable` für den Nachnamen.

```
@Column(columnDefinition = "CHARACTER(30) NOT NULL")
private String nachname;
```

Für das Beispiel des Eintrittsdatums definieren wir die Anforderung, dass das Datum nicht von der eventuell falsch eingestellten Client-Uhr, sondern von der Uhr des Datenbanksystems verwendet wird. Die folgenden Attribute der `@Column`-Annotation realisieren dies.

```
@Temporal(TemporalType.DATE)
@Column(columnDefinition = "DATE DEFAULT CURRENT_DATE",
        insertable = false, updatable = false)
private Date eintritt;
```

Beim Erzeugen der Tabelle wird die Spalte `EINTRITT` nun korrekt mit dem Default des aktuellen Tagesdatums erzeugt, das Einfügen eines Datensatzes erfolgt jedoch mit der SQL-Anweisung

```
INSERT INTO KUNDE (ID, NACHNAME, KD_VNAME, KONTOSTAND, EINTRITT)
       VALUES (?, ?, ?, ?, ?)
```

wobei als Wert für `EINTRITT` `null` verwendet wird, wenn die Anwendung keinen expliziten Wert setzt. Die Spalte `EINTRITT` darf also nicht in der `INSERT`-Anweisung auftauchen, was mit `insertable = false` erreicht wird. Die verwendete SQL-Anweisung lautet nun

```
INSERT INTO KUNDE (ID, NACHNAME, KD_VNAME, KONTOSTAND)
       VALUES (?, ?, ?, ?)
```

Um zu verhindern, dass nach dem initialen Einfügen eine spätere Änderung des Eintrittsdatums erfolgen kann, wird zusätzlich `updatable = false` definiert. Die Spalte `EINTRITT` wird dann in keiner `UPDATE`-Anweisung verwendet. Bemerkung: Das SQL-Fragment `DATE DEFAULT CURRENT_DATE` entspricht dem Standard-SQL [KK08], führt jedoch bei MySQL zu einem Syntaxfehler.

Tipp:

Die Kombination von `updatable = false` und `insertable = false` realisiert ein *read-only* Property . Dies ist dann sinnvoll einsetzbar, wenn das Datenbanksystem, eine andere Anwendung oder der JPA-Provider selbst, nicht aber die JPA-Anwendung eine Änderung der Daten vornehmen können soll. Ein Beispiel hierfür entwickeln wir in Abschnitt 12.3.

Als letztes Beispiel eines Kunden-Property wird der Kontostand ebenfalls mit einer `@Column`-Annotation versehen.

```
@Column(precision = 7, scale = 2)
private BigDecimal kontostand;
```

Dadurch wird JPA angewiesen, insgesamt 7 Ziffern, davon 2 Nachkommastellen zur Repräsentation der Zahl zu verwenden. Auch hier verhalten sich die Provider unterschiedlich. In Tabelle 2.11 sind die bei der DDL-Erzeugung verwendeten SQL-Typen für Postgres dargestellt.

TABELLE 2.11 Generierter SQL-Typ für `precision/scale` bei Postgres

Provider	ohne `@Column`	`@Column(precision = 7, scale = 2)`
EclipseLink	`numeric(38,0)`	`numeric(7,2)`
Hibernate	`numeric(19,2)`	`numeric(7,2)`
OpenJPA	`numeric`	`numeric`

OpenJPA ignoriert die Verwendung von `precision` und `scale` und nutzt `numeric` ohne weitere Angaben. In Postgres ist dies gleichbedeutend mit `numeric(30,6)`. Dies bedeutet, dass bei der Verwendung von EclipseLink und Hibernate z. B. der Wert 12,3456 auf 12,35 gekürzt in Postgres gespeichert, während bei der Verwendung von OpenJPA 12,3456 gespeichert wird. Wir verzichten auf eine Untersuchung weiterer Datenbanksysteme und bemerken lediglich, dass OpenJPA bei Oracle tatsächlich `numberic(7,2)` verwendet.

Achtung:

Die Verwendung der Attribute `precision` und `scale` erfolgt bei den JPA-Providern unterschiedlich. So wird z. B. bei Postgres von EclipseLink bei Nichtverwendung der Attribute bei der Tabellengenerierung `numeric(38,0)` verwendet, was zum Abschneiden aller Nachkommastellen führt, während OpenJPA trotz Verwendung der Attribute diese ignoriert. Dies führt zur Anwendung der Default-Werte, bei Postgres `numeric(30,6)`, also einer höheren Darstellungsgenauigkeit als durch die Annotation angegeben.

2.3 Zusammengesetzte Schlüssel

Wir sind bisher immer davon ausgegangen, dass wir aus den in Java definierten Klassen entsprechende Tabellenschemata in SQL generieren, was durch die entsprechenden Provider-Werkzeuge auch unterstützt wird. Die Realität sieht häufig ander aus und verlangt, dass die zu erstellende Java-Anwendung über JPA an eine bereits existierende Datenbank angebunden werden muss. Die Tabellenschemata dürfen nicht geändert werden, da andere Anwendungen diese ebenfalls verwenden. Eine dritte Alternative, die sogenannte *Meet-in-the-Middle*-Variante, ist ebenfalls in der Praxis anzutreffen. Hierbei existiert bereits sowohl die Java-Anwendung als auch die Datenbank. Die Persistenzschicht ist jedoch zu erneuern und durch JPA zu ersetzten.

Wir widmen uns in diesem Abschnitt und im Abschnitt 2.4 Fragen, wie sie häufig im Zusammenhang mit der Verwendung bereits bestehender Datenbanken auftauchen, nämlich der Verwendung zusammengesetzter Schlüssel und der Verwendung mehrerer Tabellen zur Realisierung der Entity-Persistenz.

Unter einem zusammengesetzten Schlüssel versteht man einen aus mindestens zwei Properties bestehenden Schlüssel. Werden etwa in einer Bank oder einem Versicherungsunternehmen Kunden und Verträge nach Abteilungen getrennt, so ist ein Schlüssel für die Abteilung und ein Schlüssel für den Kunden oder den Vertrag ein zusammengesetzter Schlüssel. Ein Nachteil natürlicher und zusammengesetzter Schlüssel ist die enthaltene Bedeutung. Wird diese durch die Fachabteilung oder den Konzern umdefiniert, etwa durch Neustrukturierung von Abteilungen oder Zukäufe und Integration anderer Unternehmen, so müssen alle Primärschlüssel der Datenbank aktualisiert werden. Dies ist eine sehr aufwendige und schwierige Aufgabe, weshalb heute in der Regel von der Verwendung natürlicher und zusammengesetzter Schlüssel abgeraten wird.

Die JPA-Spezifikation sieht zwei alternative Möglichkeiten zur Realisierung zusammengesetzter Schlüssel vor, zum einen die Verwendung der `@IdClass`-Annotation, zum anderen die Verwendung der `@EmbeddedId`-Annotation. Eine dritte Möglichkeit, die die JPA-Spezifikation nicht explizit erwähnt, erhält man durch die Verwendung der `@Embeddable`-Annotation.

Allen drei Alternativen gemein ist, dass der zusammengesetzte Primärschlüssel in einer eigenen Klasse definiert wird. Für unser Beispiel definieren wir neben der Klasse `Kunde` noch die Klasse `KundeId` als Primärschlüsselklasse. Diese muss serialisierbar sein, einen Default-Konstruktor besitzen und die Methoden `equals()` und `hashCode()` definieren.

2.3.1 Realisierung mit `@IdClass`

Wir beginnen zunächst mit der Entwicklung der Primärschlüsselklasse. Listing 2.1 zeigt die Klasse `KundeId`, die die beiden Properties `abt` und `num` enthält, die den zusammengesetzten Primärschlüssel repräsentieren.

LISTING 2.1 Schlüsselklasse `KundeId`

```
public class KundeId implements Serializable {

  private String abt;
```

```
    private int num;

    public KundeId() {
    }

    public KundeId(String abt, int num) {
      this.abt = abt;
      this.num = num;
    }

    @Override
    public boolean equals(Object obj) {
      ...
    }

    @Override
    public int hashCode() {
      ...
    }

    public String getAbt() {
      return abt;
    }

    public int getNum() {
      return num;
    }
  }
```

Da Primärschlüssel zur Identifikation von Entities verwendet werden, müssen wir die Gleichheit von Objekten der Schlüsselklasse definieren. Dies erfolgt durch Überschreiben der Methoden `equals()` und `hashCode()`. Für die beiden Instanzvariablen wurden keine Setter definiert. So verhindert man, dass nach der Erzeugung eines Primärschlüsselobjekts dieses verändert werden kann.

Der definierte Primärschlüssel wird in der Entity-Klasse auf Basis seiner Properties und nicht als Ganzes verwendet. Im Beispiel also `abt` und `num`. Diese Properties sind mit `@Id` zu annotieren. Die Klasse `Kunde` in Listing 2.2 realisiert dieses Schema.

LISTING 2.2 Kunde mit `@IdClass` annotiert

```
1  @Entity
2  @IdClass(KundeId.class)
3  public class Kunde {
4
5    @Id private String abt;
6    @Id private int num;
7    private String vorname;
8    private String nachname;
9
10   public Kunde() {
11   }
12
```

```
13    public Kunde(String abt, int num, String vorname,String nachname) {
14      this.abt = abt;
15      this.num = num;
16      this.vorname = vorname;
17      this.nachname = nachname;
18    }
19
20    ...
21  }
```

Man erkennt in Zeile 2 die `@IdClass`-Annotation mit der in Listing 2.1 definierten Primärschlüsselklasse als Wert des `value`-Attributs.

Das die Schlüsselklasse verwendende Entity muss Properties mit identischem Typ und Namen besitzen, in unserem Fall `String abt` und `int num`. Diese Properties des Schlüssels werden mit `@Id` annotiert. Sollen die Properties vom Default abweichende Eigenschaften besitzen, so sind diese in der Entity-Klasse und nicht in der Primärschlüsselklasse anzugeben. Für eine Größenbeschränkung des Abteilungskürzels also etwa

```
@Entity
@IdClass(KundeId.class)
public class Kunde {

  @Id @Column(length = 5)
  private String abt;
```

und nicht

```
public class KundeId implements Serializable {

  @Column(length = 5)
  private String abt;
```

Die Verwendung des Entities erfolgt ganz intuitiv. Die Properties werden wie gewohnt verwendet. Auch das Persistieren erfolgt ohne Änderung, wie im folgenden Beispiel.

```
Kunde kunde = new Kunde("VERTR", 4711, "Max", "Mustermann");
em.persist(kunde);
```

Lediglich bei der `find()`-Methode ist die Schlüsselklasse zu verwenden.

```
Kunde kunde = em.find(Kunde.class, new KundeId("VERTR", 4711));
```

Achtung:

Die Generierung von zusammengesetzten Schlüsseln bzw. von Teilen des Schlüssels ist nicht erlaubt. Die Annotation des `num`-Properties mit `@GeneratedValue` ist also nicht zulässig.

Projekt:

Die Verwendung eines zusammengesetzten Schlüssels mit `@IdClass` wird im Projekt *entity-id-class* demonstriert.

2.3.2 Realisierung mit `@EmbeddedId`

Die zweite Möglichkeit zur Realisierung eines Entity mit zusammengesetztem Primärschlüssel ist die Verwendung der `@EmbeddedId`-Annotation zur Kennzeichnung des Schlüsselproperties. Das Schlüsselproperty des Entity ist bei dieser Alternative vom Typ der Schlüsselklasse. Einzubettende Klassen müssen generell mit `@Embeddable` anotiert werden. Dies ist die einzige Änderung an der Schlüsselklasse des Beispiels (Listing 2.1):

```
@Embeddable
public class KundeId implements Serializable {
  ...
```

Das Entity wird wie gewohnt ausschließlich mit `@Entity`, das Property des zusammengesetzten Schlüssels mit `@EmbeddedId` annotiert. Listing 2.3 zeigt dies.

LISTING 2.3 Kunde mit `@EmbeddedId`-Annotation

```
@Entity
public class Kunde {

  @EmbeddedId
  private KundeId id;
  private String vorname;
  private String nachname;

  public Kunde() {
  }
  public Kunde(KundeId id, String vorname, String nachname) {
    this.id = id;
    this.vorname = vorname;
    this.nachname = nachname;
  }
  ...
```

Auch hier gilt wie im vorherigen Abschnitt, dass die Schlüsselklasse selbst keine Entsprechung in der Datenbank hat und die Properties der Schlüsselklasse in der Entity-Tabelle zu Spalten werden. Das Annotieren von Properties kann nur in der Schlüsselklasse geschehen, da die Properties in der Entity-Klasse nicht vorhanden sind. Dies ist unschön, weil dann die Definition von Spalten einer Entity-Tabelle in einer anderen Klasse und nicht im Entity vorgenommen werden. Einen Ausweg stellen hier die Annotationen `@AttributeOverrides` und `@AttributeOverride` bereit, die Property-Attribute überschreiben können. Die beiden Annotationen sind in den Tabellen 2.12 und 2.13 mit ihren Attributen dargestellt. Die obige Entity-Definition kann mit diesen Annotationen um abweichende Spaltennamen in der Entity-Tabelle erweitert werden:

```
@EmbeddedId
@AttributeOverrides({
  @AttributeOverride(name = "abt",
                     column= @Column(name = "abteilung", length=5)),
  @AttributeOverride(name = "num", column= @Column(name = "nummer"))
})
private KundeId id;
```

TABELLE 2.12 `@AttributeOverrides`-Attribute

`@AttributeOverrides(...)`			
Attribut	**Typ**	**Default**	**Beschreibung**
`value`*	`Attribute-Override[]`	—	Liste von zu überschreibenden Mapping-Attributen

TABELLE 2.13 `@AttributeOverride`-Attribute

`@AttributeOverride(...)`			
Attribut	**Typ**	**Default**	**Beschreibung**
`name`*	`String`	—	Spaltenname
`column`*	`Column`	—	Spaltendefinition

Die durch die Primärschlüsselklasse und das Entity mit den angegebenen `@AttributeOverride/s`-Annotationen entstehende Tabelle zeigt Abbildung 2.1. Man erkennt die Properties beider Klassen sowie die überschriebenen Spaltenbezeichner.

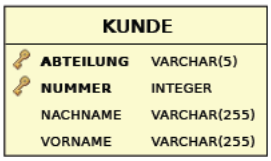


ABBILDUNG 2.1 Tabellenschema mit `@AttributeOverrides`

Bei der Verwendung der Klasse muss lediglich das neue Property im Konstruktor beachtet werden. Man hätte dies auch unverändert lassen und die `KundeId`-Instanz im Konstruktor erzeugen können.

```
...
kunde = new Kunde(new KundeId("VERTR", 4711), "Max", "Mustermann");
em.persist(kunde);
...
kunde = em.find(Kunde.class, new KundeId("VERTR", 4711));
...
```

In Vorgriff auf Kapitel 7 Datenbankabfragen weisen wir darauf hin, dass der Primärschlüssel als Property in der JPQL verwendet werden kann:

```
...
Query query = em.createQuery("select k from Kunde k where id = :id");
query.setParameter("id", new KundeId("VERTR", 4711));
Kunde kunde = (Kunde) query.getSingleResult();
...
```

Achtung:

Laut Spezifikation müssen Entity-Klassen und Embeddable-Klassen bei Java-SE in der Konfigurationsdatei der Persistenzeinheit, der Datei `persistence.xml`, mit dem `<class>`-Element aufgezählt werden. Im Beispiel besteht OpenJPA auf die Aufzählung beider Klassen. EclipseLink begnügt sich mit der Entity-Klasse, während Hibernate vollständig ohne die Verwendung von `<class>` auskommt.

Projekt:

Das Projekt *entity-id-embeddedid* verwendet einen zusammengesetzten Schlüssel mit `@EmbeddedId` und `@Embeddable`.

2.3.3 Realisierung mit `@Embeddable`

Die dritte und letzte Alternative zur Realisierung zusammengesetzter Schlüssel ist in der JPA-Spezifikation nicht explizit als solche genannt. Wir zitieren aus Abschnitt 11.1.13 der Spezifikation:

> „The Embeddable annotation is used to specify a class whose instances are stored as an intrinsic part of an owning entity and share the identity of the entity."

Wir haben die `@Embeddable` -Annotation bereits in Abschnitt 2.3.2 verwendet, um die so annotierte Klasse durch die Verwendung von `@EmbeddedId` in der Entity-Klasse als Primärschlüssel zu verwenden. Wenn man statt `@EmbeddedId` nun `@Id` verwendet, erhält man den Standardfall der Verwendung einer Klasse als persistenten Property-Typ und reduziert den oben genannten Fall einer gemeinsamen Identität (share the identity) auf die alleinige Identität durch genau dieses Property.

```
@Entity
public class Kunde {

  @Id private KundeId id;
  private String vorname;
  private String nachname;
  ...
```

Als Klasse zur Repräsentation des zusammengesetzten Schlüssels verwenden wir weiterhin die Klasse `KundeId` aus Listing 2.1. Die Klasse kann nun mit `@Embeddable` annotiert werden oder nicht. Falls man die Klasse nicht annotiert, wird sie als gewöhnlicher, benutzerdefinierter serialisierbarer Typ verwendet (siehe Anforderungen an POJO-Properties in Abschnitt 2.2.1). Falls man sie mit `@Embeddable` annotiert, aber in der Entity-Klasse wie in unserem Beispiel mit `@Id` und nicht mit `@EmbeddedId` verwendet, definiert die Spezifikation kein explizites Mapping. Das Verhalten ist also provider-spezifisch. Tabelle 2.14 auf der nächsten Seite zeigt das von den Providern zugrunde gelegte Mapping des Property `KundeId id`, das bei der Verwendung von `Embeddable` durchaus unterschiedlich ist.

TABELLE 2.14 Mapping des Property `KundeId id`

Provider	mit `@Embeddable`	ohne `@Embeddable`
EclipseLink	Property `id`	Property `id`
Hibernate	Properties `abt`, `num`	Property `id`
OpenJPA	Properties `abt`, `num`	Property `id`

2.4 Mehrere Tabellen für ein Entity

JPA erlaubt es, ein feinkörniges Objektmodell zu realisieren. Im Bereich der OR-Mapper versteht man darunter die Möglichkeit, mehr Klassen als Tabellen zu haben, um so dem objektorientierten Ziel kleiner Klassen näher zu kommen. Dies wird z. B. durch einbettbare Objekte und Vererbung (Kapitel 5) realisiert.

Es ist in der Praxis aber auch der umgekehrte Fall anzutreffen. Als Anwendungsbeispiel soll bereits eine Datenbank existieren, bei der Kunden und deren Adressen in zwei Tabellen realisiert sind. Das sogenannte Legacy-Schema (Legacy engl. für Altlast, Erbe, Vermächtnis) mit den beiden Tabellen ist in Abbildung 2.2 abgebildet. Die Beziehung ist auf Datenbankebene eine 1:n-Beziehung, worauf wir in Kapitel 4 näher eingehen. Fürs Erste gehen wir davon aus, dass ein Kunde eine Adresse besitzt und nicht mehrere und die Properties der Adresse in der Klasse `Kunde` enthalten sind. Die Adressdaten gehören zu diesem Kunden und werden nicht mit anderen Kunden geteilt.

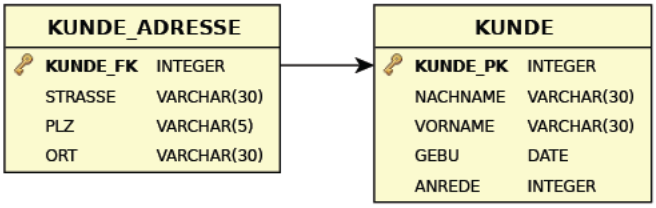


ABBILDUNG 2.2 Legacy-Kundenschema

Unser Ziel ist es, *ein* Entity `Kunde` zu entwerfen, das seine Daten in den bereits bestehenden beiden Tabellen verwaltet bzw. von JPA verwalten lässt. JPA sieht hierfür die `@SecondaryTable`-Annotation vor, deren Attribute in Tabelle 2.15 aufgezählt sind.

Die grundsätzliche Idee bei der Verwendung der `@SecondaryTable`-Annotation ist die Angabe der zusätzlichen Tabelle auf Entity-Ebene und die Annotation jedes Property, das in der Sekundärtabelle abgebildet wird, durch die `@Column`-Annotation und deren `table`-Attribut. Listing 2.4 zeigt die Realisierung des Kunden-Entity für die beiden in Abbildung 2.2 gezeigten Tabellen.

TABELLE 2.15 `@SecondaryTable`-Attribute

`@SecondaryTable(...)`			
Attribut	**Typ**	**Default**	**Beschreibung**
`name`*	`String`	—	Name der Tabelle
`catalog`	`String`	Default-Katalog	Der Katalog der Tabelle
`pkJoinColumns`	`PrimaryKey-JoinColumn[]`	`{}`	Spalte(n) für Join mit der Primärtabelle. Default sind PK-Spalten der Primärtabelle.
`schema`	`String`	Default-Schema	Das Schema der Tabelle
`unique-Constraints`	`Unique-Constraint[]`	`{}`	Spaltenname(n) für Unique-Constraints

LISTING 2.4 Kundenrealisierung mit `@SecondaryTable`

```
1  @Entity
2  @SecondaryTable(name = "KUNDE_ADRESSE",
3    pkJoinColumns={@PrimaryKeyJoinColumn(name = "KUNDE_FK")})
4  public class Kunde  {
5
6    @Id @GeneratedValue
7    @Column(name = "KUNDE_PK") private Integer id;
8    @Column(length = 30)
9    private String vorname;
10   @Column(length = 30)
11   private String nachname;
12   @Temporal(TemporalType.DATE) @Column(name = "GEBU")
13   private Date geburtsdatum;
14   private Anrede anrede;
15
16   @Column(table = "KUNDE_ADRESSE", length = 30)
17   private String strasse;
18   @Column(table = "KUNDE_ADRESSE", length = 5)
19   private String plz;
20   @Column(table = "KUNDE_ADRESSE", length = 30)
21   private String ort;
22   ...
```

Die Annotation `@SecondaryTable` in den Zeilen 2 und 3 definiert den Namen der Sekundärtabelle mit dem zwingend erforderlichen `name`-Attribut. Das Attribut `pkJoinColumns` definiert den Fremdschlüssel in die Primärtabelle. Wenn diese einen zusammengesetzten Schlüssel besitzt, wie wir ihn in Abschnitt 2.3 eingeführt haben, sind hier mehrere Spalten anzugeben. Sowohl ein Spaltenname der Primär- als auch ein Spaltenname der Sekundärtabelle werden durch `@Column`-Annotationen abweichend vom Property-Namen definiert: `KUNDE_PK` in Zeile 7 und `GEBU` in Zeile 12. Die abweichende Zuordnung eines Property in die Sekundärtabelle erfolgt ebenfalls in der `@Column`-Annotation. Die Verwendung der `table`-Option für die Properties `strasse` (Zeile 16), `plz` (Zeile 18) und `ort` (Zeile 20) definiert `KUNDE_ADRESSE` als entsprechende Tabelle für diese Properties.

Ein Entity kann mehrere Sekundärtabellen haben. Diese werden über die `@SecondaryTables`-Annotation angegeben, die eine Liste von `@SecondaryTable`-Annotationen erwartet. Da dies das einzig mögliche Attribut ist, verzichten wir auf die Darstellung des Attributs in Tabellenform.

Die Struktur bei der Verwendung mehrerer Sekundärtabellen entspricht schematisch dem folgenden Muster:

```
@Entity
@SecondaryTables({
    @SecondaryTable(
        name = "TAB_1",
        pkJoinColumns={@PrimaryKeyJoinColumn(name = "FK_1")}),
    @SecondaryTable(
        name = "TAB_2",
        pkJoinColumns={@PrimaryKeyJoinColumn(name = "FK_2")}),
    @SecondaryTable(
        name = "TAB_3",
        pkJoinColumns={@PrimaryKeyJoinColumn(name = "FK_3")})
})
public class SehrGrossesEntity {
    ...
```

Projekt:

Die Programmtexte dieses Abschnitts über Sekundärtabellen finden Sie im Projekt *secondary-table*.

3 Objektlebenszyklus

Alle persistenten Objekte einer JPA-Anwendung werden durch sogenannte Entity-Manager verwaltet. Eine oder mehrere Instanzen des Interfaces `EntityManager` sind dafür verantwortlich, die Entities der Anwendung und die Inhalte der Datenbank konsistent zu halten. Die von einem Entity-Manager verwalteten Objekte werden als *Persistenzkontext* (Persistence Context) bezeichnet. Der Entity-Manager stellt sicher, dass die Primärschlüssel innerhalb eines Persistenzkontextes eindeutig sind, dass also ein Entity in einem Persistenzkontext nur einmal existiert. Neben der Garantie dieser Objekteindeutigkeit stellt der Entity-Manager Methoden bereit, um Objekte zu persistieren, aus der Datenbank zu lesen und um Objekte zu aktualisieren und zu löschen.

Jeder Entity-Manager wird durch eine Fabrik, das Interface `EntityManagerFactory`, erzeugt. Die Konfiguration aller Entity-Manager einer solchen Fabrik erfolgt über eine *Persistenzeinheit* (Persistence Unit) mit Hilfe der Konfigurationsdatei `persistence.xml`. Diese ist mit einem eindeutigen Namen versehen, so dass in einer Anwendung zwischen mehreren Entity-Manager-Fabriken und somit auch zwischen mehreren Verbindungen zu verschiedenen Datenbanken unterschieden werden kann.

Bei der Erzeugung und Verwendung des Entity-Managers muss zwischen Java-SE- und Java-EE-Anwendungen differenziert werden. In einer Java-SE-Anwendung wird ein Entity-Manager explizit, also programmatisch erzeugt. In der verwalteten Umgebung eines Java-EE-Application-Servers erfolgt die Erzeugung (in der Regel) automatisch durch den Application-Server. Man spricht auch von *anwendungsverwalteten* und *container-verwalteten* Entity-Managern. Die Zustände, die ein Entity bezüglich der Verwaltung durch einen Entity-Manager annehmen kann, sind jedoch in Java-SE und Java-EE identisch. Wir beschreiben daher zunächst im folgenden Abschnitt diese Zustände, um danach die Verwendung des Entity-Managers in Java-SE-Anwendungen in Abschnitt 3.2 und die Verwendung innerhalb eines Java-EE-Application-Servers in Abschnitt 3.3 darzustellen.

3.1 Der Zustand eines Entity

Bevor wir auf die einzelnen Operationen für das Lesen, Schreiben, Ändern und Löschen von Entities eingehen, sollen die in JPA möglichen Zustände eines Entity charakterisiert werden. Üblicherweise wird unter dem Zustand eines Objekts die Menge der aktuellen Property-

Werte verstanden. Hier wollen wir unter einem Zustand den Verwaltungszustand eines Entity aus Sicht des Entity-Managers verstehen. Die JPA-Spezifikation kennt hierfür vier Zustände.

- **Transiente** Objekte wurden noch nicht in die Obhut des Entity-Managers übergeben und besitzen noch kein Äquivalent in der Datenbank.
- Dagegen sind **verwaltete** Objekte (managed object) zum Zeitpunkt der Betrachtung unter der Kontrolle des Entity-Managers und in der Datenbank persistent.
- **Losgelöste** Objekte (detached) sind augenblicklich nicht an einen Entity-Manager gebunden, besitzen aber im Unterschied zu transienten Objekten ein Äquivalent in der Datenbank. Sie wurden bereits von einem Entity-Manager in der Datenbank gespeichert, wurden von diesem aber wieder getrennt.
- Sind Objekte hingegen mit einem Entity-Manager bzw. dessen Persistenzkontext verbunden und in der Datenbank gespeichert, aber zum Löschen vorgemerkt, werden sie im Rahmen der Spezifikation als **gelöschte** (removed object) Objekte bezeichnet.

Abbildung 3.1 zeigt die möglichen Zustandsübergänge, die die Methoden des Interface `EntityManager` auslösen. Ein Konstruktoraufruf mit `new` ist offensichlich keine derartige Methode, aber der Vollständigkeit halber dargestellt. Mit Ausnahme der Methoden `find()`, `getReference()` und `clear()` besitzen alle dargestellten Methoden das relevante Entity-Objekt als einzigen Parameter. Die Methode `refresh()` ist überladen und besitzt weitere Parameter, z. B. einen Lock-Modus, auf den wir in Kapitel 6 eingehen.

Die Übersicht ist nicht vollständig, da zum einen das `EntityManager`-API noch weitere Methoden besitzt und zum anderen andere Ereignisse, wie etwa das Ende einer Transaktion, einen Zustandsübergang hervorrufen. Der Einfachheit halber wollen wir diese Punkte aber zunächst unberücksichtigt lassen und uns im Rahmen der Einführung auf die Zustandsübergänge durch die dargestellten Methoden des Entity-Managers konzentrieren. Für einen vollständigen Überblick über die Methoden des Interface `EntityManager` verweisen wir auf das API-Doc.

3.2 JPA in Java-SE-Anwendungen

JPA 1.0 entstand im Rahmen der EJB-3-Spezifikation. Dies unterstreicht die Enterprise-Ausrichtung von JPA. Der Einsatz in Java-EE-Umgebungen wird sicher deutlich häufiger als der Einsatz in Java-SE-Umgebungen erfolgen. JPA hat aber auch in Java-SE eine Daseinsberechtigung und ist aus didaktischen Gründen und vor allem durch den Verzicht auf einen Application-Server für eine Einführung vorzuziehen. Wir empfehlen auch dem Leser mit ausschließlichem Interesse für den Java-EE-Bereich, diesen Abschnitt durchzuarbeiten und sich erst danach dem Abschnitt 3.3 zu widmen.

Eine Java-SE-Anwendung ist eine gewöhnliche Applikation mit einer Main-Methode oder eine Servlet-Anwendung, bei der der Servlet-Container keinerlei Funktionalität zur Persistenz von Objekten bereitstellt. Die JPA-Spezifikation spricht in diesem Fall von einem Entity-Manager, der unter Verwaltung der Applikation steht (anwendungsverwaltet, application managed). Der Anwendungsentwickler muss selbst die Initialisierung des Entity-Managers, den Beginn und das Ende einer Transaktion sowie das Schließen des Entity-Managers über-

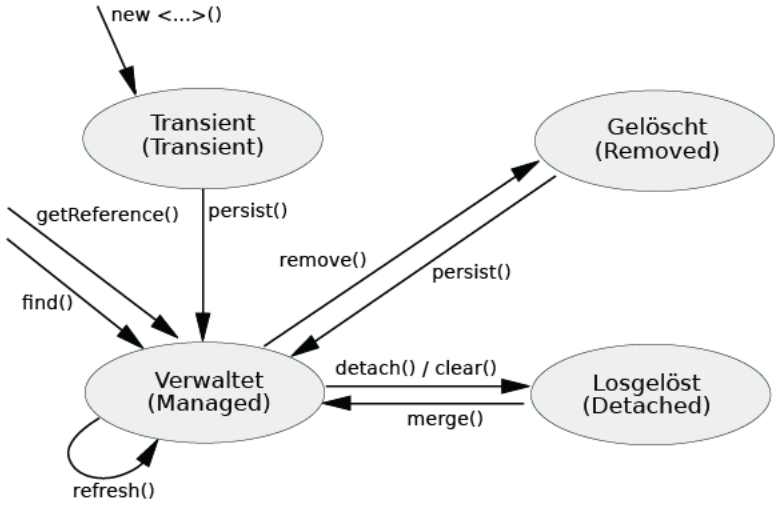


ABBILDUNG 3.1 Objektlebenszyklus eines Entity

nehmen. Die Beispiele des Kapitels 2 haben eine solche Umgebung vorausgesetzt. Der nachfolgende Programmausschnitt zeigt noch einmal schematisch das Vorgehen:

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("bank");
EntityManager em = emf.createEntityManager();

// Arbeiten mit dem Persistenzkontext ...

em.close();
emf.close();
```

Das Muster des abgebildeten Programmausschnitts, also das Erzeugen der Entity-Manager-Factory und des Entity-Managers und das Schließen dieser beiden Objekte ist immer anzuwenden, so dass wir es in den folgenden Beispielen in der Regel nicht mehr explizit aufführen, sondern uns auf das oben auskommentierte Arbeiten mit dem Persistenzkontext konzentrieren.

3.2.1 Speichern von Entities

Um transiente Entities in der Datenbank zu speichern, stellt das Interface `EntityManager` die Methode `persist()` bereit, die ein Entity persistent macht und in den Zustand *verwaltet* überführt:

```
Kunde kunde = new Kunde("Heidi", "Mustermann", "22.6.1965");
em.persist(kunde);
System.out.println("Im Persistenzkontext enthalten: "
                   + em.contains(kunde));  // true
```

Der Kunde ist nun im Persistenzkontext enthalten, wie die Methode `contains()` zeigt, aber noch nicht in der Datenbank. Der Grund hierfür ist, dass der Persistenzkontext erst am Ende einer Transaktion mit der Datenbank synchronisiert wird. Im obigen Beispiel hat damit das Entity auch noch keinen Primärschlüssel zugewiesen bekommen, falls als Generierungsart des Schlüssels `IDENTITY` gewählt wurde.

In einer Java-SE-Anwendung werden *Resource-Local*-Transaktionen verwendet, die explizit durch die Anwendung kontrolliert werden müssen. Die Schnittstelle zur Transaktionsverwaltung ist das Interface `EntityTransaction`. Eine Instanz erhält man durch die Entity-Manager-Methode `getTransaction()`. Die Schnittstelle umfasst die erwarteten Methoden `begin()`, `commit()` und `rollback()` vom Typ `void`, sowie die boolesche Methode `isActive()`, die anzeigt, ob zur Zeit (Ausführungspunkt des aktuellen Threads) eine Transaktion aktiv ist. Da in einer Java-SE-Anwendung die Anwendungs selbst für das Zurückrollen einer Transaktion verantwortlich ist, enthält `EntityTransaction` noch die beiden Methoden `setRollbackOnly()` und `getRollbackOnly()`. Wir gehen auf die Transaktionsverwaltung detailliert in Kapitel 6 ein.

In Erweiterung des Beispiels um die Verwendung einer Transaktion kann diese nun die Entity-Erzeugung und das Persistieren umfassen:

```
em.getTransaction().begin();
Kunde kunde = new Kunde("Heidi", "Mustermann", "22.6.1965");
em.persist(kunde);
em.getTransaction().commit();
```

oder aber vollständig danach ablaufen:

```
Kunde kunde = new Kunde("Heidi", "Mustermann", "22.6.1965");
em.persist(kunde);
em.getTransaction().begin();
em.getTransaction().commit();
```

Beide Fälle sind im Ergebnis identisch, da nur das Ende der Transaktion für die Synchronisation des Persistenzkontextes mit der Datenbank ausschlaggebend, der Start jedoch irrelevant ist. Die Spezifikation erlaubt es einem JPA-Provider, *jederzeit* vor dem Commit in die Datenbank zu schreiben. Ein JPA-Provider wird jedoch den Zeitpunkt des Schreibens möglichst lange hinauszögern, um im Falle eines Zurückrollens möglichst wenig Aufwand – auch in der Datenbank – zu haben. Der spätestmögliche Zeitpunkt ist direkt vor Ausführung der `commit()`-Methode.

Das Schreiben kann jedoch auch programmatisch explizit erzwungen werden. Die Methode `flush()` synchronisiert den gesamten Persistenzkontext mit der Datenbank.

```
// Erzeugen von 4 Kunden bereits erfolgt
em.getTransaction().begin();
em.persist(kunde1);
em.persist(kunde2);
em.flush();
em.persist(kunde3);
em.persist(kunde4);
em.getTransaction().commit();
```

Im Beispiel werden zunächst die INSERT-Anweisungen für die ersten beiden Kunden abgesetzt, erzwungen durch den expliziten Aufruf von `flush()`. Danach werden die beiden anderen Kunden in die Datenbank eingefügt, da die Transaktion beendet wird. Die Verwen-

dung von `flush()` kann sinnvoll sein, z. B. um bei der Schlüsselgenerierungsart `IDENTITY` den generierten Primärschlüssel verwenden zu können. Im Beispiel ist dieser nach dem `flush()`-Aufruf für `kunde1` und `kunde2` vorhanden, für `kunde3` und `kunde4` ist er `null`. In der Regel ist die Verwendung von `flush()` jedoch nicht erforderlich. An dieser Stelle muss noch angemerkt werden, dass `flush()` die Synchronisation mit der Datenbank erzwingt, aber nach wie vor die ACID-Eigenschaften der Transaktion erhalten bleiben, im obigen Fall also entweder alle vier oder keiner der Kunden in die Datenbank geschrieben werden.

Hinweis:

Die Methode `persist()` ist nicht die einzige Möglichkeit, Entities zu persistieren. In Kapitel 4 führen wir Beziehungen zwischen Entities und das Konzept der transitiven Persistenz (Persistence by Reachability) ein. Dies führt dazu, dass beim Persistieren eines Entity alle in Beziehung stehenden Entities ebenfalls persistiert werden.

3.2.2 Laden von Entities

Im Kapitel 7 wird eine textuelle Anfragesprache und in Kapitel 8 ein auf Java-Objekten basierendes Anfrage-API definiert, die beide eine Menge von Objekten in den Persistenzkontext laden. In diesem Abschnitt stellen wir die Methoden

```
public <T> T find(Class<T> entityClass, Object primaryKey)
public <T> T getReference(Class<T> entityClass, Object primaryKey)
```

vor, die bei Kenntnis des Primärschlüssels eines Entity genau dieses Entity laden, falls es existiert. Die genannten Methoden existierten bereits in JPA 1.0. In JPA 2.0 wurde die Methode `find()` überladen. Hinzu kamen drei Versionen mit Parametern für den Lock-Modus, eine Map von Properties und Hints (Hinweise mit detaillierteren Angaben zur Verwendung) und die Kombination dieser beiden Parameter. Auf den Lock-Modus gehen wir in Kapitel 6 näher ein. Ein Beispiel für einen Hint liefern wir am Ende dieses Abschnitts.

Betrachten wir zunächst die Methode `find()`. Sie liefert das gefundene Entity bzw. `null` zurück, wenn es keinen entsprechenden Datenbankeintrag gibt. Als Parameter sind die Entity-Klasse und der Primärschlüssel zu übergeben. Bei anderen Parametertypen wird eine `IllegalArgumentException` geworfen.

```
Kunde kunde = em.find(Kunde.class, 66); // null oder Kunde
System.out.println(kunde);
```

Die `find()`-Methode sucht zunächst im Persistenzkontext nach dem entsprechenden Entity. Falls es vorhanden ist, wird dieses Objekt zurückgegeben. Nur wenn sich das Objekt nicht im Persistenzkontext befindet, wird eine SELECT-Anweisung an die Datenbank geschickt. Dies bedeutet insbesondere, dass wiederholte `find()`-Aufrufe mit demselben Primärschlüssel nur zu einer SELECT-Anweisung führen.

Hinweis:

Der Persistenzkontext agiert als JPAs First-Level-Cache. Objekte, die sich bereits im Persistenzkontext befinden, werden nicht noch einmal geladen, außer durch den expliziten Aufruf noch vorzustellender Methoden. Der Second-Level-Cache, auf den wir in Abschnitt 6.3.5 eingehen, wurde in JPA 2.0 neu in die Spezifikation aufgenommen und war in JPA 1.0 vollständig provider-abhängig.

Im Unterschied zu `find()` *kann* die Methode `getReference()` ein Objekt zurückliefern, das als Proxy für das eigentliche Entity fungiert. Ein weiterer Unterschied ist das Verhalten im Falle eines nicht vorhandenen Datenbankeintrags. Während `find()` `null` zurückgibt, wirft `getReference()` eine `EntityNotFoundException`. Der Zeitpunkt des Werfens der Exception ist dem JPA-Provider freigestellt. Es kann beim Aufruf der Methode oder beim ersten Zugriff auf das Entity erfolgen, da erst dann das `SELECT` an die Datenbank abgesetzt wird.

```
em.getTransaction().begin();
Kunde kunde = em.getReference(Kunde.class, 66); // Exception moeglich
System.out.println("Id: " + kunde.getId());
System.out.println(kunde.getNachname()); // spaetestens hier Except.
em.getTransaction().commit();
```

Im Beispiel kann die Exception bei Nichtexistenz eines Kunden mit Primärschlüssel 66 erst beim Zugriff auf den Nachnamen oder bereits beim Aufruf von `getReference()` geworfen werden, je nachdem, ob der JPA-Provider von der Möglichkeit des späten Ladens der Property-Werte Gebrauch macht oder nicht.

Der erste Zugriff auf spät geladene Properties darf nicht nach dem Loslösen des Entity vom Persistenzkontext (Abschnitt 3.2.5) erfolgen, da die Properties in diesem Fall noch nicht initialisiert wurden und durch das Loslösen auch nicht mehr initialisiert werden können.

Die Freiheiten der Spezifikation bezüglich der Methode `getReference()` werden von den drei Providern sehr unterschiedlich genutzt. EclipseLink realisiert die Methode durch ein `SELECT` analog zu `find()` und erzeugt ein Objekt der Klasse `Kunde`. Hibernate setzt kein `SELECT` ab und erzeugt ein Proxy-Objekt. OpenJPA setzt ebenfalls kein `SELECT` ab, erzeugt aber ein Objekt der Klasse `Kunde`. Erfolgt der Zugriff auf die Id, setzt Hibernate ein `SELECT` ab, OpenJPA nicht. Erfolgt der Zugriff auf andere Properties, setzt OpenJPA schließlich das `SELECT` ab. Das geschilderte Verhalten betrifft den Default-Fall, d. h. ohne weitere Maßnahmen. Bei der Überarbeitung der Entity-Klasse durch provider-spezifische Werkzeuge kann sich das Verhalten ändern. Die Überarbeitung wird bei EclipseLink *Weaving*, bei Hibernate *Instrumentation* und bei OpenJPA *Enhancement* genannt. Wir gehen darauf detailliert in den Abschnitten 11.2, 12.2 und 13.2 ein.

Tipp:

Die unterschiedlichen Realisierungen von `getReference()` bei den drei Providern beeinflusst die Portabilität negativ. Andererseits ist der Aufwand für das Erzeugen des Entity über einen bekannten Primärschlüssel mit `find()` sehr gering. Wir raten daher zur generellen Verwendung von `find()`.

Die Methoden `find()` und `getReference()` lassen sich auch dann verwenden, wenn das Entity einen zusammengesetzten Primärschlüssel verwendet. In diesem Fall werden den Methoden entsprechend initialisierte Instanzen der Primärschlüssel-Klasse übergeben. Beispiele dazu findet der Leser am jeweiligen Ende der Abschnitte 2.3.1 und 2.3.2.

Um diesen Abschnitt über das Laden von Entities zu vervollständigen, gehen wir noch auf die bereits erwähnten, überladenen Versionen der `find()`- Methode ein, die in JPA 2.0 neu aufgenommen wurden.

```
public <T> T find(Class<T> entityClass, Object primaryKey,
                  LockModeType lockMode)
public <T> T find(Class<T> entityClass, Object primaryKey,
                  LockModeType lockMode, Map<String, Object> hints)
public <T> T find(Class<T> entityClass, Object primaryKey,
                  Map<String, Object> hints)
```

Wir konzentrieren uns auf die letzte Methode, da wir Lock-Modi erst in Kapitel 6 behandeln. Unter einem Hint versteht JPA zusätzliche Informationen für den Provider, z. B. um Performanzsteigerungen zu ermöglichen. Es wird zwischen JPA-Hints und provider-spezifischen Hints unterschieden. Für JPA ist das Präfix `javax.persistence` reserviert. Hints werden tatsächlich als Hinweise interpretiert, müssen also nicht vom Provider beachtet werden.

JPA definiert in Version 2.0 lediglich die beiden Hints

```
javax.persistence.lock.timeout
javax.persistence.query.timeout
```

die wir an dieser Stelle noch nicht sinnvoll einsetzen können. Wir greifen daher auf den EclipseLink-Hint `eclipselink.read-only` zurück, der dafür sorgt, dass das zurückgelieferte Entity nicht verwaltet wird und somit Änderungen am Objekt nicht mit der Datenbank synchronisiert werden.

```
em.getTransaction().begin();
Map<String,Object> map = new HashMap<String, Object>();
map.put("eclipselink.read-only", true);
Kunde kunde = em.find(Kunde.class, 66, map);
// kunde verwenden
em.getTransaction().commit();
```

Im Beispiel wird der Kunde read-only geladen. Dies bedeutet, dass Änderungen am Kunden nicht mit der Datenbank synchronisiert werden. Das Synchronisieren mit der Datenbank, also das Aktualisieren der Datenbank bezüglich des Entity-Zustands, ist das Default-Verhalten von JPA, wie wir im nächsten Abschnitt zeigen.

3.2.3 Aktualisieren von Entities

Wir erwähnten bereits, dass verwaltete Entities im Persistenzkontext überwacht werden und der Entity-Manager dafür verantwortlich ist, Änderungen am Entity mit der Datenbank zu synchronisieren. In der Literatur wird dieser Vorgang als *Automatic Dirty Checking* bezeichnet. Ein Objekt wird als *dirty* bezeichnet, wenn es Änderungen enthält, die noch nicht in der Datenbank festgehalten wurden. Durch diesen Automatismus ist es nicht notwendig, eine spezielle Methode für die Aktualisierung aufzurufen, wie folgendes Beispiel zeigt.

```
em.getTransaction().begin();
Kunde kunde1 = em.find(Kunde.class, <pk>);
// Aenderungen am verwalteten Objekt:
kunde1.setNachname("Mustermann");
// bei commit wird automatisch synchronisiert
em.getTransaction().commit();
```

Obwohl nur die `find()`-Methode des Entity-Managers aufgerufen wurde, wird vor dem Ende der Transaktion eine `UPDATE`-Anweisung an die Datenbank abgesetzt, um den Nachnamen zu aktualisieren. Auch das Muster zum Persistieren von Objekten mit `persist()` wird derartig erweitert, so dass der Entwickler die Reihenfolge der Operationen vernachlässigen kann, wie das folgende Beispiel zeigt.

```
em.getTransaction().begin();
Kunde kunde2 = new Kunde("Liesbet", "Musterfrau", "13.3.1971");
em.persist(kunde2);
//Aenderungen am verwalteten Objekt:
kunde2.setNachname("Meier");
// bei commit wird automatisch synchronisiert
em.getTransaction().commit();
```

Hinweis:

JPA definiert lediglich, dass Änderungen an verwalteten Entities am Ende der Transaktion mit der Datenbank zu synchronisieren sind, jedoch nicht, wie dies zu erfolgen hat. Im letzten Beispiel setzen EclipseLink und OpenJPA eine einzige `INSERT`-Anweisung für „Liesbet Meier" ab, während Hibernate zunächst ein `INSERT` für „Liesbet Musterfrau" und dann ein `UPDATE` für den Nachnamen (sogar für alle Properties) absetzt.

Bisher haben wir unter der Fragestellung der Aktualisierung eines Entity nur die Richtung von der JVM in die Datenbank diskutiert. Es ist jedoch auch die umgekehrte Richtung für die Praxis relevant. Da Anwendungen und Datenbank mehrbenutzerfähig sind, kann ein anderer Benutzer derselben oder einer anderen Anwendung die Daten eines Entity in der Datenbank geändert haben, während das Entity in einem Persistenzkontext verwaltet wird. Die Methode `refresh()` aktualisiert den Zustand eines verwalteten Entity mit den aktuellen Daten der Datenbank. Eventuelle Änderungen in der JPA-Anwendung werden daher überschrieben.

```
em.getTransaction().begin();
Kunde kunde = em.find(Kunde.class, <pk>);
kunde.setNachname("Mustermann");
em.refresh(kunde);
em.getTransaction().commit();
```

Im Beispiel enthalten nach dem `refresh()` alle Properties des Kunden die aktuellen Werte aus der Datenbank. Der Nachname „Mustermann" wurde ebenfalls mit dem entsprechenden Datenbankwert überschrieben und ist somit für die Anwendung verloren.

Die Methode `refresh()` erhielt mit JPA 2.0 drei zusätzliche überladene Versionen, die nach dem Muster der Methode `find()` die Angabe eines Lock-Modus sowie von Hints erlauben. Die Verwendung erfolgt analog zu `find()`, so dass wir auf eine Darstellung verzichten.

3.2.4 Löschen von Entities

Bei der gewöhnlichen Java-Programmierung werden nicht mehr referenzierte Objekte automatisch vom Garbage-Collector (vollständig) entfernt. Bei der Verwendung persistenter JPA-Entities kann dies nicht mehr funktionieren, es muss explizit gelöscht werden, um auch die entsprechenden Daten in der Datenbank zu löschen. Die Methode `remove()` löscht das als Parameter angegebene Entity.

```
em.getTransaction().begin();
Kunde kunde = em.find(Kunde.class, <pk>);
em.remove(kunde);
// alternativ natuerlich auch so moeglich:
em.remove(em.find(Kunde.class, <pk>));
em.getTransaction().commit();
```

Auch das Löschen eines Entity findet nicht beim Aufruf der Methode, sondern erst am Ende der Transaktion statt. Der für die Praxis eher hypothetische Fall, dass ein zur Löschung vorgemerktes Entity doch nicht gelöscht werden soll, kann durch einen hinzugefügten `persist()`-Aufruf realisiert werden, wie im folgenden Beispiel angedeutet.

```
em.getTransaction().begin();
Kunde kunde = em.find(Kunde.class, 80);
em.remove(kunde);
// ...
em.persist(kunde);
em.getTransaction().commit();
```

3.2.5 Weitere Operationen auf Entities

Mit den bisher vorgestellten Methoden des Entity-Managers haben wir die typische CRUD-Funktionalität (Create, Read, Update, Delete) abgedeckt und stellen diese noch einmal in Tabelle 3.1 einander gegenüber.

TABELLE 3.1 CRUD-Funktionalität im Vergleich

CRUD	SQL	Entity-Manager
Create	`INSERT`	`persist()`
Read	`SELECT`	`find()`
Update	`UPDATE`	automatisch
Delete	`DELETE`	`remove()`

Abbildung 3.1 auf Seite 57 zeigt neben den bereits vorgestellten Methoden `getReference()` und `refresh()` noch drei weitere Methoden ohne Entsprechung im CRUD-Muster, da sie mit der internen Funktionsweise des Persistenzkontextes zu tun haben. Es sind die Methoden `detach()`, `clear()` und `merge()`, wobei die beiden ersten Methoden ein bzw. alle Entities eines Persistenzkontextes von diesem lösen.

Doch was bedeutet es, dass ein Entity vom Persistenzkontext losgelöst ist? Das Entity befand sich zu einem früheren Zeitpunkt im Persistenzkontext, jedoch wurde diese Zuordnung

gelöst, so dass der Persistenzkontext den Zustand des Entity nicht mehr überwacht und insbesondere nicht mehr in der Pflicht steht, Zustandsänderungen mit der Datenbank zu synchronisieren. Das Loslösen vom Persistenzkontext kann in verschiedenen Situationen entstehen. Das Commit einer Transaktion eines transaktionsbegrenzten (transaction-scoped) Persistenzkontexts löst ebenso wie ein Rollback alle Entities vom entsprechenden Persistenzkontext. Das Schließen eines anwendungsverwalteten Entity-Managers mittels `close()` löst ebenfalls alle Entities. Im Java-EE-Container kommen noch weitere Möglichkeiten hinzu, auf die wir im Abschnitt 3.3 eingehen. Hier beschränken wir uns zunächst auf die beiden Methoden `detach()` und `merge()`, die jeweils die inverse Operation realisieren.

Das Loslösen eines Entity vom Persistenzkontext durch expliziten Aufruf von `detach()` wird selten benötigt. Auch die `merge()`-Methode findet im Java-SE-Umfeld selten Gebrauch, sehr wohl aber im Java-EE-Umfeld. Die `merge()`-Methode wird verwendet, um ein losgelöstes Entity wieder in den Persistenzkontext einzufügen. Als Parameter von `merge()` wird das einzufügende Entity verwendet. Der Rückgabewert ist das Ergebnis des Einfügens, also das wiedereingefügte Entity. Das Parameter-Entity wird nicht verändert und sollte in der Regel nicht weiterverwendet werden.

Obwohl `detach()` in Kombination mit `merge()` in Java-SE-Anwendungen selten Verwendung findet, wollen wir ein Beispiel für die beiden Methoden entwickeln. Das in Listing 3.1 wiedergegebene Beispiel verdeutlicht die Auswirkung des Loslösens und Einfügen eines Entity. Das Verständnis dieser Mechanismen ist für Java-EE besonders wichtig, da das Loslösen dort automatisch geschieht. Es kann dort aber nicht so anschaulich demonstriert werden.

LISTING 3.1 Die Methoden `detach()` und `merge()`

```
1  Kunde kunde1 = new Kunde("Kauf", "Wenig", "11.07.1960");
2  Kunde kunde2 = new Kunde("Spar", "Viel", "16.02.1958");
3
4  em.getTransaction().begin();
5  em.persist(kunde1);
6  em.detach(kunde1);
7  System.out.println("kunde1 im PC enthalten:: "
8                     + em.contains(kunde1));  // false
9  System.out.println("kunde2 im PC enthalten:: "
10                    + em.contains(kunde2));  // false
11 kunde2 = em.merge(kunde1);
12 System.out.println("kunde1 im PC enthalten:: "
13                    + em.contains(kunde1));  // false
14 System.out.println("kunde2 im PC enthalten:: "
15                    + em.contains(kunde2));  // true
16 kunde1.setNachname("Kunde1");
17 kunde2.setNachname("Kunde2");
18 em.getTransaction().commit();
```

Im Beispiel wird lediglich der Kunde `kunde1` persistiert (Zeile 5), um sofort wieder losgelöst zu werden. Die Aufrufe von `contains()` in den Zeilen 7/8 und 9/10 liefert für beide Entities `false`. Durch das Einfügen von `kunde1` in den Persistenzkontext (Zeile 11) werden dessen Property-Werte in `kunde2` übernommen, dem Rückgabewert des `merge()`-Aufrufs. `kunde1` ist damit weiterhin nicht im Persistenzkontext enthalten, `kunde2` sehr wohl. Die Ergebnisse der `contains()`-Aufrufe in den Zeilen 12/13 und 14/15 sind also `false` und `true`. Die Än-

derung der Nachnamen in den Zeilen 16 und 17 werden letztendlich nur für `kunde2` in die Datenbank geschrieben, da `kunde1` nicht verwaltet ist. Nach Ende des Programmausschnitts befinden sich die Daten *Kauf, Kunde2, 11.07.1960* in der Datenbank.

Zum Abschluss der Diskussion von `detach()` und `merge()` bleibt noch anzumerken, dass in der Regel das Einfügen eines Entity mit `merge()` nach dem Muster

```
kunde = em.merge(kunde);
```

erfolgt, also Parameter und Rückgabewert von `merge()` dasselbe Objekt sind.

Während `detach()` ein einziges Entity aus dem Persistenzkontext löst, werden beim Aufruf von `clear()` alle Entities gelöst. Dies ist in der Regel nicht nötig und nur bei lang lebenden Persistenzkontexten mit vielen Entities sinnvoll. Wenn in einer SE-Anwendung ein Entity-Manager über eine Anfrage Tausende von Entities in den Persistenzkontext lädt, einige von diesen ändert und zurückschreibt und danach keines der Entities mehr benötigt, können alle Entities losgelöst werden. Sie belegen dann innerhalb des Persistenzkontextes keinen unnötigen Platz und können vom Garbage-Collector entfernt werden.

In Kapitel 4 führen wir Beziehungen zwischen Objekten und deren Eigenschaften ein. Wenn etwa ein Kunde ein Konto besitzt und das Konto persistiert wird, so wird bei entsprechender Konfiguration der Beziehung das Konto automatisch auch persistiert. Die Beziehungen werden hierfür über das Enum `CascadeType` konfiguriert. Dieser kennt die Werte `DETACH`, `MERGE`, `PERSIST`, `REFRESH` und `REMOVE`. Sie realisieren jeweils das Kaskadieren der Funktionalität der Entity-Manager-Methoden gleichen Namens, also `detach()`, `merge()`, `persist()`, `refresh()` und `remove()`. Sollen alle Funktionalitäten kaskadiert werden, so besitzt das Enum den weiteren Wert `ALL`. Wir gehen auf das Kaskadieren detailliert in Kapitel 4 ein.

3.2.6 Konfiguration

Das Interface `EntityManager` stellt das zentrale JPA-API dar. Ein Entity-Manager verwaltet über eine bestimmte Zeit eine Menge von Entities. Diese Menge von Entities wird *Persistenzkontext* genannt. Wir haben diesen Begriff und den Begriff einer Persistenzeinheit am Anfang dieses Kapitels oberflächlich eingeführt und wollen diese nun genauer darstellen. Abbildung 3.2 zeigt auf der linken Seite die Klassen bzw. Interfaces `Persistence`, `EntityManagerFactory` und `EntityManager` sowie auf der rechten Seite die Persistenzeinheit und den Persistenzkontext. Die folgenden Darstellungen beziehen sich auf Java-SE. Die grundlegenden Konzepte gelten auch für Java-EE, weichen jedoch in Details ab, auf die wir in Abschnitt 3.3.1 eingehen.

Die Persistenzeinheit definiert eine Menge von Entity-Klassen, deren Instanzen durch `EntityManager`-Instanzen verwaltet werden. Dazu werden Klassen einer Anwendung mit ihrem Mapping (Annotationen oder XML) auf eine (einzige) Datenbank zusammengefasst und mit einem Namen versehen. Dies geschieht in der Konfigurationsdatei `persistence.xml`, die sich im `META-INF`-Verzeichnis der Anwendung bzw. des Jars befinden muss. Die Konfiguration der Persistenzeinheit `bank`, die wir in den bisherigen Beispielen des Abschnitts 3.2 verwendet haben, ist in Listing 3.2 dargestellt.

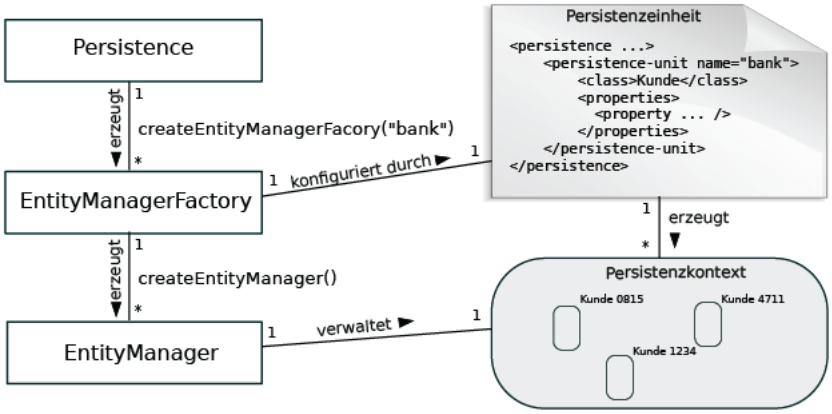


ABBILDUNG 3.2 Klassen und Konzepte der Persistenz

LISTING 3.2 Konfiguration der Persistenzeinheit in `persistence.xml`

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
    version="2.0">

    <persistence-unit name="bank" transaction-type="RESOURCE_LOCAL">
        <class>de.jpainfo.model.Kunde</class>
        <properties>
            <property name="javax.persistence.jdbc.driver"
                      value="org.h2.Driver" />
            <property name="javax.persistence.jdbc.url"
                      value="jdbc:h2:tcp://localhost/bank" />
            <property name="javax.persistence.jdbc.user"
                      value="sa" />
            <property name="javax.persistence.jdbc.password"
                      value="" />
    </properties>
    </persistence-unit>
</persistence>
```

Im `<persistence>`-Element werden über die Attribute des Elements Namensraum und Schema definiert. Falls die Neuerungen von JPA 2.0 verwendet werden sollen, ist auf die entsprechende Version (`2_0` bzw. `2.0`) zu achten. Das Attribut `name` des `<persistence-unit>`-Elements definiert den Namen der Persistenzeinheit, in unserem Fall `bank`. Das Attribut `transaction-type` definiert die Art bzw. Ebene der Transaktionsverwaltung. Der Wert `RESOURCE_LOCAL` stellt die einfachste Art, die einer lokalen Ressource dar. In unserem Fall ist dies eine Datenbank. `RESOURCE_LOCAL` ist der Default für anwendungsverwaltete Persistenzeinheiten, so dass die Angabe des `transaction-type`-Attributs im Beispiel auch entfallen kann. Die möglichen Transaktionsarten werden in der Enumeration `PersistenceUnitTransactionType` im Package `javax.persistence.spi` definiert.

Die möglichen Werte sind `RESOURCE_LOCAL` und `JTA`. Wir verwenden JTA-Transaktionen im Abschnitt 3.3 und widmen uns im Kapitel 6 allgemein dem Thema Transaktionen.

In einer Java-SE-Umgebung müssen Entity-Klassen, einbettbare Klassen und gemappte Oberklassen mit dem `<class>`-Element explizit in der Persistenzeinheit aufgezählt werden. In Beispiel ist dies nur die Klasse `Kunde`. Das `<properties>`-Element enthält beliebig viele `<property>`-Elemente, die Name/Wert-Paare definieren. Die möglichen Werte waren in JPA 1.0 ausschließlich provider-abhängig. Mit JPA 2.0 wurde das Präfix `javax.persistence` spezifiziert, das im Beispiel mit dem weiteren Bestandteil `jdbc` schließlich `driver`, `url`, `user` und `password` festlegt.

Achtung:

Persistente Entity-Klassen sind laut Spezifikation mit dem `<class>`-Element explizit zu nennen. EclipseLink und OpenJPA folgen dieser Anforderung, Hibernate nicht. Hibernate nimmt alle Entity-Klassen auch ohne explizite Nennung in die Persistenzeinheit auf.

Die Präfixe der von uns verwendeten drei Provider sind `eclipselink`, `hibernate` und `openjpa`. Das folgende Beispiel zeigt für jeden Provider ein Beispiel aus dem Bereich der Tabellenverwaltung.

```
<property name="eclipselink.ddl-generation"
    value="drop-and-create-tables" />
<property name="hibernate.hbm2ddl.auto"
    value="create-drop" />
<property name="openjpa.jdbc.SynchronizeMappings"
    value="buildSchema(SchemaAction='add,deleteTableContents')"/>
```

Während der Entwicklung einer Anwendung kann es sehr praktisch sein, dass die Tabellen-Schemata bei Änderung der Entity-Klassen ebenfalls geändert bzw. neu erzeugt werden. Die drei Beispiele realisieren ein derartiges Verhalten zum Teil sehr unterschiedlich. EclipseLink löscht beim Start der Anwendung zunächst die Tabellen der Entities, um sie danach sofort wieder zu erzeugen. Hibernate erzeugt die Tabellen beim Start der Anwendung und löscht sie am Ende der Anwendung. OpenJPA sieht keine Möglichkeit des Löschens vor, kann aber eine Tabelle erzeugen bzw., wenn sie bereits vorhanden ist, ihre Inhalte löschen.

Projekt:

Die im Abschnitt 3.2 beschriebenen Code-Beispiele zum Thema JPA in Java-SE-Anwendungen sind im Projekt *objektlebenszyklus* enthalten.

3.3 JPA in Java-EE-Anwendungen

Die für den Java-SE-Bereich vorgestellten Konzepte, Klassen und Interfaces existieren selbstverständlich auch für bzw. in einer Java-EE-Umgebung. Die Art der Verwendung ist jedoch eine andere und impliziert damit auch andere Verwendungsmuster. Abbildung 3.3 zeigt die

für Java-EE überarbeitete Version der Abbildung 3.2 auf Seite 66. Wir gehen zunächst auf den Standardfall, den container-verwalteten Entity-Manager, ein.

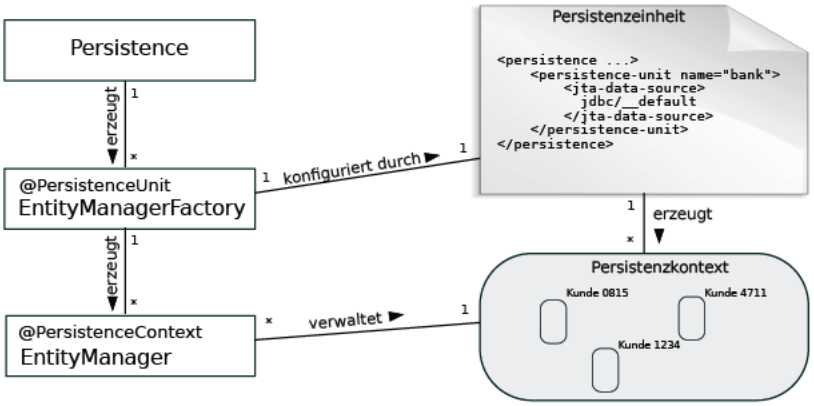


ABBILDUNG 3.3 Klassen und Konzepte der Persistenz in Java-EE

3.3.1 Konfiguration

Die Aufrufe von `createEntityManagerFactory()` und `createEntityManager()` erfolgen nicht in der Anwendung, sondern durch den Application-Server. Sie sind daher in Abbildung 3.3 nicht mehr explizit dargestellt. Weil ein Application-Server hochentwickelte Verfahren zur effizienten Ressourcenverwaltung nutzt, erfolgt die Konfiguration von JPA aus anwendungslogischer Sicht zwar weiterhin in der Konfigurationsdatei `persistence.xml`, sie wird jedoch um container-spezifische Konfigurationen erweitert. Die Open-Source-Application-Server GlassFish [URL-GF] und JBoss-AS [URL-JBoss] implementieren in den Versionen 3 und 6 bzw. 7 die Java-EE-6-Spezifikation und damit die Version 2.0 von JPA. GlassFish verwendet EclipseLink, JBoss-AS verwendet Hibernate. Da beide Application-Server einer Open-Source-Lizenz unterliegen (es gibt jeweils auch ein kommerzielles Lizenzmodell), raten wir dem Leser, einen der beiden Server zu verwenden.

Hinweis:

Welche Version eines Application-Servers tatsächlich eine Java-EE-Spezifikation erfüllt, kann auf den Kompatibilitätsseiten von Oracle [URL-EECOMP] und auf den entsprechenden Herstellerseiten nachgelesen werden. Für uns ist hier nur die Lauffähigkeit unserer Beispielanwendung interessant.

Die genannten container-spezifischen Konfigurationen erlauben z. B. die Verwendung eines Pools von Datenbankverbindungen. Da die von JPA benötigte Verbindung zur Datenbank nur zu einem Bruchteil der gesamten Anwendungszeit benötigt wird, ist es möglich, deutlich weniger Verbindungen zur Datenbank zu verwenden als z. B. Benutzer gleichzeitig angemeldet

sind. Dies spart Kosten, da das Lizenzmodell von Datenbanksystemen häufig auf der Anzahl gleichzeitiger Verbindungen basiert. Abbildung 3.4 zeigt die Konfiguration eines JDBC-Connection-Pools über das graphische Front-End des GlassFish-Application-Servers.

Hinweis:

Auf die Installation und Konfiguration eines Application-Servers können wir hier nicht im Detail eingehen. Sie gestalten sich in der Regel sehr einfach und werden von uns im Netz beschrieben.

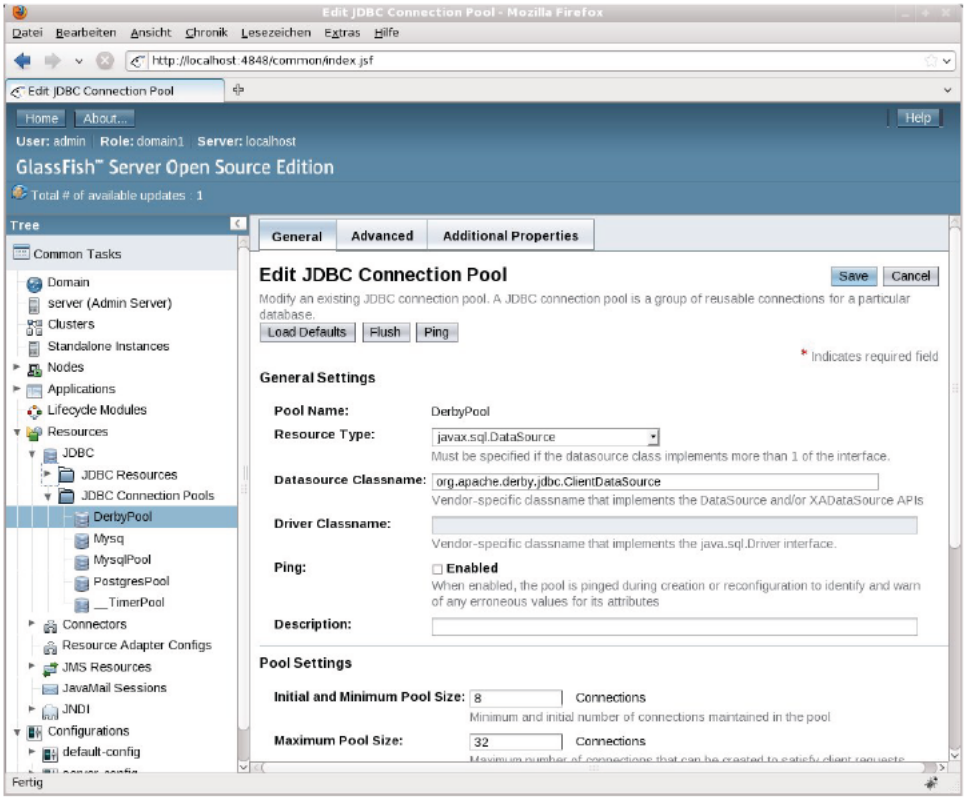


ABBILDUNG 3.4 GlassFish JDBC-Connection-Pools

Als JDBC-Connection-Pool ist „`DerbyPool`“ im linken Panel selektiert. Dies ist der Connection-Pool der in GlassFish vorhandenen Derby- bzw. Java-DB. Es muss also keine externe Datenbank für die ersten Gehversuche mit GlassFish installiert werden. Im rechten Panel erkennt man im unteren Bereich, dass die Pool-Größe auf 8–32 Verbindungen eingestellt ist.

JDBC-Connection-Pools werden im GlassFish als Basis einer JDBC-Ressource verwendet. Abbildung 3.5 zeigt die Konfiguration der JDBC-Ressource „`jdbc/__default`“, was ebenfalls der JNDI-Name der Ressource ist. Dieser JNDI-Name wird in der Datei `persistence.xml` als JTA-Data-Source verwendet.

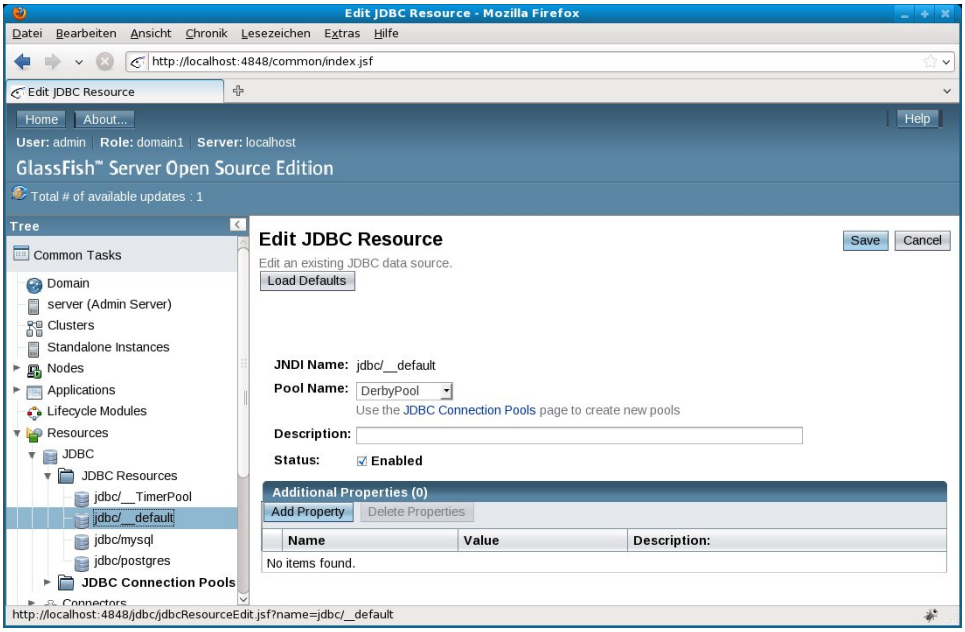


ABBILDUNG 3.5 GlassFish JDBC-Ressourcen

Listing 3.3 zeigt die Definition der Persistenzeinheit für GlassFish und die JTA-Data-Source „`jdbc/__default`“. Die Unterschiede zur Persistenzeinheit in Listing 3.2 auf Seite 66 für Java-SE sind:

- der Wegfall der JDBC-Verbindungsdaten, da diese bereits durch die Ressource-Definitionen des Application-Servers vorgenommen wurden;
- stattdessen die Angabe einer JTA-Data-Source; (`<jta-data-source>`)
- der Wegfall der Aufzählung der Entity-Klassen in `<class>`-Elementen, da in Java-EE der Application-Server den Classpath durchsucht und automatisch alle mit `@Entity` annotierten Klassen berücksichtigt werden;
- die Verwendung einer anderen Transaktionsverwaltung (JTA).

LISTING 3.3 Konfiguration der Persistenzeinheit in `persistence.xml` (Java-EE)

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
    version="2.0">

    <persistence-unit name="bank" transaction-type="JTA">
      <jta-data-source>jdbc/__default</jta-data-source>
    </persistence-unit>

</persistence>
```

In einer container-verwalteten Umgebung ist eine JTA-Transaktionsverwaltung der Default, so dass die Angabe `transaction-type="JTA"` auch entfallen kann. Eine ausführliche Darstellung aller Aspekte der Transaktionsverwaltung erfolgt in Kapitel 6. Die Angabe provider-spezifischer Properties, wie in Abschnitt 3.2.6 beispielhaft verwendet, ist ebenfalls möglich.

Eines der Ziele von Java-EE war die Spezifikation einer allgemeinen Infrastruktur und Ablaufumgebung für Unternehmensanwendungen, um den Portierungsaufwand zwischen diesen Strukturen und Umgebungen zu minimieren. Die Beispielanwendung zur Kundenverwaltung, deren JPA-spezifischen Teile wir in den folgenden Abschnitten darstellen, ist daher sowohl auf GlassFish 3 als auch auf JBoss-AS 6 bzw. 7 lauffähig. Der JBoss-Application-Server enthält die in Java implementierte Datenbank HSQLDB [URL-HSQLDB], deren Data-Source-Definition in der Administrationsoberfläche des JBoss-AS 6 in Abbildung 3.6 dargestellt ist.

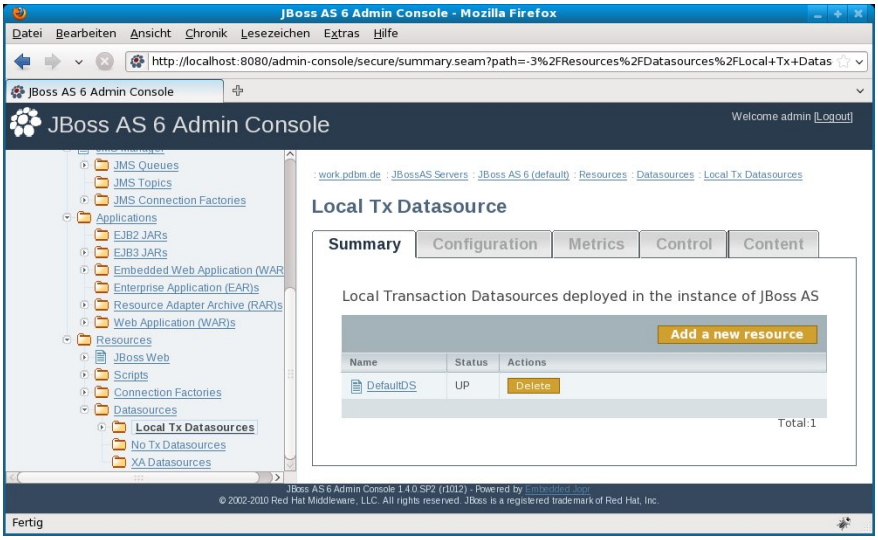


ABBILDUNG 3.6 JBoss-AS Datasource

Um das Beispielprojekt auf JBoss-AS statt auf GlassFish zu deployen, muss diese JTA-Data-Source verwendet werden. Die beiden folgenden Zeilen zeigen die jeweils alternativ zu verwendende Definition der JTA-Data-Source aus der `persistence.xml`.

```
<jta-data-source>jdbc/__default</jta-data-source> <!-- GlassFish -->
<jta-data-source>java:/DefaultDS</jta-data-source> <!-- JBoss-AS -->
```

3.3.2 Speichern und Laden von Entities

Im Abschnitt 3.2 wurden die JPA-Methoden bezüglich der CRUD-Funktionalität und deren Verwendung in Java-SE in separaten Unterabschnitten eingeführt. Da die Verwendung von JPA in einer verwalteten Java-EE-Umgebung mit einer deutlichen Code-Reduktion einhergeht, fassen wir mehrere Methoden in ihrer Darstellung zusammen.

Ein Grund für die Code-Reduktion ist der Verzicht auf eine explizite Transaktionsverwaltung. Weil in Unternehmensanwendungen in der Regel alle Verarbeitungsschritte transaktionssi-

cher erfolgen müssen, ist die transaktionale Verwendung von Methoden der Standard und muss nicht explizit programmiert werden. Der Schlüssel hierzu sind Session-Beans, die in der Enterprise-JavaBeans-Spezifikation (EJB) definiert werden. Methoden von Session-Beans werden im Default-Fall transaktional ausgeführt: Befindet sich der Thread der aktuellen Programmausführung in einer Transaktion, wird für die Ausführung der Methode diese Transaktion verwendet. Existiert keine aktive Transaktion, wird eine neue begonnen.

Die mit Java-EE 5 eingeführten EJBs der Version 3.0 [URL-EJB3] vereinfachen die Entwicklung von Unternehmensanwendungen im Gegensatz zu früheren Versionen ganz erheblich. EJBs der Version 3.1 [URL-EJB31], die mit Java-EE 6 definiert wurden, führen diese Vereinfachung weiter fort, z. B. mit einer Singleton-Session-Bean. Wir können hier nicht auf EJBs im Allgemeinen eingehen und konzentrieren uns auf die JPA-relevanten Aspekte. Diese gelten sowohl für EJBs der Version 3.0 als auch der Version 3.1.

In den bisherigen Beispielen haben wir jeweils Code-Ausschnitte mit den beteiligten JPA-Methoden angegeben. Für die Beispiele zu Java-EE gehen wir zu einer methodenbasierten Darstellung über, da diese Methoden wie oben geschildert den transaktionalen Rahmen von EJBs realisieren. Die in Listing 3.4 dargestellte Klasse `KundeService` realisiert eine Stateless Session-Bean (SLSB). Eine Stateless Session-Bean ist in EJB 3.1 eine einfache Java-Klasse (POJO) mit der Annotation `@Stateless` . In EJB 3.0 muss zusätzlich ein Interface für die öffentlichen Methoden verwendet werden. Aus Gründen der Einfachheit verwenden wir im Folgenden die 3.1-Version.

LISTING 3.4 Session-Bean `KundeService` mit `suchen()` und `speichern()`

```
@Stateless
public class KundeService {

  @PersistenceContext
  EntityManager em;

  public Kunde suchen(Integer id) {
    return em.find(Kunde.class, id);
  }

  public void speichern(Kunde kunde) {
    em.persist(kunde);
  }
  ...
}
```

Der Entity-Manager wird über die Annotation `@PersistenceContext` *injiziert*. Dies bedeutet für den Anwendungsentwickler, dass er zum Zeitpunkt eines Methodenaufrufs der Session-Bean davon ausgehen kann, dass der Application-Server eine Entity-Manager-Instanz in die so annotierte Variable geschrieben hat. Die Methoden `suchen()` und `speichern()` vereinfachen sich durch die implizite Transaktionalität ganz erheblich.

Wie wird jedoch die injizierte Entity-Manager-Instanz erzeugt? Der Application-Server erzeugt für alle Persistenzeinheiten der `persistence.xml` jeweils eine `EntityManagerFactory`-Instanz. Aus diesen werden, wie in Abschnitt 3.2 ausgeführt, `EntityManager`-Instanzen erzeugt, jedoch automatisch vom Container, nicht vom Anwendungsentwickler. Wenn nur eine Persistenzeinheit in der `persistence.xml` existiert, ist die Verwendung von

`@PersistenceContext` eindeutig. Optional kann der Name der Persistenzeinheit angegeben werden, um diese Eindeutigkeit zu garantieren:

```
@PersistenceContext(unitName = "bank")
EntityManager em;
```

Die weiteren Attribute der Annotation `@PersistenceContext` sind in Tabelle 3.2 zusammengefasst.

TABELLE 3.2 `@PersistenceContext`-Attribute

`@PersistenceContext(...)`			
Attribut	**Typ**	**Default**	**Beschreibung**
`name`	`String`	–	Name des `EntityManager` im Kontext (JNDI). Das Attribut wird nicht benötigt, wenn der Entity-Manager injiziert wird.
`properties`	`PersistenceProperty[]`	–	Implementierungsspezifische Einstellungen, die dem Container und JPA-Provider übergeben werden können.
`type`	`PersistenceContextType`	`TRANSACTION`	Flag, um zwischen einem transaktionsbezogenen und einem erweiterten Persistenzkontext zu wählen.
`unitName`	`String`	–	Name der Persistenzeinheit, definiert in der Datei `persistence.xml`

Soll direkt eine Entity-Manager-Fabrik verwendet werden, so ist die `@PersistenceUnit`-Annotation zu verwenden. Dies ist in der Regel bei Session-Beans nicht sinnvoll, kann aber z. B. bei Servlets verwendet werden, wie im folgenden Code-Ausschnitt angedeutet.

```
@WebServlet(urlPatterns = "/special")
public class SpecialServlet extends HttpServlet {

  @PersistenceUnit
  EntityManagerFactory emf;

  @Override
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response) throws IOException
                        {
    Integer id = new Integer(request.getParameter("id"));
    EntityManager em = emf.createEntityManager();
    em.getTransaction().begin();
    Kunde kunde = em.find(Kunde.class, id);
    kunde.setNachname("...");
    em.getTransaction().commit();
  }
}
```

Beim Beispiel ist zu beachten, dass Entity-Transaktionen nur mit `RESOURCE_LOCAL` und nicht mit `JTA` verwendet werden können, so dass die `persistence.xml` entsprechend definiert sein muss. Die Annotation `@PersistenceUnit` ist in Tabelle 3.3 dargestellt.

Die dargestellten Zusammenhänge werden in Abbildung 3.3 auf Seite 68 nochmals deutlich.

TABELLE 3.3 `@PersistenceUnit`-Attribute

`@PersistenceUnit(...)`			
Attribut	**Typ**	**Default**	**Beschreibung**
`name`	`String`	–	Name der `EntityManagerFactory` im Kontext (JNDI). Das Attribut wird nicht benötigt, wenn die Entity-Manager-Fabrik injiziert wird.
`unitName`	`String`	–	Name der Persistenzeinheit, definiert in der Datei `persistence.xml`

3.3.3 Aktualisieren und Löschen von Entities

Bei der Verwendung eines durch den Container verwalteten Entity-Managers bzw. Persistenzkontextes muss zwischen einem transaktionalen (transaction-scoped) und einem erweiterten (extended) Persistenzkontext unterschieden werden. Die Unterscheidung erfolgt als `type`-Attribut der `@PersistenceContext`-Annotation über die Enumeration `PersistenceContextType` mit den beiden Werten `TRANSACTION` und `EXTENDED`. Default ist `TRANSACTION`. Wir verwenden zunächst einen transaktionalen Persistenzkontext und gehen dann in Abschnitt 3.3.4 auf den erweiterten Persistenzkontext ein.

Der durch die `@PersistenceContext`-Annotation im Default injizierte Entity-Manager verwaltet, wie bereits erwähnt, einen transaktionalen Persistenzkontext. Bei einem transaktionalen Persistenzkontext werden am Ende der Transaktion alle verwalteten Entities vom Kontext gelöst (detached) und sind damit aus Sicht des Entity-Managers gewöhnliche, d. h. nicht persistente POJOs. In geschichteten Java-EE-Anwendungen ist eine Anfrage durch den Benutzer typischerweise mit HTTP und JSF realisiert. Am Ende der Verwendung eines Entity in einer EJB wird die Transaktion committed, das Entity losgelöst und über JSF dem Benutzer angezeigt. Soll beim nächsten JSF-Request dieses Entity gelöscht oder aktualisiert werden, so muss es zunächst wieder in die Obhut eines Entity-Managers überführt werden, und zwar mit Hilfe der `merge()`-Methode. Als Beispiel gibt der folgende Code die entsprechenden Methoden der Session-Bean `KundeService` aus Listing 3.4 auf Seite 72 wieder. Um die Art des Persistenzkontextes explizit zu machen, wurde das `type`-Attribut der Annotation `@PersistenceContext` mit dem Default-Wert verwendet (im Unterschied zu Listing 3.4).

LISTING 3.5 Session-Bean `KundeService` mit `loeschen()` und `aendern()`

```
@Stateless
public class KundeService {

  @PersistenceContext(type = PersistenceContextType.TRANSACTION)
  EntityManager em;
  ...
  public void loeschen(Kunde kunde) {
    em.remove(em.merge(kunde));
  }
  public void aendern(Kunde kunde) {
    em.merge(kunde);
  }
  ...
}
```

In der Methode `loeschen()` wird das Entity durch den Aufruf von `merge()` in den Persistenzkontext integriert. Das von `merge()` zurückgegebene Entity kann dann einfach gelöscht werden, analog zur Verwendung in Java-SE (Abschnitt 3.2.4). Nach Aufruf von `merge()` innerhalb der `aendern()`-Methode ist das Entity wieder im Persistenzkontext enthalten, so dass das Konzept des *Automatic Dirty Checking* greift und das Entity am Ende der Transaktion mit der Datenbank synchronisiert wird.

3.3.4 Der erweiterte Persistenzkontext

Ein Entity-Manager verwaltet die Entities eines Persistenzkontextes. Er ist damit zunächst einmal konzeptionell unabhängig von Datenbankverbindungen und Transaktionen. Der transaktionale Persistenzkontext gibt diese Unabhängigkeit auf, der erweiterte Persistenzkontext realisiert diese Unabhängigkeit und ist als eine der großen Errungenschaften der Java-EE-5-Spezifikation zu bewerten. Der erweiterte Persistenzkontext wurde als Mittel zur Realisierung beliebig langer Anwendungsfälle konstruiert, die mehrere Transaktionen andauern können und eventuell neue oder wiederverwendete (Stichwort Connection-Pool) Datenbankverbindungen benötigen. Für den Anwendungsentwickler ist dies jedoch nicht sichtbar und ermöglicht so eine sehr einfache Realisierung dieser Anwendungsfälle.

Da der Persistenzkontext über mehrere Benutzerinteraktionen hinweg existieren soll, wird er einer zustandsbehafteten Session-Bean und deren Lebensdauer zugeordnet, d.h., er wird bei der Erzeugung der Stateful Session-Bean (SFSB) (Annotation `@Stateful`) angelegt und beim Entfernen dieser Session-Bean wieder zerstört. Listing 3.6 zeigt die überarbeitete Version der Klasse `KundeService` mit der Annotation `@Stateful` und dem Persistenzkontexttyp `EXTENDED`.

LISTING 3.6 Session-Bean `KundeService` mit erweitertem Persistenzkontext

```
@Stateful
public class KundeService {

  @PersistenceContext(type = PersistenceContextType.EXTENDED)
  EntityManager em;
  ...
  public void loeschen(Kunde kunde) {
    em.remove(kunde);
  }

  public void aendern(Kunde kunde) {
  }
  ...
}
```

Weil das Speichern und Laden von Entities ohne Änderungen ganz analog zum transaktionalen Persistenzkontext erfolgt, sind in Listing 3.6 nur die Methoden `loeschen()` und `aendern()` angegeben. In beiden Methoden wird auf den Aufruf von `merge()` des Listings 3.5 auf der vorherigen Seite verzichtet, was im Falle von `aendern()` sogar zu einem leeren Methodenrumpf führt. Weil der EJB-Container am Ende der Methode `aendern()`

die Transaktion committed, werden alle als *dirty* gekennzeichneten Entities mit der Datenbank synchronisiert, im Beispiel das geänderte Kunden-Objekt. Wir erhalten also genau die gewünschte Semantik.

Die Verwendung des erweiterten Persistenzkontextes ist sehr attraktiv, da das Programmiermodell intuitiv und einfach ist. Der einzige Nachteil ist, dass der Persistenzkontext sehr groß werden *kann*. Mit den in Kapitel 7 noch einzuführenden JPA-Queries lassen sich viele Entities in den Persistenzkontext laden. Führt dies zu Speicher- und damit Performanz-Problemen, können nicht mehr benötigte Entities mit der Methode `detach()` vom Persistenzkontext gelöst und damit vom Garbage-Collector entsorgt werden. Für diesen Fall ist die Funktionsweise des Second-Level-Cache zu beachten, auf den wir später eingehen.

Tipp:

Als weitere Optimierungsmöglichkeit kann als Scope der SFSB nicht die Session, sondern eine Benutzerkonversation verwendet werden. Die Annotation `@ConversationScoped` der CDI-Spezifikation [URL-CDI] erlaubt die Definition einer Benutzertransaktion mit programmatischem Beginn und Ende. CDI ist in Java-EE 6 enthalten und somit in allen entsprechenden Application-Servern vorhanden. Eine detaillierte Darstellung der Verwendung des Scopes soll hier nicht Gegenstand sein und wird in Abschnitt 6.4 nachgeholt.

3.3.5 Analyse und Zusammenfassung

Um dem Leser eine bessere Grundlage für Entwurfsentscheidungen zur Verfügung zu stellen, sollen hier eine kurze Analyse und Zusammenfassung der bisher vorgestellten Möglichkeiten der verschiedenen Persistenzkontexte erfolgen. Die Persistenzeinheit definiert eine mit einem Namen versehene Menge von persistenten Entity-Klassen. Ein Persistenzkontext stellt eine Menge von verwalteten Entity-Instanzen dar. In Java-EE werden zwei Arten von Persistenzkontexten unterschieden:

- Transaktionaler Persistenzkontext (`PersistenceContextType.TRANSACTION`)
- Erweiterter Persistenzkontext (`PersistenceContextType.EXTENDED`)

Ein Persistenzkontext wird durch einen oder mehrere Entity-Manager verwaltet. Auch hier unterscheidet man zwei Arten:

- Anwendungsverwalteter Entity-Manager in SE/EE, (Interfaces `EntityManager` und `EntityManagerFactory`)
- Container-verwalteter Entity-Manager in EE (Annotationen `@PersistenceUnit` und `@PersistenceContext`)

Der in Listing 3.4 auf Seite 72 verwendete container-verwaltete Entity-Manager mit transaktionalem Persistenzkontext ist an die JTA-Transaktion des Containers gebunden. Beim Aufruf einer Methode des Entity-Managers (durch die Injektion des Entity-Managers ist dies ein Container-Proxy) prüft der Container, ob der aktuellen Transaktion ein Persistenzkontext zugeordnet ist. Wenn ja, verwendet der Entity-Manager diesen Persistenzkontext. Wenn nein, wird ein neuer Persistenzkontext erzeugt und der Transaktion zugeordnet. Am Ende der

Transaktion wird der Persistenzkontext bereinigt und entfernt. Dies verdeutlicht noch einmal die Notwendigkeit der Verwendung der `merge()`-Methode in Listing 3.5.

Wird ein erweiterter Persistenzkontext wie in Listing 3.6 auf Seite 75 verwendet, ist die Lebensdauer des Persistenzkontextes an die Lebensdauer der Stateful Session-Bean gebunden. Wird die Session-Bean erzeugt, wird auch der Persistenzkontext erzeugt. Der Persistenzkontext existiert so lange, wie die Session-Bean. Entities werden nie losgelöst (außer explizit durch das Anwendungsprogramm), so dass sich lange Benutzertransaktionen erstaunlich einfach realisieren lassen.

Tipp:

Um die beschriebenen Aussagen zu verifizieren, raten wir dem Leser, in die Methoden der Klasse `KundeService` eine Logging-Ausgabe des Ausdrucks `em.getDelegate()` einzufügen. Dies liefert die Implementierung des Entity-Managers zurück. Für einen transaktionalen Persistenzkontext ist dies bei jedem Methodenaufruf eine andere Instanz, für den erweiterten Persistenzkontext immer dieselbe.

Am Ende dieses Abschnitts ist noch eine Bemerkung zum Verhältnis zwischen Application-Server und JPA-Provider angebracht: JPA 2.0 ist unverzichtbarer Bestandteil von Java-EE 6, auch des Web-Profils. Damit ist in jedem Application-Server ein entsprechender JPA-Provider enthalten. Wie bereits erwähnt, ist EclipseLink der JPA-Provider von GlassFish und Hibernate der Provider von JBoss-AS. Der Application-Server WebLogic, der wie GlassFish ebenfalls von Oracle hergestellt und vertrieben wird, verwendet in den neueren Versionen ebenfalls EclipseLink. Frühere Versionen verwendeten Kodo, ein Provider, der intern OpenJPA verwendet. Die Application-Server WebSphere von IBM und Geronimo von Apache verwenden ebenfalls OpenJPA. Prinzipiell ist der Austausch des JPA-Providers im Application-Server möglich, und es existieren im Internet entsprechende Anleitungen. Wir raten hiervon jedoch ab.

Projekt:

Die im Abschnitt 3.3 beschriebenen Code-Beispiele zum Thema JPA in Java-EE-Anwendungen sind im Projekt *objektlebenszyklus-ee* enthalten.

3.4 Callback-Methoden und Entity-Listener

Um individuell auf Veränderungen im Lebenslauf eines Entity reagieren zu können, stellt JPA Callback-Methoden und Entity-Listener zur Verfügung. Callback-Methoden sind spezielle Methoden, die von JPA bei Zustandsänderungen eines Entity aufgerufen werden. Als Zustandsänderungen sind hier Ereignisse bezüglich der Datenbank und nicht bezüglich des Objektlebenszyklus (Abbildung 3.1 auf Seite 57) zu verstehen. Callback-Methoden können direkt in Entity-Klassen oder in anderen Klassen definiert werden, die man dann *Entity-Listener* nennt.

Ereignisse in diesem Sinne korrespondieren mit entsprechenden Datenbankoperationen: Einfügen, Löschen, Aktualisieren und Lesen. Mit Ausnahme des Lesens wird jeweils zwi-

schen einem Pre- und einem Post-Ereignis unterschieden, so dass sich sieben Ereignisarten ergeben. Dies führt zu sieben Annotationen, die zur Definition von Callback-Methoden verwendet werden:

- `@PrePersist`
- `@PostPersist`
- `@PreRemove`
- `@PostRemove`
- `@PreUpdate`
- `@PostUpdate`
- `@PostLoad`

Die genannten Annotationen besitzen keine Attribute. Wir verzichten daher auf die Darstellung in Tabellenform.

3.4.1 Callback-Methoden

Als erstes Beispiel soll die Vergabe eines Primärschlüssels beim Einfügen eines Entity in die Datenbank protokolliert werden. Wie in Kapitel 2 beschrieben, erfolgt bei der Generierungsart `IDENTITY` für Primärschlüssel-Properties die Vergabe des Wertes durch die Datenbank. Vor dem Aufruf der `persist()`-Methode ist das Property also `null`, nach der Ausführung der Methode hat es den durch die Datenbank vergebenen Wert. Der folgende Ausschnitt des Entity `Kunde` zeigt die beiden Callback-Methoden mit ihren entsprechenden Annotationen.

```
@Entity
public class Kunde {

  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  private Integer id;

  ...

  @PrePersist
  private void vorEinfuegen() {
    System.out.println("vorEinfuegen(): id = " + id);
  }

  @PostPersist
  private void nachEinfuegen() {
    System.out.println("nachEinfuegen(): id = " + id);
  }

}
```

Die mit `@PrePersist` annotierte Methode wird bei einem erfolgreichen Aufruf von `persist()` aufgerufen. Falls ein neues Entity durch einen `merge()`-Aufruf in einen Persistenzkontext eingefügt wird, erfolgt ebenfalls der Aufruf der Methode, da auch hier ein `INSERT` in der Datenbank erfolgt. Nach dem `INSERT` erfolgt der Aufruf der mit `@PostPersist` annotierten Methode. Vergleicht man die jeweiligen Ausgaben, so ist die Id bei der `@PrePersist`-

Methode `null`, während sie bei der `@PostPersist`-Methode den soeben von der Datenbank vergebenen Primärschlüssel enthält.

Als weiteres Beispiel soll das Laden eines Entity ebenfalls geloggt werden:

```
@PostLoad
private void nachLaden() {
  System.out.println("nachLaden(): id = " + id);
}
```

Diese Callback-Methode wird nach der Ausführung von `em.find()` aufgerufen. Sie wird aber auch für alle Entities aufgerufen, die durch eine Query in den Persistenzkontext geladen werden. Queries führen wir in Kapitel 7 ein.

3.4.2 Entity-Listener

Callback-Methoden realisieren das Event-Handling für Zustandsänderungen innerhalb der Entity-Klassen. Ist dies nicht gewünscht, können Entity-Listener verwendet werden. Entity-Listener sind einfache Klassen mit Callback-Methoden, die über die Annotation `@EntityListeners` mit der Entity-Klasse assoziiert werden. Das folgende Beispiel zeigt einen Entity-Listener mit derselben Funktionalität bezüglich des Primärschlüssel-Loggings des letzten Beispiels. Da sich die Methoden nicht in der Entity-Klasse befinden, ist das Entity als Parameter zu übergeben.

```
public class KundeListener {

  @PrePersist
  public void vorEinfuegen(Kunde kunde) {
    System.out.println("vorEinfuegen(): id = " + kunde.getId());
  }

  @PostPersist
  public void nachEinfuegen(Kunde kunde) {
    System.out.println("nachEinfuegen(): id = " + kunde.getId());
  }

}
```

Im zu überwachenden Entity muss der Entity-Listener über die schon erwähnte Annotation `@EntityListeners` bekannt gemacht werden:

```
@Entity
@EntityListeners(KundeListener.class)
public class Kunde {
  ...
```

Wie die Pluralform `@EntityListeners` andeutet, kann ein Array von Listenern angegeben werden:

```
@Entity
@EntityListeners({KundeListener.class, NochEinListener.class})
public class Kunde {   ...
```

Weil das `value`-Attribut das einzige Attribut ist, verzichten wir auf die Darstellung in Tabellenform.

JPA stellt einige grundsätzliche Anforderungen an Entity-Listener. Diese müssen einen öffentlichen Default-Konstruktor besitzen und zustandslos sein. Diese Zustandslosigkeit impliziert, dass Entity-Listener keine Instanzvariablen haben dürfen. Der Grund hierfür ist die Freiheit des JPA-Providers, Entity-Listener-Instanzen für mehrere Entity-Instanzen zu verwenden, und die Listener-Methoden in mehreren Threads aufzurufen. Außerdem sind die Verwendung von `EntityManager`- oder `Query`-Methoden, der Zugriff auf andere Entities und die Modifikation von Beziehungen zwischen Entities nicht erlaubt.

Zum Abschluss des Abschnitts sollen noch die syntaktischen Regeln für Callback-Methoden und ihre Aufrufreihenfolge genauer untersucht werden. Callback-Methoden in Entities haben `void` als Rückgabetyp und keine Parameter. Callback-Methoden in Entity-Listenern haben ebenfalls `void` als Rückgabetyp und einen Parameter vom Typ `Object` oder vom Entity-Typ, wie in unserem Beispiel. Die Sichtbarkeit der Callback-Methoden ist beliebig (`public`, `private`, `protected` oder Package-Ebene), sie dürfen jedoch nicht `static` oder `final` sein.

Falls mehrere Callback-Methoden für dasselbe Ereignis festgelegt sind, beschreibt die Spezifikation eine exakte Reihenfolge für deren Ausführung. Diese Beschreibung ist jedoch relativ umfangreich, so dass wir hier auf die exakte Darstellung verzichten und eine vereinfachte Fassung vorstellen. Sollten sich wider Erwarten Probleme ergeben, bitten wir den Leser, direkt in der Spezifikation nachzuschlagen.

Die Callback-Aufrufe erfolgen nach folgendem Verfahren:

- Callback-Methoden der Default-Listener (Abschnitt 3.4.3) in der Reihenfolge in der Konfigurationsdatei (`orm.xml`)
- Callback-Methoden der Entity-Listener in der Reihenfolge ihrer Anordnung in der `@EntityListeners`-Annotation. Befindet sich die Entity-Klasse in einer Vererbungshierarchie (Kapitel 5), werden geerbte Callbacks vor den eigenen ausgeführt. Die Reihenfolge der geerben Callbacks geht von Ober- zu Unterklasse. Die Annotation `@ExcludeSuperclassListeners` unterbindet die Aufrufe geerbter Callback-Methoden.
- Callback-Methoden der Entity-Klasse. Auch hier werden geerbte Callbacks vor den eigenen ausgeführt. Die Reihenfolge der geerben Callbacks geht von Ober- zu Unterklasse.

3.4.3 Default-Listener

Soll ein Entity-Listener für *alle* Entities registriert werden, so müssten alle Entity-Klassen entsprechend mit der Annotation `@EntityListeners` versehen werden. Dieser in der Regel nicht praktikable Aufwand kann durch einen *Default-Listener* reduziert werden. Default-Listener werden jedoch nicht im Java-Code, sondern in der XML-Konfiguration der Persistenzeinheit definiert. Die Datei `orm.xml` im Verzeichnis `META-INF` definiert Mapping-Optionen und Defaults. Die Mapping-Optionen sind die XML-Entsprechungen für Annotationen. Wir gehen darauf kurz in Abschnitt 3.7 und ausführlich in Kapitel 10 ein. Als Defaults werden in der XML-Konfiguration globale Einstellungen, wie z. B. Listener für alle Entity-Klassen definiert. Der folgende Ausschnitt aus der `orm.xml` zeigt die Deklaration eines Default-Listeners (Klasse `de.jpainfo.model.EntityListener`) mit dem Tag `<entity-listener>`. Die Callback-Methoden werden mit XML-Tags beschrieben, die durch konsistente Umbenennung der sieben Ereignis-Annotationen bestehen: `@`-Zeichen

entfernen, interne Camel-Case-Großschreibung durch „-“-Zeichen ersetzen, Kleinschreibung.

```
...
<persistence-unit-defaults>
  <entity-listeners>
    <entity-listener class="de.jpainfo.model.EntityListener">
      <pre-persist method-name="vorEinfuegen" />
      <post-persist method-name="nachEinfuegen" />
    </entity-listener>
  </entity-listeners>
</persistence-unit-defaults>
...
```

Der verwendete Default-Listener `EntityListener` besitzt die entsprechenden Methoden:

```
public class EntityListener {

  public void vorEinfuegen(Object obj) {
    System.out.println("EntityListener.vorEinfuegen() Id = " + getId(
        obj));
  }

  public void nachEinfuegen(Object obj) {
    System.out.println("EntityListener.nachEinfuegen() Id = " + getId
        (obj));
  }

  ...
}
```

Da die Methoden für beliebige Entities aufgerufen werden, muss der Parametertyp `Object` sein. Wir haben in unseren bisherigen Beispielen das Primärschlüssel-Property immer `id` genannt. Dies ist jedoch eine von uns gewählte Konvention und darf für einen allgemeinen Entity-Listener nicht vorausgesetzt werden. Es muss daher das Property gesucht werden, das mit `@Id` annotiert ist. Die Methode `getId()` leistet genau dies, wobei wir in der Darstellung auf allgemeine Fehlerfälle und Exception-Handling der Übersichtlichkeit wegen verzichten.

```
private Object getId(Object obj) {
  Field[] fields = obj.getClass().getDeclaredFields();
  for (Field field : fields) {
    Annotation[] annotations = field.getAnnotations();
    for (Annotation annotation : annotations) {
      if (annotation.annotationType()
                    .equals(javax.persistence.Id.class)) {
        field.setAccessible(true);
        return field.get(obj);
      }
    }
  }
  return null;
}
```

Hinweis:

Hier muss angemerkt werden, dass die Methode `getId()` bei allen Speicher-Operationen aufgerufen wird und Reflection sowie verschachtelte Schleifen verwendet. In der Praxis ist daher in der Regel auf derart aufwendige Berechnungen in Callback-Methoden aus Performanzgründen zu verzichten.

Wir hatten Default-Listener mit der Notwendigkeit eines Listeners für alle Entities motiviert. Sollen jedoch nur *fast* alle Entities mit einem Listener versehen werden, müssen doch wieder alle diese Entity-Klassen mit `@EntityListeners` annotiert werden. Abhilfe schafft hier die Annotation `@ExcludeDefaultListeners`. Mit ihr werden Entity-Klassen annotiert, für die dann Default-Listener von der Bearbeitung ausgeschlossen werden.

3.4.4 Callbacks und Listener zur Validierung

Unsere bisherigen Beispiele für Callback-Methoden und Entity-Listener haben Logging-Aufgaben übernommen. Ein weiteres wichtiges Einsatzgebiet für Callback-Methoden und Entity-Listener ist die Validierung.

Normalerweise muss ein Kunde einen Vor- und einen Nachnamen besitzen. Zusätzlich sind diese in der Regel in ihrer Länge beschränkt. Dies könnte mit der `@Column`-Annotation etwa wie folgt definiert werden.

```
@Entity
public class Kunde {

  @Id @GeneratedValue
  private Integer id;
  @Column(nullable = false, length = 30)
  private String vorname;
  @Column(nullable = false, length = 30)
  private String nachname;
  ...
```

Zu beachten ist hier, dass in der Datenbank die entsprechenden Constraints für die beiden Spalten existieren müssen. JPA führt auf Basis der reinen Existenz der `@Column`-Annotation keine Validierung durch.

Wenn die entsprechenden Constraints in der Datenbank vorliegen, führen diese beim Insert eine entsprechende Validierung durch und werfen im Fehlerfall eine Exception. Dies führt bei Hunderten oder Tausenden von Benutzern und vielen derartigen Constraints zu einer unnötigen Belastung der Datenbank. Sinnvoller ist es, bereits in der Anwendung zu validieren. Die Klasse `KundeValidierer` ist ein Entity-Listener, wie in Abschnitt 3.4.2 beschrieben. Er ist als solcher beim Entity `Kunde` zu deklarieren.

```
public class KundeValidierer {

  @PrePersist
  private void validiere(Kunde kunde) {
    if (kunde.getVorname() == null) {
      throw new IllegalArgumentException(
```

```
                "Vorname darf nicht 'null' sein");
        }
        if (kunde.getVorname().length() < 3
            || kunde.getVorname().length() > 30) {
          throw new IllegalArgumentException(
                "Vorname muss zwischen 3 und 30 Zeichen haben");
        }
      }

    }
```

Im Gegensatz zur `@Column`-Annotation bzw. der Längenbeschränkung auf Datenbankebene prüft der Entity-Listener auch die untere Länge des Vornamens. Wir werfen eine – in ihrer Verwendung nicht ganz korrekte – Standard-Exception. Eine anwendungsdefinierte Exception wäre eventuell sinnvoller.

Tipp:

In der Regel raten wir von der Verwendung von Callbacks oder Entity-Listener zur Validierung in JPA 2.0 ab. In JPA 1.0 ist eine derartige Verwendung in Ordnung. Mit JPA 2.0 wurde der Standard *Bean Validation* eingeführt, der eine Validierung durch einfache Annotationen ermöglicht. Wir stellen dies in Kapitel 9 vor. Die Integration von Bean Validation in JPA 1.0 ist möglich, wir gehen darauf jedoch nicht ein.

Projekt:

Die im Abschnitt 3.4 verwendeten Beispiele zu Callback-Methoden und Entity-Listener sind im Projekt *objektlebenszyklus-listener* enthalten.

3.5 Das neue Zustands-API

Mit JPA 2.0 wurden zwei neue Interfaces als Schnittstelle zwischen Anwendung und JPA-Provider eingeführt: `PersistenceUtil` und `PersistenceUnitUtil`, wobei das zweitgenannte das erstgenannte erweitert. Beide Interfaces sind noch sehr rudimentär, sollen aber in zukünftigen Releases erweitert werden. Instanzen der Interfaces erhält man über die Static-Methode `getPersistenceUtil()` der Klasse `Persistence` bzw. über `getPersistenceUnitUtil()` der `EntityManagerFactory`:

```
PersistenceUtil persistenceUtil = Persistence.getPersistenceUtil();
EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("...");
PersistenceUnitUtil persistenceUnitUtil =
        emf.getPersistenceUnitUtil();
```

Die Methoden des Interface `PersistenceUnitUtil` sind:

```
Object getIdentifier(Object entity)
boolean isLoaded(Object entity)
boolean isLoaded(Object entity, String attributeName)
```

Die Methoden geben den Wert des Primärschlüssel-Property sowie den Status des Ladezustands eines Entity bzw. eines Entity-Property zurück. Der Ladezustand wird dabei bezüglich der Entity-Properties mit `FetchType.EAGER` definiert. Diese Informationen sind in der Regel weniger für Anwendungen als für Frameworks interessant. Da wir jedoch im letzten Abschnitt eine Methode `getId()` selbst realisiert haben, soll hier ein kleines Beispiel entwickelt werden.

Die folgenden Code-Zeilen erzeugen auf unterschiedliche Art drei Kundenobjekte.

```
Kunde kunde1 = em.find(Kunde.class, 1);
Kunde kunde2 = em.getReference(Kunde.class, 2);
Kunde kunde3 = new Kunde("Max", "Mustermann", "12.3.1970");
```

Für jedes Kundenobjekt werden die beiden folgenden Methoden aufgerufen.

```
persistenceUnitUtil.isLoaded(kunde<i>));
persistenceUnitUtil.getIdentifier(kunde<i>));
```

Das Ergebnis der Methodenaufrufe für die drei Provider fasst Tabelle 3.4 zusammen.

TABELLE 3.4 Ergebnisse der Methodenaufrufe

Methode	EclipseLink	Hibernate	OpenJPA
isLoaded(kunde1)	true	true	true
getIdentifier(kunde1)	1	1	1
isLoaded(kunde2)	true	false	false
getIdentifier(kunde2)	2	null	2
isLoaded(kunde3)	true	true	false
getIdentifier(kunde3)	null	null	null

Dieses Beispiel macht noch einmal deutlich, dass die Spezifikation den JPA-Providern in einigen Dingen Realisierungsalternativen offen lässt. Im Fall von `kunde2` kann `getReference()` früh oder beim ersten Zugriff laden. Die verschiedenen Ergebnisse sind damit in Ordnung. Im Fall von `kunde3` erscheint uns die Spezifikation nicht eindeutig genug, was aber im Zweifel zu diskutieren wäre. Im Rahmen unseres Buches liegt uns lediglich daran, auf derartige Unterschiede hinzuweisen.

Projekt:

Die in diesem Abschnitt verwendeten Beispiele zum Interface `PersistenceUnitUtil` sind im Projekt *objektlebenszyklus-listener* enthalten.

3.6 Das Packaging von Anwendungen

Wie wir in Abschnitt 3.3 gesehen haben, ist die Integration von JPA in Java-EE-Anwendungen als sehr gelungen zu bezeichnen, was sich z. B. an der Injektion eines Entity-Managers oder dem automatischen Transaktionsmanagement belegen lässt. Als Standardplattform für Unternehmensanwendungen mit eventuell Tausenden von Klassen kommt dem Packaging von Entity-Klassen und weiteren Anwendungsartefakten in Java-EE-Anwendungen große Bedeutung zu. Die JPA-Spezifikation beschreibt daher auch dieses Packaging.

Im Zeitalter moderner IDEs gehen wir zunächst auf die Unterstützung durch diese IDEs ein, um danach den allgemeinen Fall darzustellen. Wir beschränken uns bei der IDE auf Eclipse, merken aber an, dass auch NetBeans, IntelliJ und – falls Sie eine andere IDE einsetzen – wahrscheinlich auch Ihre IDE eine derartige Unterstützung bieten.

3.6.1 Packaging von Web-Anwendungen

Java-EE 6 erlaubt das Packaging von EJBs, zu denen auch Entities gehören, in Web-Archiven (WAR). Das Format eines solchen WARs ist in der Servlet-Spezifikation [URL-Serv] beschrieben. Ein WAR entspricht dem Format eines Java-Archivs (JAR) und ist somit eine ZIP-Datei. Es enthält mit `WEB-INF` ein spezielles Verzeichnis, das den Deployment-Deskriptor `web.xml` der Web-Anwendung, aber auch die Unterverzeichnisse `classes` und `lib` enthält. Das `classes`-Verzeichnis enthält die Klassen der Anwendung, das `lib`-Verzeichnis die Bibliotheken in Form von JAR-Dateien.

Tipp:

Sollten Sie gerade Ihre ersten Schritte mit Java-EE tun, raten wir Ihnen dringend dazu, Ihre Anwendung als Web-Archiv in Form eines WARs zu deployen, da das Packaging eines WARs wesentlich einfacher als das Packaging eines Enterprise-Archives (EAR) ist.

Die bereits angesprochene IDE-Unterstützung erlaubt das Anlegen von Projekten für Java-Web-Anwendungen, die der Servlet-Spezifikation entsprechende WAR-Dateien erzeugen. Bei Eclipse ist als entsprechende Projektart „*Dynamic Web Project*“ auszuwählen. Während der Entwicklung kann alternativ zum Erzeugen der WAR-Datei in einer Verzeichnisstruktur deployt werden, die dem WAR-Format entspricht. Die Organisation der Ausgangsdateien ist dabei beliebig und eventuell von IDE zu IDE verschieden. Abbildung 3.7 zeigt in der linken Grafik die Darstellung des Projekts *objektlebenszyklus-ee* aus Abschnitt 3.3 im Package-Explorer von Eclipse. Wird das Projekt von Eclipse hot deployt oder als WAR exportiert, so wird die im rechten Teil der Abbildung dargestellte Verzeichnis- und Dateistruktur erzeugt, die den Anforderungen der Servlet-Spezifikation entspricht.

3.6.2 Allgemeines Packaging

Nach dieser Einführung in das Packaging eines einfachen WARs wollen wir uns die Regeln für das Packaging von JPA-Anwendungen bzw. Teilanwendungen genauer anschauen. Grundla-

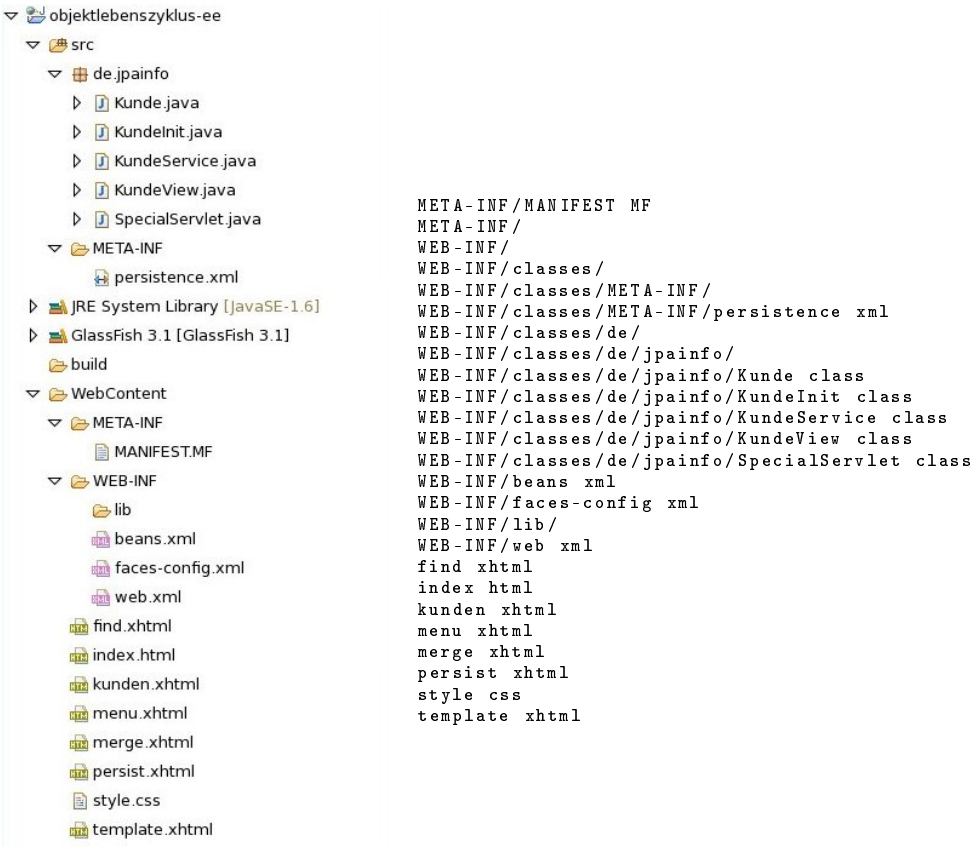


ABBILDUNG 3.7 Organisationsstruktur: Quellcode und WAR-Datei

ge ist die zu Beginn dieses Kapitels eingeführte Persistenzeinheit. Diese besteht im Einzelnen aus:

- einer Entity-Manager-Factory und ihren Entity-Managern sowie ihrer Konfigurationdaten;
- der Menge der verwalteten Klassen dieser Persistenzeinheit;
- den Mapping-Metadaten, die in Form von Annotationen oder XML die Klassen auf die Datenbank abbilden.

Die im ersten Punkt genannten Konfigurationsdaten sind in der Datei `persistence.xml` sowie in weiteren Dateien enthalten, auf die wir später noch eingehen. Das Packaging erfolgt auf der Basis einer Persistenzeinheit. Diese kann innerhalb der folgenden Alternativen eingepackt werden:

- Eine Menge von Klassen im `classes`-Verzeichnis einer WAR-Datei
- Eine oder mehrere JAR-Dateien in einer WAR- oder EAR-Datei
- Eine Menge von Klassen in einem sogenannten EJB-JAR
- Eine Kombination dieser Alternativen

Die erste Alternative haben wir im Abschnitt 3.6.1 beschrieben. Da dies die einfachste Möglichkeit ist, empfehlen wir diese Alternative für die ersten Schritte mit JPA in einer Java-EE-Umgebung.

Eine Persistenzeinheit wird über die Datei `persistence.xml` definiert. Die JAR-Datei oder das Verzeichnis, dessen `META-INF`-Verzeichnis die `persistence.xml` enthält, wird als die Wurzel (root) der Persistenzeinheit bezeichnet. In Java-EE kann die Wurzel der Persistenzeinheit eine der folgenden Alternativen sein:

- Eine EJB-JAR-Datei
- Das Verzeichnis `META-INF/classes` einer WAR-Datei
- Eine JAR-Datei im Verzeichnis `WEB-INF/lib` einer WAR-Datei
- Eine JAR-Datei in einem EAR-Bibliotheksverzeichnis
- Ein JAR-File eines Anwendungs-Clients

Wir wollen diese der Vollständigkeit halber aufgeführten Alternativen nicht alle detaillieren, sondern beschränken uns auf den häufigsten Fall. Größere Unternehmensanwendungen werden in Form von Enterprise-Archiven (EAR) zusammengefasst. Das Dateiformat entsprich einem JAR. Ein EAR enthält Module, die wiederum JAR-Dateien sind. Neben einfachen JAR-Dateien, die gewöhnliche Bibliotheken oder eine Client-Anwendung beinhalten können, sind die beiden wichtigen Modularten WAR- und EJB-JAR-Dateien. WAR-Dateien sind die schon bekannten Web-Anwendungen. EJB-JAR-Dateien sind einfache JAR-Dateien, die Enterprise-JavaBeans enthalten. Die einfachste Möglichkeit zur Erstellung der vorgeschriebenen Verzeichnisstrukturen ist die Verwendung einer IDE. In Eclipse sind Projekte der folgenden Projektarten zu erzeugen:

- Dynamic Web Project
- EJB Project
- Enterprise Application Project

Die letztgenannte Projektart erlaubt die Referenzierung von Projekten der beiden erstgenannten Arten. Beim Exportieren der Anwendung der letzten Projektart als EAR werden das WAR und das EJB-JAR der Teilprojekte automatisch erzeugt. Die Verwendung mehrerer Web- oder EJB-Projekte in einem Enterprise-Projekt ist möglich.

3.7 Mapping-Metadaten in XML

Die mit Java-EE 5 begonnene Entwicklung, Konfigurations- und Metadaten nicht in XML, sondern mit Hilfe von Annotationen in Java direkt definieren zu können, ist bereits in JPA 1.0 vollständig umgesetzt worden. Nichtsdestotrotz lassen sich in JPA Mapping-Metadaten auch in XML beschreiben.

Wir sind der Meinung, dass Annotationen in den meisten Fällen der Vorzug vor XML zu geben ist. Trotzdem kann es Gründe geben, das Mapping in XML zu beschreiben, etwa unterschiedliche Namensregeln in verschiedenen Deployment-Szenarien, z. B. bei verschiedenen Kunden. Wir gehen daher kurz auf die Definition von Mapping-Metadaten in XML ein.

Die Default-Datei für Mapping-Metadaten ist die Datei `orm.xml` (object relational mapping) im Verzeichnis `META-INF` der Anwendung. Falls diese Datei existiert, wird die in ihr befindliche Mapping-Information automatisch verwendet und überschreibt die im Java-Quellcode angegebenen Annotationen.

Um das Verhalten zu verdeutlichen, verwenden wir als Beispiel einen Kunden mit der folgenden Definition.

```
@Entity
public class Kunde {

  @Id @GeneratedValue
  private Integer id;
  private String vorname;
  private String nachname;
  @Temporal(TemporalType.DATE)
  private Date geburtsdatum;
  ...
}
```

Vor- und Nachname werden hier auf identisch bezeichneten Spalten der Tabelle `Kunde` abgebildet. Als Datentyp wird ohne weitere Konfiguration `VARCHAR(255)` verwendet. Sollen die verwendete Generierungsstrategie für den Primärschlüssel (hier `AUTO`) und das Tagesdatum als Charakterisierung des Geburtsdatums beibehalten, die Spalten für Vor- und Nachnamen aber geändert werden, so ist dies mit Hilfe der Datei `orm.xml` möglich. Der folgende Code definiert die Spaltennamen `KUNDE_VORNAME` und `KUNDE_NACHNAME` sowie jeweils den Datentyp `VARCHAR(30)`.

```
<entity-mappings
    xmlns="http://java.sun.com/xml/ns/persistence/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
         http://java.sun.com/xml/ns/persistence/orm_2_0.xsd"
    version="2.0">
    <entity class="de.jpainfo.model.Kunde">
      <attributes>
        <basic name="vorname">
          <column name="KUNDE_VORNAME" length="30" />
        </basic>
        <basic name="nachname">
          <column name="KUNDE_NACHNAME" length="30" />
        </basic>
      </attributes>
    </entity>
</entity-mappings>
```

Dieser kleine Ausflug in die Verwendung von XML zur Konfiguration von JPA ist hiermit beendet. Er sollte dem Leser klarmachen, dass es Alternativen zu den von uns bisher und in den nächsten Kapiteln weiterhin verwendeten Annotationen gibt. Da diese Alternativen für die meisten Leser wahrscheinlich nicht von vorrangigem Interesse sind, haben wir die vollständige Darstellung der XML-Konfiguration von JPA in Kapitel 10 ausgelagert.

Projekt:

Die in Abschnitt 3.7 dargestellten Code-Beispiele zum Thema *Metadaten in XML* sind im Projekt *orm* enthalten.

4 Objekte und Beziehungen

4.1 Standardbeziehungen

Beziehungen zwischen Objekten sind von fundamentaler Bedeutung im objektorientierten Entwurf. Klassen sollten nicht zu groß sein, um die Übersichtlichkeit und Wartbarkeit zu gewährleisten. Um realistische Probleme zu lösen, müssen daher mehrere, in der Regel sehr viele Klassen verwendet werden, die in der einen oder anderen Art zusammenarbeiten. Die Zusammenarbeit erfolgt dabei sowohl passiv auf der Ebene der Datenstrukturen als auch aktiv auf der Ebene der Prozesse. Aus Sicht von JPA sind die Datenstrukturen von besonderem Interesse.

Das Entity-Relationship-Modell ist ein etabliertes Modell beim Entwurf von Datenbanken, um Daten als Entitäten, Beziehungen und Attribute zu modellieren. Historisch gesehen wurde der ER-Entwurf vor dem objektorientierten Entwurf entwickelt und auch praktiziert.

Beim Entwurf kompletter Anwendungen mit Java, JPA und SQL treten in der Praxis alle möglichen Konstellationen auf, wobei die „grüne Wiese", d. h. ein Neuprojekt ohne zu verwendende Altsysteme, sicher für den Entwickler die erfreulichste ist. Es soll hier jedoch keine Analyse möglicher Projektkonstellationen erfolgen, sondern lediglich auf den Bedarf einer konsistenten Verwendung von Bezeichnungen über Programmierparadigmen hinweg hingewiesen werden. Wir führen im Folgenden daher Konzepte mitsamt ihren Bezeichnungen ein, die mit der JPA-Spezifikation korrespondieren. Obwohl JPA für den Java-Entwickler einen einfachen Zugang zur Persistenz realisieren soll, kommen die Konzepte und Bezeichnungen aus dem Datenbankbereich, wie das folgende Zitat aus der Spezifikation zeigt: „These annotations mirror common practice in relational database schema modeling". Um den objektorientiert arbeitenden Software-Entwickler entgegenzukommen, versuchen wir den gängigen objektorientierten Sprachgebrauch zu integrieren.

Abbildung 4.1 zeigt ein UML-Klassendiagramm mit zwei Klassen und einer Assoziation. Während in UML zwischen Assoziation, Aggregation und Komposition unterschieden wird, spricht man in JPA allgemein von einer *Beziehung*. Für eine Einführung in den objektorientierten Entwurf mit Hilfe der UML empfehlen wir [RQZ07].

- Unter einer *Beziehung* (Relationship) versteht man einen Bezug oder Zusammenhang zwischen zwei Klassen bzw. Entities, im Beispiel Klasse1 und Klasse2.

ABBILDUNG 4.1 Assoziation zwischen zwei Klassen

- Eine Beziehung kann aus verschiedenen Blickwinkeln gesehen werden. Ein Entity nimmt in einer Beziehung eine *Rolle* (Role) ein. Dasselbe Entity kann in einer anderen Beziehung eine andere Rolle einnehmen.
- Beziehungen können eine *Richtung* haben. Im Beispiel zeigt die Richtung von Klasse1 zu Klasse2. Falls eine Richtung existiert, ist die Beziehung *unidirektional* (unidirectional), sonst *bidirektional* (bidirectional).
- Falls eine Beziehung gerichtet ist, kann zwischen der *Quelle* (Source) und dem *Ziel* (Target) unterschieden werden. Im Beispiel ist Klasse1 die Quelle, Klasse2 das Ziel der Beziehung.
- Für die beteiligten Entities können *Kardinalitäten* (Cardinality) angegeben werden. Im Beispiel ist das für beide Klassen *1..**, was eine beliebige Zahl zwischen 1 und unendlich bedeutet.
- In fast jeder Beziehung wird für eine der beteiligten Seiten die implementierende Tabelle eine Join-Spalte enthalten. Diese Seite wird *besitzende* Seite (owning side), *Besitzer* oder *Eigentümer* genannt. Die andere Seite wird *inverse* Seite genannt.

Die ER-Modellierung kennt für diese allgemeine Art der Assoziation die vier Unterscheidungen 1:1, 1:n, n:1 und n:m, wobei 1, n und m als Kardinalitäten interpretiert werden. JPA unterstützt diese durch die Annotationen `@OneToOne`, `@OneToMany` `@ManyToOne` und `@ManyToMany`. Wir führen diese Annotationen in ihren Standardausprägungen in den folgenden Abschnitten ein, und zwar sowohl im Bezug zu objektorientierten Konzepten als auch datenbankspezifisch. Komplexeren Beziehungen ist Abschnitt 4.2 gewidmet.

4.1.1 1:1-Beziehungen

Wir beginnen mit der 1:1-Beziehung, da dies der einfachste Fall und somit für eine Einführung der am besten geeignete ist. Wenn für einen Kunden explizit eine Adresse verwaltet werden soll, ergibt sich ein Klassenmodell wie in Abbildung 4.2.

Die Beziehung im Beispiel ist unidirektional, d. h. man kann vom Kunden zur Adresse navigieren, jedoch nicht umgekehrt. Als Kardinalitäten der Beziehung ist an beiden Seiten 0 oder 1 zugelassen. Dies bedeutet, dass ein Kunde eine Adresse haben kann, aber nicht muss und dass es Adressen geben kann, die zu keinem Kunden gehören. Dies ist der allgemeinste Fall einer 1:1-Beziehung und soll zunächst als Grundlage dienen. Später werden wir diese Eigenschaften anwendungsbezogen optimieren.

JPA erlaubt die Modellierung einer 1:1-Beziehung mit der Annotation `@OneToOne`. Dabei wird das objektwertige Attribut des Beziehungsziels in der Quell-Klasse annotiert, wie der folgende Code der beiden Klassen verdeutlicht.

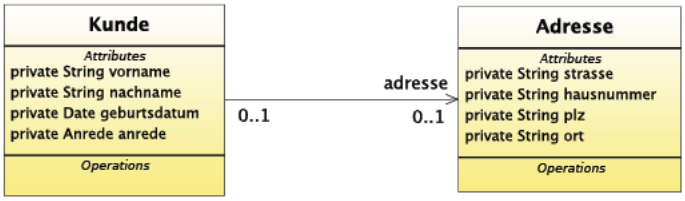


ABBILDUNG 4.2 1:1-Beziehung zwischen Kunde und Adresse

```
@Entity
public class Kunde {

  @Id @GeneratedValue
  private Integer id;
  private String vorname;
  private String nachname;
  @Temporal(TemporalType.DATE)
  private Date geburtsdatum;
  private Anrede anrede;
  @OneToOne
  private Adresse adresse;
  ...
```

```
@Entity
public class Adresse {

  @Id @GeneratedValue
  private Integer id;
  private String strasse;
  private String hausnummer;
  private String plz;
  private String ort;
  ...
```

In der zugrunde liegenden Tabellenstruktur enthält die Tabelle `Kunde` einen Fremdschlüssel in die Tabelle `Adresse`. Man bezeichnet den Kunden daher auch als den *Besitzer* (Owner) der Beziehung, die Adresse als nicht besitzende oder inverse Seite. Der Name der Fremdschlüsselspalte wird im Default-Fall wie folgt gebildet: Aneinanderkettung des Namens der Zielklasse, in unserem Fall `Adresse`, dem Unterstrich und dem Namen des Primärschlüssels der Zielklasse, in unserem Fall `id`. Falls dies nicht erwünscht ist, kann der Name überschrieben werden. Da die Fremdschlüsselspalte datenbanktechnisch als Join-Spalte für diese Assoziation verwendet wird, lautet die entsprechende Annotation `@JoinColumn`. Der Name kann über das Attribut `name` gesetzt werden, so dass das Property folgendermaßen annotiert wird:

```
@OneToOne
@JoinColumn(name = "adresse")
private Adresse adresse;
```

Das entsprechende Tabellenschema mit dem Fremdschlüssel `adresse` zeigt Abbildung 4.3.

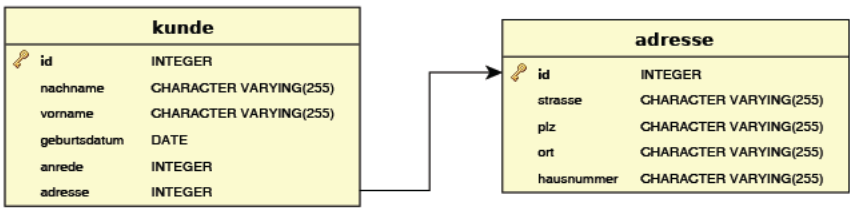


ABBILDUNG 4.3 Tabellenstruktur für die Beziehung Kunde–Adresse

Hinweis:

Im UML-Klassendiagramm wurde der Assoziationsname *adresse* gewählt. Im Java-Code des Entity `Kunde` ist dies der Name eines Property, das jedoch nicht im UML-Klassendiagramm vorhanden ist. Dies ist kein Versehen, sondern gewollt: In der Modellierung ist *adresse* der Name der Assoziation, ein derartiges Property ist nicht vorhanden. In der tatsächlichen Realisierung werden Assoziationen über Properties realisiert, so dass hier das Property `adresse` vorhanden sein muss.

Die Verwendung der beiden Klassen erfolgt intuitiv, wie sie auch ohne JPA realisiert werden würde. Um einen Kunden mitsamt Adresse zu speichern, werden die beiden Entities erzeugt, die Assoziation mit `setAdresse()` hergestellt und beide Entities persistiert.

```
Kunde kunde = new Kunde("Max", "Mustermann", ...);
Adresse adresse = new Adresse("Goethestrasse", ...);
kunde.setAdresse(adresse);
em.persist(kunde);
em.persist(adresse);
```

Nach dem Lesen eines Kunden kann über `getAdresse()` auf die Adresse zugegriffen werden. Die Assoziation wird also von JPA automatisch aufgelöst.

```
Kunde kunde = em.find(Kunde.class, 1);
System.out.println("Kunde " + kunde +
                   " wohnt in " + kunde.getAdresse());
```

Gehen wir nun daran, verschiedene Optionen beziehungsweise Anforderungen bezüglich der Assoziation zu realisieren. Eine mögliche Anforderung könnten die Beschränkung der Kardinalität der Zielseite auf 1 sein. Mit anderen Worten: Jeder Kunde muss eine Adresse haben. Dies ist relativ einfach mit dem `optional`-Attribut der `@OneToOne`-Annotation realisierbar, das mit dem Wert `false` die Existenz der Beziehung erzwingt.

```
@OneToOne(optional = false)
@JoinColumn(name = "adresse")
private Adresse adresse;
```

Leider wird die Existenz der Beziehung von EclipseLink nicht geprüft, so dass zusätzlich die Join-Column auf `not null` beschränkt werden muss.

```
@OneToOne(optional = false)
@JoinColumn(name = "adresse", nullable = false)
private Adresse adresse;
```

Die zusätzliche Verwendung der Annotationen `@Basic(optional = false)` oder `@Column(nullable = false)` statt der `@JoinColumn`-Erweiterung führt nicht zum Ziel. Die `@Basic`-Variante ist nur ein Hint an den JPA-Provider und muss von ihm nicht implementiert werden. Die `@Column`-Annotation darf nicht auf Fremdschlüssel-Properties angewendet werden. Zur Vollständigkeit sind die Annotationen `@OneToOne` und `@JoinColumn` in den Tabellen 4.1 und 4.2 mit ihren Attributen dargestellt.

Eine weitere Optimierungsmöglichkeit ist das Automatisieren der Persistenz von Beziehungsgeflechten, d. h. Objekte, die über Beziehungen verbunden sind. Im Beispiel wurden oben sowohl der Kunde als auch die Adresse persistiert:

TABELLE 4.1 `@OneToOne`-Attribute

`@OneToOne(...)`			
Attribut	**Typ**	**Default**	**Beschreibung**
`cascade`	`CascadeType[]`	—	Zu kaskadierende Operationen auf der Zielseite der Assoziation
`fetch`	`FetchType`	`EAGER`	Ladezeitpunkt der Assoziation (`EAGER`, `LAZY`)
`mappedBy`	`String`	—	Name des Property der Beziehung auf der Eigentümerseite
`optional`	`boolean`	`true`	Angabe, ob Assoziation optional ist
`orphanRemoval`	`boolean`	`false`	Angabe, ob Orphan gelöscht werden soll
`targetEntity`	`Class`	`void.class`	Klasse des Ziels der Assoziation

TABELLE 4.2 `@JoinColumn`-Attribute

`@JoinColumn(...)`			
Attribut	**Typ**	**Default**	**Beschreibung**
`columnDefinition`	`String`	—	Zusätzliches SQL-Fragment zur Definition der Spalte
`insertable`	`boolean`	`true`	Angabe, ob die Spalte in Insert-Anweisungen verwendet werden soll
`name`	`String`	—	Der Name der Fremdschlüsselspalte
`nullable`	`boolean`	`true`	Angabe, ob die Fremdschlüsselspalte Nullwerte zulässt
`referencedColumnName`	`String`	—	Name der Spalte, die durch diese Fremdschlüsselspalte referenziert wird
`table`	`String`	—	Name der Tabelle (nur sinnvoll bei Sekundärtabellen)
`unique`	`boolean`	`false`	Angabe, ob das Property ein eindeutiger Schlüssel ist
`updatable`	`boolean`	`true`	Angabe, ob die Spalte in Update-Anweisungen verwendet werden soll

```
Kunde kunde = new Kunde("Max", "Mustermann", ...);
Adresse adresse = new Adresse("Goethestrasse", ...);
kunde.setAdresse(adresse);
em.persist(kunde);
em.persist(adresse);
```

Der zweimalige `persist()`-Aufruf ist nicht effektiv und bei größeren Beziehungsgeflechten fehleranfällig. JPA unterstützt ein zentrales Konzept des OR-Mappings, nämlich *Persistence by Reachability*, auch *transitive Persistenz* genannt. Dies bedeutet, dass ein Entity nicht nur explizit persistent gemacht werden kann, sondern dass es implizit (automatisch) persistent wird, wenn es von einem bereits persistenten Entity referenziert wird.

Bei der 1:1-Beziehung zwischen `Kunde` und `Adresse` soll nun automatisch beim Persistieren eines Kunden auch seine Adresse gespeichert werden. Dabei kann die Adresse optional oder auch verpflichtend vorhanden sein. Um dies zu erreichen, wird der Kaskadierungstyp `PERSIST` für die Option `cascade` der `@OneToOne`-Annotation (siehe `@OneToOne`-Optionen

in Tabelle 4.1 auf der vorherigen Seite) verwendet. Die Option erwartet ein Array von Werten der Enumeration `javax.persistence.CascadeType`. Mögliche Werte sind `DETACH`, `MERGE`, `PERSIST`, `REFRESH`, `REMOVE` und `ALL`. Bei der Angabe einer oder mehrerer dieser Enumeration-Werte werden die gleichnamigen Methoden des Entity-Managers (`detach()`, `merge()`, `persist()`, `refresh()`, `remove()`) kaskadiert: Wenn die Methoden auf dem Entity ausgeführt werden, das die Beziehungsannotation enthält, werden sie auch auf das Beziehungsziel ausgeführt. Die Verwendung von `ALL` ist gleichbedeutend mit `cascade = {DETACH, MERGE, PERSIST, REFRESH, REMOVE}`.

In der überarbeiteten Version mit kaskadierendem Persistieren

```
@OneToOne(cascade = CascadeType.PERSIST)
private Adresse adresse;
```

genügt nun das alleinige Persistieren des Kunden. Die Adresse wird automatisch gespeichert.

```
Kunde kunde = new Kunde("Max", "Mustermann", ...);
Adresse adresse = new Adresse("Goethestrasse", ...);
kunde.setAdresse(adresse);
em.persist(kunde);
```

Oft ist es sinnvoll, alle Kaskadierungsoptionen zu verwenden, so dass sich

```
@OneToOne(cascade = CascadeType.ALL)
private Adresse adresse;
```

ergibt. Beim Löschen eines Kunden mit

```
em.remove(kunde);
```

wird dann auch die Adresse gelöscht, falls sie vorhanden ist.

Eine weitere Optimierung ist das automatische Löschen verwaister Objekte. Unter einer Waise (englisch Orphan) versteht man ein Objekt, das nicht mehr über eine Beziehung referenziert wird. Wenn ein Kunde umzieht und eine neue Adresse erhält

```
kunde.setAdresse(new Adresse("Schillerstrasse", ...));
```

oder der Kunde unbekannt verzogen ist

```
kunde.setAdresse(null);
```

existiert das alte Adresse-Objekt sowohl in der JVM als auch in der Datenbank. In der JVM wird es eventuell vom Garbage-Collector entfernt. In der Datenbank muss es explizit mit `em.remove()` gelöscht werden. Mit JPA 2.0 wurde das Orphan-Removal eingeführt, das das Objekt automatisch löscht. Hierzu muss das Attribut `orphanRemoval` (siehe Tabelle 4.1 auf der vorherigen Seite) der Beziehung gesetzt werden:

```
@OneToOne(cascade = CascadeType.ALL, orphanRemoval = true)
private Adresse adresse;
```

Projekt:

Der Beispiel-Code zum Thema unidirektionale 1:n-Beziehungen ist im Projekt *beziehungen-1-zu-1-unidirektional* enthalten.

4.1.1.1 Bidirektionale 1:1-Beziehungen

Das bisher verwendete Beispiel der Beziehung zwischen `Kunde` und `Adresse` war unidirektional mit der Möglichkeit der Navigation von einem Kunden zu seiner Adresse. Mit der in Kapitel 7 beschriebenen Abfragesprache lassen sich für eine gegebene Adresse alle dort wohnenden Kunden herausfinden. Mit den uns bisher zur Verfügung stehenden Mitteln ist dies jedoch nicht möglich. Dieses Manko kann durch eine *bidirektionale* 1:1-Beziehung behoben werden. Dazu wird sowohl im Kunden als auch in der Adresse das entsprechende Property mit `@OneToOne` annotiert. Die bisher eingeführten Attribute der `@OneToOne`-Annotation sind auch bei bidirektionalen Beziehungen erlaubt und anwendungsfallbezogen einzusetzen. Wir haben in der folgenden Darstellung der beiden Entities jedoch der Übersichtlichkeit halber darauf verzichtet.

```
@Entity
public class Kunde {

  @Id @GeneratedValue
  private Integer id;
  private String vorname;
  private String nachname;
  @Temporal(TemporalType.DATE)
  private Date geburtsdatum;
  private Anrede anrede;
  @OneToOne
  @JoinColumn(name = "adresse")
  private Adresse adresse;
  ...
```

```
@Entity
public class Adresse {

  @Id @GeneratedValue
  private Integer id;
  private String strasse;
  private String hausnummer;
  private String plz;
  private String ort;
  @OneToOne(mappedBy = "adresse")
  private Kunde kunde;
  ...
```

Man erkennt im Entity `Adresse` das zusätzliche Property `kunde` vom Typ `Kunde`, das ebenfalls mit `@OneToOne` annotiert ist. Ohne das Attribut `mappedBy` (siehe Tabelle 4.1 auf Seite 93) wird wie für das Entity `Kunde` oben dargestellt, eine unidirektionale 1:1-Beziehung mit `Adresse` als Besitzer vorausgesetzt. Man erhält also zwei voneinander unabhängige unidirektionale 1:1-Beziehungen. Um eine einzige bidirektionale 1:1-Beziehung zu erhalten, muss auf den Fremdschlüssel in der `Adresse`-Tabelle verzichtet und statt dessen die Beziehung auf das entsprechende Property der besitzenden Seite abgebildet („ge-mapped“) werden. Dies wird mit dem Attribut `mappedBy` erreicht.

Achtung:

Der in der `@JoinColumn`-Annotation verwendete Wert des Attributs `name` ist der Name der Tabellenspalte in der Datenbank, während der in der `@OneToOne`-Annotation verwendete Wert des Attributs `mappedBy` der Name des Java-Properties ist.

Auf die noch nicht beschriebenen Attribute der `@OneToOne`- und `@JoinColumn`-Annotationen gehen wir in den nächsten Abschnitten ein.

Projekt:

Der Beispiel-Code zum Thema bidirektionale 1:n-Beziehungen ist im Projekt *beziehungen-1-zu-1-bidirektional* enthalten.

4.1.2 1:n- und n:1-Beziehungen

Die 1:n-Beziehung, uni- oder bidirektional, ist sicher die am häufigsten verwendete Beziehungsart in der objektorientierten Modellierung. Da sich sowohl 1:1- als auch n:m-Beziehungen mit Hilfe von 1:n-Beziehungen realisieren lassen, ist sie auch die universellste Beziehungsart. Die n:1-Beziehung entspricht der 1:n-Beziehung, jedoch aus einem anderen Blickwinkel.

Als Grundlage zur Einführung von 1:n-Beziehungen wählen wir die Klassen `Konto` und `Buchung`, wie in Abbildung 4.4 dargestellt. Die Abbildung zeigt bereits den bidirektionalen Fall. Wir beginnen jedoch zunächst mit einer unidirektionalen 1:n-Beziehung.

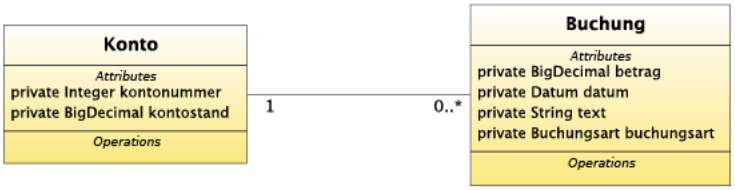


ABBILDUNG 4.4 1:n-Beziehung zwischen Konto und Buchung

Unidirektionale 1:n-Beziehungen können in JPA sehr einfach realisiert werden. Die objektwertige Seite, in unserem Fall die Klasse `Konto`, erhält ein collection-wertiges Property mit einer `@OneToMany`-Annotation. Generell sind als Property-Typen die Interfaces `Collection`, `Set`, `List` und `Map` des Package `java.util` erlaubt. Wir verwenden im Folgenden meist `Set`, um SQL-Ergebnis*mengen* zu symbolisieren. Die Verwendung des Super-Interface `Collection` wäre ebenfalls möglich. `List` und `Map` sind als Spezialfälle zu sehen und werden im Abschnitt 4.2 gesondert erläutert.

Für den einfachsten Fall einer 1:n-Beziehung ist damit die Realisierung bereits abgeschlossen, so dass sich der folgende Code für die beiden Entities ergibt.

```
@Entity
public class Konto {

  @Id @GeneratedValue
  private Integer kontonummer;
  private BigDecimal kontostand;

  @OneToMany(cascade =
             CascadeType.ALL)
  private Set<Buchung> buchungen;
  ...
```

```
@Entity
public class Buchung {

  @Id @GeneratedValue
  private Integer id;
  private BigDecimal betrag;
  @Temporal(TemporalType.DATE)
  private Date datum;
  private String text;
  private Buchungsart buchungsart;
  ...
```

Wir haben das Attribut `cascade` der `@OneToMany`-Annotation verwendet, das sich identisch zum `cascade`-Attribut der `@OneToOne`-Annotation verhält und bereits im Abschnitt 4.1.1 eingeführt wurde. Die Tabelle 4.3 zeigt alle Attribute der `@OneToMany`-Annotation, die den `@OneToOne`-Attributen entsprechen. Lediglich das nicht sinnvolle `optional` fehlt.

Für eine derartige 1:n-Beziehung muss eine JPA-Implementierung neben den beiden Tabellen für die Entity-Klassen eine Join-Tabelle vorsehen, die aus je einem Fremdschlüssel

TABELLE 4.3 `@OneToMany`-Attribute

`@OneToMany(...)`			
Attribut	**Typ**	**Default**	**Beschreibung**
`cascade`	`CascadeType[]`	—	Zu kaskadierende Operationen auf der Zielseite der Assoziation
`fetch`	`FetchType`	`LAZY`	Ladezeitpunkt der Assoziation (`EAGER`, `LAZY`)
`mappedBy`	`String`	—	Name des Property der Beziehung auf der Eigentümerseite. Bei bidirektionalen Beziehungen zwingend erforderlich
`orphanRemoval`	`boolean`	`false`	Angabe, ob Orphan gelöscht werden soll
`targetEntity`	`Class`	`void.class`	Zielklasse der Assoziation

für die beiden Entity-Klassen besteht. Auch die Bezeichner der Tabelle und der Fremdschlüssel werden in der Spezifikation definiert. Der Tabellenname ergibt sich aus den mit Unterstrich aneinandergehängten Entity-Namen (zuerst Quelle, dann Ziel) (im Beispiel also `Konto_Buchung`). Der Name der ersten Fremdschlüsselspalte wird aus dem Quell-Entity, dem Unterstrich sowie dem Primärschlüssel-Property (`Konto_kontonummer`) gebildet. Der Name der zweiten Fremdschlüsselspalte wird aus dem Beziehungs-Property, dem Unterstrich sowie dem Primärschlüssel-Property des Ziel-Entity gebildet (`Buchungen_id`). Abbildung 4.5 zeigt diese Tabellenstruktur. Da SQL-Bezeichner nicht case-sensitiv sind, ist die unterschiedliche Schreibung dem verwendeten Werkzeug geschuldet.

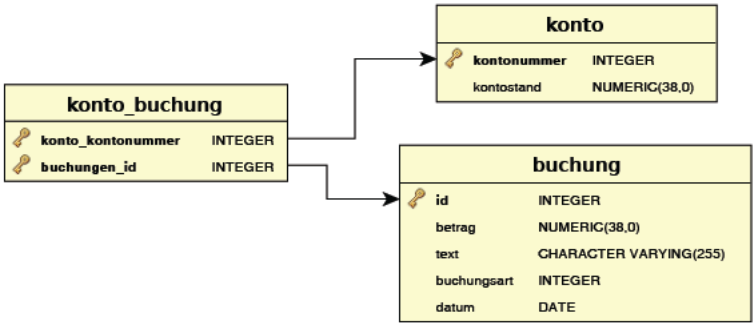


ABBILDUNG 4.5 Tabellenstruktur für die unidirektionale 1:n-Beziehung Konto–Buchung

Das bisher noch nicht verwendete Attribut `targetEntity` der `@OneToMany`-Annotation und der anderen Beziehungs-Annotationen unterstützt die Verwendung von nicht typisierten Collections. Falls im Beispiel für das Property `buchungen` eine nicht generische Menge verwendet werden soll, ist im Attribut `targetEntity` der Typ der Zielklasse der Assoziation anzugeben:

```
@OneToMany(targetEntity = Buchung.class)
private Set buchungen;
```

Wie werden nun aber 1:n-Beziehungen verwendet? Zunächst muss die Assoziationsvariable mit einem entsprechenden Datentyp initialisiert werden. Beim deklarierten Typ `Set` z. B. mit einer `HashSet`. Dies kann bei der Variablendeklaration oder im Konstruktor erfolgen. Wir wählen hier die Konstruktorvariante und setzen außerdem den Kontostand auf 0.

```
public Konto() {
  buchungen = new HashSet<Buchung>();
  kontostand = BigDecimal.ZERO.setScale(2);
}
```

Der gewählte Typ, hier also `HashSet`, wird vom JPA-Provider jedoch spätestens beim Lesen eines Kontos aus der Datenbank durch eine provider-spezifische Implementierung ausgetauscht und spielt in der weiteren Verwendung keine Rolle.

Es können nun Konten angelegt und Buchungen hinzugefügt werden. Der folgende Code-Ausschnitt verdeutlicht dies exemplarisch. Wir haben bewusst eine etwas ungewöhnliche Anweisungsreihenfolge gewählt, um zu zeigen, dass am Ende der Transaktion das Konto und die Buchung persistiert werden, falls bei der Kaskadierung `PERSIST` gesetzt ist. Die Reihenfolge der Anweisungen, zunächst Persistieren des Kontos und dann Hinzufügen der Buchung, spielt keine Rolle.

```
em.getTransaction().begin();
Konto konto = new Konto();
em.persist(konto);
Buchung buchung = new Buchung(new BigDecimal("600.00"), "12.08.2011",
                              "Miete", Buchungsart.SOLL);
konto.getBuchungen().add(buchung);
em.getTransaction().commit();
```

Dem aufmerksamen Leser ist sicher aufgefallen, dass der Kontostand trotz Hinzufügen der Buchung nach wie vor 0 ist: ein offensichtlicher Fehler. Wir werden dies berichtigen, nachdem wir die Beziehung bidirektional gemacht haben.

Projekt:

Der Beispiel-Code zum Thema unidirektionale 1:n-Beziehungen ist im Projekt *beziehungen-1-zu-n-unidirektional* enthalten.

4.1.2.1 Bidirektionale 1:n-Beziehungen

Das für die unidirektionale 1:n-Beziehung beschriebene Datenbankschema mit einer zusätzlichen Tabelle ist in der Regel wenig sinnvoll. Der übliche Weg, diesen Umstand zu beseitigen, ist die Einführung einer zusätzlichen Spalte in der Zieltabelle der Beziehung, die eine Fremdschlüsselspalte in die Quelltabelle darstellt. Dies kann sehr einfach durch eine bidirektionale 1:n-Beziehung realisiert werden. Dazu verwendet man eine zusätzliche n:1-Beziehung von `Buchung` zu `Konto`, die mit Hilfe der `@ManyToOne`-Annotation definiert wird. Diese Beziehung ist aber tatsächlich *zusätzlich*, so dass zwei unabhängige Beziehungen existieren: eine 1:n-Beziehung von `Konto` zu `Buchung` und eine n:1-Beziehung von `Buchung` zu `Konto`. Um aus diesen beiden unabhängigen Beziehungen *eine* bidirektionale Beziehung zu machen, muss in beiden Entities dieselbe Fremdschlüsselspalte für beide Beziehungen definiert werden. Auf der inversen Seite erfolgt dies über das `mappedBy`-Attribut der `@OneToMany`-

Annotation, auf der besitzenden Seite über die bereits bekannte `@JoinColumn`-Annotation (Attributübersicht in Tabelle 4.2 auf Seite 93). Es ergibt sich der folgende Code für die beiden Entity-Klassen.

```
@Entity
public class Konto {

  @Id @GeneratedValue
  private Integer kontonummer;
  private BigDecimal kontostand;

  @OneToMany(mappedBy = "konto",...)
  private Set<Buchung> buchungen;
  ...
```

```
@Entity
public class Buchung {

  @Id @GeneratedValue
  private Integer id;
  private BigDecimal betrag;
  ...
  @ManyToOne
  @JoinColumn(name = "konto",
              nullable = false)
  private Konto konto;
  ...
```

Im Entity `Buchung` wird das Property `konto` mit einer `@ManyToOne`-Annotation versehen. Zusätzlich zu dieser Annotation wird jedoch die bereits erwähnte Möglichkeit zur Definition der Fremdschlüsselspalte benötigt. Die `@JoinColumn`-Annotation erlaubt genau dies mit dem Attribut `name`. Ohne das `name`-Attribut würde als Spaltenname der Tabellenname, ein Unterstrich und das Primärschlüssel-Property (`Konto_kontonummer`) verwendet.

Tipp:

Die `@JoinColumn`-Annotation ist optional und dient zum Überschreiben des Default-Namens der Join-Spalte. Da der Default-Name für Join-Spalten stark vom Default-Namen einfacher Properties abweicht, raten wir zur Verwendung der `@JoinColumn`-Annotation.

Da eine Buchung immer genau einem Konto zugeordnet sein muss, wird in der `@JoinColumn`-Annotation mit dem `nullable`-Attribut auch die Möglichkeit eines Nullwertes in dieser Fremdschlüsselspalte verhindert.

Achtung:

Der in der `@JoinColumn`-Annotation verwendete Wert des Attributs `name` ist der Name der Tabellenspalte in der Datenbank, während der in der `@OneToMany`-Annotation verwendete Wert des Attributs `mappedBy` der Name des Java-Property ist.

Die Attribute der `@ManyToOne`-Annotation sind in Tabelle 4.4 dargestellt. Sie entsprechen zum Teil den `@OneToMany`-Attributen aus Tabelle 4.3 auf Seite 97. Eine Ausnahme ist das `optional`-Attribut. Mit diesem Attribut kann alternativ sichergestellt werden, dass eine Buchung immer an ein Konto gebunden ist. Anstatt

```
@ManyToOne
@JoinColumn(name="konto", nullable=false)
private Konto konto;
```

kann mal also auch schreiben:

```
@ManyToOne(optional=false)
@JoinColumn(name="konto")
private Konto konto;
```

TABELLE 4.4 `@ManyToOne`-Optionen

`@ManyToOne(...)`			
Option	**Typ**	**Default**	**Beschreibung**
`cascade`	`CascadeType[]`	—	Zu kaskadierende Operationen auf der Zielseite der Assoziation
`fetch`	`FetchType`	`EAGER`	Ladezeitpunkt der Assoziation (`EAGER`, `LAZY`)
`optional`	`boolean`	`true`	Beziehung ist optional
`targetEntity`	`Class`	`void.class`	Klasse des Ziels der Assoziation

Die dargestellten Mapping-Informationen definieren nun ein Datenbankschema, das pro Entity eine Tabelle mit den entsprechenden Spalten für die Properties und eine zusätzliche Spalte `konto` in der Tabelle `Buchung` als Fremdschlüssel in die Tabelle `Konto` besitzt. Wir haben dies zur Verdeutlichung in Abbildung 4.6 dargestellt. Der Leser möge sich von der Vereinfachung im Vergleich zum Datenbankschema der unidirektionalen Beziehung in Abbildung 4.6 selbst überzeugen.

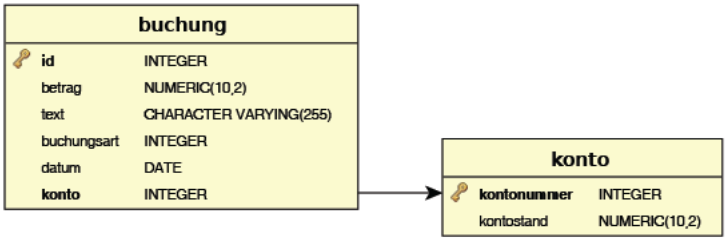


ABBILDUNG 4.6 Tabellenstruktur für die bidirektionale 1:n-Beziehung Konto–Buchung

Bei der Verwendung der beiden Klassen ist darauf zu achten, dass die Beziehung auch wirklich bidirektional verwendet wird, also auf beiden Seiten der Beziehung tatsächlich die entsprechenden Setter aufgerufen werden. Eine Buchung muss daher in die Collection der Buchungen auf der Konto-Seite aufgenommen werden, in der Buchung selbst muss aber auch das Konto gesetzt werden. Beim folgenden Beispiel gehen wir davon aus, dass die Kaskadierung der Assoziation mindestens `PERSIST` umfasst.

```
Konto konto = new Konto();
Buchung buchung = new Buchung(...);
konto.getBuchungen().add(buchung);
buchung.setKonto(konto);
em.persist(konto);
```

Der Zwang zur Verlinkung auf beiden Seiten der Beziehung ist unschön und fehleranfällig. Die eigentliche Erstellung der Beziehung gehört logisch zur Funktionalität eines Kontos und sollte dort auch realisiert werden, so dass wir die Klasse `Konto` überarbeiten.

```
@Entity
public class Konto {

  @Id @GeneratedValue
  private Integer kontonummer;
```

```
  @Column(precision = 10, scale = 2)
  private BigDecimal kontostand;
  ...
  public void addBuchung(Buchung buchung) {
    buchung.setKonto(this);
    buchungen.add(buchung);
    if (buchung.getBuchungsart() == HABEN) {
      kontostand = kontostand.add(buchung.getBetrag());
    } else if (buchung.getBuchungsart() == SOLL) {
      kontostand = kontostand.subtract(buchung.getBetrag());
    } else {
      throw new RuntimeException("Buchungsart muss bekannt sein.");
    }
  }
  ...
}
```

Neben der Verlinkung in beide Richtungen in den ersten beiden Zeilen der `addBuchung()`-Methode wurde das Kontostand-Property mit der `@Column`-Annotation versehen. Dies gewährleistet, dass wir die entsprechenden Nachkommastellen bei Euro-Beträgen zur Verfügung haben, und ist entsprechend auch in der Klasse `Buchung` für das `betrag`-Property hinzuzufügen.

JPA erlaubt den Entwurf von *Domain-Models* bzw. *Rich Domain-Models* im Gegensatz zu *Anemic Domain-Models*. Ein Anemic Domain-Model realisiert Anwendungsobjekte ausschließlich über Getter und Setter und lagert die Anwendungslogik in separate Klassen aus. Diese Art der Modellierung wurde z. B. durch das Entity-Modell der frühen EJB-Spezifikationen gestärkt, die Rich Domain-Models nicht oder nur unzureichend unterstützten. Mit der Einführung von JPA 1.0 gibt es keinen Grund mehr, ein Anemic Domain-Model zu verwenden, und wir raten dem Leser ausdrücklich, Rich Domain-Models zu erstellen. Martin Fowler bezeichnet das Anemic Domain-Model ausdrücklich als Anti-Pattern [URL-ADM].

Beim Löschen einer Buchung muss diese zunächst aus der Collection entfernt und dann über den Entity-Manager gelöscht werden.

```
konto.getBuchungen().remove(buchung);
em.remove(buchung);
```

Will man auch diesen Code in die Klasse `Konto` migrieren, so ergibt sich ein Problem mit dem Entity-Manager. Dieser ist innerhalb der Entity-Klasse nicht bekannt und sollte auch nicht z. B. durch einen Methodenparameter übergeben werden. Abhilfe schafft hier das Attribut `orphanRemoval`, das das Löschen in der Datenbank automatisiert. Der entstehende Code ist damit recht einfach. Wir verzichten aus Gründen der Übersichtlichkeit auf die Aktualisierung des Kontostands.

```
@Entity
public class Konto {
  ...
  public void deleteBuchung(Buchung buchung) {
    buchungen.remove(buchung);
  }
  ...
```

Hinweis:

Dem Leser wird aufgefallen sein, dass durch die Verwendung der Kaskadierungstypen `PERSIST`, `REMOVE` oder `ALL` sowie des Attributs `orphanRemoval` aus der einfachen Assoziation eine Komposition geworden ist. Die Lebensdauer einer Buchung ist an die Lebensdauer eines Kontos gebunden. Dies ist fachlich auch so korrekt. Wir verzichten auf eine Korrektur des UML-Klassendiagramms.

Hinweis:

Um die direkte Manipulation der Buchungen zu verhindern, kann der Setter `private` deklariert werden. Beim Getter ist dies nicht möglich. Eine Alternative ist die Verwendung eines read-only Getters, z. B. über die Methode `Collections.unmodifiableSet()`:

```
public Set<Buchung> getBuchungen() {
  return Collections.unmodifiableSet(buchungen);
}
```

Dies erzwingt eventuell weitere Änderungen des APIs und sollte daher gut überlegt werden.

Die Beziehung zwischen Konto und Buchung ist nun bidirektional. Die Verwendung wird jedoch in den meisten Fällen vom Konto zu den Buchungen stattfinden, etwa über eine For-Schleife z. B. für das Erstellen eines Kontoauszuges:

```
Konto konto = em.find(Konto.class, ...);
for (Buchung buchung : konto.getBuchungen()) {
  // hier die Buchungen be/verarbeiten
}
```

Wird ein Konto auf diese Art und Weise aus der Datenbank in die VM geladen, wird eine Select-Anweisung für die Tabelle `Konto` abgesetzt, wobei der Primärschlüssel des Entity in der Where-Klausel verwendet wird, etwa der Art

```
SELECT KONTONUMMER, KONTOSTAND
FROM KONTO WHERE (KONTONUMMER = ?)
```

Greift man nun wie im Beispiel über den Getter `getBuchungen()` auf die Buchungen dieses Kontos zu, so setzt man eine erneute Select-Anweisung ab, die strukturell die folgende Form hat:

```
SELECT ID, BETRAG, TEXT, BUCHUNGSART, DATUM, konto
FROM BUCHUNG WHERE (konto = ?)
```

Diese SQL-Anweisung wird tatsächlich erst beim Aufruf des Getters ausgeführt. Der Getter bzw. die Entity-Klasse wird hierzu über ein zwischengeschaltetes Proxy oder über Byte-Code-Manipulationen erzeugt und ausgeführt, worauf wir provider-spezifisch in den Abschnitten 11.2, 12.2 und 13.2 eingehen. Diese Art des Ladens nennt man *Lazy Loading* (spätes Laden). Der Ladezeitpunkt ist über das Attribut `fetch` der `@OneToMany`-Annotation einstellbar. Die beiden möglichen Alternativen sind die Werte `EAGER` und `LAZY` der Enum `FetchType`. Das *Eager Loading* (frühes Laden) lädt alle Buchungen eines Kontos, wenn das Konto geladen wird.

```
@OneToMany(mappedBy = "konto", fetch = FetchType.EAGER)
private Set<Buchung> buchungen;
```

Dies ist in der Regel nicht sinnvoll, da eventuell nie auf die Buchungen eines Kontos zugegriffen und damit ein unnötiger Aufwand verursacht wird. Der von JPA vorgesehene Standardfall ist daher auch Lazy Loading und muss nicht explizit angegeben werden.

```
@OneToMany(mappedBy = "konto", fetch = FetchType.LAZY) // Default
private Set<Buchung> buchungen;
```

Der unbedachte Umgang mit Lazy Loading kann jedoch zu Performanzproblemen führen. Das damit im Zusammenhang stehende n+1-Select-Problem und seine Lösung stellen wir in Abschnitt 4.4 dar.

Die Werte `EAGER` und `LAZY` für die Fetch-Art der `@OneToMany`-Annotation werden von JPA unterschiedlich behandelt. Während `EAGER` eine Anforderung (Requirement) an den JPA-Provider ist, ist `LAZY` ein Hinweis (Hint). `EAGER` muss also vom JPA-Provider umgesetzt werden, während `LAZY` umgesetzt werden kann oder auch nicht. Im Beispiel Konto-Buchung wird von allen drei Providern sowohl das frühe als auch das späte Laden realisiert.

Hinweis:

Alle drei Provider realisieren das frühe Laden von 1:n-Beziehungen, jedoch auf unterschiedliche Art. Während EclipseLink zwei Select-Anweisungen wie oben angedeutet nacheinander absetzt, generieren Hibernate und OpenJPA eine Select-Anweisung mit einem Left-Outer-Join über die Tabellen Konto und Buchung.

Bei der n:1-Beziehung stellt sich die Sachlage anders dar. Die Wahrscheinlichkeit, dass für eine Buchung auch das zugehörige Konto verwendet wird, ist relativ hoch. Außerdem ist der eventuell unnötig geleistete Aufwand beim Laden eines Kontos wesentlich geringer als beim Laden von – unter Umständen sehr vielen – Buchungen der 1:n-Beziehung. JPA sieht daher bei der `@ManyToOne`-Annotation `EAGER` als Default-Wert für das `fetch`-Attribut vor. Auch hier gilt, dass `EAGER` umgesetzt werden muss, während `LAZY` ein Hint ist. Die Umsetzung erfolgt bei EclipseLink wiederum mit zwei Selects, bei Hibernate und OpenJPA mit einem Select und einem Inner-Join.

Tipp:

Da bidirektionale 1:n-Beziehungen sehr häufig vorkommen, lohnt es sich, eine einfache Merkregel einzuprägen:

- Bei bidirektionalen 1:n-Beziehungen (oder n:1) ist die „n-Seite“ die besitzende Seite.
- Die Join-Spalte der besitzenden Seite wird optional mit `@JoinColumn` annotiert.
- Die „1-Seite“ ist die inverse Seite, daher wird das `mappedBy`-Attribut verwendet.

Projekt:

Der Beispiel-Code zum Thema bidirektionale 1:n-Beziehungen ist im Projekt *beziehungen-1-zu-n-bidirektional* enthalten.

4.1.3 n:m-Beziehungen

Das Klassendiagramm in Abbildung 4.7 realisiert die Beziehung zwischen Kunden und Konten als n:m-Beziehung. Ein Kunde besitzt mindestens ein Konto (sonst wäre er kein Kunde), kann aber beliebig viele Konten besitzen. Ein Konto ist immer (mindestens) einem Kunden zugeordnet. Bei (Ehe-)Paaren haben zwei Kunden ein oder mehrere gemeinsame Konten, bei einer GmbH sind eventuell mehrere Geschäftsführer oder Prokuristen verfügungsberechtigt.

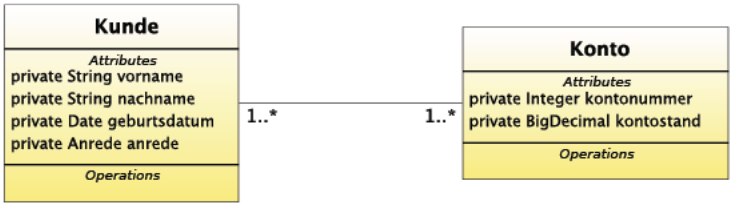


ABBILDUNG 4.7 n:m-Beziehung zwischen Kunde und Konto

Wir beschränken uns bei Kunden auf natürliche Personen. Natürliche *und* juristische Personen können mit Vererbung, die wir im Kapitel 5 einführen, beschrieben werden.

Eine n:m-Beziehung wird in JPA mit der `@ManyToMany`-Annotation definiert. Die möglichen Attribute dieser Annotation sind in Tabelle 4.5 auf der nächsten Seite dargestellt und mit Ausnahme des fehlenden `orphanRemoval` identisch zu den in Tabelle 4.3 auf Seite 97 dargestellten Attributen der `@OneToMany`-Annotation. Für eine unidirektionale n:m-Beziehung ist die `@ManyToMany`-Option für den Standardfall ausreichend, da nur die besitzende Seite annotiert werden muss. Das Attribut `mappedBy`, das in Tabelle 4.5 als verpflichtend gekennzeichnet ist, muss nur bei bidirektionalen n:m-Beziehungen angegeben werden; bei unidirektionalen ist es optional.

Zur Realisierung der unidirektionalen Kunde-Konto-Beziehung wird lediglich das collectionwertige Property `konten` in der Klasse `Kunde` mit `@ManyToMany` annotiert. Die Klasse `Konto` benötigt keinen speziellen JPA-Code.

TABELLE 4.5 `@ManyToMany`-Attribute

`@ManyToMany(...)`			
Option	**Typ**	**Default**	**Beschreibung**
`cascade`	`CascadeType[]`	—	Zu kaskadierende Operationen auf der Zielseite der Assoziation
`fetch`	`FetchType`	`LAZY`	Ladezeitpunkt der Assoziation (`EAGER`, `LAZY`)
`mappedBy`*	`String`	—	Name des Property der Beziehung auf der Eigentümerseite
`targetEntity`	`Class`	`void.class`	Zielklasse der Assoziation

```
@Entity
public class Kunde {

  @Id @GeneratedValue
  private Integer id;
  private String vorname;
  private String nachname;
  @Temporal(TemporalType.DATE)
  private Date geburtsdatum;
  private Anrede anrede;

  @ManyToMany
  private Set<Konto> konten;
  ...
```

```
@Entity
public class Konto {

  @Id @GeneratedValue
  private Integer kontonummer;
  @Column(precision = 10, scale = 2)
  private BigDecimal kontostand;
  ...
```

Um eine n:m-Beziehung in einer Datenbank zu realisieren, wird eine sogenannte Link- oder auch Join-Tabelle benötigt, die aus zwei Fremdschlüsseln besteht, die die zu verbindenden Tabellen referenzieren. JPA sieht eine solche Tabelle vor, deren Name sich durch Aneinanderhängen der beiden Entity-Namen, getrennt durch einen Unterstrich, ergibt. Der erste Spaltenname besteht aus dem Entity-Namen, einem Unterstrich und der Primärschlüsselspalte des Entity, in dem `@ManyToMany` verwendet wird. Der zweite Spaltenname besteht aus dem mit `@ManyToMany` annotierten Property-Namen, einem Unterstrich und der Primärschlüsselspalte des anderen Entity. Im Beispiel also der Tabellenname `Kunde_Konto` sowie die Spaltennamen `Kunde_id` und `konten_kontonummer`. Abbildung 4.8 zeigt diese Tabellenstruktur.

Wir haben das `cascade`-Attribut bereits bei den drei anderen Beziehungsannotationen kennengelernt. Auch in diesem Beispiel ist das kaskadierende Persistieren sinnvoll. Bei einer n:m-, aber auch bei der n:1-Beziehung kann das kaskadierende Löschen jedoch zum Löschen eines Ziel-Entity führen, das noch von anderen Entities referenziert wird. Eine derartige Kaskadierung sollte daher unterbleiben.

Achtung:

Die JPA-Spezifikation warnt ausdrücklich vor der Verwendung von `cascade = REMOVE` bei `@ManyToMany`- und `@ManyToOne`-Beziehungen. Dies führt zu nicht portablen Anwendungen.

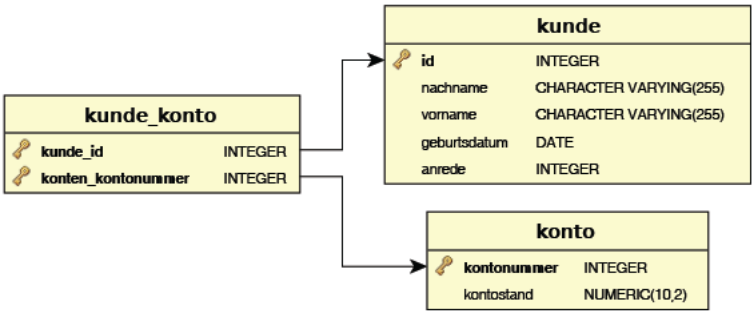


ABBILDUNG 4.8 Tabellenstruktur für die Beziehung Kunde–Konto

Eine weitere sinnvolle Code-Erweiterung ist die Möglichkeit, die Join-Tabelle und deren Spalten explizit benennen zu können. Dies erfolgt mit Hilfe der `@JoinTable`- und `@JoinColumn`-Annotationen. Die Attribute der `@JoinTable`-Annotation sind in Tabelle 4.6 dargestellt. Die Attribute der `@JoinColumn`-Annotation hatten wir bereits in Tabelle 4.2 auf Seite 93 aufgezählt. Im Beispiel entscheiden wir uns, den Tabellennamen entsprechend dem JPA-Standard zu wählen, die beiden Fremdschlüsselspalten jedoch wie die Entities zu benennen. Das Beispiel liest sich dann wie folgt:

TABELLE 4.6 `@JoinTable`-Attribute

`@JoinTable(...)`			
Attribute	**Typ**	**Default**	**Beschreibung**
`catalog`	`String`	—	Katalogname
`inverseJoinColumns`	`JoinColumn[]`	`{}`	Fremdschlüsselspalte(n) der Join-Tabelle zur nichtbesitzenden Tabelle
`joinColumns`	`JoinColumn[]`	`{}`	Fremdschlüsselspalten der Join-Tabelle zur besitzenden Tabelle
`name`	`String`	—	Name der Join-Tabelle
`schema`	`String`	—	Schema-Name
`uniqueConstraints`	`uniqueConstraint[]`	`{}`	Unique-Constraints (Benutzung nur bei Generierung)

```
@Entity
public class Kunde {

  @Id @GeneratedValue
  private Integer id;
  ...
  @ManyToMany(cascade = CascadeType.PERSIST)
  @JoinTable(
    name = "Kunde_Konto",
    joinColumns = {@JoinColumn(name = "kunde")},
    inverseJoinColumns = {@JoinColumn(name = "konto")}
  )
  private Set<Konto> konten;
  ...
}
```

Die überarbeitete Tabellenstruktur für die Klassen `Kunde` und `Konto` sowie die benötigte Join-Tabelle zeigt Abbildung 4.9.

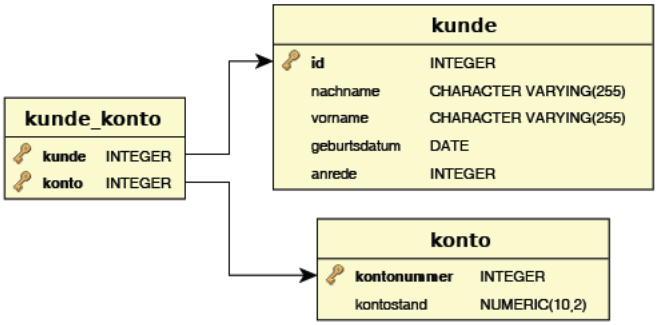


ABBILDUNG 4.9 Überarbeitete Tabellenstruktur für die Beziehung Kunde–Konto

Projekt:

Der Beispiel-Code zum Thema unidirektionale n:m-Beziehungen ist im Projekt *beziehungen-n-zu-m-unidirektional* enthalten.

4.1.3.1 Bidirektionale n:m-Beziehungen

Soll auch von einem Konto zu dem oder den besitzenden Kunden navigiert werden können, muss die Beziehung bidirektional sein. Wie bereits bei 1:n-Beziehungen motiviert, muss dazu eine inverse Beziehung von `Konto` zu `Kunde` definiert werden, hier also nochmals eine n:m-Beziehung. Dazu muss nicht noch einmal die Join-Tabelle angegeben werden, sondern lediglich das Property der Beziehung auf der Eigentümerseite über das `mappedBy`-Attribut genannt werden. Im Beispiel ist dies das Property `konten` der Klasse `Kunde`. Der Code für die beiden Klassen stellt sich damit wie folgt dar:

```
@Entity
public class Kunde {

  @Id @GeneratedValue
  private Integer id;
  ...
  @ManyToMany(cascade = PERSIST)
  @JoinTable(
    name = "Kunde_Konto",
    joinColumns =
      {@JoinColumn(name="kunde")},
    inverseJoinColumns =
      {@JoinColumn(name="konto")}
  )
  private Set<Konto> konten;
  ...
```

```
@Entity
public class Konto {

  @Id @GeneratedValue
  private Integer kontonummer;
  @Column(precision=10, scale=2)
  private BigDecimal kontostand;

  @ManyToMany(mappedBy ="konten")
  private Set<Kunde> kunden;
  ...
```

Die Verlinkung eines Kunden mit einem Konto erfolgt analog zu 1:n-Beziehungen sinnvollerweise in einer Methode `addKonto()` in der Klasse `Kunde`.

```
public void addKonto(Konto konto) {
  konto.getKunden().add(this);
  konten.add(konto);
}
```

Es kann aber auch die Erzeugung des Kontos selbst in eine entsprechende Methode ausgelagert werden.

```
public Konto addKonto() {
  Konto konto = new Konto();
  konto.getKunden().add(this);
  konten.add(konto);
  return konto;
}
```

Die n:m-Beziehung ist nun bidirektional und von beiden Entity-Seiten navigierbar. Das Tabellenschema muss nicht geändert werden und entspricht dem in Abbildung 4.9 auf der vorherigen Seite gezeigten Schema. Im folgenden Beispiel wird für jedes Konto eines bestimmten Kunden jeder weitere Verfügungsberechtigte ausgegeben.

```
Kunde kunde = em.find(Kunde.class, ...);
for (Konto konto : kunde.getKonten()) {
  System.out.print("Das Konto " + konto.getKontonummer()
                   + " gehoert " + kunde.getVorname() + " "
                   + kunde.getNachname());
  for (Kunde k : konto.getKunden()) {
    if (!k.equals(kunde)) {
      System.out.print(" und " + k.getVorname() + " "
                       + k.getNachname());
    }
  }
  System.out.println();
}
```

Das Beispiel basiert darauf, dass der Entity-Manager als First-Level-Cache die Identität von Objekten garantiert. Die Methode `equals()` verwendet für den Vergleich den Identitätsoperator von Java (==), falls sie nicht überschrieben wurde. Der in der ersten Zeile geladene Kunde wird daher in der Methode `getKunden()` nicht noch einmal in den aktuellen Kontext des Entity-Managers geladen, so dass die `equals()`-Methode auch ohne überschrieben zu sein funktioniert. Weitere Details zur Objektidentität und zum Persistenzkontext wurden in den Kapiteln 2 und 3 ausgeführt.

Projekt:

Der Beispiel-Code zum Thema bidirektionale n:m-Beziehungen ist im Projekt *beziehungen-n-zu-m-bidirektional* enthalten.

Am Ende des Abschnitts über die Standardbeziehungen `@OneToOne`, `@OneToMany`, `@ManyToOne` und `@ManyToMany` wollen wir das Attribut `fetch`, das in allen vier Annotationen verfügbar ist, in seinem Gebrauch gegenüberstellen. Tabelle 4.7 zeigt den jeweiligen Default-Wert sowie die Verbindlichkeit der Implementierung für den JPA-Provider. Als Merkregel kann formuliert werden: Der Default für Beziehungen mit objektwertigem Ziel (`@...ToOne`) ist `EAGER`, für collection-wertige Ziele (`@...ToMany`) ist `LAZY`. Frühes Laden muss für alle Beziehungen implementiert, spätes Laden kann implementiert werden.

TABELLE 4.7 Vergleich des `fetch`-Attributs

Beziehung	`fetch`-Default	EAGER / LAZY (Implementierung)
`@OneToOne`	`EAGER`	Requirement / Hint
`@OneToMany`	`LAZY`	Requirement / Hint
`@ManyToOne`	`EAGER`	Requirement / Hint
`@ManyToMany`	`LAZY`	Requirement / Hint

Hinweis:

Alle Provider realisieren auch Lazy Loading, zumindest nach entsprechender Byte-Code-Manipulation. Da diese in Application-Servern standardmäßig erfolgt, unterstützen alle Provider innerhalb ihrer jeweiligen Application-Server auch das späte Laden. Hinweise zur expliziten Byte-Code-Manipulation geben wir in den Abschnitten 11.2, 12.2 und 13.2.

4.2 Andere Beziehungsarten

4.2.1 Assoziationsklassen

Bei der Einführung von 1:n-Beziehungen hatten wir angedeutet, dass diese Art von Beziehung ausreicht, um alle anderen Beziehungsarten zu modellieren. Eine n:m-Beziehung kann durch zwei 1:n-Beziehungen realisiert werden. Dies ist insbesondere dann sinnvoll, wenn der n:m-Beziehung explizit weitere Daten zugeordnet sind. Eine zusätzliche dritte Entity-Klasse zur Verwaltung dieser Daten ist dann obligatorisch.

Zwei unabhängige 1:n-Beziehungen zur Realisierung einer n:m-Beziehung haben den Nachteil, dass sie durch zusätzlichen Programm-Code als „zusammengehörig" implementiert werden müssen. Zum Beispiel kann es immer nur beide Beziehungen oder keine geben. Ziel dieses Abschnitts ist die Implementierung einer solchen Lösung.

Abbildung 4.10 zeigt die Assoziationsklasse `Verfuegung`, die Kunden und Konten verbindet. Ziel der Modellierung sind zusätzliche Daten, die der Beziehung als solche zugeordnet sind und weder bei `Kunde` noch bei `Konto` realisiert werden können. Im Beispiel sind dies der Beginn und das Ende (`von`, `bis`) der Verfügungsberechtigung eines Kunden über ein bestimmtes Konto. Das entsprechende Tabellenschema veranschaulicht Abbildung 4.11 .

Hinweis:

Die Bezeichnung *Assoziationsklasse* ist die in der UML gebräuchliche Bezeichnung [RQZ07]. Oestereich [Oes09] nennt diese Modellierungsnotation eine *attributierte Assoziation*.
Reine n:m-Beziehungen kommen in der Praxis durchaus vor. Sie haben jedoch die Eigenschaft, über kurz oder lang zusätzliche Daten aufnehmen zu müssen. Sie sollten sich daher überlegen, ob die hier dargestellte Lösung bereits zu Projektbeginn eingesetzt werden soll.

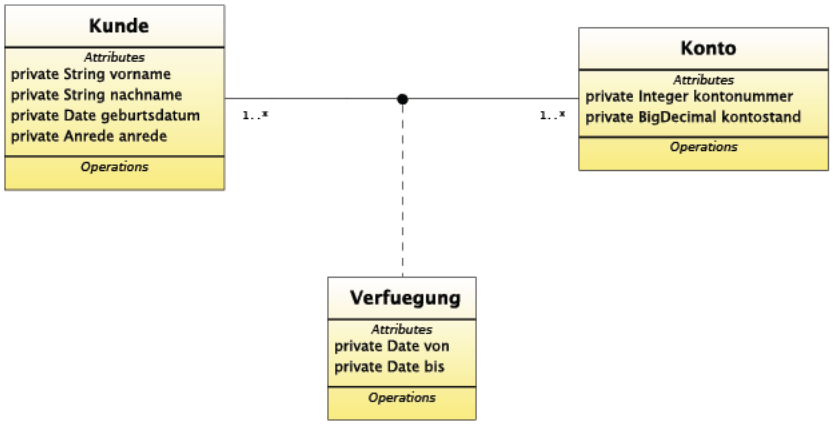


ABBILDUNG 4.10 n:m-Beziehung mit Assoziationsklasse `Verfuegung`

Die Implementierung, die wir nicht in UML darstellen, enthält die drei Entity-Klassen `Kunde`, `Konto` und `Verfuegung`. Zwischen `Kunde` und `Verfuegung` existiert eine 1:n-Beziehung, zwischen `Konto` und `Verfuegung` ebenfalls. Die Tabelle für `Verfuegung` enthält zwei Fremdschlüsselspalten in die Tabellen für `Kunde` und `Konto`. Beide Spalten sind gemeinsamer Primärschlüssel dieser Tabelle, wie für n:m-Beziehungen üblich. Da beide Spalten nicht `null` sein dürfen, haben wir der oben angesprochenen Bedingung, dass entweder beide oder keine Beziehung existieren dürfen, entsprochen.

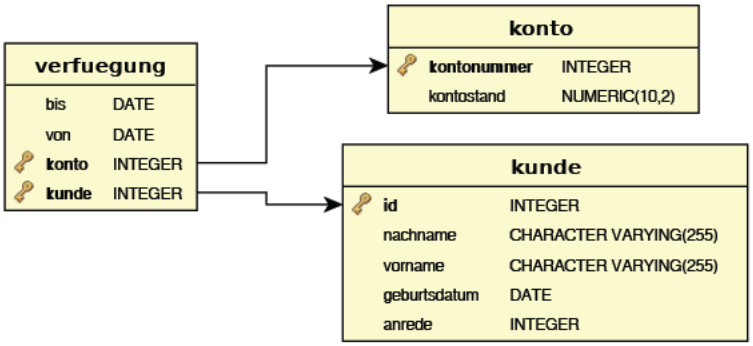


ABBILDUNG 4.11 Tabellenschema für die Assoziationsklasse aus Abbildung 4.10

Die beiden Klassen `Kunde` und `Konto` stellen sich damit wie folgt dar.

```
@Entity
public class Kunde {

  @Id @GeneratedValue
  private Integer id;
  private String vorname;
  private String nachname;
  @Temporal(TemporalType.DATE)
  private Date geburtsdatum;
  private Anrede anrede;

  @OneToMany(mappedBy="kunde")
  private Set<Verfuegung>
          verfuegungen;
  ...
```

```
@Entity
public class Konto {

  @Id @GeneratedValue
  private Integer kontonummer;
  @Column(precision = 10, scale = 2)
  private BigDecimal kontostand;

  @OneToMany(mappedBy = "konto")
  private Set<Verfuegung>
          verfuegungen;
  ...
```

Den zusammengesetzten Primärschlüssel für die Klasse `Verfuegung` realisieren wir durch eine Schlüsselklasse `VerfuegungId`, die mit der `@IdClass`-Annotation als Primärschlüsselklasse für `Verfuegung` deklariert wird. Die Primärschlüsselklasse muss `equals()` und `hashCode()` überschreiben. Wir haben diese Anforderung bereits bei der Darstellung der `@IdClass`-Annotation in Abschnitt 2.3.1 erläutert.

```
@Entity
@IdClass(VerfuegungId.class)
public class Verfuegung {

  @Id
  @ManyToOne(optional = false)
  @JoinColumn(name = "kunde")
  private Kunde kunde;

  @Id
  @ManyToOne(optional = false)
  @JoinColumn(name = "konto")
  private Konto konto;

  @Temporal(TemporalType.DATE)
  private Date von;
  @Temporal(TemporalType.DATE)
  private Date bis;
  ...
```

```
public class VerfuegungId {

  private Integer kunde;
  private Integer konto;
  ...
  @Override
  public int hashCode() {
    ...
  }

  @Override
  public boolean equals(Object obj) {
    ...
  }
}
```

Wie wird nun eine solche Beziehung mit Assoziationsklasse verwendet? Wir gehen davon aus, dass ein Kunde und ein Konto vorhanden sind und dass der Kunde ab einem bestimmten Datum über das Konto verfügen können soll. Eine entsprechende Methode wird also in der Klasse `Kunde` erstellt.

```
@Entity
public class Kunde {

  ...
  public Verfuegung addKonto(String vonString, Konto konto) {
    Date von;
    try {
      von = format.parse(vonString);
    } catch (ParseException e) {
      throw new IllegalArgumentException("Falsches Datumsformat");
```

```
    }
    Verfuegung verfuegung = new Verfuegung(this, konto, von, null);
    this.getVerfuegungen().add(verfuegung);
    konto.getVerfuegungen().add(verfuegung);
    return verfuegung;
  }
  ...
}
```

Die Methode `addKonto()` fügt für ein bestehendes Konto eine Verfügung ab einem bestimmten Datum ein. Der Code besteht abgesehen von der Konvertierung des Datums aus dem Konstruktoraufruf

```
public Verfuegung(Kunde kunde, Konto konto, Date von, Date bis) {
  ...
}
```

mit explizitem `von`-Datum und `null` als `bis`-Datum. Weiterhin werden im Code für `addKonto()` die collection-wertigen Properties der beiden `@OneToMany`-Beziehungen aktualisiert. Die schreibende Verwendung in der Anwendung erfolgt dann durch einfachen Aufruf der `addKonto()`-Methode und Persistieren des Rückgabewertes:

```
// 'kunde' und 'konto' bereits persistent, oder
em.persist(kunde);
em.persist(konto);
// dann persistieren der Verfuegung:
em.persist(kunde.addKonto("1.7.2011", konto));
```

Beim Lesen in der Anwendung können die Verfügungsdaten für einen Kunden ganz normal verwendet werden, da eine einfache 1:n-Beziehung existiert.

```
Kunde kunde = em.find(Kunde.class, ...);
Set<Verfuegung> verfuegungen = kunde.getVerfuegungen();
for (Verfuegung verfuegung : verfuegungen) {
  System.out.println(verfuegung.getVon());
  System.out.println(verfuegung.getBis());
  System.out.println(verfuegung.getKonto());
}
```

Wir haben bei diesem Beispiel den Primärschlüssel von Verfügungen auf den Kunden und das Konto dieser Verfügung beschränkt. Somit kann ein Kunde nur eine Verfügung für ein Konto haben, und eine historische Datenführung ist nicht möglich. Es kann z. B. nicht vermerkt werden, dass der Kunde letztes Jahr vom 1.1. bis zum 30.6. verfügungsberechtigt war, danach aber nicht mehr, und jetzt aktuell ab heute wieder verfügungsberechtigt sein soll. Hierzu muss das `von`- und das `bis`-Property in den Primärschlüssel aufgenommen werden, was aber problemlos möglich ist.

Projekt:

Das Beispiel zur Assoziationsklasse ist im Projekt *beziehungen-assoziationsklasse* enthalten.

4.2.2 Rekursive Beziehungen

Für die Bankanwendung sollen nun Angestellte inklusive der Mitarbeiter-Hierarchie modelliert werden. Für einen Vorgesetzten sollen alle direkten Mitarbeiter, für einen Mitarbeiter sein direkter Vorgesetzter bekannt sein. Um keine bestimmte Hierarchie vorwegzunehmen, sollen beliebige Höhen der Hierarchie möglich sein. Es ergibt sich eine rekursive Datenstruktur, die als UML-Klassendiagramm in Abbildung 4.12 dargestellt ist.

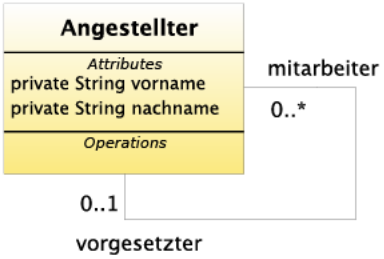


ABBILDUNG 4.12 Angestellter in rekursiver Beziehung

Ein Angestellter hat in der Regel genau einen Vorgesetzten. Hat ein Angestellter keinen Vorgesetzten, so ist er der Geschäftsführer, Vorstand oder Ähnliches – datentechnisch die Wurzel des Baumes. Ein Abteilungsleiter hat mehrere direkte Mitarbeiter. Ein einfacher Angestellter hat keine direkten Mitarbeiter, ist also datentechnisch ein Blatt des Baumes. Diese Beziehung ist eine einfache bidirektionale 1:n-Beziehung, wie sie in Abschnitt 4.1.2 detailliert wurde. Im Unterschied zu Abschnitt 4.1.2 ist jedoch die Quelle und das Ziel dieselbe Klasse, sonst ändert sich nichts. Um das Beispiel möglichst einfach zu halten, beschränken wir uns auf Vor- und Nachnamen der Angestellten. Zur Realisierung der Beziehung führen wir die Properties `mitarbeiter` und `vorgesetzter` ein, die mit `@OneToMany` und `@ManyToOne` zu annotieren sind.

```
@Entity
public class Angestellter {

  @Id @GeneratedValue
  private Integer id;
  private String vorname;
  private String nachname;

  @OneToMany(mappedBy = "vorgesetzter", cascade = CascadeType.ALL)
  private Set<Angestellter> mitarbeiter;

  @ManyToOne
  @JoinColumn(name = "vorgesetzter")
  private Angestellter vorgesetzter;
  ...
}
```

Wie in Abschnitt 4.1.2 erläutert, wird in der `@OneToMany`-Annotation das `mappedBy`-Attribut verwendet, die `@ManyToOne`-Annotation wird durch die `@JoinColumn`-Annotation ergänzt. Die Tabelle besteht aus den beiden fachlichen Spalten sowie Primär- und Fremdschlüssel-

spalte, wobei der Fremdschlüssel dieselbe Tabelle referenziert und den Vorgesetzten repräsentiert. Abbildung 4.13 zeigt die Tabelle für die Klasse `Angestellter`.

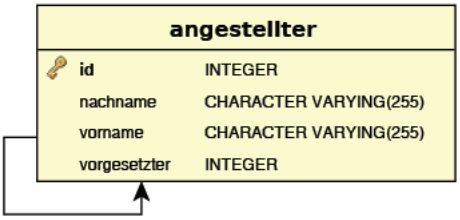


ABBILDUNG 4.13 Tabelle für die Klasse `Angestellter`

Der Aufbau der Hierarchie wird durch das Hinzufügen eines neuen Mitarbeiters zu einem Abteilungsleiter realisiert. Hierzu findet wieder das Schema der 1:n-Beziehung aus Abschnitt 4.1.2 Verwendung.

```
public void neuerMitarbeiter(Angestellter angestellter) {
  this.getMitarbeiter().add(angestellter);
  angestellter.setVorgesetzter(this);
}
```

Strukturrelevante Modifikationen an der Hierarchie, wie der Austausch eines Abteilungsleiters oder die Fusion zweier Abteilungen, sind wesentlich schwieriger zu programmieren. Der Grund liegt jedoch einzig und allein in der Datenstruktur Baum begründet und nicht in JPA. Wir verzichten auf die Entwicklung entsprechender Methoden.

Andere Methoden, wie die Prüfung, ob ein Knoten im Baum die Wurzel oder ein Blatt ist, sowie die Berechnung der Sohnknoten gestalten sich sehr einfach:

```
public boolean isChef() {
  return vorgesetzter == null;
}

public int anzahlMitarbeiter() {
  return mitarbeiter.size();
}

public boolean isAbteilungsleiter() {
  return anzahlMitarbeiter() != 0;
}
```

Projekt:

Das Beispiel für rekursive Beziehungen ist im Projekt *beziehungen-rekursiv* enthalten.

4.2.3 Geordnete Beziehungen

In den bisherigen Beispielen dieses Kapitels haben wir collection-wertige Beziehungen mit dem Typ `java.util.Set` modelliert. Dieses Interface stellt die mathematische Abstraktion

einer Menge dar und erlaubt daher keine Duplikate. Die Reihenfolge der Elemente eines Iterators (Methode `iterator()`) ist nicht definiert. Dies entspricht der mengentheoretischen Fundierung relationaler Datenbanken, der Relationenalgebra. In der Praxis sind Reihenfolgen und Ordnungen aber durchaus relevant und auch in relationalen Datenbanken möglich (Order By). Auf Java-Seite benötigt man hierfür andere Datentypen als den der Menge. JPA sieht neben `Set` auch die Typen `Collection`, `List` und `Map` im Package `java.util` zur Verwendung in collection-wertigen Beziehungen vor. Das Interface `Collection` ist das übergeordnete Interface von `Set` und `List`. Die Verwendung von `Collection` statt `Set` kann in allen bisherigen Beispielen des Buches erfolgen, da die Methoden von `Collection` und `Set` identisch sind. Unserer Meinung nach ist die Verwendung von `Set` intuitiver, da `Set` die mathematische Mengensemantik realisiert und der typischen Datenbanksemantik entspricht.

Wir wollen uns in diesem Abschnitt zunächst der Verwendung des Interfaces `List` widmen, um dann in Abschnitt 4.2.4 die `Map` zu verwenden. Als Beispiel zur Demonstration der Listenimplementierung einer Beziehung verwenden wir das bereits bekannte Beispiel von Konten mit ihren Buchungen. Die Entity-Klassen entsprechen den Klassen aus Abschnitt 4.1.2 für bidirektionale 1:n-Beziehungen und damit dem UML-Klassendiagramm in Abbildung 4.4 auf Seite 96. Wir geben den Code der beiden Klassen hier noch einmal wieder, verzichten aber auf nicht relevante Details.

```
@Entity
public class Konto {

  @Id @GeneratedValue
  private BigDecimal kontostand;

  @OneToMany(mappedBy = "konto")
  @OrderBy
  private List<Buchung> buchungen;
  ...
```

```
@Entity
public class Buchung {

  @Id @GeneratedValue
  private Integer id;
  private BigDecimal betrag;
  @Temporal(TemporalType.DATE)
  private Date datum;
  private String text;
  private Buchungsart buchungsart;

  @ManyToOne(optional = false)
  @JoinColumn(name = "konto",
              nullable = false)
  private Konto konto;
  ...
```

In der Klasse `Konto` wird die 1:n-Beziehung wie im Abschnitt 4.1.2 beschrieben definiert. Das Property `buchungen` ist aber vom Typ `List` und erhält zusätzlich eine `@OrderBy`-Annotation. Als Option wird das Property des Assoziationsziels angegeben, nach der die Liste sortiert werden soll. Wenn wie in diesem Fall kein Property angegeben ist, wird nach dem Primärschlüssel-Property sortiert. Soll nach den Buchungsbeträgen sortiert werden, so ist das Property `betrag` zu verwenden:

```
@OneToMany(mappedBy = "konto")
@OrderBy("betrag")
private List<Buchung> buchungen;
```

Die `@OrderBy`-Annotation erlaubt die Sortierung nach mehreren Properties, die durch Komma getrennt angegeben werden. Zusätzlich kann jedes Property den Zusatz `ASC` oder `DESC` für *ascending* (aufsteigend) und *descending* (absteigend) erhalten. Der Default ist `ASC`. Die folgende Verwendung von `@OrderBy` führt zu einer Reihenfolge mit aufsteigenden Bu-

chungsbeträgen. Sollte derselbe Buchungsbetrag mehrfach vorkommen, werden diese Buchungen anhand ihres Buchungsdatums in umgekehrter Reihenfolge gelesen.

```
@OneToMany(mappedBy = "konto")
@OrderBy("betrag ASC, datum DESC")
private List<Buchung> buchungen;
```

Der JPA-Provider setzt die `@OrderBy`-Annotation direkt in SQLs `ORDER BY` um. Für das obige Beispiel erzeugen alle drei Provider SQL-Code, der ähnlich zu folgender Where-Klausel ist:

```
... WHERE konto = <pk> ORDER BY betrag ASC, datum DESC
```

Da die `@OrderBy`-Annotation nur das Attribut `value` kennt, verzichten wir auf die tabellarische Darstellung dieses Annotationsattributs.

Die `@OrderBy`-Annotation ist datenbankunabhängig in dem Sinne, dass sie keine Entsprechung in Tabellenstrukturen hat. Das Vorhanden- oder Nichtvorhandensein der Annotation zieht keine Änderungen an Tabellen nach sich. Die Sortierung wird beim Lesen aus der Datenbank erstellt und danach von JPA nicht mehr überwacht. Insbesondere Änderungen an der Liste werden nicht mit der Datenbank synchronisiert.

Anders verhält es sich mit der in JPA 2.0 eingeführten Annotation `@OrderColumn`. Diese ist nicht nur für das sortierte Lesen aus der Datenbank zuständig, sondern muss auch die Reihenfolge der Liste *in* der Datenbank gewährleisten. Damit müssen ordnungsrelevante Operationen in der Liste (Vertauschen, Einfügen, Löschen von Elementen) am Ende der Transaktion mit der Datenbank synchronisiert werden, und es muss, wie der Name der Annotation andeutet, eine entsprechende Spalte in der Tabelle existieren.

Als Beispiel verwenden wir den Anwendungsfall, dass Nachrichten automatisiert an Kunden versendet werden können. Nachrichten werden zunächst in den Fachbereichen erstellt und dann Kunden zugewiesen. In einem Batch-Lauf werden dann alle Nachrichten für einen Kunden abgeschickt. Um Marketing-Maßnahmen aufeinander abstimmen zu können, sind die Nachrichten in einer bestimmten Reihenfolge zu versenden. Der folgende Code zeigt die relevanten Teile der beiden Entity-Klassen `Kunde` und `Nachricht`.

```
@Entity
public class Kunde {

  @Id @GeneratedValue
  private Integer id;
  private String vorname;
  private String nachname;
  ...
  @OneToMany(mappedBy="kunde", ...)
  @OrderColumn(name="reihenfolge")
  private List<Nachricht> nachrichten;
  ...
```

```
public class Nachricht {

  @Id @GeneratedValue
  private Integer id;
  private String text;

  @ManyToOne
  @JoinColumn(name="kunde",
              nullable=false)
  private Kunde kunde;
  ...
```

Über die beiden Properties `nachrichten` und `kunde` und die Annotationen `@OneToMany`, `@ManyToOne` und `@JoinColumn` wird eine bidirektionale 1:n-Beziehung definiert. Dies erfolgt exakt so, wie in Abschnitt 4.1.2 beschrieben. Lediglich die Verwendung des Typs `List` für das Property `nachrichten` sowie dessen Annotation mit `@OrderColumn` unterscheiden sich hiervon. Die Annotation `@OrderColumn` definiert eine Spalte, die zur Definition der Reihenfolge des collection-wertigen Property verwendet wird. Obwohl die Annotation

auf der objektwertigen Seite der Beziehung verwendet wird, definiert sie eine Spalte in der Tabelle der besitzenden Seite, hier `Nachricht`. Im Beispiel wird die Spalte über das Attribut `name` mit `reihenfolge` benannt. Ohne die Verwendung des Attributs besteht der Default-Name aus dem collection-wertigen Property mit angehängtem `_ORDER`, hier also `nachrichten_ORDER`. Das Attribut `name` sowie die weiteren Attribute der Annotation `@OrderColumn` zeigt Tabelle 4.8.

TABELLE 4.8 `@OrderColumn`-Attribute

`@OrderColumn(...)`			
Attribute	**Typ**	**Default**	**Beschreibung**
`column-Definition`	`String`	—	DDL-Anteil für Spalte
`insertable`	`boolean`	`true`	Spalte in Insert-Anweisung enthalten
`name`	`String`	Property + `_ORDER`	Name der Ordnungsspalte
`nullable`	`boolean`	`true`	NULL-Werte zugelassen
`updatable`	`boolean`	`true`	Spalte in Update-Anweisung enthalten

Die Abbildung 4.14 zeigt das zugrunde liegende Tabellenschema. Der Leser möge sich bewusst machen, dass die Spalte `reihenfolge` in der Tabelle `nachricht` keine Entsprechung im Entity `Nachricht` besitzt und damit insbesondere auch deren Existenz durch ausschließliche Betrachtung des Java-Codes für `Nachricht` nicht offensichtlich wird.

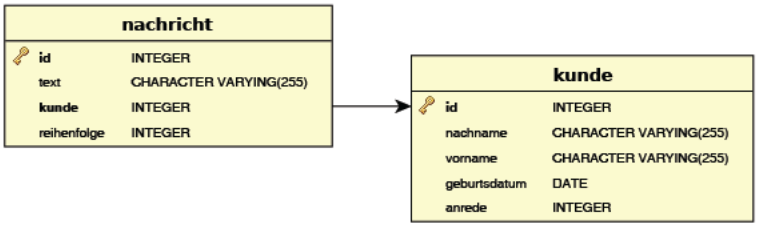


ABBILDUNG 4.14 Tabellenschema für 1:n-Beziehung mit Reihenfolge

Die Reihenfolge der Elemente wird von JPA automatisch verwaltet. Als Datentyp der Tabellenspalte ist ein ganzzahliger Typ zu verwenden. Bei der Generierung mit den Provider-Werkzeugen wird von allen drei Providern `Integer` verwendet. Die Elemente der Liste werden fortlaufend und mit 0 beginnend indiziert. Um eine neue Nachricht für einen Kunden zu registrieren, dient die Methode `neueNachricht()`.

```
public void neueNachricht(Nachricht nachricht) {
  nachricht.setKunde(this);
  nachrichten.add(nachricht); // hinten anhaengen
}
```

Da die Methode `List.add(Object)` ein Objekt an das Ende der Liste anhängt, definiert sich die Reihenfolge der Nachrichten durch die Reihenfolge der Methodenaufrufe. Das folgende Beispiel erzeugt drei Nachrichten, in derselben Reihenfolge wie im Programm-Code.

```
Kunde kunde = new Kunde(...);
kunde.neueNachricht(
        new Nachricht("Neuer Zinssatz: 2,5 % ab 1.7.2011"));
kunde.neueNachricht(
        new Nachricht("Sichern Sie sich attraktive ... "));
kunde.neueNachricht(
        new Nachricht("Für jeden geworbenen Neu-Kunden ..."));
em.persist(kunde);
```

Die Nachrichten erhalten als Wert der Spalte `reihenfolge` die Werte 0, 1 und 2, wenn die Persistenzeinheit mit der Datenbank synchronisiert wird. Das tatsächliche Verhalten der JPA-Provider ist jedoch sehr unterschiedlich. EclipseLink setzt drei Inserts ohne Beachtung der Spalte `reihenfolge`, gefolgt von drei Updates der Spalte `reihenfolge`, ab. OpenJPA verwendet drei Inserts inklusive der Spalte `reihenfolge`. Hibernate verwendet wie Eclipselink drei Inserts ohne Beachtung der Spalte `reihenfolge`, unterlässt dann aber den Update der Spalte, so dass diese jeweils `null` enthält. Hibernate verhält sich also nicht spezifikationskonform.

Achtung:

Hibernate setzt das Beispiel nicht korrekt um. Der Fehler ist im Hibernate-Jira bekannt:

`https://hibernate.onjira.com/browse/HHH-5390`

Für geordnete Beziehungen gelten dieselben Randbedingungen wie für gewöhnliche Beziehungen. In obigem Beispiel sind wir z. B. davon ausgegangen, dass für die 1:n-Beziehung mindestens die Persistenz kaskadiert wird, so dass das Persistieren des Kunden das Persistieren der drei Nachrichten nach sich zieht. Manipulationen an der Liste, also Einfügen, Löschen oder Ändern der Reihenfolge, werden von JPA automatisch mit der Datenbank synchronisiert. Wir möchten den Leser jedoch davor warnen, dies unbedacht zu tun. Beim Einfügen einer neuen Nachricht an den Anfang der Liste erzeugt EclipseLink n Update-Anweisungen für die bereits existierenden n Elemente, bei denen der Wert der Ordnungsspalte jeweils um 1 erhöht wird. Für das einzufügende Element werden eine Insert- und eine Update-Anweisung in der Datenbank ausgeführt, in Summe also $n+1$ Updates. Dies führt bei großen Listen zu einem erheblichen Aufwand.

Achtung:

Änderungen an einer `@OrderColums`-Liste führen zu Update-Anweisungen, deren Anzahl der Größenordnung der Liste entspricht. Die `@OrderColumn`-Annotation sollte daher nur für kleinere Listen Verwendung finden.

Projekt:

Die Beispiele für beide Arten von geordneten Beziehungen sind im Projekt *beziehungen-order* enthalten.

4.2.4 Beziehungen als Abbildungen

Der Zugriff auf ein einzelnes Entity einer collection-wertigen Beziehung erfolgt durch sequenzielles Durchsuchen der Menge oder Liste von Entities. Bei größeren Mengen oder Listen kann sich dies in der Laufzeit bemerkbar machen. Abhilfe schafft hier die Verwendung einer Map – mathematisch einer Abbildung – als Beziehungstyp. Man erhält damit eine Abbildung vom Schlüssel auf das Entity. Als Schlüssel der Map wird ein eindeutiger Identifikator des Entity verwendet, so dass prinzipiell das Primärschlüssel-Property oder Properties mit einem Unique-Constraint (Annotation `@Column(unique = true)`, siehe Tabelle 2.10 auf Seite 42) in Betracht kommen. Im bisherigen Konto-Buchung-Beispiel haben wir kein solches Property, so dass wir als Erweiterung des Beispiels die Anforderung einer eindeutigen Buchungsidentifikation auch außerhalb der Anwendung und der Datenbank aufnehmen. Dies kann z. B. durch eine UUID erfolgen. Der folgende Code gibt die Entities `Konto` und `Buchung` wieder, wobei die bekannte 1:n-Beziehung besteht. Beim Entity `Konto` wird zusätzlich die Annotation `@MapKey` verwendet, die als Wert des `name`-Attributs den Schlüssel der Map definiert. Der Typ des collection-wertigen Property `buchungen` ist `Map<String, Buchung>`.

```
@Entity
public class Konto {

  @Id @GeneratedValue
  private Integer kontonummer;
  private BigDecimal kontostand;

  @OneToMany(mappedBy = "konto")
  @MapKey(name = "buchungsschluessel")
  private Map<String, Buchung> buchungen;
  ...
}
```

Das Entity `Buchung` erhält das zusätzliche Property `buchungsschluessel` mit der Anforderung der Eindeutigkeit durch die `@Column`-Annotation mit `unique = true`.

```
@Entity
public class Buchung {

  @Id @GeneratedValue
  private Integer id;
  private BigDecimal betrag;
  @Temporal(TemporalType.DATE)
  private Date datum;
  private String text;
  private Buchungsart buchungsart;
  @Column(nullable = false, unique = true)
  private String buchungsschluessel;

  @ManyToOne(optional = false)
  @JoinColumn(name = "konto", nullable = false)
  private Konto konto;
  ...
}
```

Das `name`-Attribut ist das einzige Attribut der `@MapKey`-Annotation. Tabelle 4.9 auf der nächsten Seite zeigt dies der Vollständigkeit halber.

TABELLE 4.9 `@MapKey`-Attribute

`@MapKey(...)`			
Option	**Typ**	**Default**	**Beschreibung**
`name`	`String`	`""`	Name des Property des Beziehungs-Entities, das als Map-Schlüssel verwendet werden soll.

Die Methode `addBuchung()` der Klasse `Konto` wird derart überarbeitet, dass das Hinzufügen der Buchung über die Methode `put()` des Interface `Map` realisiert wird.

```
public void addBuchung(Buchung buchung) {
  buchung.setKonto(this);
  buchungen.put(buchung.getBuchungsschluessel(), buchung);
  if (buchung.getBuchungsart() == Buchungsart.HABEN) {
    kontostand = kontostand.add(buchung.getBetrag());
  } else if (buchung.getBuchungsart() == Buchungsart.SOLL) {
    kontostand = kontostand.subtract(buchung.getBetrag());
  } else {
    throw new RuntimeException("Buchungsart muss bekannt sein.");
  }
}
```

Beim lesenden Zugriff auf die Map muss der Schlüssel bekannt sein, wie im folgenden Beispiel.

```
String buchungsschluessel = ...
Konto konto = em.find(Konto.class, ...);
// hier Zugriff ueber Map-Schluessel:
Buchung buchung = konto.getBuchungen().get(buchungsschluessel);
```

Alternativ kann über alle `Map.Entry`-Instanzen iteriert werden, indem die Rückgabe der Methode `entrySet()` in der erweiterten For-Schleife verwendet wird.

```
Set<Entry<String,Buchung>> entrySet =
    konto.getBuchungen().entrySet();
for (Entry<String, Buchung> entry : entrySet) {
  ... entry.getKey() ...  entry.getValue() ...
}
```

Projekt:

Das Beispiel für Beziehungen als Abbildungen bzw. Maps ist im Projekt *beziehungen-map* enthalten.

4.3 Wertemengen

In Abschnitt 2.1 haben wir die Unterscheidung zwischen Werten (primitive Typen) und Objekten (Referenztypen) aus Sicht von Java diskutiert, um schließlich die Notwendigkeit einer Id für JPA-Objekte zu motivieren. Aus Anwendungssicht können aber auch komplexere Ob-

jekte eine Wertesemantik haben. Dies äußert sich dadurch, dass keine explizite Möglichkeit der Identifikation des Objekts besteht, sowie der Bindung der Existenz und Lebensdauer an ein anderes Objekt.

JPA 1.0 konnte nur in sehr begrenztem Umfang mit Mengen solcher Werte umgehen. JPA 2.0 führt eine Reihe von Erweiterungen ein, die die Verwaltung von Wertemengen analog zu 1:n-Beziehungen ermöglichen. In Abschnitt 4.3.1 erläutern wir zunächst die Möglichkeiten von JPA 1.0, in den weiteren Abschnitten die Neuerungen von JPA 2.0.

4.3.1 Wertemengen ohne JPA

Die Serialisierbarkeit einer Klasse ist nach den im Abschnitt 2.2.1 definierten Anforderungen ausreichend, um ein Property vom Typ dieser Klasse innerhalb einer Entity-Definition verwenden zu können. Weitere Mapping-Optionen werden von JPA nicht benötigt. Man erhält damit eine sehr einfache Möglichkeit, eine Komposition, also eine Aggregation, die die Lebensdauer der Teile an die Existenz des Ganzen bindet. Als Beispiel dient uns ein Kunde, dem mehrere Adressen zugeordnet sind, etwa Erst- und Zweitwohnsitz sowie alternative Postanschriften. Wir verwenden eine Klasse `Adresse`, wie wir sie bereits mehrfach verwendet haben.

```
public class Adresse implements Serializable {

  private String strasse;
  private String hausnummer;
  private String plz;
  private String ort;
  ...

}
```

Weil die Klasse das Interface `Serializable` implementiert, kann sie in Entities wie etwa der Klasse `Kunde` verwendet werden.

```
@Entity
public class Kunde {

  @Id @GeneratedValue
  private Integer id;
  private String vorname;
  private String nachname;
  @Temporal(TemporalType.DATE)
  private Date geburtsdatum;
  private Anrede anrede;
  private Set<Adresse> adressen;

  public Kunde() {
    adressen = new HashSet<Adresse>();
  }
  ...
}
```

Dabei ist es unerheblich, ob nur eine Adresse oder eine Menge von Adressen verwendet wird, da die `Set`-Implementierungen des JDK, sowie die von den JPA-Providern verwendeten `Set`-Implementierungen ebenfalls das Interface `Serializable` implementieren.

Die entsprechende Tabelle enthält neben den Spalten für den Primärschlüssel und die anderen Properties eine Spalte `adressen`. Diese Spalte hat einen datenbankabhängigen binären Datentyp (siehe Tabelle 2.8 auf Seite 40) und enthält jeweils die serialisierte Menge von Adressen eines Kunden. Diese bestechend einfache Lösung hat zwei gravierende Nachteile. Zum einen können die Adressen nicht in der Datenbank direkt mit SQL oder mit einer anderen Programmiersprache, etwa COBOL, bearbeitet werden. Zum anderen wird beim Serialisieren und Deserialisieren eines Objekts eine Versionsnummer, die sogenannte `serialVersionUID` verwendet. Diese kann explizit durch eine Long-Konstante definiert werden oder wird implizit durch das Serialisierungslaufzeitsystem berechnet. Erfolgt nun eine Änderung an der Klasse, so wird mit unterschiedlichen Versionsnummern serialisiert und deserialisiert, was zu einer `InvalidClassException` führt. Weil Datenbanken in der Regel eine Lebensdauer von vielen Jahren haben, ist eine derartige Änderung im Laufe der Zeit nicht unwahrscheinlich. Eine Verwendung dieser Alternative von Wertemengen ist also nur zu empfehlen, wenn sichergestellt ist, dass sich die `serialVersionUID` nicht ändert.

Tipp:

Bei der Diskussion der Serialisierbarkeit von Entities in Abschnitt 2.2.3 haben wir bereits allgemein zu serialisierbaren Entities geraten. Moderne Application-Server erlauben das Hot-Deployment von Anwendungen inklusive der Wiederherstellung von Sessions. Wegen der oben genannten Gründe sollte das `serialVersionUID`-Property auch hier explizit gesetzt werden. Lediglich aus Gründen einer übersichtlicheren Darstellung stellen die Code-Listings dies nicht dar.

Achtung:

Hibernate kommt mit dem verwendeten Datentyp `Set<Adresse>` nicht zurecht und wirft eine `MappingException`, ein offensichtlicher Bug. EclipseLink und OpenJPA funktionieren problemlos.

4.3.2 Einfache Datentypen

Mit der in JPA 2.0 eingeführten Annotation `@ElementCollection` können Basistypen und einbettbare Datentypen als Grundtypen für Collections dienen. Analog zu den Standardbeziehungen können die Typen `Collection`, `Set`, `List` und `Map` verwendet werden, um eine Menge von Werten zu realisieren. Die für Standardbeziehungen bereits diskutierten Eigenschaften der vier Datentypen gelten auch für hier. In Anlehnung an die Schreibweise der Annotation wird im JPA-Jargon auch von *Element Collections* gesprochen.

Als Anwendungsbeispiel definieren wir für ein Konto eine Menge von TANs (Transaktionsnummern), die wir später in eine Liste von TANs überführen. TANs sind einfache Werte und sollen in unserem Beispiel Strings der Länge 6 sein. Das die TANs repräsentierende Property wird dazu mit `@ElementCollection` annotiert, wie der folgende Code-Ausschnitt zeigt.

```
@Entity
public class Konto {

  @Id @GeneratedValue
  private Integer kontonummer;
  @Column(precision = 10, scale = 2)
  private BigDecimal kontostand;

  @ElementCollection
  private Set<String> tans;
  ...
}
```

Die Attribute der Annotation `@ElementCollection` sind in Tabelle 4.10 zusammengefasst. Das Attribut `fetch` entspricht dem identischen Attribut der Beziehungsannotationen `@OneToOne`, `@OneToMany`, `@ManyToOne` und `@ManyToMany`. Auch hier ist analog zu den collection-wertigen Beziehungsannotationen `LAZY` der Default. Ebenfalls übernommen ist die Verpflichtung des JPA-Providers zur Realisierung der `EAGER`-Strategie während `LAZY` einen Hint darstellt. Das Attribut `targetClass` entspricht dem Attribut `targetEntity` der Beziehungsannotationen und ist bei nicht generischen Collection-Typen zu verwenden.

TABELLE 4.10 `@ElementCollection`-Attribute

`@ElementCollection(...)`			
Option	**Typ**	**Default**	**Beschreibung**
`fetch`	`FetchType`	`LAZY`	Ladezeitpunkt der Assoziation (`EAGER`, `LAZY`)
`targetClass`	`Class`	`void.class`	Basistyp oder einbettbare Klasse der Collection

Das Tabellenschema zum obigen Konto-Beispiel entspricht strukturell dem einer 1:n-Beziehung. Die collection-wertige Seite enthält einen Fremdschlüssel zur objektwertigen Seite. JPA sieht als Tabellenname die Verknüpfung von Entity-Namen und Collection-Property, getrennt durch Unterstrich, vor. Die Fremdschlüsselspalte wird durch die Verknüpfung des Entity-Namens mit dem Primärschlüsselnamen, ebenfalls durch Unterstrich getrennt, gebildet. Abbildung 4.15 stellt dieses Schema dar.

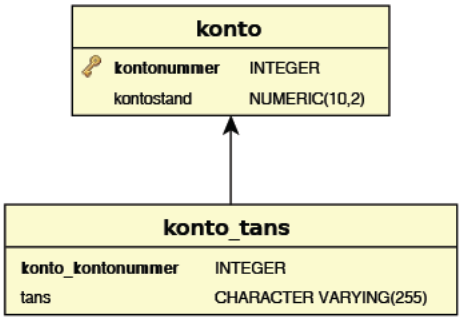


ABBILDUNG 4.15 Konto- und TAN-Tabelle

Achtung:

Die Werkzeuge zur Schema-Generierung von EclipseLink und OpenJPA erzeugen keinen Fremdschlüssel-Constraint, was aber in der weiteren Verwendung nicht problematisch ist. OpenJPA geht vom Defaultnamen `element` für die Wertespalte aus, was ebenfalls nicht spezifikationskonform ist.

Die genannten Regeln zur Benennung von Tabellen und Spalten basieren auf den Defaults der Annotation `@CollectionTable`, die in JPA 2.0 zur Spezifikation von Wertetabellen eingeführt wurde. Wir verwenden nun diese und andere Annotationen, um das Beispiel zu optimieren. Die Attribute der Annotation sind in Tabelle 4.11 aufgeführt. Sie sind mit Ausnahme des Attributs für die Join-Column der `@Table`-Annotation (siehe Tabelle 2.4 auf Seite 35) nachempfunden.

TABELLE 4.11 `@CollectionTable`-Attribute

`@CollectionTable(...)`			
Attribut	**Typ**	**Default**	**Beschreibung**
`catalog`	`String`	Default-Katalog	Katalogname
`joinColumns`	`JoinColumn[]`	—	Fremdschlüsselspalten
`name`	`String`	Entity-Name + Unterstrich + Property-Name	Tabellenname
`schema`	`String`	Default-Schema	Schema-Name
`uniqueConstraint`	`UniqueConstraint[]`	—	Spaltenname(n) für Unique-Constraint

Zunächst soll der Collection-Typ in `List` geändert werden, da die TANs in einer bestimmten Reihenfolge verwendet werden sollen. Die einfachste Möglichkeit zur Definition einer Reihenfolge ist die Annotation `@OrderBy`, die wir bereits in Abschnitt 4.2.3 kennengelernt haben. Als Tabellenname soll `tans`, als Fremdschlüsselspalte bzw. Join-Column `kontoid` verwendet werden. Dies wird mit der `@CollectionTable`-Annotation realisiert. Die Spalte für die TANs soll im Singular benannt werden, und eine einzelne TAN soll sechs Zeichen umfassen. Hierzu verwendet man die aus Kapitel 2 bekannte Annotation `@Column`. Die übererarbeitete Version der Wertemenge liest sich nun wie folgt.

```
@ElementCollection
@OrderBy
@Column(name = "tan", length = 6)
@CollectionTable(name = "tans",
                 joinColumns = @JoinColumn(name="kontoid"))
private List<String> tans;
```

Achtung:

Die Reihenfolge/Sortierung der TANS mit `@OrderBy` wird nur von Hibernate und OpenJPA realisiert. EclipseLink ignoriert die Annotation.

Die Reihenfolge der TANs definiert sich bei der Verwendung von `@OrderBy` über die natürliche Sortierung, also z. B. bei Strings alphabetisch, bei Integer numerisch. Für TANs ist dies nicht sehr sinnvoll. Abhilfe schafft hier eine explizite, persistente Reihenfolge, die über die Annotation `@OrderColumn` realisiert werden kann. Da wir die Annotation bereits in Abschnitt 4.2.3 ausführlich diskutiert haben, stellen wir hier nur die Lösung vor:

```
@ElementCollection
@OrderColumn(name = "reihenfolge")
@Column(name = "tan", length = 6)
@CollectionTable(name = "tans",
                 joinColumns = @JoinColumn(name="kontoid"))
private List<String> tans;
```

Die Tabelle `tans` besitzt nun eine zusätzliche Spalte `reihenfolge`, die eine der Reihenfolge entsprechende Ordinalzahl, beginnend bei 0, enthält.

4.3.3 Einbettbare Datentypen

Die Annotation `@ElementCollection` erlaubt nicht nur Wertemengen von Basistypen, sondern auch von sogenannten einbettbaren Typen. Ein einbettbarer Typ ist selbst kein Entity und wird mit `@Embeddable` annotiert. In Abschnitt 2.3.3 wurde mit dieser Annotation ein zusammengesetzter Schlüssel realisiert. Wir haben dort auch die entsprechende Definition der Annotation aus der JPA-Spezifikation zitiert, die im API-Doc der Annotation wiederholt wird: „...whose instances are stored as an intrinsic part of an owning entity ...“. Dies ist bei der Verwendung von `@ElementCollection` nicht korrekt. In diesem Fall wird analog zu Basistypen eine Tabelle für die mit `@Embeddable` annotierte Klasse verwendet, die einen Fremdschlüssel in die Entity-Tabelle besitzt, um so die Zugehörigkeit der einbettbaren Daten zu einem Entity auszudrücken.

Zur Verwendung innerhalb eines Beispiels definieren wir die Anforderung, dass die Adressen eines Kunden als entsprechende Wertemenge modelliert werden sollen. Um vom Default der Tabellen- und Spaltennamen abzuweichen, verwenden wir die bereits bekannte Annotation `@CollectionTable`.

```
@Entity
public class Kunde {

  @Id @GeneratedValue
  private Integer id;
  private String vorname;
  private String nachname;
  @Temporal(TemporalType.DATE)
  private Date geburtsdatum;
  private Anrede anrede;

  @ElementCollection
  @CollectionTable(
    name = "kundenadressen",
    joinColumns =
      @JoinColumn(name = "kundenid"))
  private Set<Adresse> adressen;
  ...
}
```

```
@Embeddable
public class Adresse {

  private String strasse;
  private String hausnummer;
  private String plz;
  private String ort;
  ...
}
```

Das zugrunde liegende Tabellenschema ist identisch zur entsprechenden Modellierung mit `@OneToOne` oder `@OneToMany` und in Abbildung 4.16 dargestellt. Der einzige Unterschied ist der fehlende Primärschlüssel in der Tabelle `Kundenadressen`. EclipseLink und OpenJPA erzeugen auch hier bei der Tabellengenerierung keinen Fremdschlüssel-Constraint.

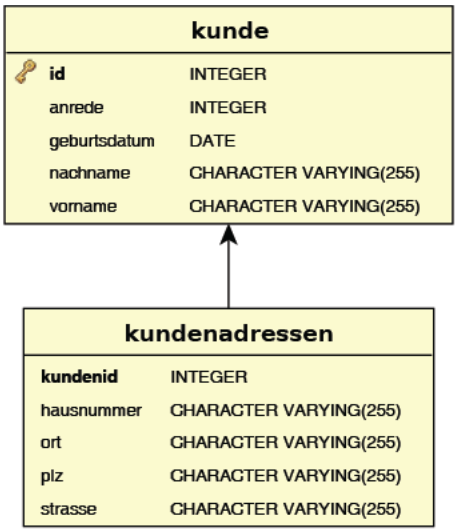


ABBILDUNG 4.16 Kundenadressen

Die bei den Basistypen verwendeten Annotationen `@OrderBy` und `@OrderColumn` für Listen können auch bei einbettbaren Datentypen unverändert eingesetzt werden, so dass wir auf entsprechende Beispiele verzichten.

Die letzte Erweiterung von JPA 2.0 im Hinblick auf Wertemengen ist die Möglichkeit der Verwendung von Wertemengen über Maps. Auf einzelne Werte wird also über einen Schlüssel zugegriffen. Die Deklaration erfolgt ebenfalls mit der Annotation `@ElementCollection`, als Typ wird das Interface `Map` verwendet, wobei der Wertetyp der Map ein einbettbares Objekt sein muss. Als Anwendungsbeispiel sollen für einen Kunden verschiedene Arten von telefonischen Kontakten möglich sein, hier speziell eine Telefonnummer in der Firma und privat sowie eine Mobiltelefonnummer. Diese drei Arten können über eine Enumeration definiert werden. Das Telefon (die Verbindungsalternative) ist als einbettbares Objekt mit der Telefonart und der eigentlichen Telefonnummer realisiert.

```
@Entity
public class Kunde {

  @Id @GeneratedValue
  private Integer id;
  private String vorname;
  private String nachname;
  @Temporal(TemporalType.DATE)
  private Date geburtsdatum;
  private Anrede anrede;

  @ElementCollection
  private Map<Telefonart,
              Telefon> telefone;
  ...
}
```

```
public enum Telefonart {
  FIRMA, PRIVAT, MOBIL;
}


@Embeddable
public class Telefon {

  public Telefonart art;
  public String nummer;

  ...
}
```

Weil das Map-Property nur mit `@ElementCollection` annotiert ist, greifen die Default-Namensregeln, die wir bereits erläuterten. Das entsprechende Tabellenschema ist in Abbildung 4.17 dargestellt.

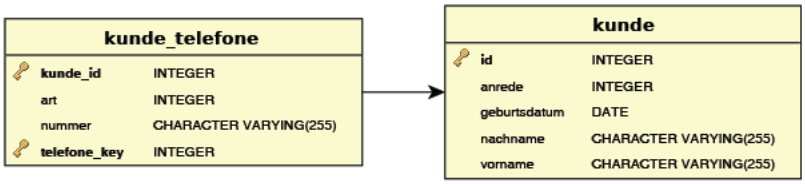


ABBILDUNG 4.17 Kundentelefone mit einer `Map`

Weil die Klasse Telefon explizit die Telefonart enthalten sollte, sind die Spalten `art` und `telefone_key` in der Tabelle `kunde_telefone` redundant, was hier aber nicht weiter stören soll.

Die bereits bekannte Annotation `@CollectionTable` kann verwendet werden, um von den Default-Namen abzuweichen. Weitere Möglichkeiten für explizite Mapping-Optionen sind mit Map-spezifischen Annotationen realisierbar. Für Properties der Typen `Date` und `Calendar` muss die Annotation `@MapKeyTemporal` verwendet werden. Die Annotation entspricht der `@Temporal`-Annotation für einfache Properties (siehe Tabelle 2.7), so dass wir auf eine Darstellung verzichten. Für Enum-Properties kann die Annotation `@MapKeyEnumerated` verwendet werden. Auch diese Annotation ist der entsprechenden Annotation für einfache Properties nachempfunden, der `@Enumerated`-Annotation (siehe Tabelle 2.9). Der Default ist das Mapping auf eine Integer-Spalte, wie die Abbildung 4.17 verdeutlicht. Wir wollen nun alternativ auf einen String abbilden und weitere Änderungen vornehmen:

```
@ElementCollection
@CollectionTable(name = "kundentelefone",
                 joinColumns = @JoinColumn(name = "kundenid"))
@MapKeyEnumerated(EnumType.STRING)
@MapKeyColumn(name = "MAP_KEY")
private Map<Telefonart, Telefon> telefone;
```

Das überarbeitete Tabellenschema ist in Abbildung 4.18 dargestellt.

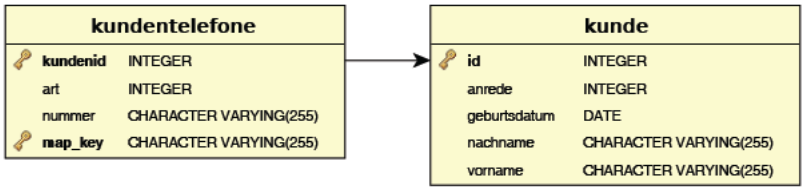


ABBILDUNG 4.18 Überarbeitete Version der Kundentelefone

Auch die explizite Benennung der Tabellenspalten ist möglich. Hier werden jedoch keine neuen Annotationen speziell für Maps benötigt, sondern es können die Annotationen `@AttributeOverrides` und `@AttributeOverride` verwendet werden, deren Attribute in den Tabellen 2.12 und 2.13 auf Seite 49 beschrieben sind. Als Beispiel sollen die beiden Spalten für die Klasse `Telefon` das Präfix `TEL_` bekommen.

```
@AttributeOverrides(
  {@AttributeOverride(name = "nummer",
                      column = @Column(name = "TEL_NR")),
   @AttributeOverride(name = "art",
                      column = @Column(name = "TEL_ART"))})
private Map<Telefonart, Telefon> telefone;
```

Als Schlüssel für Maps sind nicht nur einfache Datentypen oder Enumerations, sondern auch Entity-Typen erlaubt. Für die Entities `Kunde` und `Konto` kann mit der folgenden Verwendung von `@ElementCollection` für jeden Kunden und jedes Konto ein separater Zinssatz definiert werden.

```
@Entity
public class Kunde {

  @ElementCollection
  @Column(name = "zinssatz", length = 4, scale = 2)
  private Map<Konto, BigDecimal> zinssaetze;
  ...
}
```

Dieser Anwendungsfall ist für die Praxis sicher nicht relevant, da z. B. Ehepartner für ihr gemeinsames Konto nicht unterschiedliche Zinssätze bekommen würden. Das Property für den Zinssatz sollte daher im Konto-Entity definiert werden. Ein sinnvoller Anwendungsfall ist jedoch das Beispiel der Assoziationsklasse aus Abschnitt 4.2.1. Hier sollte ein Kunde für ein Konto für eine bestimmte Zeit ein Verfügungsrecht eingeräumt bekommen. Bei der Modellierung über eine Map wird als Schlüssel das Konto verwendet.

```
@Entity
public class Kunde {

  @Id @GeneratedValue
  private Integer id;
  ...

  @ElementCollection
  @CollectionTable(name = "verfuegung",
                   joinColumns = @JoinColumn(name = "kunde"))
```

```
  @MapKeyJoinColumn(name = "konto")
  private Map<Konto, VonBis> verfuegungen;
  ...
}
```

Die Klasse `VonBis` wird als einbettbarer Wert, nicht als Entity modelliert.

```
@Embeddable
public class VonBis {

  @Temporal(TemporalType.DATE)
  private Date von;
  @Temporal(TemporalType.DATE)
  private Date bis;
  ...
}
```

Das Tabellenschema für das Beispiel ist in Abbildung 4.19 dargestellt. Es ist identisch zur Realisierung über eine Assoziationsklasse mit drei Entities (siehe Abbildung 4.11 auf Seite 110). Das unterschiedliche Layout ist dem verwendeten Werkzeug zur Visualisierung geschuldet.

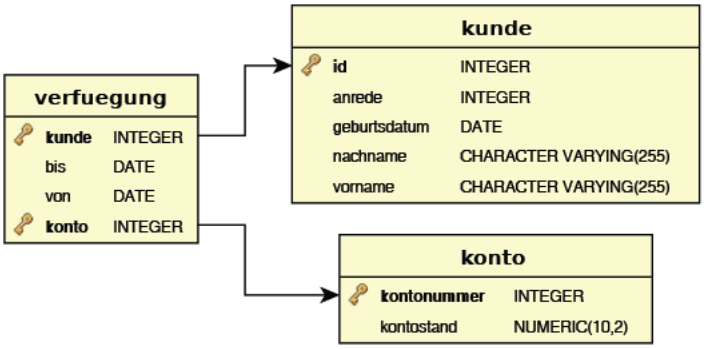


ABBILDUNG 4.19 Tabellenschema für Wertemenge von Verfügungsdaten

Um die identische Struktur zu erhalten, haben wir die bereits bekannte `@CollectionTable`-Annotation verwendet, um die Tabelle selbst und den Fremdschlüssel in die Tabelle `kunde` umzubenennen. Zur Umbenennung der Fremdschlüsselspalte in die Tabelle `konto` kann `@AttributeOverride` nicht verwendet werden, weil damit nur einfache Properties, nicht aber Fremdschlüssel umbenannt werden können. Abhilfe schafft hier die Annotation `@MapKeyJoinColumn`, die auch in der Pluralform `@MapKeyJoinColumns` existiert. Die `@MapKeyJoinColumn`-Attribute sind in Tabelle 4.12 aufgezählt.

Weil die Klasse `Verfügung` kein Entity ist, können wir keinen Vorteil aus JPAs transitiver Persistenz ziehen. Methodenaufrufe des Entity-Managers müssen sowohl auf Kunde- als auch auf Konto-Objekten erfolgen. Durch das etwas komplexere `Map`-API werden auch einige Methoden im Vergleich zum Abschnitt 4.2.1 etwas aufwendiger, wie etwa die Methode `Kunde.getKonten()`:

```
public Set<Konto> getKonten() {
  return verfuegungen.keySet();
}
```

TABELLE 4.12 `@MapKeyJoinColumn`-Attribute

`@MapKeyJoinColumn(...)`			
Attribut	**Typ**	**Default**	**Beschreibung**
`columnDefinition`	`String`	—	Zusätzliches SQL-Fragment zur Definition der Spalte
`insertable`	`boolean`	`true`	Angabe, ob die Spalte in Insert-Anweisungen verwendet werden soll
`name`	`String`	—	Der Name der Fremdschlüsselspalte des Map-Schlüssels
`nullable`	`boolean`	`true`	Angabe, ob die Fremdschlüsselspalte Nullwerte zulässt
`referencedColumnName`	`String`	—	Name der Spalte, die durch diese Fremdschlüsselspalte referenziert wird
`table`	`String`	—	Name der Tabelle (nur sinnvoll bei Sekundärtabellen)
`unique`	`boolean`	`false`	Angabe, ob das Property ein eindeutiger Schlüssel ist
`updatable`	`boolean`	`true`	Angabe, ob die Spalte in Update-Anweisungen verwendet werden soll

Wir haben nun praktisch alle mit JPA realisierbaren Arten von Beziehungen exemplarisch wiedergegeben. Trotzdem bleiben einige Lücken, da die Möglichkeiten von JPA sehr umfangreich sind. Wir fordern den Leser auf, die im API-Doc der Annotationen enthaltenen Beispiele zu studieren und auszuprobieren.

Projekt:

Das Projekt *beziehungen-element-collection* enthält die Beispiele zu Wertemengen, also der Abschnitte 4.3.1, 4.3.2 und 4.3.3.

4.4 Beziehungen und Performanz

Das Java inhärent zugrunde liegende Objektmodell, bei dem Objekte über Referenzen zu einem komplexen Objektgeflecht verbunden werden, ist Java-Entwicklern vertraut und wird ganz intuitiv benutzt. Es gestattet eine sehr effiziente Navigation zwischen Objekten durch den Aufruf von Gettern, einer Form des Dereferenzierens. Mit der Einführung von JPA ändert sich dies. Die Navigation kann zwar in großen Teilen konzeptionell unverändert übernommen werden, die hohe Performanz der Navigation geht jedoch verloren. In der Regel ist das Dereferenzieren einer JPA-Beziehung über den entsprechenden Getter mit einem Datenbankaufruf verbunden und damit um Größenordnungen langsamer als ein JVM-internes Dereferenzieren. Bei der Verwendung von JPA sollte dies berücksichtigt werden, um nicht in Per-

formanzfallen zu tappen. Eine dieser Performanzfallen, das sogenannte *n+1-Select-Problem*, sowie eine Möglichkeit, diese Falle zu umgehen, sind der Gegenstand dieses Abschnitts.

Zur Vermeidung des Problems kommt die *Java Persistence Query Language* zum Einsatz, die wir in Kapitel 7 einführen. Bei Verständnisproblemen sollte der Leser zunächst Kapitel 7 durcharbeiten und dann wieder hierher zurückkehren.

Als Struktur des Anwendungsbeispiels sind einem Kunden über eine 1:n-Beziehung mehrere Konten zugeordnet. Auf diesen Konten finden Buchungen statt, die ebenfalls über eine 1:n-Beziehung modelliert werden. Als fachliche Aufgabe soll für jedes Konto die Buchung mit dem höchsten Betrag bestimmt werden. Der folgende Code zeigt einen Programmausschnitt, der dies realisiert.

```
1   ...
2   Kunde kunde = em.find(Kunde.class, ...);
3   List<Buchung> maximalbuchungen = new ArrayList<Buchung>();
4   for (Konto konto : kunde.getKonten()) {
5     Buchung maximalbuchung = null;
6     for (Buchung buchung : konto.getBuchungen()) {
7       if (maximalbuchung == null) {
8         maximalbuchung = buchung;
9       } else if (buchung.getBetrag()
10                  .compareTo(maximalbuchung.getBetrag()) > 0) {
11        maximalbuchung = buchung;
12      }
13    }
14    maximalbuchungen.add(maximalbuchung);
15  }
16  ...
```

Die Ausführung der Methode `getKonten()` in Zeile 4 löst das Proxy für die Konten auf und setzt eine Select-Anweisung an die Datenbank ab. Wenn der Kunde n Konten besitzt, so werden in der Zeile 6 zur Auflösung der Buchungs-Proxies n weitere Select-Anweisungen an die Datenbank geschickt, insgesamt also n+1 Select-Anweisungen. Dieses Muster ist als *n+1-Select-Problem*, manchmal auch nur *n+1-Problem* genannt, in der Literatur bekannt und wurde dort bereits ausreichend diskutiert.

Achtung:

Das n+1-Select-Problem sollte in der Regel *immer* vermieden werden, um keine potenziellen Performanzprobleme hervorzurufen. Die Vermeidung erfolgt durch eine explizit formulierte JPQL-Anfrage.

Eine Lösung des Problems wäre, die Assoziation nicht spät, sondern früh zu laden, d. h. als `FetchType` nicht `LAZY`, den Default, sondern `EAGER` beim Mapping der Assoziation zu verwenden:

```
@OneToMany(mappedBy="konto", fetch=FetchType.EAGER, ...)
private Set<Buchung> buchungen;
```

In unserem Fall würde dies jedoch nicht zum Ziel führen, da die `EAGER`-Definition nur beim expliziten Laden eines Kontos mit `find()` die Buchungsassoziation mit frühem Laden realisiert. Da wir ein Konto über den Aufruf von `getKonten()` laden, wird die Assoziation *nicht* in einem `Select` geladen.

Falls in Ihrem Anwendungsfall das frühe Laden möglich ist, sollten Sie dessen Einsatz aber trotzdem gut abwägen. Frühes Laden birgt in der Regel auch ein Performanzrisiko, da Sie *global* das frühe Laden einstellen, es eventuell aber nur in sehr seltenen Fällen benötigen.

In unserem Anwendungsfall ist es sinnvoll zu versuchen, das n+1-Select-Problem *lokal* zu vermeiden. Hierzu wird eine JPQL-Anfrage mit einem *Outer Join* verwendet. Der folgende Ausschnitt zeigt die überarbeitete Fassung.

```
1   ...
2   Kunde kunde = em.find(Kunde.class, ...);
3   List<Buchung> maximalbuchungen = new ArrayList<Buchung>();
4   List<Konto> konten = em.createQuery(
5       "select distinct k from Konto k
6        left join fetch k.buchungen where k.kunde = :kunde",
7       Konto.class).setParameter("kunde", kunde).getResultList();
8   for (Konto konto : konten) {
9     Buchung maximalbuchung = null;
10    for (Buchung buchung : konto.getBuchungen()) {
11      if (maximalbuchung == null) {
12        maximalbuchung = buchung;
13      } else if (buchung.getBetrag().compareTo(maximalbuchung.
          getBetrag()) > 0) {
14        maximalbuchung = buchung;
15      }
16    }
17    maximalbuchungen.add(maximalbuchung);
18  }
```

Statt in der äußeren For-Schleife über den Getter `getKonten()` auf die Konten zuzugreifen, wird eine JPQL-Abfrage formuliert, über deren Ergebnis in der Variablen `konten` iteriert wird. Weil die JPQL-Abfrage über den Outer Join auch die Buchungen für die Konten des Kunden lädt, müssen diese nicht in einem weiteren `Select` explizit geladen werden. Die weitere Code-Struktur wurde nicht verändert. Nach der Überarbeitung setzt ein JPA-Provider nur noch zwei statt n+1 Select-Anweisungen ab. Weil die Minimierung der Datenbankzugriffe aus Performanzgesichtspunkten in der Regel immer anzustreben ist, ist dieser Alternative der Vorzug zu geben.

Eine weitere Möglichkeit wäre, den Kunden ebenfalls in die JPQL-Anfrage aufzunehmen und einen Join über drei Tabellen zu realisieren. Damit würde nur noch eine Select-Anweisung in Richtung Datenbank abgeschickt.

Projekt:

Das Projekt *n-plus-eins-selects* enthält den Beispiel-Code. Die beiden Alternativen sind als jeweils eigene Klasse mit Main-Methode realisiert.

5 Vererbung

In Kapitel 4 haben wir uns mit Beziehungen zwischen Objekten beschäftigt und als Beispiel unter anderem Kunden mit ihren Konten und Buchungen auf diesen Konten betrachtet. Konten existieren aber in verschiedenen Ausprägungen: Sparkonten, Girokonten, Festgeldkonten und andere. In objektorientierten Sprachen werden derartige Ausprägungen in der Regel mit Hilfe von Vererbung modelliert. Dieses Kapitel beschreibt die verschiedenen Möglichkeiten, die JPA zur Realisierung von Vererbungsbeziehungen bereitstellt, und schließt damit die Darstellung des Mappings objektorientierter Konzepte mit JPA ab.

Wie bereits in Kapitel 1 motiviert, ist die Abbildung einer objektorientierten Vererbungshierarchie in relationale Datenbanken eines der zentralen Probleme von OR-Mappern. Heute besteht Konsens darüber, dass prinzipiell drei alternative Realisierungsmöglichkeiten existieren, die Gegenstand dieses Kapitels sind:

- eine einzige Tabelle für eine gesamte Vererbungshierarchie
- eine Tabelle je Unterklasse
- eine Tabelle je konkreter Klasse

Das oben angesprochene Beispiel verschiedener Kontoarten soll nun konkret werden. Wir verwenden eine abstrakte Oberklasse `Konto` sowie die beiden Unterklassen `Sparkonto` und `Girokonto`. Die in Kapitel 4 verwendeten Beziehungen zu Kunden und Buchungen sollen ebenfalls realisiert werden. Die Abbildung 5.1 zeigt das entsprechende UML-Klassenmodell.

5.1 Eine Tabelle für eine Vererbungshierarchie

Wir beginnen mit der einfachsten Alternative, die alle Klassen einer Vererbungshierarchie in eine einzige Tabelle integriert. Abbildung 5.2 zeigt schematisch die entstehende Struktur inklusive zweier Datensätze, wenn die Klassen `Konto`, `Sparkonto` und `Girokonto` in einer Tabelle zusammengefasst werden. Man erkennt von links nach rechts zuerst die sogenannte *Diskriminatorspalte* `disc`. Der Wert dieser Spalte wird verwendet, um den Entity-Typ der Daten dieser Tabellenzeile zu bestimmen. In unserem Beispiel ist die Klasse `Konto` abstrakt, so dass nur zwischen `Sparkonto` und `Girokonto` unterschieden werden muss. Dies kann

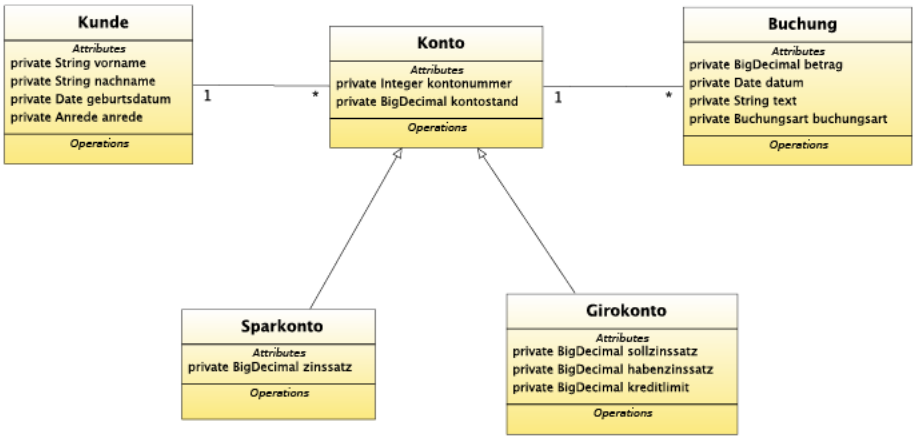


ABBILDUNG 5.1 Vererbungshierarchie der Konten

z. B. durch einen Integer-Wert oder, wie im Beispiel, durch die Strings `"Spar"` und `"Giro"` geschehen. Die nächsten Spalten, `kontonummer` und `kontostand`, sind Properties der Klasse `Konto` und damit in jeder Instanz von `Sparkonto` und `Girokonto` enthalten. Es folgt die Spalte `zinssatz`, das einzige Property der Klasse `Sparkonto`. Die letzten drei Spalten repräsentieren die Properties der Klasse `Girokonto`. In der Darstellung haben wir die Spalte `kunde` unterschlagen, da sie nichts mit Vererbung zu tun hat. Sie fungiert als Fremdschlüssel in die Kundentabelle.

disc	konto-nummer	konto-stand	zinssatz	soll-zinssatz	haben-zinssatz	kreditlimit
			...			
Spar	100000	100,00	0,45			
			...			
Giro	100001	2500,00		14,50	0,20	4000
			...			
	Konto		Sparkonto	Girokonto		

ABBILDUNG 5.2 Tabellenstruktur für die Vererbungshierarchie

Bei den beiden Beispieldatensätzen kann man erkennen, dass die von einer Oberklasse geerbten Properties auch in den Spalten der Unterklassen vorhanden sind, während die Properties der nicht abstrakten Klassen immer nur in einem Datensatz vorhanden und die Spalten der jeweils anderen Unterklasse(n) NULL sind.

Die Umsetzung der Alternative *eine Tabelle für eine Vererbungshierarchie* erfolgt in JPA mit der `@Inheritance`-Annotation und Setzen des Attributs `strategy` auf `SINGLE_TABLE`. Dies ist gleichzeitig der Default-Wert dieses Attributs. Tabelle 5.1 gibt die `@Inheritance`-Annotation komplett wieder.

TABELLE 5.1 `@Inheritance`-Attribute

`@Inheritance(...)`			
Attribut	**Typ**	**Default**	**Beschreibung**
`strategy`	`InheritanceType`	`SINGLE_TABLE`	Strategie zur Realisierung der Vererbung. Mögliche Werte: `SINGLE_TABLE`, `JOINED`, `TABLE_PER_SUBCLASS`

Die in der Tabellenstruktur in Abbildung 5.2 dargestellte zusätzliche Diskriminatorspalte wird mit der `@DiscriminatorColumn`-Annotation definiert. Tabelle 5.2 zeigt die Attribute dieser Annotation.

TABELLE 5.2 `@DiscriminatorColumn`-Attribute

`@DiscriminatorColumn(...)`			
Attribut	**Typ**	**Default**	**Beschreibung**
`columnDefinition`	`STRING`	—	DDL für Diskriminator-Spalte
`discriminatorType`	`DiscriminatorType`	`STRING`	Typ des Diskriminators. Mögliche Werte sind `CHAR`, `INTEGER`, `STRING`
`length`	`int`	31	Länge von String-Diskriminatoren
`name`	`String`	`DTYPE`	Spaltenname des Diskriminators

Tipp:

Die Verwendung eines Strings als Diskriminatorwert erleichtert das Verständnis und die Nutzung der Tabelle auf SQL-Ebene.

Das Listing 5.1 gibt die ersten Zeilen der Oberklasse `Konto` wieder. Die Definition der Diskriminatorspalte entspricht dem Beispiel aus Abbildung 5.2. Das Listing ist recht umfangreich, was jedoch nicht der Vererbung geschuldet ist. Beim Generieren des Primärschlüssels wird bei 100 000 begonnen, um möglichst realistische Kontonummern zu erhalten. Die Verwendung von Sequenzen zur Primärschlüsselgenerierung haben wir in Abschnitt 2.1.1 erläutert. Die Beziehungsannotationen werden verwendet, um die 1:n- bzw. die n:1-Beziehung zu Buchung und Kunde zu realisieren.

LISTING 5.1 Klasse `Konto` mit Vererbungsannotationen

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "Disc",
                     discriminatorType = DiscriminatorType.STRING)
public abstract class Konto {

  @Id @GeneratedValue(generator = "KtoSeq",
                      strategy = GenerationType.SEQUENCE)
  @SequenceGenerator(name = "KtoSeq", sequenceName = "KTO_SEQ",
                     allocationSize = 100, initialValue = 100000)
```

```
    private Integer kontonummer;
    @Column(precision = 10, scale = 2)
    private BigDecimal kontostand;

    @ManyToOne(optional = false)
    @JoinColumn(name = "kunde", nullable = false)
    private Kunde kunde;

    @OneToMany(mappedBy = "konto", cascade = CascadeType.ALL)
    private Set<Buchung> buchungen;
    ...
```

In den Unterklassen ist lediglich noch der Wert des Diskriminators für diese Klasse anzugeben. Dies erfolgt über die `@DiscriminatorValue`-Annotation, die in Tabelle 5.3 dargestellt ist. Falls diese Annotation nicht verwendet wird, generiert die JPA-Implementierung einen in der Spezifikation nicht näher beschriebenen, herstellerspezifischen Wert. Im Falle eines String-Diskriminators wird der Entity-Name verwendet. Falls Sie die `@DiscriminatorValue`-Annotation verwenden, muss ihr String-Wert dem Diskriminatortyp entsprechen, also z. B. bei `INTEGER` ein ganzzahliger Wert sein.

TABELLE 5.3 `@DiscriminatorValue`-Optionen

`@DiscriminatorValue(...)`			
Option	**Typ**	**Default**	**Beschreibung**
`value`	`String`	—	Wert zur Repräsentation des Entity-Typs. Beim Diskriminatortyp `String` ist der Default der Entity-Name, sonst ein provider-abhängiger Wert.

Im folgenden Code-Ausschnitt entscheiden wir uns bei der Klasse `Sparkonto` für den String `"Spar"`, der auch im Beispiel verwendet wurde. Weil das `Id`-Property bereits in der Oberklasse definiert wurde, wird es in der Unterklasse nicht angegeben. Wir verzichten auf eine Darstellung der Girokonto-Klasse.

```
@Entity
@DiscriminatorValue("Spar")
public class Sparkonto extends Konto {

  @Column(precision = 5, scale = 2)
  private BigDecimal zinssatz;

  ...
```

Das Schema für die drei Tabellen ist relativ offensichtlich. Wir stellen es in Abbildung 5.3 trotzdem dar, damit wir die noch vorzustellenden Alternativen zur Realisierung der Vererbung im Vergleich besser beurteilen können.

Man erkennt in der Tabelle `konto` die Spalten, die wir bereits in Bezug zur Tabelle 5.2 diskutiert haben, sowie die in Tabelle 5.2 unterschlagene Spalte `kunde`, die den Fremdschlüssel zur Kundentabelle repräsentiert.

Das herunterladbare Projekt enthält Datensätze, die in Abbildung 5.4 in der H2-Konsole nach der Abfrage „`Select * from konto`“ dargestellt sind. Die verwendete Vererbungsalternative *eine Tabelle für eine Hierarchie* unterstützt polymorphe Assoziationen vollständig, so

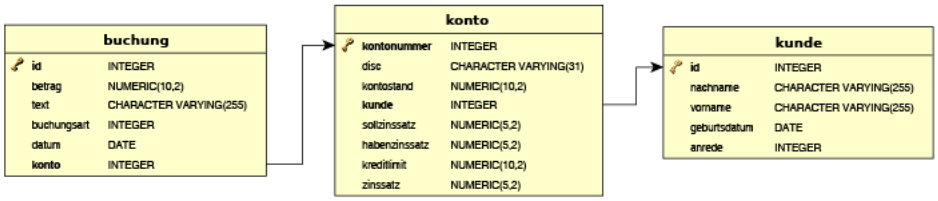


ABBILDUNG 5.3 Tabellenschema für die Vererbungsstrategie `SINGLE_TABLE`

dass ein Kunde mit dem Getter `getKonten()` nach *allen* Konten, sowohl Giro- als auch Sparkonten, gefragt werden kann.

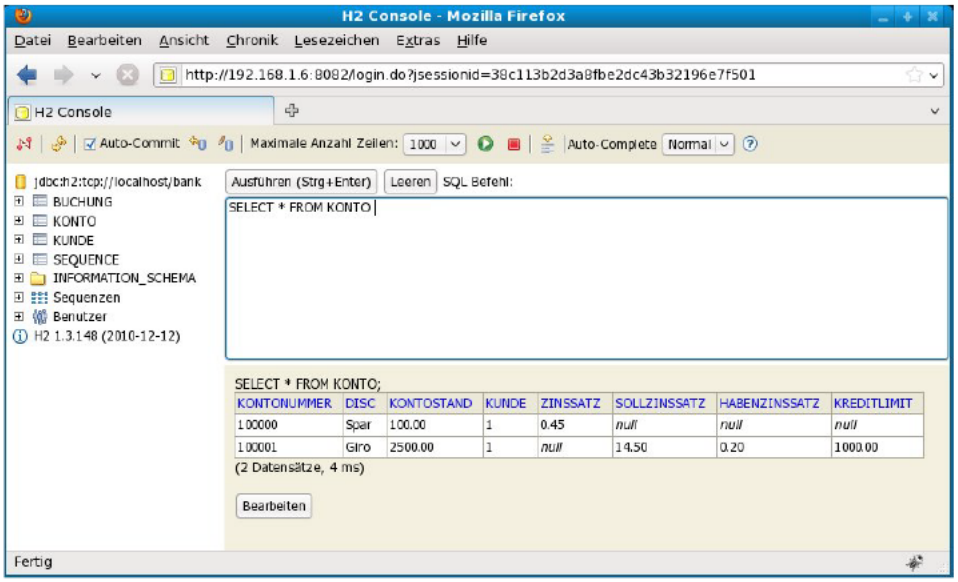


ABBILDUNG 5.4 Beispieldatensätze in der Datenbank

Der folgende Code-Ausschnitt zeigt dies exemplarisch, indem über das Ergebnis dieses Getter-Aufrufs iteriert wird.

```
Kunde k = em.find(Kunde.class, ...);
System.out.println("gelesener Kunde: " + k.getNachname());
for (Konto konto : k.getKonten()) {
  System.out.println("Konto-ID: " + konto.getKontonummer()
                     + ", Klasse: " + konto.getClass().getSimpleName
                         ());
}
```

Die Ausgabe des dargestellten Code-Ausschnitts lautet

```
gelesener Kunde: Mustermann
Konto-ID: 100001, Klasse: Girokonto
Konto-ID: 100000, Klasse: Sparkonto
```

Auch die in Kapitel 7 noch vorzustellenden Datenbankabfragen mit JPQL arbeiten polymorph. Die Abfrage „`Select k from Konto k`“ selektiert alle Instanzen der Entity-Klasse `Konto`. Syntaktisch wird dies im Vorgriff auf Kapitel 7 wie folgt dargestellt.

```
List<Konto> konten = em.createQuery("Select k from Konto k",
                                    Konto.class)
                        .getResultList();
```

Die abstrakte Klasse `Konto` besitzt keine Instanzen. In unserem Beispiel existiert jedoch je eine Instanz der nicht abstrakten Unterklassen. Der folgende Code zeigt die Iteration über das Abfrageergebnis sowie die textuelle Ausgabe.

```
for (Konto konto : konten) {
  System.out.println(konto.getClass().getSimpleName()
     + " mit Kontonummer " + konto.getKontonummer());
}

Sparkonto mit Kontonummer 100000
Girokonto mit Kontonummer 100001
```

Hinweis:

Soll das Property `habenzinssatz` der Klasse `Girokonto` in `zinssatz` umbenannt werden, so werden die identisch benannten Properties der beiden Unterklassen auf dieselbe Tabellenspalte abgebildet. Ist weiterhin die Existenz zweier Spalten gewünscht, so kann mit der `@Column`-Annotation ein alternativer Spaltenname definiert werden.

Das Paradigma der *Convention over Configuration* wird von JPA vollständig realisiert, wie viele Beispiele des Kapitels 2 exemplarisch gezeigt haben. Im Falle der Vererbung ist die Strategie `SINGLE_TABLE` der Default. Für die Diskriminator-Spalte sowie deren Werte existieren ebenfalls Defaults: Der Spaltenname ist `DTYPE`, der Typ `String` und der Wert der Entity-Name. Es können also alle vererbungsrelevanten JPA-Annotationen des Beispiels entfallen, ohne die Korrektheit und Vollständigkeit des Beispiels zu gefährden. Die folgenden Klassendefinitionen entsprechen dem obigen Beispiel, wobei das Mapping implizit von einer String-Diskriminatorspalte `DTYPE` und den Werten `Sparkonto` und `Girokonto` als einzigem Unterschied ausgeht.

```
@Entity
public abstract class Konto {
  ....
}

@Entity
public class Sparkonto extends Konto {
  ...
}

@Entity
public class Girokonto extends Konto {
  ...
}
```

Achtung:

Die in diesem und den nächsten Abschnitten vorgestellten Möglichkeiten zur Realisierung von Vererbungsbeziehungen werden von OpenJPA nur implementiert, wenn die kompilierten Klassen einem sogenannten Enhancement unterzogen werden. Wir gehen hierauf in Abschnitt 13.2 ein.

Projekt:

Das Projekt *vererbung-single-table* enthält den Code für die erste Vererbungsalternative *eine Tabelle für eine Vererbungshierarchie*, die durch die Vererbungsstrategie `SINGLE_TABLE` realisiert wird.

5.2 Eine Tabelle je Unterklasse

Das zugrunde liegende Problem bei der Abbildung objektorientierter Vererbung auf relationale Datenbanken ist der Umstand, dass objektorientierte Sprachen sowohl die Vererbungsbeziehung als auch eine „normale" Beziehung, in der objektorientierten Welt als Assoziation bezeichnet, kennen. Man spricht auch von „is-a"- und „has-a"-Beziehungen. Relationale Datenbanken kennen aber lediglich die zweite Beziehungsart. Die Realisierungsalternative *eine Tabelle je Unterklasse* macht sich dies zunutze und bildet die Vererbung auf eine Beziehung zwischen Tabellen ab. Um auf die Daten eines Unterklasseobjekts zugreifen zu können, muss die JPA-Implementierung daher über diese Tabellen joinen.

Wie sieht nun die konkrete Realisierung dieser Alternative aus? Der folgende Code-Ausschnitt gibt die Annotationen der Oberklasse wieder. Als Vererbungsstrategie wird `JOINED`, als Diskriminatorspalte `Disc` verwendet.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@DiscriminatorColumn(name = "Disc")
public abstract class Konto {

  @Id @GeneratedValue
  private Integer kontonummer;
  ...
```

Als Diskriminatorwert für die Unterklassen bleiben wir bei den Werten `Spar` und `Giro` des Beispiels. Die Unterklassen benötigen kein mit `@Id` annotiertes Primärschlüssel-Property.

```
@Entity
@DiscriminatorValue("Spar")
public class Sparkonto
       extends Konto {
  ...
}
```

```
@Entity
@DiscriminatorValue("Giro")
public class Girokonto
       extends Konto {
  ...
}
```

Das entsprechende Tabellenschema für die beschriebene Struktur ist in Abbildung 5.5 dargestellt. Für die Vererbungshierarchie existieren die drei Tabellen `Konto`, `Sparkonto` und `Girokonto`, deren Spalten den jeweiligen Properties der Klassen entsprechen. Die Klasse

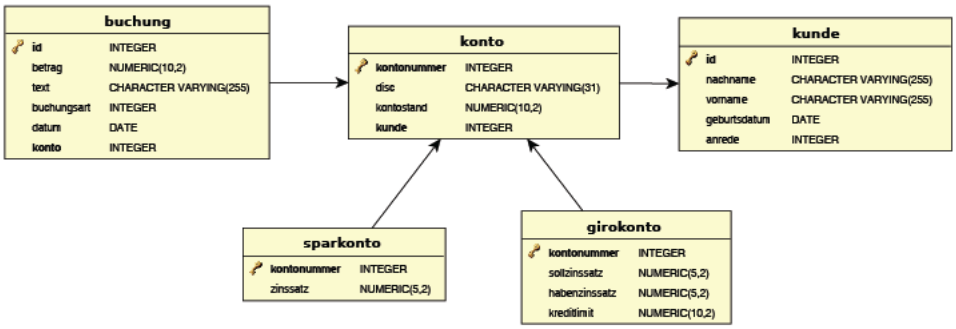


ABBILDUNG 5.5 Tabellenschema für die Vererbungsstrategie `JOINED`

`Konto` enthält das Primärschlüssel-Property `kontonummer`, das sich auch in der Tabelle wiederfindet. Die Klassen `Sparkonto` und `Girokonto` besitzen ein derartiges Property nicht, sehr wohl aber die entsprechenden Tabellen die jeweiligen Spalten. Diese werden für den benötigten Join als Fremdschlüssel verwendet.

Falls die Tabellen schon existieren oder mit abweichenden Bezeichnern generiert werden sollen, kann mit der `@PrimaryKeyJoinColumn`-Annotation ein abweichender Name in der Unterklasse angegeben werden. Dieser wird dann für den Join mit der Tabelle der Oberklasse verwendet. Soll etwa der Fremdschlüssel in der Tabelle `Sparkonto` `Konto_kontonummer` heißen, so lässt sich dies folgendermaßen realisieren:

```
@Entity
@DiscriminatorValue("Spar")
@PrimaryKeyJoinColumn(name = "Konto_kontonummer")
public class Sparkonto extends Konto {
  ...
}
```

Bei der Verwendung der Klassen ergibt sich keine Änderung. Auch diese Vererbungsalternative realisiert polymorphe Abfragen vollständig. Das Beispiel der Abfrage aller Konten eines Kunden sowie die JPQL-Abfrage aller existierender Konten aus Abschnitt 5.1 können problemlos ausgeführt werden und ergeben dieselben Ergebnisse.

Achtung:

Die Vererbungsalternative *eine Tabelle je Unterklasse* verwendet Joins, um die Daten von Unterklasseninstanzen zusammenzustellen. Bei tiefen Vererbungshierarchien kann dies zu Performanzproblemen führen. Die Spezifikation warnt ausdrücklich: „In deep class hierarchies, this may lead to unacceptable performance."

Die Notwendigkeit zur Verwendung von `@DiscriminatorColumn` in der Oberklasse und `@DiscriminatorValue` in der Unterklasse sowie die Default-Regeln bei Nichtexistenz sind in der Spezifikation (Abschnitt 11.1.10) leider nicht exakt formuliert. EclipseLink als Referenzimplementierung realisiert bei der Nichtexistenz die Default-Regeln, die auch für die Alternative `SINGLE_TABLE` gelten, also `DTYPE` als Spaltenname, Klassennamen als Werte. Hibernate und OpenJPA verzichten vollständig auf die Diskriminatorspalte.

Achtung:

Falls die Annotationen `@DiscriminatorColumn` und `@DiscriminatorValue` nicht verwendet werden, bilden Hibernate und OpenJPA die Oberklasse ohne Diskriminatorspalte auf die entsprechende Tabelle ab, während EclipseLink die Default-Regeln für die beiden Annotationen anwendet. Dies führt zu nicht portablen Anwendungen.

Projekt:

Das Projekt *vererbung-joined* enthält den Code für die zweite Vererbungsalternative *eine Tabelle je Unterklasse*, die durch die Vererbungsstrategie `JOINED` realisiert wird.

5.3 Eine Tabelle je konkreter Klasse

Bei der Alternative *eine Tabelle je konkreter Klasse* wird eine Klassenhierarchie auf eine Tabellenstruktur abgebildet, bei der eine konkrete, nicht abstrakte Klasse durch eine Tabelle repräsentiert wird, die die Properties der Klasse selbst, aber auch die geerbten Properties aller Oberklassen enthält. Abstrakte Klassen werden im Gegensatz zur Alternative aus Abschnitt 5.2 nicht durch Tabellen repräsentiert, sondern sind implizit in den Tabellen der konkreten Unterklassen vorhanden. Der JPA-Provider wird daher mit dem SQL-Vereinigungsoperator `Union` oder getrennten Abfragen je Unterklasse arbeiten, um die Daten für polymorphe Operationen zu ermitteln. Die Umsetzung erfolgt durch die Vererbungsstrategie `TABLE_PER_CLASS`, so dass sich als Code für die Oberklasse `Konto` der folgende Ausschnitt ergibt

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Konto {

  @Id @GeneratedValue(strategy = GenerationType.SEQUENCE)
  private Integer kontonummer;
  ...
```

In den konkreten Unterklassen wird keine spezielle Syntax benötigt, so dass wir auf eine Darstellung verzichten.

Hinweis:

Die beschriebene Alternative *eine Tabelle je konkreter Klasse* ist in der JPA-Spezifikation sowohl in der Version 1.0 als auch in der Version 2.0 als optional gekennzeichnet. Die bedeutet, dass ein JPA-Provider diese Alternative anbieten kann, aber nicht muss, um spezifikationskonform zu sein. Die drei von uns verwendeten JPA-Provider realisieren diese Alternative, wenn auch in unterschiedlichem Umfang.

Die Verwendung der Klassen erfolgt ohne Einschränkungen, soweit die Vererbungshierarchie isoliert betrachtet wird. Werden Assoziationen auf Entity-Klassen verwendet, die keine Blät-

ter der Vererbungshierarchie sind, kommt es zu Problemen bzw. Einschränkungen. Bei unserem zugrunde liegenden Beispiel ist genau dies der Fall. In Abbildung 5.1 auf Seite 134 ist die abstrakte Oberklasse der Vererbungshierarchie sowohl an einer n:1- als auch an einer 1:n-Beziehung beteiligt. Problematisch ist hier die 1:n-Beziehung, da diese auf Datenbankebene durch einen Fremdschlüssel in der Tabelle `Buchung` zu realisieren ist mit der Tabelle `Konto` als Ziel. Diese existiert jedoch nicht, sondern nur die Tabellen `Sparkonto` und `Girokonto`. Zur Verdeutlichung dient die Darstellung des Datenbankschemas in Abbildung 5.6.

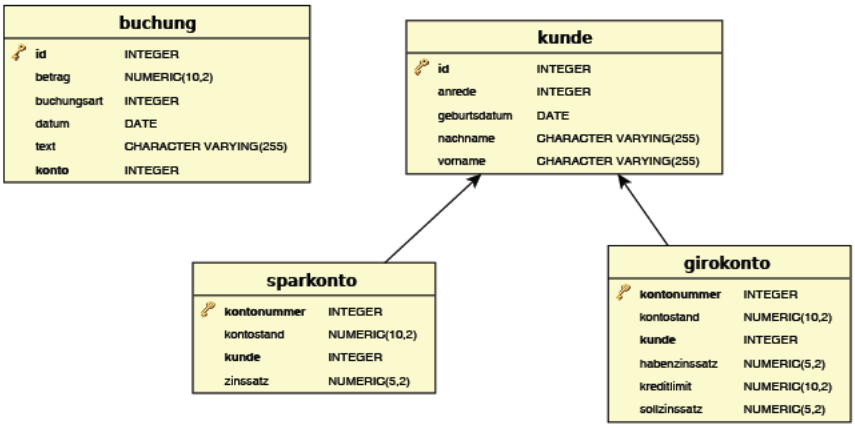


ABBILDUNG 5.6 Tabellenschema für die Vererbungsstrategie `TABLE_PER_CLASS`

Man erkennt, dass *keine* Fremdschlüsselbeziehung von der Tabelle `Buchung` zu den beiden Kontentabellen existiert und auch nicht existieren kann, da SQL derartige Fremdschlüsselbeziehungen nicht unterstützt. Das Kritische an dieser Realisierungsalternative ist also, dass der JPA-Provider einerseits durch die entsprechenden Annotationen prinzipiell über die Informationen bezüglich der Annotationen verfügt, um zu korrekten Ergebnissen zu kommen, andererseits aber andere Anwendungen, eventuell mit anderen Programmiersprachen realisiert, diese Informationen nicht besitzen. Außerdem kann das Datenbanksystem die Integrität der Anwendungsdaten nicht garantieren, da die entsprechenden Fremdschlüssel-Constraints nicht existieren.

EclipseLink und Hibernate kommen mit diesem Problem zurecht, OpenJPA nicht: Für das gegebene Beispiel mit bidirektionalen 1:n- und n:1-Beziehungen können EclipseLink und Hibernate auf das in Abbildung 5.6 dargestellte Schema mappen, OpenJPA nicht. Details über die von OpenJPA nicht unterstützten Assoziationsarten bei der Verwendung von `TABLE_PER_CLASS` sind im OpenJPA-Handbuch dokumentiert.

Achtung:

OpenJPA verbietet die Verwendung von 1:1- und n:1-Beziehungen zu Entities einer `TABLE_PER_CLASS`-Hierarchie, die keine Blätter der Hierarchie sind.

Um die Eindeutigkeit der Primärschlüssel in den konkreten Unterklassen garantieren zu können, müssen aber auch EclipseLink und Hibernate eine gewisse Einschränkung erzwingen:

Die Generierungsstrategie für das Primärschlüssel-Property für das Assoziationsziel, in unserem Beispiel die Klasse `Konto`, muss `TABLE` oder `SEQUENCE` sein. Der JPA-Provider muss garantieren können, dass Spar- und Girokonten verschiedene Primärschlüssel benutzen, da im Property `konto` der Klasse `Buchung` ein eindeutiger Wert verwendet werden muss. Dies ist mit `IDENTITY` jedoch nicht zu garantieren.

Interessant bei dieser Vererbungsalternative ist die Umsetzung von Anfragen, die in der Spezifikation bereits mit `Union` oder separaten `Select`-Abfragen pro Unterklasse vorgeschlagen wird. EclipseLink und OpenJPA erzeugen für unser Beispiel bei der JPQL-Abfrage „`Select k from Konto k`" zwei einzelne `Select`-Abfragen während Hibernate insgesamt drei verschachtelte `Select`s mit `Union` erzeugt.

Projekt:

Das Projekt *vererbung-table-per-class* enthält den Code für die dritte Vererbungsalternative *eine Tabelle je konkreter Klasse*, die durch die Vererbungsstrategie `TABLE_PER_CLASS` realisiert wird.

5.4 Vergleich der Vererbungsstrategien

Wir haben in den letzten drei Abschnitten die in JPA vorgesehenen Möglichkeiten zur Realisierung von Vererbungshierarchien vorgestellt und zum Teil bereits deren Vor- und Nachteile genannt. Hier sollen nun noch einmal die Vor- und Nachteile explizit und vollständig aufgezählt werden, um dem Leser einen Vergleich der Vererbungsstrategien zu ermöglichen. Es gibt keine generell zu bevorzugende Alternative, und es sollten bei jeder zu realisierenden Vererbungshierarchie erneut die Vor- und Nachteile in der konkreten Aufgabenstellung abgewogen und eine Entscheidung für eine der drei Alternativen getroffen werden. Im Abschnitt 5.6 stellen wir anhand eines einfachen Beispiels die Kombination von Vererbungsstrategien vor. Dies erfolgt aus Gründen der Vollständigkeit. Im Allgemeinen ist von der Kombination abzuraten, da sie zu komplexeren, schlechter wartbaren und somit unerwünschten Strukturen führt.

5.4.1 Eine Tabelle für eine Vererbungshierarchie

Die Strategie `SINGLE_TABLE` ist der Default bei den Vererbungsstrategien. Durch die von JPA praktizierte *Convention over Configuration* legt dies eine gewisse Präferenz des Spezifikationsgremiums für diese Strategie nahe.

Vorteile `SINGLE_TABLE`:

- Volle Unterstützung objektorientierter Polymorphie
- Sehr performante Abfragen möglich, da keine Joins
- Einfaches konzeptionelles Modell
- Einfache Syntax. Im Minimalfall kein zusätzliches JPA-Mapping benötigt

Nachteile `SINGLE_TABLE`:

- NOT-NULL-Constraints nicht möglich
- Minimaler Speicherbedarf für die NULL-Werte
- Datenbank-Schema nicht in Normalform, da funktionale Abhängigkeiten

5.4.2 Eine Tabelle je Unterklasse

Die Strategie `JOINED` entspricht der objektorientierten Denkweise, da es eine 1:1-Entsprechung zwischen einer Klasse und einer Tabelle gibt.

Vorteile `JOINED`:

- Volle Unterstützung objektorientierter Polymorphie
- 1:1-Entsprechung zwischen Klasse und Tabelle
- Tabellen in Normalform

Nachteile `JOINED`:

- Abfragen verwenden Joins und sind daher aufwendig. Der Join findet jedoch über einen singulären Fremdschlüssel statt, so dass der Aufwand begrenzt ist.

5.4.3 Eine Tabelle je konkreter Klasse

Die Strategie `TABLE_PER_CLASS` ist optional, muss also von einem JPA-Provider nicht implementiert werden, so dass sich Portabilitätsprobleme ergeben können. Die drei von uns verwendeten Provider stellen die Strategie zur Verfügung.

Vorteile `TABLE_PER_CLASS`:

- Sehr performante Abfragen möglich, da keine Joins
- Einfache Verwendung in anderen Programmiersprachen
- Keine Änderung an bestehenden Tabellen beim Einfügen von Blättern in der Vererbungshierarchie

Nachteile `TABLE_PER_CLASS`:

- Aufwendige Abfrage durch mehrere `Select`- und/oder `Union`-Anweisungen
- Schwache objektorientierte Polymorphie
- Schemaredundanz in Unterklassen, daher nicht in Normalform
- Keine expliziten Fremdschlüssel auf Datenbankebene
- Verwendung von Beziehungen eventuell eingeschränkt
- In der Spezifikation als optional gekennzeichnet; daher eventuell nicht portabel

5.5 Mapping von Oberklassen

Entities können von Oberklassen erben, die persistente Daten und Mapping-Informationen bereitstellen, selbst aber keine Entities sind. Eine solche Oberklasse wird *Mapped Super-*

class genannt und mit `@MappedSuperclass` annotiert. Die Klasse dient als Verteiler von Mapping-Informationen in die Unterklassen, darf aber selbst nicht als Entity verwendet werden: Die Klasse lässt sich nicht in Entity-Manager-Methoden und JPQL-Abfragen verwenden. Sie ist auch nicht als Ziel von Beziehungen erlaubt.

Im folgenden Beispiel werden Firmen- und Privatkunden und deren Adressen verwaltet. Dazu wird die Klasse `Kunde` als gemappte Oberklasse mit `@MappedSuperclass` annotiert. Die Klasse `Adresse` ist eine gewöhnliche Entity-Klasse.

```
@MappedSuperclass
public abstract class Kunde {

  @Id @GeneratedValue
  private Integer id;
  @Version
  private Integer version;
  @ManyToOne
  @JoinColumn(name = "adresse")
  private Adresse adresse;
  ...
```

```
@Entity
public class Adresse {

  @Id @GeneratedValue
  private Integer id;
  private String strasse;
  private String hausnummer;
  private String plz;
  private String ort;
  ...
```

Die Beziehung zwischen Kunden und Adressen darf nicht bidirektional sein, da gemappte Oberklassen nicht als Ziel einer Beziehung erlaubt sind. Firmen- und Privatkunden werden ohne zusätzliche vererbungsspezifische Annotationen als Unterklasse definiert.

```
@Entity
public class Privatkunde
       extends Kunde {

  private String vorname;
  private String nachname;
  @Temporal(TemporalType.DATE)
  private Date geburtsdatum;
  private Anrede anrede;
  ...
```

```
@Entity
public class Firmenkunde
       extends Kunde {

  private String
      firmenname;
  ...
```

Die gemappte Oberklasse kann abstrakt sein, muss es aber nicht. Für den abstrakten Fall entspricht das Mapping der Vererbungsalternative `TABLE_PER_CLASS`, da die Properties in die Unterklassen wandern. Der Unterschied liegt in der nicht vorhandenen Tabelle für die Oberklasse `Kunde`. Abbildung 5.7 zeigt das entsprechende Schema. Die bei `TABLE_PER_CLASS` fehlenden Fremdschlüssel-Constraints sind vorhanden.

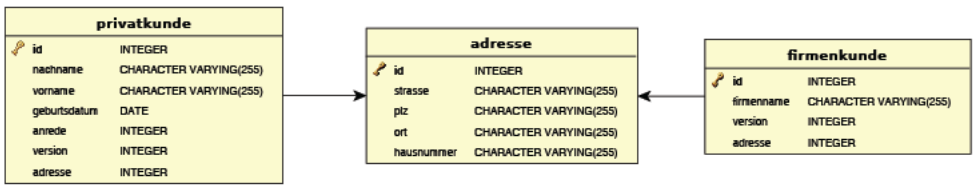


ABBILDUNG 5.7 Tabellenschema für das Beispiel mit `@MappedSuperclass`

Hinweis:

OpenJPA erlaubt die Verwendung von gemappten Oberklassen in Queries und Entity-Manager-Methoden. Das Ergebnis enthält entsprechende Unterklasseninstanzen.

Wir haben im Beispiel die gemappte Oberklasse fachlich verwendet und Kunden sowie Firmen- und Privatkunden modelliert. Die Verwendung von `@MappedSuperclass` ist technisch motiviert: Sollen bestimmte Entities rein technisch motivierte persistente Daten erhalten, so scheidet hierfür ein Entity als Oberklasse aus, da es keine Fachlichkeit (keine fachlichen Daten) enthält. Eine gemappte Oberklasse kann diese technisch motivierten Daten jedoch einfach an alle Unterklassen vererben. Ein Beispiel hierfür sind etwa Primärschlüssel (`@Id`) und Versionsproperty (`@Version`), wie wir sie bereits im Beispiel verwendet haben, sowie ein Zeitstempel für den initialen Eintrag in der Datenbank und ein Zeitstempel für die letzte Änderung. Für derartige Erweiterungen in Richtung historische Datenverarbeitung existieren jedoch auch dedizierte Frameworks. Eines davon, Envers, stellen wir in Abschnitt 11.4 vor.

Projekt:

Den Programm-Code für gemappte Oberklassen finden Sie im Projekt *vererbung-mapped-superclass*.

5.6 Kombination von Vererbungsstrategien

In den Abschnitten 5.1, 5.2 und 5.3 wurde die `@Inheritance`-Annotation in ihren drei alternativen Verwendungen für die jeweils komplette Vererbungshierarchie der Beispiele eingesetzt. Die `@MappedSuperclass`-Annotation im Abschnitt 5.5 wurde verwendet, um Objektzustände ohne `@Entity` persistent zu machen. Die Kombination all dieser Annotationen und die Verwendung nicht persistenter Klassen innerhalb ein und derselben Vererbungshierarchie ist jedoch auch möglich. Die JPA-Spezifikation erwähnt explizit, dass der Einsatz verschiedener Vererbungsstrategien innerhalb einer Entity-Hierarchie durch einen JPA-Provider *nicht* unterstützt werden muss und daher nicht portabel ist. Wir raten Ihnen von einer Kombination verschiedener Vererbungsstrategien innerhalb einer Hierarchie ab, da die Tabellenstrukturen, aber auch die Handhabung auf Java-Seite naturgemäß komplexer werden. Nichtsdestotrotz ist die Kombination möglich, und wir wollen sie an dieser Stelle exemplarisch erläutern.

Hinweis:

Die Kombination verschiedener Vererbungsstrategien innerhalb einer Hierarchie muss von einem JPA-Provider nicht unterstützt werden. Wir raten daher im allgemeinen Fall von einer derartigen Kombination ab. Falls entsprechende Gründe für eine Kombination sprechen, raten wir zur ausschließlichen Verwendung von `SINGLE_TABLE` und `JOINED`.

Um die Vererbungsstrategien `SINGLE_TABLE` und `JOINED` einsetzen zu können, müssen wir unsere Vererbungshierarchie vergrößern. Abbildung 5.8 zeigt die neue Hierarchie, in die ein Kontokorrent- und ein Tagesgeldkonto integriert wurden. Falls die Fachlichkeit nicht der Realität entspricht, bitten wir den bankkaufmännisch gebildeten Leser um Nachsicht.

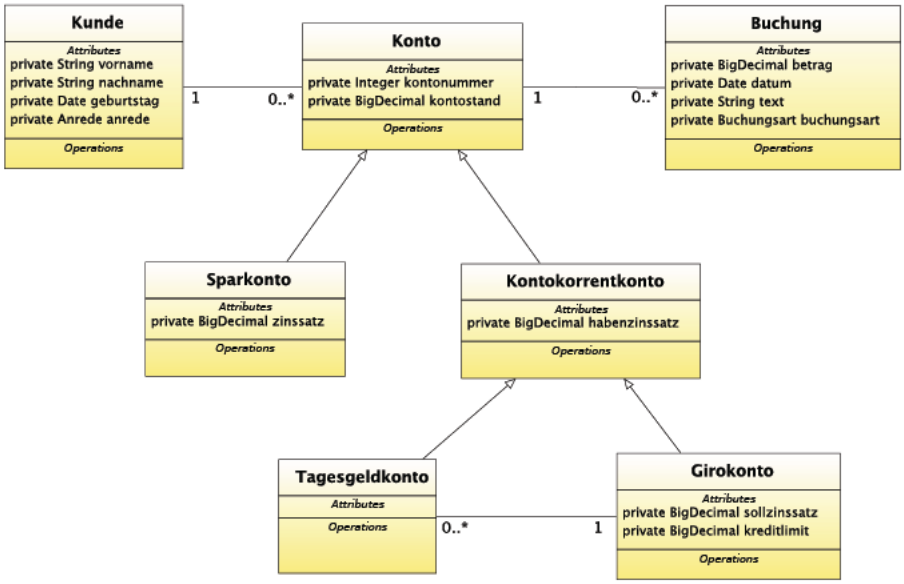


ABBILDUNG 5.8 Klassenhierarchie für kombinierte Vererbungsstrategie

Die Klassen `Konto` und `Kontokorrentkonto` sind abstrakt und jeweils die Wurzel einer entsprechenden Vererbungsstrategie für den darunter befindlichen Teilbaum. Die verwendeten Strategien sind `JOINED` für das Entity `Konto` und `SINGLE_TABLE` für das Entity `Kontokorrentkonto`.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Konto {

  @Id @GeneratedValue
  private Integer kontonummer;
  ...
}
```

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public abstract class Kontokorrentkonto extends Konto {
  ...
}
```

Das zugrunde liegende Tabellenschema ist in Abbildung 5.9 dargestellt. Bemerkenswert bei diesem Schema ist eine doppelte Fremdschlüsselbeziehung von der Tabelle `Kontokorrentkonto` zur Tabelle `Konto`. Eine Fremdschlüsselbeziehung ist der Vererbungsstrategie `JOINED` geschuldet, damit Instanzen von `Konto`-Unterklassen mit den entsprechenden Daten von `Konto` gejoint werden können. Die zweite Fremdschlüsselbeziehung repräsentiert die n:1-Beziehung eines Tagegeldkontos zu seinem Referenzkonto, einem Girokonto. Der folgende Code-Ausschnitt skizziert dies.

```
@Entity
public class Tagesgeldkonto extends Kontokorrentkonto {

  @ManyToOne
  private Girokonto referenzkonto;
  ...
}
```

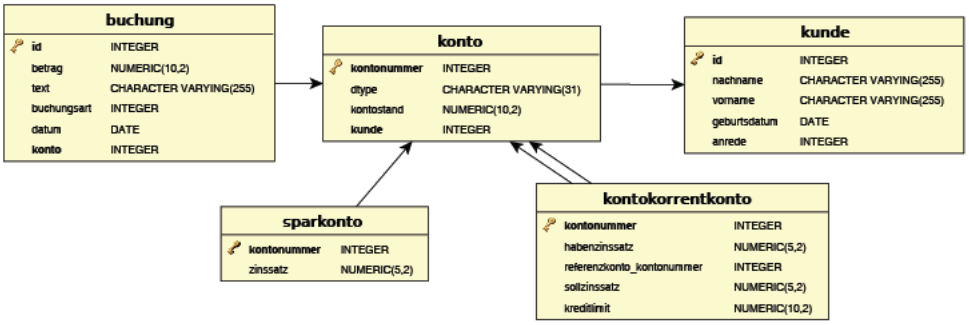


ABBILDUNG 5.9 Tabellenschema für kombinierte Vererbungsstrategie

Die in Abbildung 5.8 dargestellten Klassen sind durch den Verzicht auf die Vererbungsstrategie `TABLE_PER_CLASS` in vollem Umfang polymorph verwendbar: Über einen Kunden kann zu seinen Konten navigiert werden, und JPQL-Anfragen für Konten liefern Instanzen sämtlicher Unterklassen. Beide Anwendungsfälle sind als Beispiele im herunterladbaren Projekt enthalten.

Projekt:

Den Programm-Code für die Kombination von Vererbungsstrategien finden Sie im Projekt *vererbung-mixed*.

6 Transaktionen

Früher oder später muss sich jeder Entwickler dem Problem des simultanen Zugriffs mehrerer Anwender auf ein und denselben Datensatz stellen. Die Möglichkeiten, dabei auftretende Probleme zu behandeln, reichen vom „Ignorieren des Problems“ bis zum Abschalten des parallelen Zugriffs, was bedeutet, dass die Zugriffe auf die Sätze einer Datenbank nur noch hintereinander, also serialisiert, erfolgen können. Zwischen diesen beiden Maßnahmen sind eine Reihe weiterer Lösungsansätze möglich, die wir in diesem Abschnitt diskutieren wollen. Dazu sei uns zunächst eine theoretische Einführung in das Thema des Mehrbenutzerbetriebs und der Transaktionen erlaubt.

Eine Transaktion ist eine logische Einheit von Operationen innerhalb der Datenbank. Sobald eine dieser Operationen aus welchem Grund auch immer fehlschlägt, muss das Datenbanksystem sicherstellen, dass auch die anderen zur Transaktion gehörenden Operationen nicht durchgeführt werden bzw., wenn sie bereits durchgeführt wurden, dass die Änderungen wieder rückgängig gemacht werden. Eine Transaktion ist erst dann erfolgreich beendet, wenn auch die letzte Operation innerhalb dieser Transaktion ohne Fehler zum Ende kommt. In der Datenbank-Theorie sind diese Eigenschaften als ACID-Eigenschaften bekannt. Wir wollen sie in der folgenden Übersicht näher erläutern.

A wie Atomicity: Transaktionen werden ganz oder gar nicht ausgeführt.

C wie Consistency: Transaktionen überführen die Datenbank von einem konsistenten Zustand in einen anderen.

I wie Isolation: Transaktionen werden so ausgeführt, als wenn sie allein auf der Datenbank operieren würden.

D wie Durability: Transaktionen sind nach erfolgreichem Abschluss dauerhaft und gehen auch durch Fehler nicht mehr verloren.

Für eine Datenbank spielt es keine Rolle, ob eine Transaktion aus einer oder mehreren Operationen besteht. Jedes ausgeführte SQL-Statement gehört zu einer Transaktion. Die Verbindung zu einer Datenbank mit der *Java Database Connectivity (JDBC)* befindet sich per Default im Auto-Commit-Modus. Wird diese Einstellung nicht verändert, wird jeder auf der Datenbank ausgeführte Befehl, der Daten definiert oder manipuliert, automatisch mit einem `commit()` versehen, die Existenz einer Transaktion bleibt dem Entwickler zunächst komplett verborgen.

Tipp:

In der Regel verwenden auch die von den Datenbank-Herstellern mitgelieferten Clients den Auto-Commit-Modus. Jeder abgesetzte SQL-Befehl wird mit dieser Einstellung automatisch in die Hülle einer Transaktion gebracht und diese dann abgeschlossen. Dieses Verhalten lässt sich mit dem Befehl `SET AUTOCOMMIT OFF` oder bei grafischen Clients über eine Option in den Einstellungen deaktivieren.

6.1 Probleme des Mehrbenutzerbetriebs

Datenbanken erlauben es, dass mehrere Transaktionen zur gleichen Zeit aktiv sind. Diese Gleichzeitigkeit kann Probleme mit sich bringen, die im SQL-Standard in die Bereiche *Dirty Read*, *Non-repeatable Read* und *Phantom Read* eingeteilt werden. Diese Problemklassen wollen wir im Folgenden kurz skizzieren.

Dirty Reads

Dirty Reads treten dann auf, wenn eine Transaktion Daten einer anderen Transaktion lesen kann, obwohl die andere Transaktion ihre Arbeiten noch nicht vollständig abgeschlossen hat. Wir haben den Vorgang in Abbildung 6.1 skizziert. Innerhalb der Transaktion 1 wird der Datensatz `x1` gelesen, verändert und zurückgeschrieben. Genau zu diesem Zeitpunkt greift Transaktion 2 auf diesen Datensatz zu und liest ihn. Tritt danach in Transaktion 1 ein Fehler auf, werden sämtliche Operationen dieser Transaktion rückgängig gemacht. Leider hat die zweite Transaktion die bereits geschriebenen Daten schon gelesen und besitzt jetzt einen falschen Inhalt des Datensatzes `x1`.

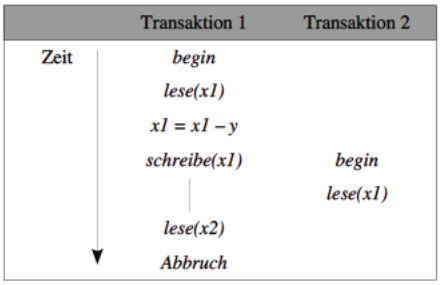


ABBILDUNG 6.1 Darstellung des Dirty-Read-Problems

Non-repeatable Reads

Ein Non-repeatable Read liegt vor, wenn innerhalb einer Transaktion eine Abfrage mehrmals ausgeführt wird, diese Abfrage dabei aber unterschiedliche Ergebnisse liefert. Sobald

	Transaktion 1	Transaktion 2
Zeit		*begin*
	begin	*lese(x1)*
	lese(x1)	
	x1 = x1 − y	
	schreibe(x1)	
		lese(x1)

ABBILDUNG 6.2 Darstellung des Non-Repeatable-Read-Problems

das Datenbanksystem es zulässt, mehrere Transaktionen gleichzeitig auszuführen, und keine weiteren Locking-Maßnahmen ergreift, können Situationen entstehen, in denen gegen den Grundsatz der Isolation verstoßen wird. In Abbildung 6.2 wird in der Transaktion 2 der Datensatz `x1` mehrmals gelesen. Da keine Operation in der Transaktion 2 diesen Datensatz verändert, kann sie beim erneuten Einlesen des Datensatzes erwarten, dass dieser unverändert geblieben ist. Wird jedoch nach dem ersten Lesen dieser Datensatz durch Transaktion 1 verändert, liefert das erneute Einlesen einen anderen Inhalt. Die Isolation, eine wichtige Eigenschaft, die jeder Transaktion zugesichert werden sollte, ist verletzt.

Phantom Reads

Eine Transaktion handelt mit Phantom-Sätzen, wenn die gleichen Suchkriterien innerhalb einer Transaktion auf unterschiedliche Datensätze zutrifft, weil eine andere Transaktion die Menge der Elemente durch Hinzufügen oder Entfernen verändert hat. In Abbildung 6.3 erhält

	Transaktion 1	Transaktion 2
Zeit	*begin*	*begin*
	*select * from kunde where id < 10*	
		insert into kunde (id, name) values (5, 'Meier')
	*select * from kunde where id < 10*	

ABBILDUNG 6.3 Darstellung des Phantom-Problems

die Transaktion 1 bei der zweiten Operation eine andere Menge von Kunden zurückgeliefert als bei der ersten Abfrage, weil die Transaktion 2 einen Kunden hinzugefügt hat, der den ungeänderten Suchkriterien entspricht.

6.2 Isolation-Level

Um die geschilderten Probleme zu umgehen, finden Sie im SQL-Standard verschiedene Isolation-Level. Jeder Level stellt einen bestimmten Kompromiss aus einer Menge von in Kauf genommenen potenziellen Problemen des Mehrbenutzerbetriebs und einer möglichst parallelen Ausführung vieler Transaktionen dar. Die jeweiligen Auswirkungen eines bestimmten Isolation-Level sind in Tabelle 6.1 dargestellt.

Wie aus der Tabelle ersichtlich ist, handelt man sich mit jeder weiteren Stufe, ausgehend von `Serializable` immer mehr potenzielle Probleme ein. Zu beachten ist auch, dass nicht jede Datenbank jedes Isolation-Level bereitstellt.

Wir wollen diesen kleinen Exkurs in die Welt der Transaktionen und der Mehrbenutzerfähigkeit von Datenbanken mit einem weiteren Beispiel abschließen. Abbildung 6.4 zeigt die Verwendung des *Serializable*-Isolation-Level in einer Oracle-Datenbank mit Hilfe von 3 verschiedenen Transaktionen. Der Leser kann dieses Beispiel selbst nachvollziehen, indem er einfach 3 unterschiedliche Client-Anwendungen zu einer Datenbank öffnet. Dabei muss jedoch sichergestellt sein, dass sich die Anwendungen nicht im Auto-Commit-Modus befinden. Das dargestellte Beispiel lässt sich auch mit einer Postgres-Datenbank nachstellen.

(a) Client 1 beginnt seine Transaktion. Er stellt für seine Transaktion den Isolation-Level `Serializable` ein. Die anderen Clients befinden sich ohne explizite Konfiguration automatisch im Level `Read Committed`, dem Standardwert von Oracle.

(b) Client 1 liest innerhalb seiner Transaktion den Nachnamen des Kunden mit der Id 801. Er bekommt als Ergebnis den Namen `Voigt`.

(c) Client 2 beginnt an dieser Stelle seine Arbeit und ändert den Nachnamen des Kunden 801. Diese Änderung wird zunächst noch nicht mit einem `commit` abgeschlossen.

(d) Client 3 liest den Nachnamen des gleichen Kunden. Da 2 seine Arbeit noch nicht abgeschlossen hat, bekommt er ebenfalls das Ergebnis `Voigt`.

(e) Mit dem `commit` wird die Änderung des Nachnamens abgeschlossen.

(f) Der Nachname `Wehr` ist für den Client 3 sichtbar.

(g) Da sich Client 1 im Isolation-Level Serializable befindet, sind die oben beschriebenen Problemklassen ausgeschlossen. Die Datenbank sichert ihm zu, *isoliert* zu arbeiten. Aus diesem Grund bekommt er mit seiner Abfrage den Nachnamen `Voigt`, obwohl dieser bereits geändert wurde.

(h) Client 1 beendet seine begonnene Transaktion.

(i) Erst jetzt sieht auch Client 1 die Änderung des Nachnamens.

TABELLE 6.1 SQL-Isolation-Level

Isolation-Level	Dirty Read	Non-repeatable Read	Phantom Read
Read Uncommitted	Möglich	Möglich	Möglich
Read Committed	Nicht möglich	Möglich	Möglich
Repeatable Read	Nicht möglich	Nicht möglich	Möglich
Serializable	Nicht möglich	Nicht möglich	Nicht möglich

Zeit ↓

Client 1	Client 2	Client 3	
Set Transaction Isolation Level Serializable;			*(a)*
Select nachname from kunde where id = 801			*(b)* → 'Voigt'
	update Kunde set nachname = 'Wehr' where id = 801;		*(c)*
		select nachname from kunde where id = 801;	*(d)* → 'Voigt'
	commit;		*(e)*
		select nachname from kunde where id = 801;	*(f)* → 'Wehr'
select nachname from kunde where id = 801;			*(g)* → 'Voigt'
commit;			*(h)*
select nachname from kunde where id = 801;			*(i)* → 'Wehr'

ABBILDUNG 6.4 Isolation-Level „Serializable" in einer Oracle-Datenbank

Projekt:

Als Ausgangspunkt für die Betrachtung der Konflikte des Mehrbenutzerbetriebs stellen wir dem Leser das Projekt `transaktionen-konkurrenz` mit den Java-Klassen `Client1Test` und `Client2Test` zur Verfügung. Mit den von uns theoretisch betrachteten Problemen des Mehrbenutzerbetriebs laden wir den Leser ein, diese nachzustellen, um anschließend gemeinsam Lösungen der JPA zu betrachten.

6.3 Transaktionen in der JPA

Transaktionen werden in Programmen entweder durch den Anwendungsentwickler bzw. sein Programm selbst oder aber durch den ausführenden Java-EE-Container verwaltet. Den ersten Fall haben wir schon in vielen vorangegangenen Beispielen verwendet, wir wollen ihn aber in diesem Kontext noch einmal genauer erläutern.

```
EntityManager em = null;
EntityTransaction tx = null;
try {
  em = emf.createEntityManager();
  tx = em.getTransaction();
  tx.begin();
  Query q = em.createQuery("select k from Girokonto k "
         + " where k.nummer = 1");
  Konto k = (Konto) q.getSingleResult();
```

```
    k.einzahlung(new BigDecimal("234.00"));
    tx.commit();
  } catch (RuntimeException e) {
    if (tx != null && tx.isActive()) {
      tx.rollback();
  }
  e.printStackTrace();
  // throw e;
  // Exception Handling an anderer Stelle
  // (Darstellung in Oberflaeche ggf. Abbruch)?
  }
  finally {
    em.close();
  }
```

Die Transaktion `tx` wird durch die Anwendung selbst verwaltet. Sie muss mit der Methode `begin()` begonnen und mit `commit()` abgeschlossen werden. Des Weiteren muss bei auftretenden Fehlern ein `rollback()` durchgeführt werden. Wir haben in diesem Beispiel zum ersten Mal ein Exception-Handling mit aufgenommen. JPA kennt die folgenden Arten von Ausnahmen:

- PersistenceException,
- EntityExistsException,
- EntityNotFoundException,
- LockTimeoutException,
- NonUniqueResultException,
- NoResultException,
- OptimisticLockException,
- PessimisticLockException,
- QueryTimeoutException,
- RollbackException,
- TransactionRequiredException.

Alle von einem Entity-Manager oder einem Query-Objekt ausgelösten Ausnahmen führen zum automatischen Rollback der aktuellen Transaktion. Nur die Klassen `NoResultException`, `NonUniqueResultException`, `LockedTimeoutException` und `QueryTimeoutException` lösen diesen Rollback nicht automatisch aus. Im obigen Beispiel wird die Exception abgefangen und die Transaktion manuell zurückgesetzt.

Das Beispiel soll nur zu Anschauungszwecken dienen. In größeren Projekten sind der Datenbankzugriff und das Exception-Handling an unterschiedlichen Stellen der Anwendungsarchitektur implementiert. Die `throw`-Klausel deutet an, dass die Ausnahme weiter nach *oben* gereicht wird.

Eine Transaktion der obigen Art wird als *Resource-Local* bezeichnet. Sie wird von der jeweiligen JPA-Implementierung bereitgestellt, jedoch von der Anwendung selbst verwaltet. Innerhalb eines Java-EE-Containers übernimmt diese Arbeiten der Container selbst. Das Programm muss sich nicht um die Bereitstellung, den Beginn, das Commit oder ggf. ein Rollback der Transaktion kümmern. Diese Art der Transaktionsverwaltung wird deklarative Verwaltung genannt. Der Entwickler muss einzig die Art der Transaktion mit der Annotation `@TransactionAttribute` deklarieren:

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void speichereKunde(Kunde k) {
  System.out.println("Verwaltet ? " + em.contains(k));
  em.persist(k);
}
```

Hier beschränkt sich die Transaktion auf den Kontext der Methode. Zu Beginn der Methode wird die Transaktion begonnen, mit Ende der Methode findet ein Commit in der Datenbank statt. Hier könnten auch mehrere Datenbanken (unter Verwendung mehrerer Entity-Manager) an einer Transaktion beteiligt sein. Der Persistenzkontext ist in diesem Fall transaktionsbezogen, was bedeutet: Mit Ende der Transaktion sind alle vom Kontext verwalteten Objekte im Status losgelöst. Weitere Informationen zu EJB-Transaktionen führen wir in Abschnitt 6.4 auf.

6.3.1 Konkurrierende Zugriffe

In der JPA-Version 1.0 konnte der Anwender nur das sogenannte optimistische Locking benutzen. Die Spezifikation setzte dabei voraus, dass sich die Transaktionen der zugrunde liegenden Datenbank im Isolation-Level `Read Committed` befinden.

Dieses optimistische Locking ist für Anwendungen sinnvoll, bei denen konkurrierende Zugriffe auf ein- und denselben Datensatz selten bis gar nicht auftreten. Gleichzeitige Zugriffe werden erkannt und entsprechend unterbunden, allerdings erst am Ende bzw. beim Schreiben der entsprechenden Entity-Daten. Mit der Version 2.0 ist darüber hinaus auch das pessimistische Locking möglich. Dabei werden Locks auf Zeilen der Datenbank gesetzt, die über einen gewissen Zeitraum bestehen bleiben können. Damit werden konkurrierende Zugriffe nicht erst am Ende der Transaktion, sondern bereits am Anfang erkannt.

Schauen wir uns jedoch zunächst ein Beispiel an, bei dem wir den gleichzeitigen Zugriff zweier Anwender auf den gleichen Datensatz simulieren. In der Klasse `Client1Test` haben wir mit der Methode `Thread.sleep(30000)` eine Wartezeit eingebaut. Die Main-Methode der anderen Klasse (`Client2Test`) sieht bis auf einen anderen Nachnamen und das Fehlen der `sleep()`-Methode genauso aus.

```
public class Client1Test {
  public static void main(String[] args) throws Exception {
    Provider.select("hibernate");
    EntityManagerFactory emf =
      Persistence.createEntityManagerFactory("bank", DB.h2);
    EntityManager em = emf.createEntityManager();
    Kunde k = em.find(Kunde.class, 1);
    k.setNachname("Schulze");
    // wir simulieren ein Warten
    Thread.sleep(30000);
    em.getTransaction().begin();
    em.persist(k);
    em.getTransaction().commit();
    em.close();
    emf.close();
  }
}
```

Was passiert, wenn wir die Java-Klasse `Client1Test` und kurz darauf `Client2Test` starten? Der *erste* Anwender liest den Kunden mit der Id 1 und ändert dessen Namen. Er bestätigt diese Änderung jedoch nicht sofort. Wir simulieren dieses Verhalten mit `sleep()`. Kurze Zeit später beginnt der Anwender 2 mit seiner Arbeit. Er liest den gleichen Datensatz, die Objekte befinden sich jedoch in unterschiedlichen Persistenzkontexten. Seine Änderung wird sofort ausgeführt und in die Datenbank geschrieben. Von dieser Änderung erfährt der erste Anwender jedoch nichts. Nachdem die Wartezeit beendet ist, bestätigt er die Änderung und überschreibt damit die bereits vorgenommene Änderung des zweiten Anwenders. Die Konsolen-Ausgaben der beiden Clients zeigen diesen Vorgang noch einmal sehr deutlich. Der Leser sollte besonders auf die Zeitstempel der einzelnen Vorgänge achten. Zur besseren Übersicht wurde das Select-Statement durch uns manuell angepasst. Die Selektion einzelner Spalten haben wir durch den Stern ersetzt:

```
run-Client1Test:
21:36:51,725 DEBUG SQL:111 - select * from Kunde kunde0_ where
    kunde0_.id=?
21:36:51,752 - binding parameter [1] as [INTEGER] - 1
21:36:51,795 - found [1985-10-04] as column [geburtsd2_0_0_]
21:36:51,798 - found [m] as column [geschlecht0_0_]
21:36:51,799 - found [928139] as column [kundennu4_0_0_]
21:36:51,800 - found [Thomas] as column [nachname0_0_]
21:36:51,800 - found [Maximilian] as column [vorname0_0_]
21:36:51,803 - found [1] as column [WOHNORT7_0_0_]
21:37:21,887 DEBUG SQL:111 -
               update Kunde set geburtsdatum=?, geschlecht=?,
               kundennummer=?, nachname=?, vorname=?, WOHNORT_ID=?
               where id=?
21:37:21,891 - binding parameter [1] as [DATE] - 1985-10-04
21:37:21,895 - binding parameter [2] as [CHAR] - m
21:37:21,896 - binding parameter [3] as [INTEGER] - 928139
21:37:21,897 - binding parameter [4] as [VARCHAR] - Schulze
21:37:21,898 - binding parameter [5] as [VARCHAR] - Maximilian
21:37:21,901 - binding parameter [6] as [INTEGER] - 1
21:37:21,902 - binding parameter [7] as [INTEGER] - 1

run-Client2Test:
21:37:01,361 DEBUG SQL:111 - select * from Kunde kunde0_ where
    kunde0_.id=?
21:37:01,389 - binding parameter [1] as [INTEGER] - 1
21:37:01,428 - found [1985-10-04] as column [geburtsd2_0_0_]
21:37:01,430 - found [m] as column [geschlecht0_0_]
21:37:01,432 - found [928139] as column [kundennu4_0_0_]
21:37:01,432 - found [Thomas] as column [nachname0_0_]
21:37:01,432 - found [Maximilian] as column [vorname0_0_]
21:37:01,433 - found [1] as column [WOHNORT7_0_0_]
21:37:01,504 DEBUG SQL:111 -
               update Kunde set geburtsdatum=?, geschlecht=?,
               kundennummer=?, nachname=?, vorname=?, WOHNORT_ID=?
               where id=?
21:37:01,509 - binding parameter [1] as [DATE] - 1985-10-04
21:37:01,511 - binding parameter [2] as [CHAR] - m
21:37:01,512 - binding parameter [3] as [INTEGER] - 928139
21:37:01,512 - binding parameter [4] as [VARCHAR] - Meier
21:37:01,512 - binding parameter [5] as [VARCHAR] - Maximilian
21:37:01,514 - binding parameter [6] as [INTEGER] - 1
21:37:01,515 - binding parameter [7] as [INTEGER] - 1
```

Dieser Vorgang zeigt uns zwei Probleme, für die eine Lösung bereitstehen muss:

1. Die Änderungen des zweiten Anwenders werden ohne Fehlermeldung überschrieben. Dieses Problem wird in der Literatur auch als Lost-Update-Problem beschrieben.
2. Anwender 1 führt seine Änderung durch, dabei ging er aber von einem anderen Sachverhalt aus. Wenn er von dem zwischenzeitlichen Update erfahren hätte, würde er mit hoher Wahrscheinlichkeit von seiner Änderung absehen.

Wir sprechen an dieser Stelle von „Problemen". Gleichwohl kann es eine Strategie einer Anwendung sein, konkurrierende Zugriffe nicht zu behandeln und Situationen wie die obige zuzulassen. Dieses Verfahren kann beim Benutzer zu einer gewissen Verwirrung führen, der letzte Commit gewinnt immer, ohne dass auch nur eine Fehlermeldung erscheint.

Entscheidet sich der Entwickler dafür, die beschriebenen potenziellen Probleme zu behandeln, stehen ihm verschiedene Möglichkeiten des Lockings zur Verfügung. Wir beschreiben Sie in den nachfolgenden Abschnitten.

Projekt:

Unsere nachfolgenden Programme findet der Leser in den Eclipse-Projekten `transaktionen-konkurrenz-optimistic` und `transaktionen-konkurrenz-pessimistic` zum eigenständigen Nachvollziehen der Lösungsansätze von JPA im Mehrbenutzerbetrieb.

6.3.2 Optimistisches Locking

Wie bereits erwähnt, waren die Lock-Möglichkeiten in JPA-Version 1.0 eingeschränkt. Die alte Spezifikation kannte die beiden Modi `LockModeType.READ` und `LockModeType.WRITE`. Wenn Sie einen Blick in die API der JPA 2.0 werfen, werden Sie erkennen, dass aus Kompatibilitätsgründen diese Varianten des Lockings von der Version 1.0 übernommen wurden. Ihnen äquivalent sind die neuen Lock-Varianten `LockModeType.Optimistic` und `LockModeType.OPTIMISTIC_FORCE_INCREMENT`. Nur diese sollte der Leser in seinen neuen Projekten verwenden. Darüber hinaus lohnt es sich, auch auf die Möglichkeiten des pessimistischen Lockings zurückzugreifen. Diese Varianten beschreiben wir im Abschnitt 6.3.3.

6.3.2.1 LockModeType.OPTIMISTIC

Das optimistische Locking geht, wie der Name schon sagt, davon aus, dass *alles gut geht*. Zu Beginn einer Transaktion wird deshalb auch kein Objekt tatsächlich gesperrt. Vielmehr wird sein Zustand in einer Variablen bzw. einem Zähler gespeichert. Erst beim Schreiben eines Objektes vergleicht die JPA-Implementierung die Inhalte des Zählers der Transaktion mit dem der Datenbank und zeigt einen zwischenzeitlichen Zugriff einer anderen Transaktion an.

Der erwähnte Zähler wird mit Hilfe der `@Version`-Annotation bereitgestellt. Diese Annotation sollte bei allen Entities verwendet werden, bei denen ein konkurrierender Zugriff wahrscheinlich ist bzw. Entities im Laufe ihres Lebens losgelöst und mit `merge()` dem Kontext wieder hinzugefügt werden.

`@Version` kann auf einem Entity-Attribut mit folgendem Datentyp verwendet werden: `int`, `Integer`, `short`, `Short`, `long`, `Long` sowie `Timestamp`. Die Anwendung sollte niemals

selbst den Inhalt dieser Variable ändern. Ausnahmen von dieser Regel stellen nur Batch-Vorgänge dar, die wir in Abschnitt 7.6 behandeln.

Wir können unser Beispiel aus dem vorangegangenen Abschnitt erneut verwenden. Die Test-Clients bleiben unverändert. Lediglich unsere Klasse `Kunde` wurde um ein Property `version` ergänzt, das mit `@Version` annotiert wurde. Listing 6.1 zeigt die Klasse `Kunde` mit den ausgezeichneten Attributen.

LISTING 6.1 `Kunde`-Klasse mit `@Version`-Attribut

```
@Entity
public class Kunde implements Serializable {
  @Id
  @GeneratedValue
  private Integer id;
  @Version
  private Integer version;
  private Integer kundennummer;
  private String vorname;
  private String nachname;
  @Temporal(TemporalType.DATE)
  private Date geburtsdatum;
  private Character geschlecht;
  @OneToOne(cascade = CascadeType.ALL)
  @JoinColumn(name = "WOHNORT_ID")
  private Adresse wohnort;
  @ManyToMany(cascade = CascadeType.ALL)
  private Set<Konto> konten;

  // ... Getter und Setter
}
```

Anwender, die ihr DB-Schema nicht automatisiert erstellen, müssen die `Kunde`-Tabelle um die entsprechende Spalte für die Versions-Information ergänzen.

Was passiert nun, wenn wir wiederum die Klasse `Client1` und kurz danach `Client2` ausführen? Der erste Anwender liest den entsprechenden Datensatz des Kunden sowie den Inhalt der Spalte `version`. Im initialen Zustand ist dieser Wert 0, wie man im Listing erkennen kann. Client 1 führt seine Operation zunächst nur auf dem Objekt in der JVM aus, durch die `sleep()`-Methode wird die Änderung noch nicht in die Datenbank geschrieben.

```
run-Client1Test:
[java] Hibernate: select * from Kunde kunde0_ where kunde0_.id=?
[java] 22:30:00,836 - binding parameter [1] as [INTEGER] - 1
[java] 22:30:00,875 - found [1985-10-04] as column [geburtsd2_0_0_]
[java] 22:30:00,877 - found [m] as column [geschlecht0_0_]
[java] 22:30:00,879 - found [928139] as column [kundennu4_0_0_]
[java] 22:30:00,879 - found [Wehr] as column [nachname0_0_]
[java] 22:30:00,879 - found [0] as column [version0_0_]
[java] 22:30:00,879 - found [Maximilian] as column [vorname0_0_]
[java] 22:30:00,880 - found [1] as column [WOHNORT8_0_0_]
```

Kurze Zeit später liest der zweite Anwender denselben Datensatz, auch hier ist der Inhalt des `version`-Feldes 0. Wir erinnern uns: Das Programm von Client 1 hat seine Transaktion

noch nicht committed, alle Änderungen sind im Arbeitsspeicher verblieben und für andere Benutzer unsichtbar.

```
run-Client2Test:
[java] Hibernate: select * from Kunde kunde0_ where kunde0_.id=?
[java] 22:30:08,061  - binding parameter [1] as [INTEGER] - 1
[java] 22:30:08,099  - found [1985-10-04] as column [geburtsd2_0_0_]
[java] 22:30:08,101  - found [m] as column [geschlecht0_0_]
[java] 22:30:08,102  - found [928139] as column [kundennu4_0_0_]
[java] 22:30:08,103  - found [Wehr] as column [nachname0_0_]
[java] 22:30:08,103  - found [0] as column [version0_0_]
[java] 22:30:08,103  - found [Maximilian] as column [vorname0_0_]
[java] 22:30:08,104  - found [1] as column [WOHNORT8_0_0_]

[java] Hibernate: update Kunde set geburtsdatum=?, geschlecht=?,
       kundennummer=?, nachname=?, version=?, vorname=?,
       WOHNORT_ID=? where id=? and version=?
[java] 22:30:08,168 - binding parameter [1] as [DATE] - 1985-10-04
[java] 22:30:08,170 - binding parameter [2] as [CHAR] - m
[java] 22:30:08,171 - binding parameter [3] as [INTEGER] - 928139
[java] 22:30:08,171 - binding parameter [4] as [VARCHAR] - Meier
[java] 22:30:08,171 - binding parameter [5] as [INTEGER] - 1
[java] 22:30:08,172 - binding parameter [6] as [VARCHAR] - Maximilian
[java] 22:30:08,173 - binding parameter [7] as [INTEGER] - 1
[java] 22:30:08,174 - binding parameter [8] as [INTEGER] - 1
[java] 22:30:08,174 - binding parameter [9] as [INTEGER] - 0
```

Da Anwender 2 jedoch seine Änderung gleich committed, finden sich seine Änderungen in der Datenbank wieder. Bei dieser Operation wird die `version`-Spalte des betroffenen Kunden um 1 erhöht. Parameter 5 enthält die *Version* des neuen geschriebenen Objektes mit dem Wert 1.

Beendet nun der erste Kunde seine Transaktion, kommt es zu einer Exception. Das Programm `Client1` hat die Änderungen des anderen Benutzers nicht wahrgenommen. Sein optimistischer Lock erkennt erst beim Update die zwischenzeitliche Änderung. Doch wie entdeckt dieser Lock diese Änderung? Der Leser wird erkennen, dass nach der Erweiterung unseres Entities `Kunde` mit der `@Version`-Annotation die `Where`-Bedingung nicht mehr nur die Id des Kunden als eindeutigen Schlüssel, sondern auch das zuvor gelesene `Version`-Attribut enthält.

```
run-Client1Test:
[java] Hibernate: update Kunde set geburtsdatum=?, geschlecht=?,
       kundennummer=?, nachname=?, version=?, vorname=?,
       WOHNORT_ID=? where id=? and version=?
[java] 22:30:30,960 - binding parameter [1] as [DATE] - 1985-10-04
[java] 22:30:30,963 - binding parameter [2] as [CHAR] - m
[java] 22:30:30,964 - binding parameter [3] as [INTEGER] - 928139
[java] 22:30:30,965 - binding parameter [4] as [VARCHAR] - Schulze
[java] 22:30:30,966 - binding parameter [5] as [INTEGER] - 1
[java] 22:30:30,967 - binding parameter [6] as [VARCHAR] - Maximilian
[java] 22:30:30,970 - binding parameter [7] as [INTEGER] - 1
[java] 22:30:30,971 - binding parameter [8] as [INTEGER] - 1
[java] 22:30:30,972 - binding parameter [9] as [INTEGER] - 0

Exception in thread "main" javax.persistence.RollbackException:
Error while committing the transaction
at org.hibernate.ejb.TransactionImpl.commit(TransactionImpl.java:93)
```

```
at de.jpainfo.test.Client1Test.main(Client1Test.java:31)
Caused by: javax.persistence.OptimisticLockException:
org.hibernate.StaleObjectStateException:
Row was updated or deleted by another transaction
```

Allerdings ist mit dieser `Where`-Bedingung kein Datensatz mehr für ein Update zu finden, da die gesuchte Zeile inzwischen vom zweiten Programm geändert wurde. Das Programm `Client1` bricht seine Verarbeitung mit einer Ausnahme ab und führt einen Rollback der Transaktion aus.

6.3.2.2 LockModeType.OPTIMISTIC_FORCE_INCREMENT

Mit diesem Lock-Type muss eine JPA-Implementierung im Grunde die gleichen Sachverhalte ausschließen wie der bereits beschriebene Typ `LockModeType.OPTIMISTIC`. Diese sind im Einzelnen

- *Dirty Reads* und
- *Non-repeatable Reads.*

Darüber hinaus muss der Entity-Manager auf jeden Fall den Zähler (`@Version`-Attribut) des betroffenen Entity um 1 erhöhen. Dies geschieht unabhängig davon, ob das Entity in der Transaktion verändert wurde oder nicht. Diese Einstellung kann z. B. verwendet werden, um die Zugriffshäufigkeit auf bestimmte Entities zu protokollieren.

6.3.3 Pessimistisches Locking

Das pessimistische Locking versucht bereits am Anfang einer Transaktion bzw. am Anfang des Setzens einer Sperre den Datensatz direkt auf der Datenbank zu sperren. Damit soll erreicht werden, dass ein gleichzeitiger Zugriff mehrerer Transaktionen auf einen Datensatz gleich zu Beginn des konkurrierenden Zugriffs und nicht erst beim Update des Satzes erkannt wird.

Voraussetzung für eine konsistente Verwendung dieser Lock-Modi ist natürlich die Unterstützung der zugrunde liegenden Datenbank. Wir werden in unseren Beispielen zeigen, dass z. B. ein zeilenbasierter angeforderter Lock nicht immer nur eine Zeile einer Datenbank, sondern bei bestimmten Datenbank-Herstellern gleich zum Lock der gesamten Tabelle führt. Die Abstraktion der JPA von der zugrunde liegenden Datenbank führt nicht immer zu einem deterministischen Verhalten beim Wechsel der Datenbank für eine Anwendung. Dem Leser wird geraten, die Eigenschaften einer Datenbank vor deren Einsatz genau zu untersuchen.

Eine JPA-Implementierung muss das pessimistische Locking auf Entities mit `@Version`-Attribut und auch ohne dieses unterstützen.

Tipp:

Dem Leser wird in unseren nachfolgenden Beispielen auffallen, dass obwohl das pessimistische Locking eine Sperre direkt auf der Datenbank setzt, auch das `@Version`-Attribut bei einer Veränderung des Entity inkrementiert wird. Dieses Verhalten wird von der JPA-Spezifikation [URL-JPA2] ausdrücklich gefordert, um ein möglichst korrektes Verhalten mit Applikationen zu ermöglichen, die sich rein auf das optimistische Locking verlassen.

6.3.3.1 LockModeType.PESSIMISTIC_READ

Natürlich muss auch der Modus des pessimistischen Lesens wie beim optimistischen Lesen die Phänomene des *Dirty Read* und *Non-repeatable Reads* ausschließen. Der Modus stellt also sicher, dass eine Transaktion während der gesamten Laufzeit davon ausgehen kann, dass sich die Daten eines Entity nicht verändern. Dabei geht die Spezifikation aber von einem sogenannten *Shared Lock* aus. Andere Transaktionen, die ebenfalls nur ein *Repeatable Read* sicherstellen wollen, können ebenfalls ein Lock für das Entity setzen und diesen bekommen. Existiert auf einem Entity bereits ein Lock mit dem Modus `PESSISMISTIC_READ`, können andere Transaktionen ebenfalls nur noch einen Lock mit `PESSIMISTIC_READ` setzen.

6.3.3.2 LockModeType.PESSIMISTIC_WRITE und LockModeType.PESSIMISTIC_FORCE_INCREMENT

Der Lock-Modus `PESSIMISTIC_WRITE` setzt einen exklusiven Lock auf der Datenbank. Anderen Transaktionen ist nicht erlaubt, auch nur einen weiteren Lock zu setzen. Selbst ein Lock im Modus `PESSIMISTIC_READ` wird nicht zugelassen, da sonst die Eigenschaft des *Repeatable Reads* der zweiten Transaktion nicht sichergestellt wäre. `PESSIMISTIC_FORCE_INCREMENT` verhält sich genauso, zusätzlich muss die Implementierung zwingend den Zähler des `@Version`-Attributs erhöhen. Hat das Entity keine `@Version`-Annotation, muss dies durch eine Exception angezeigt werden.

6.3.3.3 Pessimistische Locks in der Praxis

Die Spezifikation überlässt es einer Implementierung, ob sie alle Lock-Modi unterstützt. Statt des `PESSIMSTIC_READ` darf eine Implementierung auch `PESSIMISTIC_WRITE` verwenden bzw. statt eines `PESSIMISTIC_WRITE` kann auch ein `PESSIMISTIC_FORCE_INCREMENT` verwendet werden. Die umgekehrte Richtung ist nicht erlaubt.

Doch der Blick auf die JPA-Implementierung allein reicht an dieser Stelle nicht mehr aus. Da beim pessimistischen Locking die Sperre sofort auf der zugrunde liegenden Datenbank ausgeführt wird, ist auch das Verhalten der Datenbank zu betrachten. Erlaubt der SQL-Dialekt eine Zeilen- oder gar Feld-basierte Sperrmöglichkeit? Oder aber: Führt eine Schreibsperre gar zum Sperren der gesamten Tabelle? Dem Leser muss an dieser Stelle Folgendes bewusst werden: Die JPA-Spezifikation abstrahiert mit ihrer Definition vom verwendeten Datenbanksystem, werden aber wie beim pessimistischen Locking Sperren direkt auf der Datenbank gesetzt, ist das Ergebnis von der Unterstützung der verwendeten Datenbank abhängig.

Wir wollen das pessimistische Locking zunächst mit Hibernate und unserer H2-Datenbank untersuchen. Dazu verändern wir den Code unseres vorangegangenen Beispiels leicht und benutzen nun statt des optimistischen Lockings den Modus `PESSIMISTIC_READ`.

```
EntityManagerFactory emf = Persistence
    .createEntityManagerFactory("bank", DB.h2);
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Kunde k = em.find(Kunde.class, 1, LockModeType.PESSIMISTIC_READ);
k.setNachname("Schulze");
// wir simulieren ein Warten
Thread.sleep(30000);
em.persist(k);
em.getTransaction().commit();
```

Das Programm `Client2` sieht bis auf die `sleep`-Anweisung genauso aus. Werden nun beide Programme zeitgleich ausgeführt, beendet sich das zweite Programm sofort mit einer Exception. Warum ist das so? Wenn der Leser sich das abgesetzte SQL-Statement von Programm 1 genauer ansieht, wird er erkennen, dass Hibernate den Zusatz `for update` am Ende ausführt. Weil die H2-Datenbank noch keinen Shared-Modus für einen lesenden Modus kennt, weicht der von Hibernate gewählte Datenbank-Dialekt auf diese exklusive Sperre aus.

Doch selbst wenn der Leser den Quellcode von Programm 2 dahingehend ändert, dass hier ein Kunde mit einer ganz anderen ID gesperrt werden soll, tritt weiterhin eine Exception auf. Mit der H2-Version 1.3 werden Locks auf Zeilen noch nicht im Standard unterstützt. Erst mit der (noch) experimentellen Aktivierung der so genannten *Multi Version Concurrency* wird die exklusive Tabellensperre beim Befehl `select ... for update` nicht mehr benutzt. Dies muss bereits beim Starten der Datenbank mit dem Zusatz `MVCC=TRUE` erfolgen. Erfolgt dies nicht, führt der Befehl `for update` zum Sperren der gesamten Tabelle!

Weichen wir also mit unseren Beispielen auf eine Oracle-Datenbank aus. Diese unterstützt die zeilenbasierte Sperre schon seit Langem. Das (von uns gekürzte) abgesetzte SQL-Statement hat die gleiche Struktur wie das Richtung H2 abgesetzte Statement. Allerdings führt diese Selektion tatsächlich nur zum Sperren des durch die `where`-Bedingung referenzierten Satzes.

```
select * from Kunde kunde0_ where kunde0_.id=? for update
```

Ein Blick in die Dokumentation von Oracle verrät uns, dass wir mit diesem Befehl auch einen exklusiven Lock auf die selektierte Zeile bekommen, obwohl wir in unseren Transaktionen doch eine Sperre zum Lesen angefordert hatten.

Starten wir zeitgleich Client 2, sollte eine Exception geworfen werden. Zwei Transaktionen versuchen zeitgleich, den gleichen Satz zu sperren. Doch genau dies passiert nicht. Client 2 *scheint* so lange zu warten, bis die erste Transaktion beendet ist. Erst danach kann er mit seinem Update weitermachen. Doch ist hier nur das Update blockiert oder auch bereits das Lesen des Satzes? Schließlich müssen die JPA-Implementierung in Verbindung mit der Datenbank sicherstellen, dass auch *Dirty Reads* unmöglich sind. Liefert die Datenbank also richtigerweise die alten Daten aus oder bereits die neuen? Ein genauer Blick auf die Timestamps der Ausgaben zeigt uns die Vorgehensweise:

```
select * from Kunde kunde0_ where kunde0_.id=? for update
20:01:53,864 - binding parameter [1] as [INTEGER] - 1
20:01:53,934 - found [1985-10-04] as column [geburtsd2_0_0_]
20:01:53,937 - found [m] as column [geschlecht0_0_]
20:01:53,938 - found [928139] as column [kundennu4_0_0_]
20:01:53,938 - found [Schulze] as column [nachname0_0_]
20:01:53,939 - found [11] as column [version6_0_0_]
20:01:53,939 - found [Maximilian] as column [vorname0_0_]
20:01:53,939 - found [1] as column [WOHNORT8_0_0_]
```

Um 20:01:53 Uhr bekommt die Transaktion einen exklusiven Lock auf den Datensatz mit der Id 1. Transaktion 2, die ein wenig später den gleichen Datensatz sperren will, bekommt dagegen keine Antwort vom DBMS; sie verbleibt in Wartestellung:

```
select * from Kunde kunde0_ where kunde0_.id=? for update
20:02:00,326 - binding parameter [1] as [INTEGER] - 1
```

Nach dem Ende unserer simulierten Wartezeit beendet Client1 die Transaktion, indem die Änderungen mit einem Commit in die Datenbank geschrieben werden.

```
update Kunde set geburtsdatum=?, geschlecht=?, kundennummer=?,
  nachname=?, version_id=?, vorname=?, WOHNORT_ID=? where id=? and
      version_id=?
20:02:23,999 - binding parameter [1] as [DATE] - 1985-10-04
20:02:24,002 - binding parameter [2] as [CHAR] - m
20:02:24,003 - binding parameter [3] as [INTEGER] - 928139
20:02:24,004 - binding parameter [4] as [VARCHAR] - Schulze1
20:02:24,006 - binding parameter [5] as [INTEGER] - 12
20:02:24,009 - binding parameter [6] as [VARCHAR] - Maximilian
20:02:24,012 - binding parameter [7] as [INTEGER] - 1
20:02:24,013 - binding parameter [8] as [INTEGER] - 1
20:02:24,014 - binding parameter [9] as [INTEGER] - 11
```

Erst nach dem Ende von Client 1 bekommt die zweite Transaktion einen Lock auf den gleichen Satz. Die zu lesenden Informationen werden erst nach Ende von Transaktion 1 bereitgestellt. Ein *Dirty Read* ist somit ausgeschlossen.

```
20:02:24,175 - found [1985-10-04] as column [geburtsd2_0_0_]
20:02:24,178 - found [m] as column [geschlecht0_0_]
20:02:24,180 - found [928139] as column [kundennu4_0_0_]
20:02:24,180 - found [Schulze1] as column [nachname0_0_]
20:02:24,181 - found [12] as column [version6_0_0_]
20:02:24,182 - found [Maximilian] as column [vorname0_0_]
20:02:24,182 - found [1] as column [WOHNORT8_0_0_]
20:02:24,216 - update Kunde set geburtsdatum=?, geschlecht=?,
 kundennummer=?, nachname=?, version_id=?, vorname=?, WOHNORT_ID=?
 where id=? and version_id=?
 ...
```

Wie kommt es nun dazu, dass die zweite Transaktion auf die Freigabe wartet. Die JPA-Spezifikation erwähnt hierfür den Parameter

```
javax.persistence.lock.timeout // time in milliseconds
```

Die Dauer dieses Timeouts bestimmt, wie lange eine Transaktion auf einen Lock warten soll. Ist dieser Wert nicht angegeben, wird automatisch der vom jeweiligen Datenbanksystem gesetzte Timeout genutzt. Der Parameter kann innerhalb der Datei `persistence.xml` gesetzt oder als Property der Methode `Persistence.createEntityManagerFactory` mitgegeben werden. Darüber hinaus kann dieses Verhalten mit Methoden des `EntityManager` bzw. des `Query`-Interfaces überschrieben werden.

Ein Wert von 0 sollte ein Statement generieren, dass keinerlei Wartezeit zulässt. Ergänzen wir nun das Programm von Client 2 um dieses Property, welches anschließend der Lock-Methode übergeben wird:

```
Map<String, Object> properties = new HashMap<String, Object>();
properties.put( "javax.persistence.lock.timeout", 0L );

Kunde k = em.find(Kunde.class, 1, LockModeType.PESSIMISTIC_READ,
    properties);
```

Das daraus resultierende SQL-Statement inklusive der Programm-Ausgabe ergibt sich wie folgt:

```
select * from Kunde kunde0_ where kunde0_.id=? for update nowait
21:39:49,874 - binding parameter [1] as [INTEGER] - 1
21:39:49,898 ERROR JDBCExceptionReporter:234 - ORA-00054: Versuch,
    mit
  NOWAIT eine bereits belegte Ressource anzufordern
Exception in thread "main" javax.persistence.PersistenceException:
 org.hibernate.exception.LockAcquisitionException: could not load an
 entity: [de.jpainfo.model.Kunde#1]
```

Wie der Leser erkennt, wird nun bei gleichzeitigem Ausführen zweier Transaktionen die letztere mit einer Ausnahme sofort beendet. Client 2 wurde angewiesen, nicht auf den exklusiven Lock zu warten, und beendet sich mit der `javax.persistence.PersistenceException`.

Tipp:

Oracle kennt zwar einen Shared Mode, der auch zeilenbasiert angewendet werden kann. Hibernate nutzt ihn aber an dieser Stelle nicht. `PESSIMISTIC_READ` und `PESSIMISTIC_WRITE` werden in der gleichen Art und Weise mit einem `select ... for update` angewendet. Dies ist laut Spezifikation auch ausdrücklich erlaubt. Das in einer Oracle-Datenbank mögliche *Shared Locking* mit dem Befehl `Lock Table ... in Row Share Mode` kommt nicht zur Anwendung.

Kommen wir nun zu unserem letzten Beispiel des pessimistischen Lockings, in dem wir unsere Transaktionen auf einer Postgres-Datenbank absetzen. Dabei versuchen wir zunächst wieder einen Shared Lock zu erzeugen. Dies wollen wir mit dem bereits bekannten Lock-Modus `PESSIMISTIC_READ` im Programm von Client 1 erreichen. Mit der zweiten Transaktionen versuchen wir kurz danach, eine exklusive Sperre zu bekommen. Die erste Transaktion benutzt den von Postgres implementierten *Shared Modus* mit dem Befehl `select ... for share`. Damit sind weitere lesende Transaktionen im Modus *Shared* zugelassen. Darüber hinaus muss die erste Transaktion nicht noch einmal die Daten lesen, wenn sie sich später entscheidet, die Daten doch noch zu ändern. Je nachdem, wie der Leser die zweite Transaktion startet, beendet sich diese entweder mit einer Exception (bei `timeout=0`) oder aber führt seine Änderungen nach dem Ende der ersten Transaktion durch.

Wir verzichten an dieser Stelle auf weitere Beispiele mit den anderen JPA-Implementierungen. Der Leser wird erkennen, wie vielfältig das Verhalten der jeweiligen Implementierung bei den verschiedenen Datenbanken sein kann. Weil das pessimistische Locking eine Sperre direkt auf der Datenbank setzt, beeinflusst die verwendete Datenbank ganz offensichtlich das Verhalten des entwickelten Programms. Dafür erhält der Entwickler die Gewissheit, gleich am Anfang einer Transaktion eine (exklusive) Sperre auf den zu ändernden Datensatz zu bekommen. Bei häufigen konkurrierenden Zugriffen kann dies ein entscheidender Vorteil sein. Sind konkurrierende Zugriffe eher selten, hat das optimistische Locking klare Vorteile, da es zwischenzeitliche Änderungen anhand der `@Version`-Annotation erkennt. Dieser Vorgang vollzieht sich nahezu unabhängig von der verwendeten Datenbank mittels einer einfachen `select ... where`-Bedingung ohne weitere Zusätze.

Tipp:

Was eine JPA-Implementierung im Falle eines bestimmten Lock-Modus und einer bestimmten Datenbank letztendlich auf der Datenbank macht, ist nicht immer leicht zu erkennen. Dies liegt zum einen daran, dass die Implementierungen nicht immer sauber zwischen JPA-Lock-Modi und eigenen Erweiterungen trennen. Darüber hinaus sind die API-Dokumentationen oft nicht eindeutig. Letzter Ausweg kann dann nur noch ein Blick in den Quellcode der Implementierung bzw. in den generierten SQL-Code sein.

6.3.4 Locks für Objekte erhalten

In diesem Abschnitt wollen wir dem Leser noch die verschiedenen Möglichkeiten vorstellen, eine Sperre für ein Entity zu setzen. Dabei spielt es keine Rolle, ob es sich um eine optimistische oder pessimistische Sperre handelt.

Eine Methode ist bereits hinlänglich aus den vorangegangenen Beispielen dieses Kapitels bekannt. Durch das Überladen der Instanz-Methode `find` im Entity-Manager können nicht nur der Lock-Modus gesetzt, sondern darüber hinaus weitere Parameter übergeben werden, die das Verhalten der Transaktion beeinflussen.

```
public <T> T find(Class<T> entityClass,
  Object primaryKey,
  LockModeType lockMode);

public <T> T find(Class<T> entityClass,
  Object primaryKey,
  LockModeType lockMode,
  Map<String, Object> properties);
```

Über den Parameter `properties` lassen sich sowohl JPA-Properties als auch darüber hinausgehende herstellerbezogene Lock-Eigenschaften setzen. Diese dürfen allerdings nicht mit dem Präfix `javax.persistence` beginnen. Die JPA-Spezifikation definiert derzeit die folgenden Properties:

javax.persistence.lock.scope

Der Scope legt fest, wie *weit* eine Sperre auf der Datenbank gelten soll. Mögliche Werte sind in der Enumeration `PessimisticLockScope` festgelegt. Der Modus `EXTENDED` sperrt im Gegensatz zum Modus `NORMAL` auch Beziehungen zu anderen Entities, die z. B. in Join-Tabellen abgelegt sind. Die abhängigen Entities werden nicht mitgesperrt. Diese müssen durch die Anwendung selbst explizit gesperrt werden.

javax.persistence.lock.timeout

Wie bereits oben dargestellt, bestimmt dieser Parameter die Zeit in Millisekunden, die auf die Bereitstellung einer Sperre gewartet werden soll.

Weitere Sperr-Methoden in der Klasse `EntityManager` sind die Methoden `lock` und `refresh`. Während `lock` ein im Kontext befindliches Entity zu sperren versucht, überschreibt `refresh` bereits durchgeführte Änderungen im Kontext, indem es das Entity erneut von der Datenbank liest und sperrt. Die Methoden sind analog zur `find`-Methode überladen:

```
public void lock(Object entity,
  LockModeType lockMode);
public void lock(Object entity,
  LockModeType lockMode,
  Map<String, Object> properties);
public void refresh(Object entity,
  LockModeType lockMode);
public void refresh(Object entity,
  LockModeType lockMode,
  Map<String, Object> properties);
```

Der Vollständigkeit halber erwähnen wir an dieser Stelle noch die Sperrmöglichkeiten über das Interface `Query` sowie die `@NamedQuery`-Annotation. Während bei Ersterer die Methode `setLockMode()` zum Einsatz kommt, wird bei Verwendung der Annotation das Sperrverhalten über das Attribut `lockMode` gesteuert. Auf beide Elemente gehen wir noch ausführlich in Kapitel 7 ein.

6.3.5 Caching

In der JPA-Version 2.0 wurde das Caching von Objekten in begrenztem Umfang standardisiert. Damit ist nicht das Zwischenspeichern von Objekten im Rahmen des Entity-Managers einer Persistenzeinheit gemeint. Dies ist der so genannte *First Level Cache*. Eine Instanz der Klasse `EntityManager` liefert uns bei zwei Abfragen nach dem gleichen Schlüssel-Attribut das identische Objekt zurück. Die folgende Abfrage unseres Programms liefert bei Prüfung auf Objektidentität also `true`.

```
EntityManager em = emf.createEntityManager();
Kunde k1 = em.find(Kunde.class, 1);
Kunde k2 = em.find(Kunde.class, 1);
System.out.println(k1 == k2);
System.out.println(k1);
System.out.println(k2);
em.close();
```

Die korrespondierende Ausgabe stellt sich wie folgt dar. Dies sollte den Leser an dieser Stelle des Buches nicht weiter überraschen. Im Kapitel 3 wurde das Verhalten bereits beschrieben.

```
true
de.jpainfo.model.Kunde@13c2d7f
de.jpainfo.model.Kunde@13c2d7f
```

Soll über den Persistenzkontext hinaus das Zwischenspeichern von Objekten aktiviert werden, muss der *Second Level Cache* (L2-Cache) der Implementierung benutzt werden. Die Spezifikation verlangt von einem JPA-Provider nicht, dass er diese Art des Zwischenspeicherns überhaupt implementiert. Alle nachfolgend beschriebenen Anweisungen im Quellcode sind vom Provider dann komplett zu ignorieren.

Projekt:

Beispiele zum `First-` und `Second-Level-Cache` sind im Eclipse-Projekt `jpacache` zu finden.

Aktivieren des Cache

Um den Second-Level-Cache zu aktivieren, wird das XML-Element `<shared-cache-mode>` in der Datei `persistence.xml` benutzt. Das Element kann folgende Werte annehmen:

`ALL`
: Alle Entities und deren relevante Zustandsdaten werden im Cache zwischengespeichert.

`NONE`
: Das Zwischenspeichern ist für die gesamte Persistenzeinheit untersagt.

`ENABLE_SELECTIVE`
: Das Zwischenspeichern in der Persistenzeinheit ist erlaubt. Dabei werden jedoch nur die Entities berücksichtigt, die mit der Annotation `@javax.persistence.Cacheable(true)` gekennzeichnet sind. Diejenigen Entities, die keine `@Cachable`-Annotation haben bzw. mit `@Cacheable(false)` markiert sind, werden nicht zwischengespeichert.

`DISABLE_SELECTIVE`
: Um nur wenige ausgewählte Entities vom Caching auszuschließen, lässt sich `DISABLE_SELECTIVE` verwenden. Die Zwischenspeicherung wird nur für jene Entities ausgelassen, die explizit mit der Annotation `@Cacheable(false)` versehen sind. Nicht annotierte Entities unterliegen der Ablage im Zwischenspeicher.

`UNSPECIFIED`
: Diese Option ist von der Spezifikation dafür vorgesehen, provider-spezifische Default-Einstellungen heranzuziehen. Dies gilt auch dann, wenn die Anweisung `<shared-cache-mode>` komplett fehlt.

Hinweis:

Um sicherzugehen, dass die JPA-Implementierungen den Second-Level-Cache überhaupt nicht benutzen, sollte der Leser den Schalter `<shared-cache-mode>` mit `NONE` explizit setzen. Damit sind eventuelle Standard-Einstellungen der Provider entsprechend der Spezifikation deaktiviert.

Der Second-Level-Cache ist unabhängig von einzelnen `EntityManager`-Instanzen und deren Transaktionen. Weil er für einen Persistenzkontext konfiguriert wird, muss der Cache somit über den Kontext selbst bzw. dessen Factory der einzelnen Manager erreichbar sein. Die Instanz der Klasse `EntityManagerFactory` liefert uns mit der Methode `getCache()` eine Instanz von `javax.persistence.Cache` einen von der Implementierung unabhängigen Zugriff auf den Zwischenspeicher mit folgenden Methoden:

`contains(Class c, Object o)`
: Der Rückgabewert ist `true`, wenn der Cache für die angegebene Entity-Klasse und das Schlüsselattribut eine Entity-Instanz enthält.

`evict(Class c)`
: Löscht alle Instanzen der übergebenen Klasse aus dem Speicher.

`evict(Class c, Object o)`
: Löscht alle Daten, die zur übergebenen Klasse und Schlüssel gehören.

`evictAll()`
: Löscht den gesamten Second-Level-Cache der Persistenzeinheit.

In unserem ersten Beispiel wollen wir uns das Verhalten des Cache näher ansehen. Uns interessiert dabei zunächst die Tatsache, ob der Zwischenspeicher überhaupt benutzt wird. Wir

lagern dazu den Zugriff auf unsere Persistenzeinheit, die `EntityManagerFactory`, in eine zentrale Klassenvariable aus. Darüber hinaus konfigurieren wir die Cache-Nutzung mit der Anweisung `<shared-cache-mode>ALL</shared-cache-mode>`.

```
public class SecondLevelCache {
  public static EntityManagerFactory emf;
  public static void main(String[] args) throws Exception {
    Provider.select("eclipselink");
    emf = Persistence.createEntityManagerFactory("bank", DB.oracle);
    readKunde(1);
    readKunde(1);
    emf.close();
  }

  public static void readKunde(Integer i) {
    EntityManager em = emf.createEntityManager();
    System.out.println("Im Cache vorhanden: "
      + emf.getCache().contains(Kunde.class, i));
    Kunde k1 = em.find(Kunde.class, i);
    System.out.println(k1);
    em.close();
  }

}
```

Die Ausgabe bei Ausführung des Programms mit dem JPA-Provider EclipseLink stellt sich wie folgt dar:

```
Im Cache vorhanden: false
-- SELECT ID, GEBURTSDATUM, GESCHLECHT, KUNDENNUMMER,
   NACHNAME, VORNAME, WOHNORT_ID FROM KUNDE WHERE (ID = ?)
   bind => [1]
de.jpainfo.model.Kunde@1a5af9f
Im Cache vorhanden: true
de.jpainfo.model.Kunde@159780d
```

Bei der ersten Ausführung der `read()`-Methode ist das Entity mit Id 1 noch nicht im Cache vorhanden. Die zweite Ausführung gibt uns dann das erwartete `true` zurück: die JPA-Implementierung hat nach der ersten Abfrage der Datenbank das Entity in den Second-Level-Cache gelegt. Es erfolgt nur ein Datenbankzugriff.

Es ist jedoch ein wesentlicher Unterschied zum Programmausschnitt von Seite 166 zu beobachten. Wir bekommen hier nicht die Referenz auf ein und dasselbe Objekt zurückgeliefert. Obwohl die Daten nach dem ersten `Select`-Befehl im Zwischenspeicher liegen, erzeugt der Provider eine neue Instanz. Dieses Verhalten ist korrekt und würde auch ohne Cache-Benutzung so erfolgen. Bei mehreren `EntityManager`-Instanzen muss gewährleistet sein, dass die Daten in Form von Objekt-Instanzen voneinander unabhängig bearbeitet werden. Der Transaktionsgedanke der Datenbanken gilt an dieser Stelle natürlich auch für unterschiedliche `EntityManager`-Ausprägungen. Jeder aufgebaute Persistenzkontext muss seine Transaktionen unabhängig von anderen Kontexten abarbeiten. Referenzen auf denselben Speicherbereich würden diesen Grundsatz jedoch aufheben.

Unser Programmbeispiel von oben funktioniert mit EclipseLink ohne provider-spezifische Konfigurationen direkt nach Aktivierung des Caching in der Datei `persistence.xml`. EclipseLink bringt wie OpenJPA auch eine eigene Cache-Implementierung mit sich und ist bei

einfachen Caching-Verfahren ohne Synchronisation über mehrere virtuelle Maschinen nicht auf andere Cache-Anbieter angewiesen. Um den Second-Level-Cache für eine Instanz einer `EntityManagerFactory` in OpenJPA zu aktivieren, muss folgendes provider-abhängige Property gesetzt werden:

```
<property name="openjpa.DataCache" value="true"/>
```

Hibernate greift auf bestehende Cache-Implementierungen anderer Anbieter zurück. Die entsprechenden JAR-Dateien des Anbieters müssen dann im Klassenpfad zur Verfügung stehen. Hibernate aktiviert die Cache-Benutzung durch Angabe einer Factory-Klasse, die die Brücke zum Cache-Anbieter darstellt. In unseren Beispielen in diesem Kapitel referenzieren wir die Implementierung von Ehcache [URL-EH].

```
<property name="hibernate.cache.use_second_level_cache"
  value="true"/>
<property name="hibernate.cache.region.factory_class"
  value="net.sf.ehcache.hibernate.SingletonEhCacheRegionFactory"/>
```

Hinweis:

In vielen Dokumentationen zum Thema Hibernate wird auch noch auf das alte Property `hibernate.cache.provider_class` verwiesen. Es kommt in älteren Hibernate-Versionen (unter 3.3) zum Einsatz.

Damit schließen wir unser Eingangsbeispiel ab. Alle JPA-Implementierungen nutzen nun den Second-Level-Cache aus. Wie einzelne Abfragen im JPA-Standard die Cache-Nutzung beeinflussen können, sehen wir uns im nächsten Abschnitt an.

Projekt:

Die im aktuellen Kapitel beschriebenen Code-Beispiele zum Thema Caching in Java-SE-Anwendungen sind im Projekt *jpacaching* enthalten.

Die Properties `CacheRetrieveMode` und `CacheStoreMode`

Ob eine Abfrage den Cache benutzt, lässt sich mit folgenden Properties auf der Ebene des `EntityManagers` bzw. einer `Query` steuern.

`javax.persistence.CacheRetrieveMode`

Dieses Property steuert das Verhalten, wenn Daten mit der `find()`-Methode des `EntityManager` oder durch Ausführung einer `Query` gelesen werden sollen. In der Enumeration stehen die Werte `BYPASS` und `USE` zur Verfügung. Mit der Default-Einstellung `USE` versucht man, die betroffenen Entities aus dem Cache zu lesen. Sind sie dort nicht vorhanden, erfolgt der Zugriff über die Datenbank. Die Einstellung `BYPASS` lässt die Cache-Abfrage außen vor und weist den JPA-Provider an, auf die Datenbank zuzugreifen.

`javax.persistence.CacheStoreMode`

Mit dem Property `CacheStoreMode` konfiguriert man die Veränderung der Entities im Cache während des Lesens von Entities oder beim Update von Änderungen auf der Datenbank. Zur Verfügung stehen die Werte `USE`, `BYPASS` und `REFRESH`. Mit dem Default-Schalter `USE` werden Entities beim Lesen und Schreiben auch in den Cache der Anwen-

dung gebracht. Bereits im Cache enthaltene Entities werden beim Lesen aber nicht aktualisiert. Erst die Einstellung `REFRESH` aktualisiert auch die Entities im Cache, die bereits enthalten sind. Mit `BYPASS` wird der Cache wie oben umgangen.

Die Einstellungen lassen sich auf folgende Art und Weise der JPA-Implementierung übergeben:

- bei der Erstellung des `EntityManager` über die Methode `createEntityManager(Map m)` und dem Parameter einer `Map`-Instanz;
- über die Instanzmethode `setProperty(String s1, String s2)` durch direkte Angabe des Property;
- über die Methoden `find()` und `refresh()`, die einen optionalen `Map`-Parameter entgegennehmen und
- über die `setHint(String s1, String s2)`-Methode einer `Query`.

Die Einstellungen, die direkt mit den Methoden `find()`, `refresh()` und `setHint()` gemacht werden, überschreiben temporär das Verhalten, das beim `EntityManager` selbst definiert wurde. Darüber hinaus wird bei der Methode `refresh()` das Property `CacheRetrieveMode` komplett ignoriert, da diese Methode immer den Zustand des Entity aus der Datenbank holt und Änderungen des lokalen Objektes überschreibt.

Nachfolgender Programmausschnitt soll die Anwendung dieser Properties noch einmal verdeutlichen.

```
public static void em1Work() {
  EntityManager em = emf.createEntityManager();
  // Default-Einstellungen, zu Demo-Zwecken explizit gesetzt!
  em.setProperty("javax.persistence.cache.storeMode", CacheStoreMode.
      USE);
  em.setProperty("javax.persistence.cache.retrieveMode",
      CacheRetrieveMode.USE);
  // Mit find-Methode in Cache geschrieben
  Kunde k1 = em.find(Kunde.class, 1);
  // Mit find-Methode in Cache geschrieben
  Kunde k2 = em.find(Kunde.class, 2);
  // Cache umgangen mit ueberschriebenem Property
  Map m = new Hashtable<String,Object>(){{
      put("javax.persistence.cache.storeMode", CacheStoreMode.BYPASS)
          ;
  }};
  Kunde k3 = em.find(Kunde.class, 3, m);
  System.out.println(k1);
  System.out.println(k2);
  System.out.println(k3);
  Thread.sleep(100000);
  em.close();
}

public static void em2Work() {
  EntityManager em = emf.createEntityManager();
  // Lesen aus Cache, da Default-Einstellung
  Kunde k1 = em.find(Kunde.class, 1);
  // Umgehen des Cache
  Map m = new Hashtable<String,Object>(){{
    put("javax.persistence.cache.retrieveMode", CacheRetrieveMode.
        BYPASS);
```

```
    }};
    Kunde k2 = em.find(Kunde.class,2, m);
    System.out.println(k1);
    System.out.println(k2);
    em.close();
  }
```

Wir erzeugen zwei `EntityManager`-Instanzen, die den gleichen (lokalen) Cache benutzen. Die erste Instanz verändert zunächst die Default-Einstellungen des `EntityManagers` nicht. Die Ergebnisse der ersten beiden `find()`-Abfragen werden auch im Cache abgelegt. Das Entity mit der Id 3 jedoch nicht, hier umgehen wir die Ablage im Cache. Die Instanz `em2` liest den Kunden mit der Id 1 direkt aus dem Cache, da hier der JPA-Standard bzw. dessen Default-Einstellungen greifen. Entity 2 wird trotz Vorhandensein im Cache aus der Datenbank gelesen, da die `find()`-Methode mit dem Property `CacheRetrieveMode` verändert wurde. Zur Verdeutlichung des Sachverhaltes haben wir den Systemzustand noch einmal grafisch in Abbildung 6.5 dargestellt.

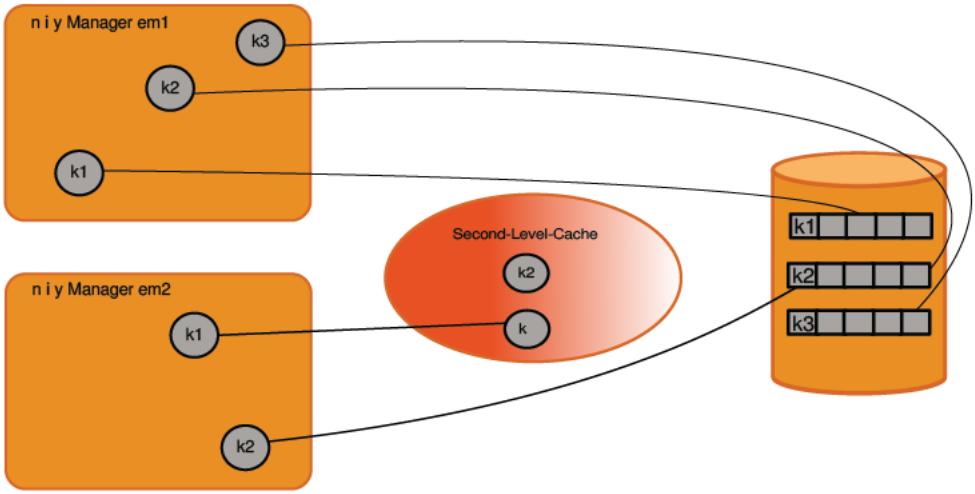


ABBILDUNG 6.5 Second-Level-Cache: Store- und Retrieve-Strategien

Hinweis:

In älteren Versionen von EclipseLink funktionieren die aktuellen Properties zur direkten Cache-Konfiguration nicht. Erst ab Version 2.3.0 werden die Einstellungen `javax.persistence.cache.storeMode` und `javax.persistence.cache.retrieveMode` von EclipseLink korrekt verarbeitet.

6.3.6 Vor- und Nachteile des Caching

Die Benutzung des Second-Level-Cache reduziert die Anzahl der Datenbankzugriffe deutlich. Dies kann, wenn die Datenbank-Anbindung langsam ist, zu hohen Performance-Gewinnen führen. Mit der weiteren Cache-Benutzung holt sich der Entwickler jedoch eine Reihe von Problemen ins Haus, die er kennen muss:

- Abfragen können dazu führen, dass veraltete Daten gelesen werden. Dieser Gedanke ist beim First-Level-Cache gerade gewünscht, um dem Transaktionsgedanken Rechnung zu tragen. Bei Benutzung des Second-Level-Cache werden aber auch bei einer erfolgreichen Änderung der Datenbank die existierenden Caches nicht davon informiert, dass es diese Änderungen gab. Mit Methoden wie `evict()`, `evictAll()`, aber auch einem `refresh()` des `EntityManager`, kann der Zwischenspeicher aktualisiert werden.
- Die Cache-Benutzung führt zu einem höheren Bedarf an Arbeitsspeicher. Die JVM, in der der Cache läuft, hält eine höhere Anzahl von Objekten parallel zur Datenbank bereit. Die JPA-Implementierungen bieten mit ihren speziellen Konfigurationen die Möglichkeit an, die Anzahl der Objekte zu begrenzen bzw. den Cache nach einer gewissen Zeitspanne automatisch zu erneuern.

Die Benutzung des Second-Level-Cache kann problemlos bei Entities eingesetzt werden, auf die meist lesend zugegriffen wird. Der Anwender muss bei schreibenden Zugriffen immer damit rechnen, veraltete Daten zu bearbeiten. Es liegt in seiner Verantwortung, die damit einhergehenden Probleme abzuschätzen und in Kauf zu nehmen oder aber die Cache-Benutzung entsprechend einzuschränken.

Hinweis:

Die verschiedenen JPA-Implementierungen bieten jeweils interessante Ansätze im Caching-Bereich, die zum Teil weit über den Standard hinausgehen. Wir haben einige provider-spezifische Ansätze in den Abschnitten 11.1, 12.1 und 13.1 beschrieben.

6.4 Konversationen

Unter Konversationen oder langlaufenden Transaktionen (long-running transactions), weniger häufig auch Benutzer- oder Anwendungstransaktionen genannt, versteht man in der Regel mehrere Interaktionen des Benutzers mit dem System, die in ihrer Gesamtheit die charakteristischen Transaktionseigenschaften (ACID) besitzen sollen. Als typischer Anwendungsfall kann etwa der Workflow einer Kundenneuanlage in einer Web-Anwendung dienen, bei der auf der ersten Seite die Kundendaten (Vorname, Nachname, Geburtsdatum, ...) auf der zweiten Seite die Adressdaten (Straße, Ort, PLZ, E-Mail, ...) und auf der dritten Seite die Bankverbindung (Bank, BLZ, Kontonummer, ...) einzugeben sind. Problematisch ist hierbei, dass die Daten jeweils per HTTP-Request übertragen werden, HTTP jedoch ein zustandsloses Protokoll ist. Die Daten der ersten Seiten müssen kumuliert und bei der Verarbeitung der letzten Seite komplett und endgültig gespeichert werden. Die in der Vergangenheit hierfür gewählten Ansätze sind zum Teil recht aufwendig zu realisieren. Mit Java-EE 6 vereinfacht sich die Realisierung zum einen durch einen Konversationskontext, zum anderen durch die Möglichkeiten von JPA.

Java-EE 6 enthält als eine von vielen Spezifikationen CDI 1.0 (Contexts and Dependency Injection for the Java EE platform) [URL-CDI]. CDI erweitert die durch das Servlet-Programmiermodell vorgegebenen Kontexte (Scopes) Request, Session und Application um Konversationen. Eine Konversation umfasst einen oder mehrere Requests, wobei die erste Art

als transient, die zweite Art als long-running bezeichnet wird. Abbildung 6.6 macht deutlich, dass der Konversationskontext zwischen dem Request-Kontext und dem Session-Kontext liegt.

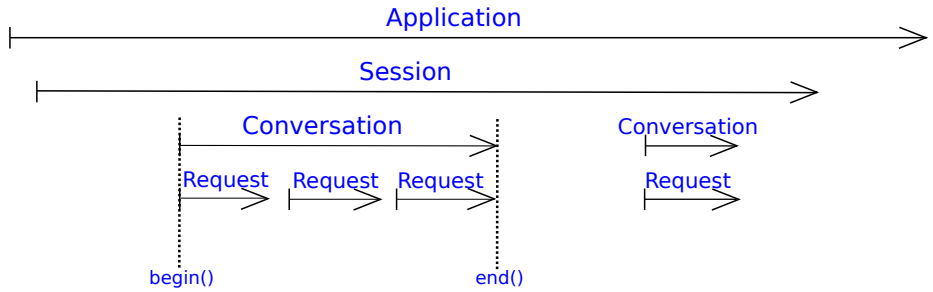


ABBILDUNG 6.6 Typische Kontexte bei Web-Anwendungen

CDI enthält für die genannten Kontexte entsprechende Annotationen: `@RequestScoped`, `@ConversationScoped`, `@SessionScoped` und `@ApplicationScoped`. Diese sind im Package `javax.enterprise.context` enthalten. Das Package enthält noch zwei weitere Annotationen, auf die wir hier jedoch nicht eingehen.

Die Annotationen können bei einfachen POJOs, aber auch bei EJB-Session-Beans verwendet werden und legen die Lebensdauer der annotierten Objekte fest. JPA-Entities sollten nicht auf diese Art und Weise annotiert werden, da der Persistenzkontext den Lebenszyklus und die Lebensdauer dieser Objekte steuert. Während bei den Kontexten Request, Session und Application die Lebensdauer der Objekte automatisch bestimmt wird, muss bei Konversationen programmatisch nachgeholfen werden, was die beiden Methoden `begin()` und `end()` in Abbildung 6.6 andeuten.

Im Folgenden entwickeln wir ein Beispiel für eine Konversation, in der die Daten eines Kunden über mehrere JSF-Seiten eingegeben und in der letzten Seite persistiert werden. Um den Code-Umfang gering zu halten, werden lediglich der Vorname, der Nachname und das Geburtsdatum des Kunden in jeweils einer JSF-Seite eingegeben. In einer realistischen Anwendung könnten etwa die Kundendaten auf der ersten, die Adressdaten auf der zweiten und die Bankverbindungsdaten auf der dritten Seite eingegeben werden, wie wir dies bereits früher angedeutet haben. Wir beginnen bei der Oberfläche und stellen im Folgenden den relevanten Teil der JSF-Seite `vorname.xhtml` dar.

```
<h:form>
  <h:outputLabel value="Vorname:" for="vorname" />
  <h:inputText id="vorname" value="#{kundeView.vorname}"
               required="true" />
  <br />
  <h:commandButton action="#{kundeView.vornameEingegeben}"
                   value="Weiter" />
  <h:commandButton action="#{kundeView.abbrechen}"
                   value="Abbrechen" immediate="true"/>
</h:form>
```

Man erkennt in der JSF-Komponente `<h:inputText>` die Möglichkeit der Eingabe des Kundenvornamens. Über den Ausdruck `#{kundeView.vorname}` der Expression-Language (EL) ist die Eingabe an das Property `vorname` der JSF-Managed-Bean namens `kundeView`

gebunden und wird von JSF automatisch zugewiesen. Ein JSF-Formular kann z. B. über eine Schaltfläche an den Server geschickt werden, was im Beispiel über die Komponente `<h:commandButton>` erfolgt. Die erste Verwendung bindet über den EL-Ausdruck `#{kundeView.vornameEingegeben}` eine sogenannte Action-Methode, die bei Betätigung der Schaltfläche aufgerufen wird. Action-Methoden sind parameterlose Methoden und werden zur Realisierung der Anwendungslogik und zur Navigation zwischen Seiten verwendet. Die in allen EL-Ausdrücken benutzte JSF-Managed-Bean ist in Listing 6.2 dargestellt.

LISTING 6.2 JSF-Managed-Bean für die Konversation zur Kundenneuanlage

```
1  @ManagedBean
2  public class KundeView {
3
4    private String vorname;
5    private String nachname;
6    private Date geburtsdatum;
7
8    @Inject
9    KundeService kundeService;
10
11   public KundeView() { }
12
13   public String neuanlage() {
14     kundeService.neuanlage();
15     return "vorname?faces-redirect=true";
16   }
17
18   public String vornameEingegeben() {
19     kundeService.vorname(vorname);
20     return "nachname?faces-redirect=true";
21   }
22
23   public String nachnameEingegeben() {
24     kundeService.nachname(nachname);
25     return "geburtsdatum?faces-redirect=true";
26   }
27
28   public String geburtsdatumEingegeben() {
29     kundeService.geburtsdatum(geburtsdatum);
30     return "fertig?faces-redirect=true";
31   }
32
33   public String abbrechen() {
34     kundeService.abbrechen();
35     return "index?faces-redirect=true";
36   }
37
38   // Getter und Setter
39   ...
40 }
```

Die in Zeile 1 verwendete Annotation `@ManagedBean` erzeugt eine JSF-Managed-Bean, deren Name dem Klassennamen mit kleingeschriebenem ersten Buchstaben entspricht. Das in der JSF-Seite verwendete Property `vorname` ist in Zeile 4 definiert; das benötigte

Getter/Setter-Paar ist nicht dargestellt. Die Action-Methode `vornameEingegeben` stellen die Zeilen 18 bis 21 dar. Der Return-Ausdruck verlangt nach einer Erläuterung: Seit JSF 2.0 ist die direkte Navigation durch Angabe des Navigationsziels im Rückgabewert der Action-Methode möglich. Im Beispiel wird zur Seite `nachname.xhtml` navigiert. JSF basiert auf HTTP-Post-Requests. Bei der Navigation zu einer anderen Seite zeigt die URL-Leiste des Browsers noch das URL der abgeschickten Formularseite an, während der Browser-Inhalt dem Navigationsziel entspricht. Über ein HTTP-Redirect (`faces-redirect=true`) kann dieses für Benutzer seltsam erscheinende Verhalten vermieden werden. Für weitere Einblicke in die Funktionsweise von JSF verweisen wir auf unser Buch [Mü10].

Die für unser Beispiel wichtige Methode ist `neuanlage()`. Diese Action-Methode beginnt den Vorgang einer Kundenneuanlage. Der Aufruf erfolgt über eine JSF-Seite, deren einziger Inhalt eine Schaltfläche mit `<h:commandButton>` zum Aufruf der Methode darstellt. Wir sehen daher von einer Darstellung der Seite ab. Das Objekt `kundeService` vom Typ `KundeService` wird durch die `@Inject`-Annotation *injiziert*. Vor Java-EE 5 wurden Objekte über JNDI-Aufrufe explizit beschafft. Java-EE 5 ermöglichte mit der `@EJB`-Annotation, dass EJBs automatisch injiziert wurden. Mit Java-EE 6 wurde diese Möglichkeit mit der `@Inject`-Annotation auf nahezu beliebige Objekte erweitert. Im Beispiel wird eine Instanz der Stateful Session-Bean `KundeService` injiziert, deren Klassendefinition in Listing 6.3 angegeben ist.

LISTING 6.3 Stateful Session-Bean mit Conversation-Scope

```
1  @Stateful
2  @ConversationScoped
3  @TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
4  public class KundeService implements Serializable  {
5
6    @PersistenceContext(type = PersistenceContextType.EXTENDED)
7    EntityManager em;
8
9    @Inject
10   Conversation conversation;
11
12   private Kunde kunde;
13
14   public void neuanlage() {
15     conversation.begin();
16     kunde = new Kunde();
17     em.persist(kunde);
18   }
19
20   public void vorname(String vorname) {
21     kunde.setVorname(vorname);
22   }
23
24   public void nachname(String nachname) {
25     kunde.setNachname(nachname);
26   }
27
28   @TransactionAttribute(TransactionAttributeType.REQUIRED)
29   public void geburtsdatum(Date geburtsdatum) {
30     kunde.setGeburtsdatum(geburtsdatum);
31     conversation.end();
32   }
```

```
33
34    public void abbrechen() {
35      conversation.end();
36    }
37  }
```

In Zeile 1 wird die Klasse durch die Annotation `@Stateful` zu einer Stateful Session-Bean. Deren Lebensdauer wird mit der Annotation `@ConversationScoped` auf eine Konversation beschränkt. Session-Beans zeigen im Default-Fall transaktionales Verhalten, wie im Abschnitt 3.3 dargestellt. Wir wollen aber gerade nicht, dass nach jeder Seite der geänderte Entity-Zustand mit der Datenbank synchronisiert wird. Daher schalten wir mit der `@TransactionAttribute`-Annotation das transaktionale Verhalten aller Methoden dieser Bean aus (`NOT_SUPPORTED`). In den Zeilen 6 und 7 wird ein `EntityManager` mit erweitertem Persistenzkontext injiziert und der Stateful Session-Bean zugeordnet, so dass aufeinanderfolgende Requests derselben HTTP-Session Zugriff auf denselben Persistenzkontext haben.

Obwohl die EJB einen Konversationskontext hat, muss der Beginn der Konversation explizit programmiert werden. Mit der Methode `begin()` des Objekts `conversation` wird dies in Zeile 15 realisiert. Das Objekt wurde in den Zeilen 9 und 10 injiziert und ist vom Typ `Conversation`, einem Interface des CDI-Package `javax.enterprise.context` zur Realisierung von Konversationen. Im weiteren Verlauf der Methode `neuanlage()` wird ein Kundenobjekt erzeugt (Zeile 16) und persistiert (Zeile 17). Da die Methode `neuanlage()` nicht in einer Transaktion ausgeführt wurde, wird der Persistenzkontext nicht mit der Datenbank synchronisiert. Es findet also wie beabsichtigt kein Einfügen des Kunden in die Datenbank statt. Die Methoden `vorname()` und `nachname()` werden über die JSF-Managed-Bean jeweils durch die entsprechende Benutzeraktivität initiiert aufgerufen und setzen den Vor- und Nachnamen des Kunden-Entity. Der Wizard-ähnliche Workflow wird durch die Methode `geburtsdatum` abgeschlossen. Da diese Methode mit `@TransactionAttribute` annotiert ist und das Transaktionsattribut auf `REQUIRED` gesetzt ist, erfolgt der Methodenaufruf in einer Transaktion. Durch den Aufruf der Methode `end()` wird die Konversation beendet, so dass die Session-Bean selbst, aber auch alle im Rahmen der Konversation erzeugten abhängigen Objekte entfernt werden. In diesem Beispiel sind keine derartigen Objekte erzeugt worden, der Leser kann sich aber bestimmt entsprechende Beispiele vorstellen. Weil die Transaktion beendet wird, wird das Kunden-Entity mit einer Insert-Anweisung in die Datenbank geschrieben. Alternativ kann an beliebiger Stelle des Workflows die Konversation durch die Methode `abbrechen()` beendet werden, ohne dass SQL-Anweisungen an die Datenbank abgesetzt werden.

Unter dem Strich lässt sich zusammenfassen, dass wir das Ziel einer Konversation, also einer langlaufenden Transaktion, erreicht haben. Mit dem Zusammenspiel von CDI, EJB und JPA konnten wir mehrere Web-Seiten zu einer zusammenhängenden Einheit mit ACID-Eigenschaften integrieren. Nicht verschwiegen werden soll an dieser Stelle, dass das Beispiel für den Produktivbetrieb noch vervollständigt werden müsste, z. B. um sicherzustellen, dass ein Benutzer nicht in der Mitte des Wizards beginnen kann. Dies liegt aber außerhalb der Ziele eines Buches über JPA.

Projekt:

Das Projekt *konversationen* enthält den Beispiel-Code aus Abschnitt 6.4.

7 JPQL (Java Persistence Query Language)

Wir haben in Kapitel 3 bereits gezeigt, wie man mit den Methoden `find()` und `getReference()` Objekte aus der Datenbank laden kann. Dazu musste aber immer der identifizierende Schlüssel des Objekts bekannt sein. Dass dieser Mechanismus nicht ausreicht, um effektiv auf den Datenbestand zuzugreifen, scheint einleuchtend. Die Java-Persistence-API bietet deshalb die Möglichkeit, über die Klasse `Query` entsprechende Java-Instanzen zu erzeugen. Dies kann zum einen über die „Java Persistence Query Language (JPQL) " oder durch direkte Eingabe von nativem SQL erfolgen. Darüber hinaus wurde die so genannte *Criteria API* in die Version 2.0 der JPA-Spezifikation übernommen. Wir sprechen hier bewusst von *übernommen*, weil es für diese Arten von Abfragen bereits Vorbilder in Implementierungen wie z. B. Hibernate gab. Diese existierten zwar schon zur JPA 1.0 waren jedoch nicht Bestandteil der damaligen Spezifikation.

Um unsere Abfragen realistisch zu gestalten, liegt diesem Kapitel das in Abbildung 7.1 dargestellte Klassenmodell zugrunde.

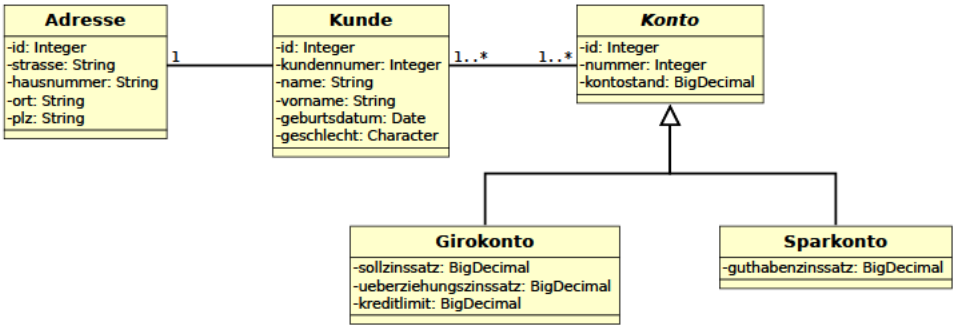


ABBILDUNG 7.1 Klassenmodell Kapitel Datenbankabfragen

Damit der Leser über die dargestellten Beispiele hinaus eigene Abfragen entwickeln kann, befinden sich in den Datenbanken des Projektes generierte Testdaten. Unsere mitgelieferte H2-Datenbank enthält ca. 1000 Kunden- und Adressdaten. Jeder Kunde besitzt außerdem ein Giro- und Sparkonto mit zufällig generiertem Saldo. Darüber hinaus kann der Leser für die Datenbanken Postgres, Oracle und MySQL die jeweiligen DDL-Konstrukte und entsprechende Insert-Anweisungen in Form von einfachen Textdateien herunterladen und sich so eigene Testdaten neben der H2-Datenbank bereitstellen.

Projekt:

Das diesem Kapitel zugehörige Eclipse-Projekt `datenbankabfragen` enthält im Java-Package `de.jpainfo.test` alle Beispiele in einzelnen Java-Klassen. Die Klasse `JPQLTest` besitzt ein Vielzahl von JPQL-Anweisungen in statischen Variablen, die der Leser einfach zur Ausführung bringen kann. Darüber hinaus haben wir eine einfache Ausgabe-Klasse `OutputHelper` erstellt, die möglichst flexibel die Rückgabe-Struktur der Abfragen darstellt.

Das Schema der Datenbank stellen wir in Abbildung 7.2 dar. Die Datentypen der einzelnen Spalten können je nach Datenbankhersteller leicht voneinander abweichen.

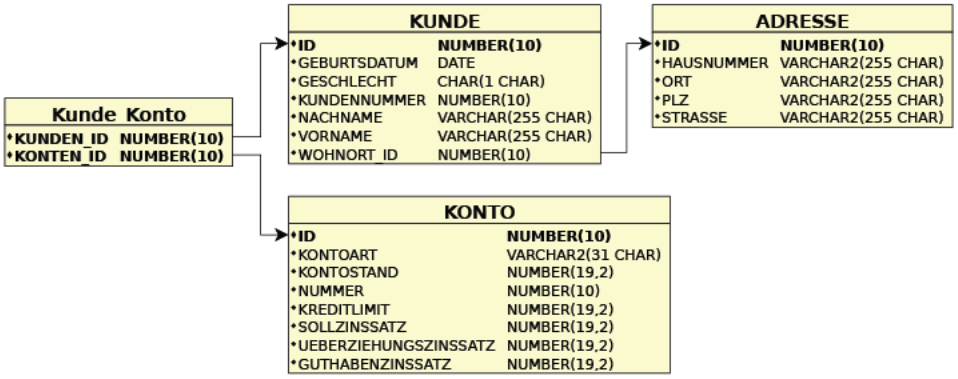


ABBILDUNG 7.2 Datenbankschema für das Kapitel Datenbankabfragen

7.1 JPA-Query-Interfaces

Eine Instanz der Klasse `Query` erhält man mit einer der drei Methoden des bereits bekannten Entity-Managers:

- `createNamedQuery()`
- `createNativeQuery()`
- `createQuery()`.

Neu hinzugekommen in JPA 2.0 ist das Interface `TypedQuery`. Dieses Interface erbt vom bereits vorgestellten Interface `Query`. Entscheidender Vorteil dieses Interfaces ist der Umstand, dass im Ergebnis der Abfrage bereits typisierte Objekte zurückgegeben werden. Der Typ, auf den der JPA-Provider die Ergebnisliste casten soll, wird bei Erzeugung des Abfrage-Objektes als zweiter Parameter übergeben. Das Interface `TypedQuery` wird darüber hinaus in der neu hinzugekommenen *Criteria-API* benutzt. Diese API stellen wir in Abschnitt 8 vor. Um ein bereits typisiertes Ergebnis zu bekommen, wurden die aus JPA 1.0 bekannten Methoden `createQuery` und `createNamedQuery` überladen:

```
public <T> TypedQuery<T> createQuery(String name,
  Class<T> resultClass);

public <T> TypedQuery<T> createNamedQuery(String name,
  Class<T> resultClass);
```

Der Zugriff auf die Ergebnisse von Abfragen ist unabhängig davon, ob es sich um natives SQL oder um eine Abfrage in der Abfragesprache der JPA handelt. Die Methode `getResultList()` liefert eine Liste vom Typ `java.util.List`, deren Elemente wie gewohnt verarbeitet werden können. Welche Typen diese Liste an welcher Stelle beinhaltet, bestimmt der Ausdruck hinter der Select-Anweisung. Die Methode `getResultList()` des Interfaces `TypedQuery` liefert bereits eine vollständige typisierte Liste von Objekten zurück.

Wenn die Abfrage genau ein Ergebnis erwarten lässt, können Sie die Methode `getSingleResult()` verwenden. Finden sich dann mehrere oder gar keine Zeilen im Ergebnis, verursacht diese Methode eine `NonUniqueResultException` bzw. `NoResultException`.

Die Methoden `getResultList()` und `getSingleResult()` benötigen keine Transaktion, innerhalb derer sie ausgeführt werden. Wenn die Abfrage innerhalb einer Transaktion ausgeführt wird, sind die Entities so lange im Zustand *verwaltet*, wie die Transaktion andauert. Nach einem Commit oder Rollback der Transaktion handelt es sich um losgelöste Objekte. Dies gilt aber nur, wenn der Persistenzkontext transaktionsbezogen ist und es sich um keinen erweiterten Kontext handelt. Einzelheiten dazu erläutert Kapitel 6.

Kommen wir nun zu einem einfachen einführenden Beispiel, in dem wir die Verwendung der beiden Query-Interfaces zeigen.

```
EntityManagerFactory emf = Persistence
  .createEntityManagerFactory("bank", DB.h2);
EntityManager em = emf.createEntityManager();
Query q = em.createQuery("select k from Kunde k where nachname "
  + " = 'Meier'");
TypedQuery<Kunde>  tq = em.createQuery("select k from Kunde k where "
  + " k.nachname = 'Meier'", Kunde.class);
List<Kunde> l = (List<Kunde>)q.getResultList();
List<Kunde> tl = tq.getResultList();
for (Kunde kunde : tl) {
  System.out.println(kunde.getNachname() + " " +kunde.getVorname());
}
```

Die Vorteile von `TypedQuery` sind klar ersichtlich. *Type-Casts* auf das Ergebnis sind nicht mehr erforderlich. Die zurückgegebene Liste bzw. das zurückgegebene Objekt bei `getSingleResult()` besitzen bereits den gewünschten Typ.

Tipp:

Wenn der Rückgabewert einer Abfrage klar vorbestimmt ist, sollte der Leser dem Interface `TypedQuery` den Vorrang geben. Das Programm bekommt damit höhere Typsicherheit, Compiler-Warnungen mit dem Inhalt *Unchecked Cast* werden vermieden. Bei nativen Abfragen muss weiterhin das Interface `Query` verwendet werden.

Möchte man nicht alle Ergebnisse auf einmal haben, sondern soll die Anwendung gewissermaßen durch die Datenmenge iterieren, können die Methoden `setFirstResult()` und

`setMaxResults()` verwendet werden. Mit ihrer Hilfe konfiguriert man zum einen den Startpunkt des Ergebnisses und zum anderen die Anzahl der zurückzuliefernden Zeilen.

```
TypedQuery<Kunde> tq = em.createQuery("select k from Kunde k "
  + " order by k.id", Kunde.class);
tq.setFirstResult(100);
tq.setMaxResults(10);
List<Kunde> tl = tq.getResultList();
for (Kunde kunde : tl) {
  System.out.println(kunde.getNachname());
}
```

Die Methode `setFirstResult()` im obigen Beispiel setzt den Startpunkt auf den 100. Eintrag. Da sie aber bei 0 anfängt zu zählen, wird der Kunde mit der Id 101 als Erster gelesen und zurückgeliefert.

Wichtig bei diesen Methoden ist jedoch, dass die jeweilige JPA-Implementierung nicht erst alle Datensätze des abzufragenden Entity aus der Datenbank lädt und im Arbeitsspeicher sortiert, um die angeforderten Sätze zurückzuliefern. Wesentlich effizienter wäre es, an dieser Stelle die Sortier-Mechanismen der zugrunde liegenden Datenbank zu verwenden. Wir wollen uns die Ergebnisse bzw. die generierten SQL-Statements für unsere verwendeten Datenbanken H2, Oracle, Postgres und MySQL ansehen. Aus Platzgründen beschränken wir uns dabei auf die JPA-Implementierung Hibernate. Der Leser ist aufgefordert, die Ergebnisse mit den anderen Implementierungen ebenfalls zu untersuchen. Wir haben die Ergebnisse dahingehend gekürzt, dass die selektierten einzelnen Spalten nicht separat ausgewiesen werden. Interessant sind für uns die `where`- und `order by`-Klausel bzw. herstellerspezifische SQL-Erweiterungen.

```
-- H2
select * from Kunde kunde0_
  order by kunde0_.id
  limit 10 offset 100

-- Oracle
select * from
  (
    select row_.*, rownum rownum_ from
      (
        select * from Kunde kunde0_ order by kunde0_.id
      ) row_
    where rownum <= 110
  )
where rownum_ > 100

-- Postgres
select * from Kunde kunde0_
  order by kunde0_.id
  limit 10 offset 100

-- MySQL
select * from Kunde kunde0_
  order by kunde0_.id
  limit 100, 10
```

Bei Postgres und H2 sind die SQL-Statements identisch, MySQL hat Abweichungen bei der `limit`-Klausel, die Ausführung in Oracle führt zu `subselect`-Anweisungen mit Hilfe der

nur in Oracle verfügbaren `rownum`-Variablen. Genau hier kann der Leser den Vorteil der JPA-Spezifikation erkennen. Wir verwenden in Java nur eines – ein von allen Datenbanken abstrahiertes Select-Statement – zur Ausführung kommen aber ganz unterschiedliche Anweisungen, die die JPA-Implementierung gezielt nach verwendetem Datenbanksystem generiert.

Hinweis:

Die generierten SQL-Statements können je nach verwendetem SQL-Dialekt sehr unterschiedlich sein. Unterschiedliche SQL-Abfragen können verschiedene Ausführungsstrategien in den jeweiligen Datenbanksystemen mit unterschiedlichem Laufzeitverhalten zur Folge haben. Ergeben sich bei einem Wechsel des zugrunde liegenden Datenbanksystems Performanzeinbußen, müssen die erzeugten SQL-Befehle näher analysiert werden. Bei einem Wechsel der JPA-Implementierung können ähnliche Probleme in einer sonst unveränderten Anwendung auftreten.

7.1.1 Named Queries

Um zu vermeiden, dass SQL-Fragmente über den gesamten Quellcode verteilt sind, lassen sich *Named Queries* verwenden. Eine Abfrage bekommt hierzu einen in der Persistenzeinheit eindeutigen Namen und kann so an verschiedenen Stellen verwendet werden. Die Konfiguration kann man sowohl über eine Annotation oder aber in der Mapping-Datei analog zu den Default-Listenern durchführen. Die XML-Konfiguration überschreibt eine Abfrage, die mit einer Annotation definiert ist, sobald diese den gleichen Namen besitzt. Das folgende Beispiel definiert die Abfrage `loadSparkonten` als Annotation:

```
@NamedQuery(
  name="loadSparkonten",
  query="select k from Sparkonto k where k.nummer = :nummer")
```

Benötigt man mehrere Abfragen in einem Entity, werden diese innerhalb von `@NamedQueries` durch ein Komma getrennt aufgeführt:

```
@NamedQueries({
  @NamedQuery(
    name="loadSparkonten",
    query="select k from Sparkonto k where k.nummer = :nummer"
  ),
  @NamedQuery(
    name="loadGirokonten",
    query="select k from Girokonto k where k.nummer = :nummer")
})
```

Hinweis:

Der Leser sollte sich in größeren Projekten auf jeden Fall Gedanken über eine Namenskonvention der im Projekt verankerten `NamedQuery`-Abfragen machen. Am Entity `Konto` hinterlegte Abfragen könnten z. B. mit `@NameQuery(name="Konto.loadSparkonten", ...)` beginnen. Damit ist bei einer Verwendung im Quellcode immer der Ort der Definition zu erkennen. Wir verzichten in unseren Programmen auf diesen Ansatz, um die Beispiele möglichst kurz zu halten.

TABELLE 7.1 `@NamedQuery`

`@NamedQuery(...)`			
Option	**Typ**	**Default**	**Beschreibung**
`name`*	`String`	–	Name der Abfrage
`query`*	`String`	–	Die eigentliche Abfrage
`hints`	`QueryHint[]`	–	Implementierungsspezifische Parameter für die Abfrage
`lockMode`	`LockModeType`	`NONE`	Lock-Modus der Abfrage

Tabelle 7.1 listet neben den bereits im Beispiel verwendeten Elementen `name` und `query` die optionalen Elemente `QueryHint[]` und `LockModeType` auf. Auf die verschiedenen Sperr-Möglichkeiten sind wir bereits in Kapitel 6 eingegangen. Mit der Option `QueryHint[]` lässt sich ein `Query`-Objekt an eine zugrunde liegende JPA-Implementierung anpassen. Die Spezifikation benutzt dafür `Key-Value`-Paare. Jede Implementierung muss die `Keys` mit ihrer eindeutigen Domain als Präfix versehen, um Inkompatibilitäten bzw. Irrtümer beim Wechsel der JPA-Implementierung auszuschließen.

Für das Element `QueryHint` definiert die Spezifikation das Property `javax.persistence.query.timeout`. Es *soll* die Zeitspanne definieren, wie lange eine Abfrage auf die Freigabe eines exklusiv gesperrten Satzes warten soll, ohne vorher abzubrechen. Die Spezifikation gibt hier aber den klaren Hinweis, dass Anwendungen sich nicht auf diesen Hinweis verlassen dürfen. Insbesondere beim Wechsel der zugrunde liegenden Datenbank oder dem Austausch der JPA-Implementierung kann nicht gewährleistet werden, dass die Anwendung deterministische Ergebnisse analog zur vorigen Konfiguration liefert.

Die Provider definieren eigene Konfigurationsmöglichkeiten, die weit über den Standard hinausgehen. Hibernates spezielle Query-Hints findet der Leser in der Klasse `org.hibernate.ejb.QueryHints`. Die möglichen Parameter von EclipseLink hat der Provider in der Klasse `org.eclipse.persistence.config.QueryHints` implementiert. Leider hat OpenJPA keine zentrale Stelle, an der alle Query-Parameter übersichtlich zu finden sind. Wir verweisen den Leser dafür auf die OpenJPA-Dokumentation.

Wir wollen uns die Verwendung der `QueryHints` nun mit einem abschließenden Beispiel ansehen. Dazu benutzen wir ein in der Oracle-Welt oft gebrauchtes Feature: die sogenannten *Optimizer-Hints*. Mit diesen Hints lässt sich der Datenbank-Optimizer zu einem bestimmten (besseren) Ausführungsplan bringen, den er unter gewissen Umständen nicht selbst findet. Benutzt wird eine `NamedQuery` in der Klasse `de.jpainfo.model.Konto` mit folgendem Inhalt:

```
@NamedQuery(name = "loadKontenHints",
  query = "select k from Girokonto k where k.kontostand > 9500",
  hints = {
    @QueryHint(name = "org.hibernate.comment",
               value = "/*+ FIRST_ROWS(10)*/"),
    @QueryHint(name = "openjpa.hint.OracleSelectHint",
               value = "/*+ FIRST_ROWS(10) */"),
    @QueryHint(name = "eclipselink.sql.hint",
               value = "/*+ FIRST_ROWS(10)*/")
  }
)
```

Die Verwendung von Hinweisen unterschiedlicher Hersteller ist in der Spezifikation genau festgelegt. Präfixe anderer Hersteller muss eine Implementierung ohne Fehler ignorieren. Erzeugt wird diese Abfrage durch die Methode `createNamedQuery()`. Die darüber hinausgehende Verarbeitung ist identisch mit den anderen, bereits vorgestellten `Query`-Objekten.

```
TypedQuery<Girokonto> tq = em.createNamedQuery(
  "loadKontenHints",
  Girokonto.class);
```

Schauen wir uns nun das Ergebnis dieser `NamedQuery` für die verschiedenen Implementierungen an. OpenJPA und EclipseLink verarbeiten die Hinweise korrekt. Direkt nach dem Wort `select` werden die Hinweise für den Datenbank-Optimizer ausgegeben:

```
SELECT /** FIRST_ROWS(10)*/
ID, kontoart, KONTOSTAND, NUMMER, SOLLZINSSATZ,
UEBERZIEHUNGSZINSSATZ, KREDITLIMIT
FROM KONTO
WHERE ((KONTOSTAND > 9500) AND (kontoart = 'GIRO'))
```

Bei Hibernate erscheint zunächst gar keine Ausgabe. Mit der Konfiguration des Parameters

```
<property name="hibernate.use_sql_comments" value="true" />
```

erscheint zumindest der Kommentar, dieser wird jedoch am Anfang des Befehls generiert und ist somit nicht für den Datenbank-Optimizer von Oracle sichtbar. Die Benutzung von Datenbank-Hints ist bei großen Oracle-Datenbanken ein oft genutztes Mittel, um das Datenvolumen innerhalb einer akzeptablen Zeitspanne abzufragen. Zumindest bei Hibernate muss dann auf eine native Query ausgewichen werden. Diese Abfragen behandeln wir in Abschnitt 7.1.3.

Tipp:

Hints lassen sich nicht nur über eine vordefinierte `NamedQuery` setzen. Die Methode `setHint()` des `Query`-Interfaces erlaubt es auch, direkt einen Hinweis zur Laufzeit des Programms hinzuzufügen.

7.1.2 Parameter für Abfragen

Der Leser wird erkennen, dass die in den vorangegangenen Beispielen abgebildeten `Select`-Statements so noch nicht gültig sind. Platzhalter wie `:nummer` aus unserem Beispiel auf Seite 181 müssen natürlich vor Ausführung einer Abfrage mit Werten gefüllt werden. Bei der anschließenden Zuweisung von Werten zu Parametern unterscheidet die JPA-Spezifikation zwei Arten:

NamedParameter

NamedParameter werden über ihren Namen identifiziert und mit Werten versorgt. Die Spezifikation empfiehlt nicht, sie in nativen `Query`-Objekten zu verwenden. Ein Parameter wird immer mit einem Doppelpunkt eingeleitet. Für die Zuweisung der case-sensitiven Parameter stehen folgende Methoden zur Verfügung:

```
setParameter(String name, Object value)
setParameter(String name, Calendar value,
```

```
    TemporalType temporalType)
  setParameter(String name, Date value,
    TemporalType temporalType)
```

PositionalParameter

Positionsparameter werden mit einem Fragezeichen definiert. Man kann sie direkt über eine Position ansprechen und auch in Abfragen mit nativem SQL problemlos verwenden. Sie werden mit Methoden versorgt, die statt des übergebenen `String`-Parameters einen `int`-Wert übergeben bekommen:

```
setParameter(int position, Object value)
setParameter(int position, Calendar value, TemporalType
    temporalType)
setParameter(int position, Date value, TemporalType temporalType)
```

Achtung:

Die Nummerierung der Positionsparameter beginnt bei 1. Grund ist die enge Verwandtschaft mit den Prepared-Statements aus der JDBC-Welt. Der Zugriff auf Positionen eines Arrays oder einer Liste beginnt im Gegensatz dazu immer bei 0.

Mit Hilfe der Klasse `TemporalType` teilt man der Abfrage mit, ob es sich um ein Datum (`TemporalType.DATE`), eine Uhrzeit (`TemporalType.TIME`) oder um beides (`TemporalType.TIMESTAMP`) handelt. Alle weiteren Typen werden mit der jeweils ersten Methode übergeben.

Neu hinzugekommen in Version 2.0 sind darüber hinaus die `setParameter`-Methoden, die ein Objekt vom Typ `javax.persistence.Parameter<T>` an erster Stelle erwarten. Auf deren Verwendung gehen wir in Kapitel 8 noch näher ein.

7.1.3 Direkte SQL-Abfragen

Bevor wir uns in Abschnitt 7.2 der Abfragesprache der JPA zuwenden, wollen wir zunächst untersuchen, wie herkömmliche SQL-Abfragen mit den Instrumenten der Java-Persistence-API auszuführen sind. Native Datenbankabfragen sollten dort benutzt werden, wo die JPQL keine Unterstützung bietet, bzw. wo man Datenbank-Konstrukte benötigt, die von der jeweiligen JPA-Implementierung nicht unterstützt werden. Mit dem Einsatz bestimmter datenbankabhängiger SQL-Konstrukte wird die Portabilität der JPA-Anwendung an sich noch nicht eingeschränkt, da alle Implementierungen diese nativen Abfragen unterstützen müssen. Man verliert somit aber die Möglichkeit, auf einfache Weise eine andere Datenbank einzusetzen.

Um ein `Query`-Objekt einer nativen SQL-Abfrage zu erhalten, können die folgenden 3 Methoden des Entity-Managers benutzt werden:

```
createNativeQuery(String sqlString)
createNativeQuery(String sqlString, Class resultClass)
createNativeQuery(String sqlString, String resultSetMapping)
```

Die einfachste Form einer nativen SQL-Abfrage gelingt mit der Methode `createNativeQuery(String abfrage)`. Dabei wird jedoch keines der in der Anwendung konfigurierten

Entities instanziiert. Zurückgegeben wird lediglich eine Liste von `Object[]`-Arrays, die entsprechend auszuwerten sind.

```
Query q = em.createNativeQuery("select distinct ort, plz " +
  " from adresse ");
Iterator<Object[]> it = q.getResultList().iterator();
while (it.hasNext()) {
  Object[] zeile = it.next();
  String ort = (String) zeile[0];
  String plz = (String) zeile[1];
  System.out.println(plz + " " + ort);
}
```

Eine erste Erweiterungsmöglichkeit bietet die Methode `createNativeQuery(String abfrage, Class clazz)`. Ihr wird neben der eigentlichen Abfrage das zu erzeugende Entity als zweiter Parameter übergeben. Zu beachten ist, dass die Abfrage sämtliche Spalten enthalten muss, die laut konfiguriertem Mapping auch das Entity enthält. Des Weiteren müssen die Spaltennamen bei Verwendung dieser Methode ebenfalls mit den im Mapping konfigurierten Spaltennamen übereinstimmen.

```
Query q = em.createNativeQuery(
  "select id, ort, strasse, hausnummer nummer, plz from adresse ",
  Adresse.class);
List<Adresse> it = q.getResultList();
for (Adresse adresse : it) {
  System.out.println(adresse.getStrasse()
    + " "
    + adresse.getHausnummer());
}
```

Die Abfrage enthält einen Alias, mit dem die Spalte `hausnummer` den Namen `nummer` erhält. Die Vergabe dieses Aliases ist hier natürlich unsinnig, soll aber zeigen, dass, solange kein explizites Mapping angegeben wird, wirklich alle Spalten vorhanden sein müssen und auch deren Namen mit dem im Entity konfigurierten SQL-Namen (sofern überhaupt vorhanden), übereinstimmen.

Das Ergebnis dieses Versuchs ist sehr überraschend, da unsere drei Implementierungen wieder unterschiedliche Ergebnisse im Programmablauf zeigen. Hibernate bricht die Verarbeitung sofort ab. Folgende Fehlermeldung wird dabei ausgegeben:

```
Caused by: java.sql.SQLException: Ungueltiger Spaltenname
```

Bei der Verwendung von EclipseLink kommen wir schon einen Schritt weiter. Die `Adresse`-Objekte werden instanziiert, das Feld Hausnummer bleibt aber leer. Die Ausgabe der Hausnummer liefert den Wert `null`.

OpenJPA instanziiert die `Adresse`-Objekte auch fehlerfrei. Beim Zugriff auf das nicht initialisierte Attribut wird die Information nachgelesen – dies passiert allerdings einzeln für jeden Datensatz und ist somit nicht sehr performant. Der Eintrag `@Column(name="nummer")` behebt zwar den Fehler und lässt die Abfrage zu einem Ende kommen, ist aber keine Lösung, da nun sämtliche anderen Abfragen, die diesen Alias nicht verwenden, ab jetzt einen Abbruch verursachen. Wird der Alias weggelassen, funktioniert die obige Abfrage wieder mit allen Implementierungen.

Hinweis:

Wir wollen dem Leser anhand dieses Beispiels verdeutlichen, dass mit nativem SQL die Portabilität zwischen den JPA-Implementierungen leidet. Die Spezifikation gibt hier eindeutig die Empfehlung ab, solange wie möglich dem Standard der JPQL zu folgen. Erst wenn deren Möglichkeiten erschöpft sind, kann die Verwendung von nativem SQL sinnvoll sein. Der Leser hat sich ja gerade durch die Verwendung von JPA dafür entschieden, möglichst unabhängig von der Datenbank und JPA-Implementierung zu bleiben.

Möchte man Spalten explizit auf Attribute eines Entity abbilden, kann die in Tabelle 7.2 dargestellte Annotation `@SqlResultSetMapping` verwendet werden. Diese Annotation muss in einem Entity bzw. in einer seiner ebenfalls gemappten Oberklassen definiert werden. Jedes `SqlResultSetMapping` benötigt einen in der Persistenzeinheit eindeutigen Bezeichner.

TABELLE 7.2 `@SqlResultSetMapping`-Attribute

`@SqlResultSetMapping(...)`		
Option	**Typ**	**Beschreibung**
`name`*	`String`	Name des ResultSet-Mapping
`columns`	`ColumnResult[]`	Mapping für Spalten, die skalare Werte enthalten
`entities`	`EntityResult[]`	Mapping, mit dem Entities in die entsprechenden Ergebnisse gemappt werden

Die folgende native SQL-Abfrage selektiert alle Kunden und bildet gleichzeitig mit einem Subselect innerhalb des Select-Blocks die Gesamtsumme aller Konten des Kunden. Diese Abfrage ist in der vorliegenden Form mit der JPQL nicht möglich, da die Sprache Subselects nur in der `where`-Bedingung zulässt. Vorteil dieser Abfrage: Um das Gesamtguthaben eines Kunden zu berechnen, müssen wir in Java nicht über die einzelnen Konten iterieren und diese einzeln aufsummieren, sondern lassen die Datenbank (falls sie Subselects zulässt) die Arbeit durchführen:

```
Query q = em.createNativeQuery(
    "select k.*, "
    + " (select sum(kto.kontostand) "
    + "  from Konto kto, Kunde_Konto k2kto "
    + "  where k.id = k2kto.kunden_id and k2kto.konten_id = kto.id "
    + " ) gesamt"
    + " from kunde k", "GesamtGuthabenMapping");
List<Object[]> l = q.getResultList();
for (Object[] o : l) {
  Kunde k = (Kunde) o[0];
  BigDecimal gesamt = (BigDecimal) o[1];
  System.out.println(k.getNachname() + " " + gesamt);
}
```

Das Mapping mit dem Namen `GesamtGuthabenMapping` definieren wir in der Klasse `Kunde`:

```
@Entity
@SqlResultSetMapping(
  name = "GesamtGuthabenMapping",
```

```
    entities = {
      @EntityResult(entityClass = Kunde.class)
    },
    columns = {
      @ColumnResult(name = "gesamt")
    }
  )
  public class Kunde implements Serializable {
  // Attribute sowie Getter und Setter ...}
```

Als Attributwerte der Annotation `@SqlResultSetMapping` lassen sich neben dem obligatorischen Namen die beiden Annotationstypen `@ColumnResult[]` sowie `@EntityResult[]` verwenden. `@ColumnResult` besitzt als einziges Element `name` und referenziert damit ein skalares Ergebnis oder dessen Alias innerhalb einer `select`-Anweisung. Wir verwenden dieses Konstrukt, um auf das von der Datenbank berechnete Gesamtguthaben zugreifen zu können. `@EntityResult` besitzt das obligatorische Element `entityClass`, mit dem eine als Entity konfigurierte Klasse referenziert wird.

Das `Object[]`-Array der Rückgabe enthält zunächst die im Element `entities` aufgeführten Entities in der hinterlegten Reihenfolge. Daran anschließend finden sich die mit `columns` aufgeführten skalaren Werte ebenfalls entsprechend ihrer Reihenfolge im `@SqlResultSetMapping`. Sollte ein Entity mehrere `@SqlResultSetMapping` benötigen, müssen diese in der Annotation `@SqlResultSetMappings` zusammengefasst werden.

Neben der Entity-Klasse lässt `@EntityResult` noch das Element `@EntityField[]` zu, mit dessen Hilfe `column`-Namen bzw. Aliase entsprechenden Properties des Entity zugeordnet werden.

Analog zu `NamedQueries` in der JPQL lassen sich auch native Abfragen an zentraler Stelle vordefinieren. Tabelle 7.3 fasst die Annotation zusammen.

TABELLE 7.3 `@NamedNativeQuery`-Attribute

`@NamedNativeQuery(...)`		
Option	**Typ**	**Beschreibung**
`name`*	`String`	Name der Abfrage
`query`*	`String`	Die eigentliche Abfrage
`hints`	`QueryHint[]`	Implementierungsspezifische Parameter für die Abfrage
`resultClass`	`Class`	Ergebnistyp der Abfrage
`resultSetMapping`	`String`	Eindeutiger Name des definierten ResultSet-Mappings

Tipp:

Die Spezifikation weist bei der Verwendung von nativem SQL ausdrücklich darauf hin, alle Spalten zu selektieren, die zu Attributen eines Entity gemappt werden. Dies schließt auch Werte ein, mit denen Beziehungen zu anderen Entities abgebildet werden. Passiert dies nicht, können die Ergebnisse der einzelnen JPA-Implementierungen undefiniert sein.

7.2 Elemente der Java Persistence Query Language (JPQL)

In diesem Kapitel wollen wir uns der eigentlichen Abfragesprache der JPA zuwenden. Wir können jedoch nicht den gesamten möglichen Sprachumfang beschreiben, sondern möchten vielmehr diejenigen Konstrukte genauer erläutern, die wir für die Praxis als relevant betrachten. Leser, die sich für den gesamten Sprachaufbau der JPQL interessieren, verweisen wir auf das Kapitel „Query Language - BNF“ in der Spezifikation zur JPA [URL-JPA2] . Dort ist die vollständige Syntax in Backus-Naur-Form beschrieben.

Überblick

Die Wörter *Projektion*, sowie *Join* und *Selektion* sind Begriffe aus der relationalen Datenbanktheorie und sollen zunächst kurz erläutert werden, da wir sie im weiteren Verlauf immer wieder verwenden werden.

Projektion bedeutet, dass Spalten mit einem bestimmten Namen einer oder mehrerer Tabellen aus der Gesamtmenge aller möglichen Spalten als Ergebnis einer Abfrage dargestellt (projiziert) werden. Projektion wird erreicht, indem eine Reihe von Spaltennamen hinter dem Schlüsselwort `select` angegeben werden. Der Name der Tabelle (genauer gesagt der Relation) folgt hinter dem Schlüsselwort `from`.

Die Selektion wählt bestimmte Zeilen einer Tabelle aus, die einem gegebenen Kriterium entsprechen. Mit Hilfe der `where`-Bedingung werden diese Kriterien einer Abfrage hinzugefügt.

Werden zwei Tabellen miteinander verknüpft, spricht man von einem sogenannten Verbund bzw. Join dieser Tabellen. Das Ergebnis dieses Joins ist wieder eine Tabelle mit allen Spalten der verbundenen Ursprungstabellen. Aus diesem Verbund kann ebenfalls mittels Projektion und Selektion das gewünschte Ergebnis ermittelt werden [HS00].

7.3 Die From-Klausel

Mit Hilfe des Schlüsselwortes `from` werden Variablen definiert, die an entsprechende Entities gebunden sind. Diese Variablen sind Platzhalter und repräsentieren die eigentlichen Instanzen der persistierten Java-Klassen. Mit diesen Variablen lassen sich später durch `select` die gewünschten Daten abfragen bzw. mit `where` die Anzahl der zurückzuliefernden Daten einschränken.

```
select k.name from Kunde k
where k.vorname = 'Jens'
```

Im obigen Beispiel wird deutlich, dass die Variable `k` an das Entity `Kunde` gebunden wird. Sie ist der Platzhalter für alle `Kunde`-Instanzen, aber auch für Instanzen der erbenden Subklassen. Optional kann zwischen Entity und Variable das Schlüsselwort `AS` aufgeführt werden.

Wie Verbunde zwischen Entities bzw. ihren zugrunde liegenden Tabellen durchgeführt werden, beschreiben wir in den nachfolgenden Abschnitten.

7.3.1 Implizite Joins

Implizite Joins sehen in der Abfragesprache gar nicht nach einem Verbund aus. Mittels Punktnotation wird zum entsprechenden Attribut navigiert. Dabei kann es vorkommen, dass auf Datenbankebene ein Join zwischen zwei oder mehr Tabellen ausgeführt wird, ohne dass explizit diese Tabellen über das Schlüsselwort `Join` und ihre Verknüpfung von Primär- und Fremdschlüssel miteinander verbunden werden. Als Beispiel eines solchen Joins benutzen wir die Assoziation zwischen `Kunde` und `Adresse`. Zwischen beiden Klassen besteht eine 1:1-Beziehung.

```
select k.nachname, k.vorname, k.wohnort.ort
from Kunde k
```

SQL-Entwickler werden hier zum wiederholten Male etwas überrascht sein. Im gesamten JPQL-Ausdruck ist kein Join zu erkennen. Die From-Klausel greift nur auf ein Entity bzw. dessen Tabelle zu. Der Join wird aber durch die Navigation vom Kunden zu dessen Wohnort über die Projektion `k.wohnort.ort` durchgeführt. Die JPA-Implementierung führt automatisch einen Join anhand der in den Konfigurationen hinterlegten Beziehungen durch. Dieser Join wird, wie alle anderen Abfragen der JPQL, nicht in Java durchgeführt. Es findet immer eine Übersetzung in den SQL-Dialekt der Datenbank statt, so dass die jeweilige Operation (hier der Join) durch die Datenbank zur Ausführung gebracht wird. Dargestellt ist der von EclipseLink erzeugte SQL-Befehl für eine Oracle-Datenbank.

```
SELECT t0.NACHNAME, t0.VORNAME, t1.ORT
FROM KUNDE t0, ADRESSE t1
WHERE (t1.ID = t0.WOHNORT_ID)
```

Implizite Joins können immer dort angewendet werden, wo eine `ManyToOne`- bzw. `OneToOne`-Beziehung vorhanden ist. Gleichwohl ist es möglich, mit einem impliziten Join einen Verbund über mehr als zwei Tabellen herzustellen. Implizite Joins sind immer sogenannte *Inner-Joins*.

Rückgabewerte der Abfragen können einzelne oder mehrere Attribute der referenzierten Entities, aber auch Entities selbst sein. Auf die Möglichkeiten der Selektion gehen wir in Kapitel 7.5 ein.

7.3.2 Inner-Joins

Implizite Joins sind nicht in allen Fällen anwendbar. Will man z. B. alle Kunden selektieren, deren Konto einen negativen Saldo aufweist, ist ein expliziter Join anzuwenden. Dabei wird mit Hilfe des Schlüsselwortes `Join` der Verbund zwischen zwei Relationen hergestellt. Auch hier muss keine Verknüpfung über die betreffenden Primär- und Fremdschlüssel hergestellt werden, solange es eine konfigurierte Beziehung zwischen den Entities gibt.

```
select  k, kto
from Kunde k join k.konten kto
where kto.kontostand < 0
```

Das generierte SQL-Statement (im Falle von EclipseLink) sieht wie folgt aus:

```
SELECT t0.ID, t0.NACHNAME, t0.GEBURTSDATUM, t0.VORNAME,
  t0.GESCHLECHT, t0.KUNDENNUMMER, t0.WOHNORT_ID, t1.ID, t1.kontoart,
  t1.KONTOSTAND, t1.NUMMER, t1.GUTHABENZINSSATZ, t1.SOLLZINSSATZ,
  t1.UEBERZIEHUNGSZINSSATZ, t1.KREDITLIMIT
FROM KUNDE t0, KUNDE_KONTO t2, KONTO t1
WHERE
(
  (t1.KONTOSTAND < 0)
  AND
  (
    (t2.kunden_ID = t0.ID)
    AND
    (t1.ID = t2.konten_ID)
  )
)
```

Tipp:

Der Leser sollte, sofern er die Beispiele in den zum Herunterladen bereitgestellten Applikationen nachvollziehen möchte, sich nicht von der Vielzahl der Ausgaben auf der Kommandozeile verwirren lassen. Bei der oben dargestellten Abfrage finden eine Reihe von weiteren Select-Statements auf der Datenbank statt, die mit unserem dargestellten Join nichts zu tun haben. Dies liegt an den Default-Werten der konfigurierten Assoziationen zwischen `Kunde` und `Adresse` sowie `Kunde` und `Konto`.

Die Definition unseres Entity `Kunde` besitzt in diesem Beispiel folgendes Aussehen:

```
@Entity
public class Kunde implements Serializable {

  @OneToOne(cascade = CascadeType.ALL)
  @JoinColumn(name = "WOHNORT_ID")
  private Adresse wohnort;

  @ManyToMany(cascade = CascadeType.ALL)
  private Set<Konto> konten = new HashSet<Konto>();

  // weitere Attribute und Getter/Setter
}
```

Zwischen `Kunde` und `Konto` liegt eine n:m-Beziehung vor, die, wenn nicht anders angegeben, `Lazy` aufgelöst wird (vgl. dazu Kapitel 4). Eine 1:1-Beziehung wird hingegen sofort ausgewertet (*Eager*) und führt dazu, dass zu den selektierten Kunden jeweils noch die zugehörige Adresse aus der Datenbank gelesen wird. Dies kann, wie in diesem Beispiel zu sehen ist, zu einer Reihe weiterer SQL-Abfragen auf der Datenbank führen, obwohl die Abfrage nach Kunden mit negativem Girokonto-Saldo eigentlich in einer Abfrage zu erledigen gewesen wäre.

Eine weitere Möglichkeit, einen Join auszuführen, stellt das Schlüsselwort `IN` bereit. Die Abfrage von Kunden mit negativem Saldo sieht dann folgendermaßen aus:

```
select k, kto
from Kunde k, in(k.konten) kto
where kto.kontostand < 0
```

Wie der Leser erkennt, sind Abfragen in JPQL wesentlich übersichtlicher als ihr Äquivalent im nativen SQL. Join-Beziehungen ergeben sich direkt aus den Beziehungen der beteiligten En-

tities und müssen nicht über geschachtelte `where`-Bedingungen in SQL-Abfragen nochmals nachgebaut werden.

7.3.3 Left-Joins

Ein Left-Join liefert alle Ergebnisse des auf der linken Seite der Join-Beziehung stehenden Entity zurück, auch wenn die Bedingungen der Join-Beziehung nicht erfüllt sind. So kann es in unserer Beispieldatenbank Kunden geben, die kein Konto besitzen, aber als Kunde geführt sind, weil vielleicht Depots oder Vollmachten für andere Konten bestehen, zu denen diese Kunden Zugang haben. Möchte man nun alle Kunden und deren Konten abfragen, zusätzlich aber auch die Kunden ohne Konto haben, bietet sich ein *Left Outer Join* an. In der Syntax ist dabei das Schlüsselwort `outer` optional:

```
select distinct k from Kunde k left join k.konten kto
where k.wohnort.ort = 'Braunschweig'
```

Kunden ohne Konto haben an der entsprechenden Stelle im Object[]-Array `null` stehen, werden aber trotzdem in der Ergebnisliste aufgeführt. Wir haben diese Selektion jedoch unterdrückt, um ein weiteres Schlüsselwort zu erläutern: `distinct`. Hat ein Kunde zwei oder mehr Konten, wird die Referenz auf das entsprechende Objekt mehrmals zurückgeliefert. Dies ist kein Fehler der Spezifikation oder des Providers, sondern liegt in der Natur von SQL bzw. dem Relationenmodell, dem SQL unterliegt. `Distinct` unterbindet diese doppelten Vorkommen erst auf Java-Ebene, die Datenbank liefert zunächst doppelte Einträge an die JVM zurück.

Konditionale Left-Joins sind mit der JPQl derzeit noch nicht möglich. Dies ist jedoch ein oft benötigtes Feature, das nach derzeitigem Stand in die JPA-Version 2.1 Einzug halten wird. Wir geben einen ersten Ausblick dazu in Kapitel 14 auf Seite 325.

Wir wollen unsere obige Suche noch ein wenig ergänzen: Gesucht sind alle Kunden mit dem Wohnort „Braunschweig“. Darüber hinaus wollen wir zu diesen Kunden genau die Konten lesen, deren Kontostand kleiner 0 ist. Haben diese Kunden aber gar kein Konto, sollen sie trotzdem im Ergebnis auftauchen.

Diese Abfrage ist so mit JPQL nicht möglich. Schränkt man in der `Where`-Bedingung auf den `kto.kontostand < 0` ein, verwirft die Datenbank alle Kunden ohne Konto. Hibernate geht an dieser Stelle über den JPA-Standard hinaus und ermöglicht genau diese Abfrage mit dem Schlüsselwort `with`:

```
// Nur Hibernate !!
Query q = em.createQuery(" select k, kto "
    + " from Kunde k left join k.konten kto"
    + " with kto.kontostand < 0"
    + " where k.wohnort.ort = 'Braunschweig'");
List<Object[]> l = q.getResultList();
```

Wir werden uns im Rahmen der Criteria-API im Kapitel 8 ansehen, wie solche Probleme mit JPA gelöst werden können.

7.3.4 Fetch-Joins

Wie in Kapitel 4 bereits beschrieben, lassen sich Assoziationen zwischen Klassen auf verschiedene Art und Weise konfigurieren. Mit der Anweisung `fetch = FetchType.LAZY` wird das referenzierte Objekt nicht sofort aus der Datenbank gelesen, wenn das referenzierende Objekt aufgebaut wird. An seine Stelle tritt ein so genanntes Proxy-Objekt, das erst bei Zugriff mit den entsprechenden Werten gefüllt wird.

Will man nun aber bei einer bestimmten Abfrage dieses Verhalten ändern, müsste eigentlich die Konfiguration des Entity angepasst werden. Dies hätte aber gleichzeitig Auswirkung auf sämtliche anderen Zugriffe auf dieses Entity im Programm und könnte an anderer Stelle z. B. zu erheblichen Performance-Einbußen führen. Um die global eingestellte Strategie für bestimmte Abfragen zu ändern, bietet die JPQL so genannte Fetch Joins an.

Als Beispiel verwenden wir die nachstehende Klasse `Kunde`. Die Assoziation zur Klasse `Konto` ist eine `ToMany`-Beziehung. Sie ist ohne explizite Angabe des `FetchType` bereits `lazy`. Die `ToOne`-Beziehung zur Klasse `Adresse` wird entsprechend konfiguriert.

```
@Entity
public class Kunde implements Serializable {

  @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
  @JoinColumn(name = "WOHNORT_ID")
  private Adresse wohnort;

  @ManyToMany(cascade = CascadeType.ALL)
  private Set<Konto> konten = new HashSet<Konto>();

  // Attribute und Getter/Setter
}
```

Durch Verwenden des Schlüsselwortes `fetch` kann dieser `LAZY`-Modus außer Kraft gesetzt werden:

```
select k from Kunde k
left join fetch k.wohnort
where k.geschlecht = 'm'
```

Bei diesem Ergebnis ist entgegen der eingestellten Konfiguration das Feld `wohnort` bei sämtlichen `Kunde`-Instanzen bereits komplett initialisiert und mit den entsprechenden Werten gefüllt. Um auch noch die dazugehörigen Konten in einem Select-Befehl zu lesen, muss die Abfrage folgendermaßen lauten:

```
select k from Kunde k
left join fetch k.wohnort
left join fetch k.konten
where k.geschlecht = 'm'
```

Der Leser sollte auch hier wieder beachten, dass die Referenz auf ein Kundenobjekt durchaus mehrfach in der Ergebnisliste enthalten sein kann. Die Anzahl seines Auftretens wird durch die Anzahl seiner Konten bestimmt.

Die JPA-Spezifikation erlaubt es nicht, der rechten Seite eines Fetch-Joins Variablen zuzuweisen. Damit ist es auch nicht möglich, diese Variablen der rechten Seite in der Select- oder

Where-Bedingung zu verwenden. Hibernate geht an dieser Stelle über die Grenzen der Spezifikation hinaus und lässt auch nachstehende Abfrage zu.

```
// nur Hibernate !
select k, kto.kontostand from Kunde k
left join fetch k.wohnort
left join fetch k.konten kto
where k.geschlecht = 'm' and kto.kontostand < 0
```

Die dargestellte Abfrage liefert alle männlichen Kunden und den negativen Kontostand. Das Kunden-Objekt ist im Ergebnis komplett initialisiert. Es enthält auch die anderen Konten des Kunden, deren Saldo möglicherweise nicht negativ ist. Ausreichend, um in die Ergebnismenge zu gelangen, ist ein Konto, das sich im negativen Bereich bewegt. Bei OpenJPA und EclipseLink schlägt die Abfrage dagegen fehl.

7.3.5 Joins ohne definierte Mapping-Beziehungen

Die vorangegangenen Joins haben bisher immer definierte Beziehungen zwischen den beteiligten Entities vorausgesetzt. Da dies nicht immer der Fall bzw. für eine Reihe von Abfragen schlichtweg nicht sinnvoll ist, kann die Join-Beziehung explizit in der Abfrage angegeben werden. Die Spezifikation spricht bei diesen Joins von *Theta*-Joins.

Wir wollen mit einer Abfrage herausfinden, welche Konten einen niedrigeren Kontostand haben als ein bestimmtes anderes Konto. Hierfür müssen wir das Entity `Konto` mit sich selbst verknüpfen, was in unserem Business-Modell nicht sinnvoll ist. Die Abfrage dazu lautet:

```
select kto1 from Konto kto1, Konto kto2
where kto1.kontostand <= kto2.kontostand
and kto2.nummer = 837024
```

Das als Referenz benutzte Konto `837024` ist selbst auch in der Ergebnismenge enthalten. Grund dafür ist die <=-Bedingung, die alle Konten aus Menge `kto1` nimmt, deren Kontostand kleiner-gleich des einzigen Kontos aus `kto2` ist.

Achtung:

Bitte verwenden Sie Joins dieser Art mit Bedacht. Ist keine Beziehung zwischen den beteiligten Entities konfiguriert und die Where-Bedingung nicht richtig definiert, ergeben sich aufgrund der Natur relationaler Datenbanken kartesische Produkte, die sehr große Ergebnismengen mit sich bringen können. Lässt man z. B. bei obiger Abfrage die Where-Bedingung weg, ergibt sich eine Liste mit einer Million Einträgen, wenn die Tabelle `Konto` 1000 Sätze umfasst.

7.4 Selektionen mit Where

In den vorangegangenen Abschnitten haben wir schon öfter die Where-Bedingung verwendet, ohne diese zu erklären. Das wollen wir jetzt nachholen.

Durch die Where-Bedingung wird das Ergebnis einer Abfrage oder das Löschen bzw. Verändern von Objekten auf bestimmte Objekte eingegrenzt, die der angegebenen Bedingung genügen. Mehrere Bedingungen können wie in SQL über die logischen Operatoren `AND`, `OR` und `NOT` miteinander verknüpft werden. Bedingungen sind Vergleiche von Objekt-Properties mit anderen Attributen, Zeichenfolgen, Zahlen, aber auch Werten wie `true` oder `false`. Für die Vergleiche stehen die in Tabelle 7.4 aufgeführten Operatoren zur Verfügung.

TABELLE 7.4 Operatoren der JPQL

Operator	Bedeutung
.	Navigieren zum ausgewählten Entity-Attribut
+,–	Vorzeichen von Zahlen
*,/	Multiplikation und Division
+,–	Addition und Subtraktion
=, >, >=, <, <=, <>, [NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL, IS [NOT] EMPTY, [NOT] MEMBER OF, [NOT] EXISTS, ALL, ANY, SOME	Vergleichsoperatoren

Die meisten Operatoren aus Tabelle 7.4 sollten dem Leser bereits aus der SQL-Welt bekannt sein. Zunächst stellen wir einige einfache Beispiele dar, bevor wir auf Subselects eingehen.

Die folgende Abfrage liefert alle Kunden, deren Name mit dem Buchstaben 'M' oder 'N' anfängt und die in Hannover wohnen:

```
select k, k.nachname, k.wohnort from Kunde k
where (k.nachname like 'M%' or k.nachname like 'N%')
and k.wohnort.ort = 'Hannover'
```

Wichtig, wie in SQL auch, ist die korrekte Klammerung. Würde im obigen Beispiel die Klammerung weggelassen, liefert das Ergebnis alle Kunden, deren Nachnamen mit 'N' anfangen **und** in Hannover leben, sowie alle anderen Kunden, deren Nachname mit 'M' anfängt.

Mit einer weiteren Abfrage suchen wir die Kunden, deren Saldo auf einem Sparkonto zwischen 7000 und 8000 Euro liegt und die älter als 60 Jahre sind.

```
Query q = em.createQuery(" select k, k.geburtsdatum, kto.kontostand "
  + " from Kunde k join k.konten kto "
  + " where k.geburtsdatum < :stichtag "
  + " and kto.kontostand between 7000 and 8000 " );
GregorianCalendar heuteCal = new GregorianCalendar(); // heute
heuteCal.add(GregorianCalendar.YEAR, -60); // heute minus 60 Jahre
q.setParameter("stichtag", heuteCal.getTime());
List result = q.getResultList();
```

Seit Version 2.0 lässt die JPA-Spezifikation die Angabe von Date-, Time- bzw. Timestamp-Variablen zu. Das obige Beispiel dazu abgewandelt liest sich dann wie folgt:

```
select k, k.geburtsdatum, kto.kontostand
from Kunde k join k.konten kto
where kto.kontostand between 7000 and 8000
and k.geburtsdatum < {d '1951-01-01'}
```

Analog zur `Date`-Variablen wird die Uhrzeit in der Form `{t hh-mm-ss}` bzw. der Kombination aus beiden `{ts yyyy-mm-dd hh-mm-ss}` angewendet. Wie der Leser erkennen wird, steht der Buchstabe `d` für ein Datum, `t` für eine Uhrzeit und die Buchstaben `ts` für die Angabe von Datum und Uhrzeit. Die Literale sind angelehnt an die Escape-Sequenzen, die auch schon JDBC bereitstellt.

7.4.1 Unterabfragen (Subselects)

Unterabfragen dürfen laut der JPA-Spezifikation in der Where- und Having-Bedingung verwendet werden. Der Einsatz im Bereich des Select-Ausdruckes wird gegenwärtig noch nicht unterstützt, ist aber in einer späteren Version geplant. Wenn die Datenbank selbst keine Subselects unterstützt, bricht die Verarbeitung der Abfrage mit einer Fehlermeldung ab.

Wir suchen in einem weiteren Beispiel unsere „guten“ Kunden. Dies sollen Kunden sein, die ein Konto besitzen, das einen größeren Kontostand hat als der Durchschnitt aller Konten.

```
select kto, kto.kontostand from Konto kto
where kto.kontostand >
  (select avg(kto2.kontostand) from Konto kto2)
```

Die Funktion `avg` liefert genau ein Ergebnis im Subselect. Deshalb ist es auch möglich, den Kontostand aller einzelnen Konten mit diesem Ergebnis zu vergleichen. Liefert eine Unterabfrage dagegen mehrere Ergebnisse, müssen Operatoren wie `ANY`, `ALL` oder `EXISTS` benutzt werden.

Schauen wir uns nun noch weitere Beispiele mit den Operatoren `ANY`, `ALL` und `EXISTS` an. Diese Konstrukte ermöglichen die Anwendung von Subselects, die mehrere Ergebnisse liefern.

Gibt es ein Girokonto, das einen höheren Saldo aufweist als alle anderen Sparkonten?

```
select giro, giro.kontostand
from Girokonto giro
where giro.kontostand > ALL
  (select spar.kontostand from Sparkonto spar)
```

Diese Abfrage ließe sich auch mit einer `MAX`-Funktion formulieren. Genaueres dazu findet der Leser in Abschnitt 7.5.1.

Als Nächstes möchten wir Kunden, die einen negativen Saldo auf ihrem Girokonto haben, gleichzeitig aber ein Sparkonto mit einem Guthaben größer 7000 Euro, ein Kreditlimit einrichten. Welche Kunden kommen dafür in Frage?

```
select k, k.id
from Kunde k
where exists (select spar from Sparkonto spar join spar.kunden sk
      where spar.kontostand > 7000
      and k.id = sk.id)
and exists (select giro from Girokonto giro join giro.kunden gk
      where giro.kontostand < 0
      and k.id = gk.id)
```

Seit Version 2.0 lassen sich auch `Collections` als Parameter für eine Abfrage verwenden. Wir zeigen dieses am Beispiel der `IN`-Funktion:

```
Query q = em.createQuery("select k.nachname, k.kundennummer "
  + " from Kunde k "
  + " where k.nachname in :namen");
List<String> nl = Arrays.asList("Meier", "Schulze", "Wehr");
q.setParameter("namen", nl);
List<Object[]> l = q.getResultList();
```

7.4.2 Funktionen

Mit Hilfe von Funktionen lässt sich die JPQL noch einmal wesentlich erweitern. Funktionen dürfen sowohl in der Where-Bedingung als auch nach dem `select` stehen. Folgende Liste erklärt diese Funktionen im Detail.

- `ABS`: Liefert den absoluten Betrag einer Zahl (Integer, Float, Double) und behält dabei den übergebenen Typ bei.

  ```
  select abs(k.kontostand), k.kontostand from Konto k
  ```

- `CASE`: Mit Hilfe der `CASE`-Anweisung lassen sich komplexe Wenn-dann-Bedingungen in einem SQL-Befehl abbilden. Der `ELSE`-Zweig ist laut Spezifikation nicht optional, Hibernate gestattet die Ausführung auch ohne ihn.

  ```
  CASE
    WHEN ... THEN
    WHEN ... THEN
    ELSE ...
  END

  select
    case when k.kontostand < 0 THEN 'Schlecht'
         when (k.kontostand > 0 and k.kontostand <= 5000) THEN 'Gut'
         when k.kontostand > 5000 THEN 'Sehr gut'
         ELSE ''
    END as bonitaet, k.kontostand
  from Konto k
  ```

  Die weiter unten erläuterten Funktionen `NULLIF` und `COALESCE` ließen sich auch mit einer CASE-Anweisung realisieren. Da diese Anweisungen, wie auch `CASE`, bereits in einigen Datenbank-Dialekten existierten, wurden sie auch in den Sprachumfang von JPA 2.0 übernommen.

- `CONCAT`: Verbindet die übergebenen Argumente zu einem Ergebnis.

  ```
  select concat(k.nachname, k.vorname) from Kunde k
  ```

- `COALESCE`: Liefert den ersten Wert aus der Parameterliste, der nicht `NULL` ist.

  ```
  select COALESCE(k.nachname, k.vorname, k.geschlecht) from Kunde k
  ```

- `CURRENT_DATE`: Das aktuelle Datum.
- `CURRENT_TIME`: Die aktuelle Uhrzeit.
- `CURRENT_TIMESTAMP`: Aktuelles Datum + Uhrzeit.
- `INDEX`: Ermöglicht das direkte Zugreifen auf ein Element einer Liste über dessen Position. Die abgebildete Abfrage funktioniert mit unserem zugrunde liegenden Modell nicht, da die

1:n-Beziehung vom Kunden zu seinen Konten über ein `Set` implementiert ist und nicht über eine Liste.

```
select k.nachname, kto.kontostand from Kunde k join k.konten kto
where index(kto) = 1
```

- `LENGTH`: Liefert die Länge eines Strings.

```
LENGTH(string)
```

- `LOCATE`: Liefert die Position des Parameters `suchstring` in `string`, startet aber erst ab der Stelle `position`. Wird der String nicht gefunden, liefert die Funktion den Wert 0 zurück. Die erste Stelle eines Strings hat die Position 1.

```
LOCATE(suchstring, string, position)

select LOCATE('eier', k.nachname, 1 ), k.nachname from Kunde k
where LOCATE('eier', k.nachname, 1 ) > 0
```

- `LOWER`: Wandelt einen String in Kleinbuchstaben um.

```
LOWER(string)
```

- `MOD`: Division zwischen `zahl1` und `zahl2`, die den Rest als Ergebnis liefert. Erwartet werden zwei Integer, Rückgabewert ist ebenfalls ein Integer. Bei OpenJPA und Hibernate funktioniert auch die unten stehende Abfrage. EclipseLink verweigert die Ausführung mit dem Hinweis, dass das Attribut unseres Business-Modells kein Integer ist.

```
MOD(zahl1, zahl2)

select mod(k.kontostand, k.kontostand), k.kontostand from Konto k
```

- `NULLIF`: Vergleicht den ersten Parameter mit dem zweiten. Sind beide Werte gleich, wird `NULL` zurückgegeben, sonst wird der erste Parameter retourniert.

```
select nullif(k.nachname, k.vorname) from Kunde k
```

- `SIZE`: Gibt die Größe der übergebenen `collection` wieder.

```
SIZE(collection)
select k.nachname, size(k.konten) from Kunde k
```

- `SUBSTRING`: Liefert den Wert von `string` ab der Stelle `position`. Die Anzahl der Zeichen entspricht dem Argument `laenge`.

```
SUBSTRING(string, position,
        laenge)

select SUBSTRING(k.nachname,0 , 5), k.nachname from Kunde k
```

- `SQRT`: Berechnet die Wurzel einer Zahl und liefert das Ergebnis als Double zurück.

```
SQRT(zahl)
```

- `TRIM`: Entfernt den Buchstaben `char` aus dem angegebenen String. Die Angabe eines Zeichens ist optional. Wenn nicht konfiguriert, wird `SPACE` angenommen. Bei `LEADING` wird nur am Anfang des Strings entfernt, `TRAILING` entfernt nur am Ende, `BOTH` auf beiden Seiten.

```
TRIM([[BOTH|LEADING|TRAILING]
  [char] FROM]
  string
)
```

- `TYPE`: Liefert den exakten Typ des Argumentes zurück. Die Funktion kann verwendet werden, um eine Abfrage auf eine bestimmte Klasse (z. B. Sparkonto) einzuschränken und nicht alle Subklassen einer Oberklasse zu erhalten.

```
select type(k), k.nummer, k.kontostand from Konto k
where type(k) = Sparkonto
```

- `UPPER`: Wandelt einen String in Großbuchstaben um.

```
UPPER(string)
```

Mit der Where-Bedingung sind wir nun in der Lage, die Menge der zurückgelieferten Entities entsprechend unseren Suchbedingungen einzuschränken. Was wir letztendlich als Ergebnis bekommen, legt die Projektion innerhalb des Select-Statements fest, das wir im folgenden Abschnitt betrachten.

7.5 Projektionen mit Select

In der JPQL können nach dem Schlüsselwort `select` die folgenden Elemente (auch mehrfach) enthalten sein:

- **Pfadausdrücke** geben genau ein Attribut eines Entity an. Der Typ des Ergebnisses entspricht dabei dem Typ des Attributs im Entity. Handelt es sich dabei um einen primitiven Datentyp, wird die Wrapper-Klasse verwendet. Bei 1:1- und n:1-Beziehungen kann es auch vorkommen, dass der Pfadausdruck auf ein weiteres Entity zeigt. Der Rückgabetyp entspricht dem Typ aus den Mapping-Informationen. Pfadausdrücke haben immer die folgende Form:

  ```
  variable.attribut[.attribut]*
  ```

  Mittels der Punktnotation werden die entsprechenden Properties der Entity-Instanz ausgewählt.
- Wird statt eines Pfades die **Variable** selbst im Select-Statement aufgeführt, wird nicht ein Wert der Instanz, sondern das gesamte Objekt zurückgegeben.
- **Aggregationen** sind spezielle SQL-Funktionalitäten, die eine Reihe von Instanzen bzw. deren Felder unter bestimmten Gruppierungen zusammenfassen. Ihr Rückgabewert ist abhängig von der Art der Aggregation (Summe, Durchschnitt, Maximum, Minimum) und des zugrunde liegenden Attributs. Beispiele dazu geben wir in Abschnitt 7.5.1.
- Sollen Objekte erzeugt werden, die nicht als Entity im Sinne der JPA definiert sind, werden **Konstruktoren** im Select-Statement benutzt. Der Rückgabewert entspricht der Klasse des verwendeten Konstruktors (Abschnitt 7.5.4).

Wie in der obigen Aufzählung beschrieben, können nach einem `select` durch Kommata getrennt mehrere Elemente stehen. Ist dies der Fall, liefert die Abfrage ein Objekt-Array (`Object[]`) pro Zeile zurück.

Die nachfolgende, nicht sehr sinnvolle Abfrage soll dies verdeutlichen:

```
select k, k.wohnort.ort, k.geburtsdatum from Kunde k
```

Das Ergebnis von `Query.getResultList()` ist eine Liste von `Object[]`-Arrays, die an Stelle 0 Kunden-Objekte, an Stelle 1 Strings und an Stelle 2 Date-Objekte enthalten.

```
java.util.List
  |_[de.jpainfo.model.Kunde, java.lang.String, java.util.Date]
  |_[de.jpainfo.model.Kunde, java.lang.String, java.util.Date]
  |_[de.jpainfo.model.Kunde, java.lang.String, java.util.Date]
  |_[de.jpainfo.model.Kunde, java.lang.String, java.util.Date]
  ...
```

Verwenden wir nun in einem weiteren Beispiel die Abfrage der Kunden mit negativem Saldo auf dem Girokonto aus einem der vorangegangenen Abschnitte:

```
select k, kto
from Kunde k join k.konten kto
where kto.kontostand < 0
```

Welche Typen enthält die Ergebnisliste der Query an welcher Stelle? Da wir mehrere Ergebnisse erwarten, fragen wir das Ergebnis mit `getResultList()` ab. Über die zu iterierende Liste gelangen wir auf der nächsten Ebene zu einem `Object[]-Array`. An der ersten Stelle des Array steht der jeweilige Kunde, die zweite Stelle hält das Konto-Objekt mit negativem Saldo bereit.

7.5.1 Aggregationsfunktionen

Aggregationsfunktionen haben je nach Art der Aggregation unterschiedliche Rückgabewerte. JPA definiert die folgenden Funktionen, die dem Leser mit SQL-Kenntnissen sicher bekannt sind:

AVG liefert den Durchschnitt eines Attributs einer bestimmten Menge von selektierten Daten. Rückgabewert dieser Funktion ist immer `java.lang.Double`.

COUNT zählt die ausgewählten Datensätze und liefert ein `java.lang.Long`.

MAX, MIN Die Funktionen zum Ermitteln des Maximums und Minimums liefern den Rückgabewert des Feldes, auf das sie angewendet worden sind.

SUM gibt die Summe eines Feldes aus. Ist das Attribut vom Typ `BigDecimal` bzw. `BigInteger`, wird dieser Typ zurückgegeben. Bei Gleitkommatypen wie `Float` oder `Double` ist der Rückgabetyp ein `Double`, in allen anderen Fällen ein `Long`.

Beispiele

Die folgende Abfrage ermittelt die Anzahl aller Kunden:

```
select count(k) from Kunde k
```

Die ältesten Kunden der Bank lassen sich mit folgender Abfrage herausfinden:

```
select k.geburtsdatum, k.nachname, k.vorname
from Kunde k
```

```
where k.geburtsdatum = ( select min(k2.geburtsdatum)
                         from Kunde k2 )
```

Bis jetzt liefern uns die beschriebenen Funktionen nur eine einzige Zeile als Ergebnis. Wir können die Aggregation aber auch dazu verwenden, um Ergebnisse für bestimmte Bereiche unserer Ergebnismenge zu berechnen. Die Konstrukte `Group by` und `Having` stellen diese Funktionalität zur Verfügung.

7.5.2 Gruppieren von Daten mit Group By und Having

`Group by` gruppiert die Ergebnismenge anhand bestimmter Eigenschaften. `Having` wendet ähnlich wie die Where-Klausel eine Selektion auf die mit `group by` erzeugten Gruppen an, die die aufgezeigten Bedingungen erfüllen müssen.

Wir wollen die Konstrukte zunächst anhand eines einfachen abstrakten Beispiels auf SQL-Ebene beschreiben. Erst danach wenden wir uns der Anwendung in JPQL zu.

Gegeben sei die nachfolgende Tabelle `Zahlen` mit vier Spalten, `A`, `B`, `C` und `D`. Die Tabelle besitzt insgesamt 8 Zeilen:

A	B	C	D
1	2	3	4
1	2	4	5
2	3	3	4
3	3	4	4
3	3	6	7
5	3	3	4
5	3	4	4
5	3	6	7

In welchen Schritten wird das nachfolgende SQL-Statement auf dieser Tabelle ausgeführt?

```
select A, sum(D)
from zahlen
where A < 5
group by A,B
having A < 4 and sum(D) < 10 and max(C) = 4;
```

Zunächst gelangen die From- und Where-Bedingungen zur Ausführung. Übrig bleiben die Zeilen der Tabelle, die in der Spalte A einen Wert < 5 haben.

A	B	C	D
1	2	3	4
1	2	4	5
2	3	3	4
3	3	4	5
3	3	6	7

In der verbliebenen Datenmenge gruppieren sich die Zeilen zu bestimmten Zeilengruppen. Jede Gruppenausprägung tritt dabei nur einmal auf.

A	B	N	
		C	D
1	2	3	4
		4	5
2	3	3	4
3	3	4	5
		6	7

Um die Having-Klausel zu berücksichtigen, muss die jeweilige Gruppensumme der Spalte D berechnet werden. Diese wird außerdem für das anschließende `select` benötigt.

A	sum(D)	N	
		C	D
1	9	3	4
		4	5
2	4	3	4
3	12	4	5
		6	7

`Having` filtert bestimmte Gruppierungen heraus. Nur die Zeilen, deren Summe aus D < 10 ist und deren Maximum in Spalte C gleich 4 ist, werden berücksichtigt. `Select` wählt aus dieser Restmenge die geforderten Spalten und deren Werte aus.

A	sum(D)
1	9

Wie in SQL muss auch in der JPQL jedes Feld, das nicht in einer Aggregatsfunktion im `select` steht, auch in der Group-By-Anweisung auftauchen.

Mit der nun folgenden Abfrage kehren wir zu unserem Beispiel zurück und suchen die Orte unseres Marktgebietes, an denen wenige Kunden in Summe sehr große Guthaben bei uns unterhalten:

```
select k.wohnort.ort, count(k), sum(kto.kontostand)
from Kunde k join k.konten kto
group by k.wohnort.ort
having count(kto) < 10
and sum(kto.kontostand) > 40000
```

Ergebnisse dieser Art sind meist keine Entities im herkömmlichen Sinne. JPA liefert als Ergebnis der obigen Abfrage eine Liste von `Object[]`-Arrays der Form `String`, `Integer`, `BigDecimal`. Wie die Werte dieser Arrays gleich in neue Java-Objekte eingebettet werden können, zeigen wir in Kapitel 7.5.4.

7.5.3 Ergebnisse sortieren mit Order By

JPQL bietet auch die Möglichkeit, das Ergebnis einer Abfrage zu sortieren. Die Order-By-Anweisung kann folgende Elemente (auch mehrfach) enthalten:

- Variablen, die an ein Entity in der From-Klausel gebunden sind;
- Pfadausdrücke, die auf ein Attribut zeigen, das keine `Collection` enthält.

Generell müssen alle Elemente, die in der Order-By-Anweisung enthalten sind, auch im entsprechenden Select-Statement aufgeführt sein. Ist in der Projektion das Element schon in Form des gesamten Entity vorhanden, kann auf die nochmalige Nennung verzichtet werden. Wir geben im Beispiel zur Kontrolle des Ergebnisses nochmals alle Elemente an.

```
select k, k.nachname, k.vorname
from Kunde k
order by k.nachname desc, k.vorname desc
```

Folgende Anweisung ist unzulässig, weil die Order-By-Anweisung ein Element enthält, das nicht in der Select-Anweisung aufgeführt ist. Hibernate, OpenJPA und EclipseLink führen diesen Befehl entgegen der Spezifikation trotzdem korrekt aus:

```
select k.nachname, k.vorname, k.kundennummer
from Kunde k join k.konten kto
order by kto.kontostand desc
```

Die Anweisungen `ASC` für aufsteigend bzw. `DESC` für absteigend sind äquivalent zu Standard-SQL und beeinflussen die Sortierrichtung.

7.5.4 Konstruktoren im Select-Statement

Mit Hilfe der JPQL ist es möglich, Instanzen von Java-Klassen zu erzeugen, die keine Entities sind. Dazu lassen sich im Select-Statement Java-Konstruktoren aufrufen. Diesem Konstruktor können beliebige Parameter aus der eigentlichen Abfrage übergeben werden. Es ist außerdem zulässig, auf diesem Wege Instanzen zu erzeugen, deren Klasse als Entity deklariert ist. Diese Java-Objekte sind dann automatisch im Zustand *neu* und noch nicht an den Persistenzkontext gebunden.

```
select new de.jpainfo.model.auswertung.KundeStadt(
  k.wohnort.plz, k.wohnort.ort, count(k))
from Kunde k
group by k.wohnort.plz, k.wohnort.ort
```

Die Klasse `KundeStadt` ist nicht als Entity definiert. Sie soll lediglich die Hülle für die Auswertung bereitstellen, wie viele Kunden in einer bestimmten Stadt bzw. einem bestimmten Stadtgebiet wohnen. Der Konstruktor dieser Klasse muss drei Parameter entgegennehmen: `String`, `String`, `Long`. Andernfalls würde diese Abfrage mit einer Fehlermeldung abbrechen.

```
// KundeStadt muss kein Entity sein
public class KundeStadt {
  private Long anzahlKunden;
  private String plz;
  private String ort;

  public KundeStadt(String plz, String wohnort, Long anzahl) {
    this.plz = plz;
    this.ort = wohnort;
    this.anzahlKunden = anzahl;
  }
}
```

7.6 Update- und Delete-Statements

Die JPA-Query-Language bietet die Möglichkeit, Update- und Delete-Statements durchzuführen, ohne dass dafür Java-Objekte (Entities) instanziiert werden müssen. In der Spezifikation werden diese Batch-Befehle auch als *Bulk Updates* und *Deletes* bezeichnet.

Statements dieser Art können z. B. bei Datenmanipulationen erforderlich sein, mit denen Hunderttausende und mehr Datensätze effizient verändert werden müssen. Entgegen den vorangegangenen Beispielen, wo mit Hilfe von Join-Bedingungen mehrere Entities in einer

Abfrage kombiniert werden konnten, kann hier nur genau ein Entity im JPQL-Statement stehen.

```
em.getTransaction().begin();
Query q = em.createQuery("delete from Konto k " +
  " where k.nummer > 10000 ");
int result = q.executeUpdate();
em.getTransaction().commit();
System.out.println("Saetze geloescht: "+result);
```

Diese Abfrage bezieht sich nicht nur auf die Klasse `Konto`. Sie bezieht auch die erbenden Klassen `Girokonto` und `Sparkonto` in das DML-Statement mit ein. Rückgabewert der Methode `executeUpdate` ist die Anzahl der Entities, die von der Ausführung betroffen sind.

Tipp:

Referenzierte Entities werden von einem Batch-Befehl nicht berücksichtigt. Darüber hinaus berücksichtigen sie keine optimistischen Sperren. Dies wird der Leser auch bemerken, wenn er den obigen Befehl gegen unsere zur Verfügung gestellte Datenbank absetzt. Das Programm beendet sich mit einer Fehlermeldung, da noch referentielle Abhängigkeiten zwischen der Tabelle `Kunde_Konto` und unserer `Konto`-Tabelle bestehen.

Zum Testen des obigen Befehls können wir zeitweise in der H2-Datenbank die referentielle Integrität deaktivieren:

```
ALTER TABLE KUNDE_KONTO SET REFERENTIAL_INTEGRITY false;
```

Danach erfolgt die Löschung ohne weitere Fehlermeldung. Die Datenbank ist danach natürlich in einem nicht konsistenten Zustand. Durch Stopp und Neustart von H2 über unser ANT-Target kann sie aber jederzeit wiederhergestellt werden. Bei Oracle, Postgres und MySQL muss der Leser die Skripte erneut einspielen.

Ein Update-Statement hat die folgende Form:

```
em.getTransaction().begin();
Query q = em.createQuery("update Adresse a set a.ort "
  + "= :newName where a.ort = :oldName ");
q.setParameter("oldName", "Hannover");
q.setParameter("newName", "Hannover-Stadt");
int result = q.executeUpdate();
em.getTransaction().commit();
System.out.println("Saetze veraendert: " + result);
```

Achtung:

Zu beachten ist, dass diese Art von Anweisungen Entities in einem geraden aktiven Persistenzkontext nicht verändern. Batch-Updates und -Deletes sollten immer in einer eigenen Transaktion durchgeführt werden. Update- und Delete-Operationen werden außerdem ohne Berücksichtigung vorhandener Locks ausgeführt. Sie verändern keine `version`-Attribute der Entities, dies muss, soweit erforderlich, manuell durchgeführt werden.

7.7 Flush-Modi und Abfragen

Die Klasse `EntityManager` unterscheidet zwischen zwei verschiedenen Flush-Modi. Sie werden mit der Methode `setFlushMode(FlushModeType flushMode)` entsprechend konfiguriert. Voreingestellt ist der Modus `FlushModeType.AUTO`. Er gewährleistet bei einer Abfrage der Datenbank, dass zunächst **alle** Änderungen der vorhandenen Java-Objekte, die die Abfrage beeinflussen könnten, in die Datenbank geschrieben werden. Erst dann wird die eigentliche Abfrage durchgeführt. Sollte keine Abfrage in der Transaktion erfolgen, wird die Synchronisation mit der Datenbank spätestens bei `commit()` der Transaktion durchgeführt.

`FlushModeType.COMMIT` lässt den `EntityManager` Änderungen erst bei Beendigung der Transaktion in die Datenbank bringen. Dieser Modus kann dazu führen, dass bei einer Abfrage Daten gelesen werden, die nicht aktuell sind, weil Änderungen noch im Persistenzkontext vorhanden sind, aber noch nicht mit der Datenbank synchronisiert wurden. Die Spezifikation überlässt es an dieser Stelle aber dem JPA-Provider, auch schon früher die Änderungen in die Datenbank zu schreiben. Konkret heißt es dazu in der Spezifikation:

If FlushModeType.COMMIT is specified, flushing will occur at transaction commit; the persistence provider is permitted, but not required, to perform to flush at other times.

Konkret bedeutet dieser Satz, dass ein Entwickler von der JPA-Implementierung nicht erwarten kann, dass die Updates bei `FlushModeType.COMMIT` erst am Ende einer Transaktion in die Datenbank geschrieben werden. Dieser Flush-Modus kann also nur als Hinweis für den Provider angesehen werden, dass es nicht nötig ist, Änderungen sofort in die Datenbank zu schreiben.

Auch das Query-Interface kennt die Methode `setFlushMode()`. Damit kann das beim `EntityManager` gesetzte Verhalten für diese eine Abfrage überschrieben werden. Schauen wir uns diese Auswirkungen an einem Beispiel einmal näher an.

```
// Auto (Default) FlushModeType
EntityManager em1 = emf.
    createEntityManager();
em1.setFlushMode(FlushModeType.
    AUTO);
Query q1 = em1.createQuery("");
Query q2 = em1.createQuery("");
Query q3 = em1.createQuery("");
q3.setFlushMode(FlushModeType.
    COMMIT);
```

```
// Commit FlushModeType
EntityManager em2 = emf.
    createEntityManager();
em1.setFlushMode(FlushModeType.
    COMMIT);
Query q4 = em1.createQuery("");
Query q5 = em1.createQuery("");
Query q6 = em1.createQuery("");
q6.setFlushMode(FlushModeType.
    AUTO);
```

Im linken Code-Beispiel können nur die Abfragen 1 und 2 sicher davon ausgehen, dass der JPA-Provider vor der eigentlichen Ausführung der Abfrage auch alle Änderungen des Persistenzkontextes in die Datenbank schreibt. Dabei müssen nur jene Änderungen sofort in die Datenbank, die das Ergebnis der Abfragen beeinflussen könnten. Der rechte Programmausschnitt sichert nur der letzten Abfrage 6 zu, dass vor Ausführung der Abfrage alle relevanten Änderungen vorher in der Datenbank sind.

Wir wollen uns die Auswirkungen des Flush-Modus bewusst machen, indem wir uns die Ausführungsreihenfolge des verwendeten JPA-Providers in der Konsole näher ansehen. Dabei gehen wir von folgendem Beispielprogramm aus. Wichtig ist die Tatsache, dass der Kunde mit der Id 1 noch nicht den Nachnamen „Mueller-Wehr“ hat. Ansonsten würde die Implementierung gar kein Datenbank-Update durchführen.

```
EntityManager em = emf.createEntityManager();
em.setFlushMode(FlushModeType.COMMIT);
em.getTransaction().begin();
Kunde k = em.find(Kunde.class, 1);
k.setNachname("Mueller-Wehr");
Query q = em.createQuery("select k from Kunde k where nachname like '
    Mueller-Wehr'");
List<Kunde> l = q.getResultList();
```

Mit unserem Beispiel haben wir den Flush-Modus von der Default-Einstellung `AUTO` auf `COMMIT` gesetzt. Die Log-Ausgabe der abgesetzten Statements geben wir verkürzt wieder. Für unser Beispiel haben wir Hibernate verwendet:

```
...
21:22:44,606 DEBUG SQL:111 - /* load de.jpainfo.model.Kunde */
...
21:22:44,733 DEBUG SQL:111 - /* select k from Kunde k
                                 where nachname = 'Mueller-Wehr' */
...
21:22:44,800 DEBUG SQL:111 - /* update de.jpainfo.model.Kunde */
...
```

Mit dem Hinweis an die JPA-Implementierung, Änderungen erst bei Commit der Transaktion in die Datenbank zu schreiben, können wir nicht mehr erwarten, diesen Kunden bei unserem anschließenden Select-Statement in der Ergebnismenge zu bekommen. Dies kann bei bestimmten Anwendungsfällen, z.B. umfangreichen Update-Statements, notwendig sein.

Lassen wir hingegen die Zeile mit der Anweisung `setFlushMode` weg, greift die Standard-Einstellung `AUTO` wieder. Die Ausführungsreihenfolge der Statements ändert sich wie folgt:

```
...
21:34:10,247 DEBUG SQL:111 - /* load de.jpainfo.model.Kunde */
...
21:34:10,398 DEBUG SQL:111 - /* update de.jpainfo.model.Kunde */
...
21:34:10,409 DEBUG SQL:111 - /* select k from Kunde k where
                                 nachname = 'Mueller-Wehr' */
...
```

Nur dieser Modus gibt uns die Sicherheit, den zuvor geänderten Kunden auch in unserer Abfrage zu haben. Der Leser ist aufgefordert, dieses Beispiel nachzuvollziehen. Das Programm-Listing kann wie alle anderen Beispiele von der Homepage zu unserem Buch heruntergeladen werden.

Achtung:

Das Setzen des `FlushModeType.COMMIT` ist nur ein Hinweis für den Provider. Die Spezifikation erlaubt ihm bei `COMMIT`, auch vor dem eigentlichen Commit der Transaktion die Änderungen bereits in die Datenbank zu schreiben. Wir haben das Beispiel auch mit OpenJPA und EclipseLink durchgeführt. Sie verhalten sich ebenfalls so, wie wir die Ausgabe von Hibernate dargestellt haben.

8 Das Criteria-API

In unserem Buch zur JPA 1.0 [MW07] sind wir bereits auf Hibernates *Query by Criteria* und *Query by Example* eingegangen. Damals waren diese Möglichkeiten zur Datenabfrage noch nicht in der JPA-Spezifikation festgelegt.

In Version 2.0 wurden nun wesentliche Bestandteile dieser Techniken adaptiert und festgelegt. Weil wir den Leser weitestgehend dazu bewegen möchten, provider-unabhängig zu implementieren, werden wir im aktuellen Buch den JPA-Standard beschreiben.

Wir gehen zunächst auf das so genannte Meta-Modell ein. Es ist Voraussetzung für ein korrektes Verständnis der Criteria-API. In weiteren Abschnitten demonstrieren wir dann die eigentliche Funktionsweise wie gewohnt anhand von Beispielen.

Das Meta-Modell der JPA bildet die Elemente der Persistence-API in Java-Strukturen ab. Zu finden sind diese Elemente im Paket `javax.persistence.metamodel`. Die dort definierten Interfaces werden von den Providern genutzt, um ihrerseits die eigentliche Implementierung und Persistierung beliebiger Business-Modelle des Entwicklers im Rahmen der Criteria-API durchzuführen.

Diese Tatsache wird den Leser jedoch nur am Rande interessieren. Schließlich ist es nicht von Belang, wie eine Implementierung die Daten in die Datenbank bringt und wieder ausliest, solange sie den Konventionen der JPA folgt und bestimmte Eigenschaften wie etwas die Performance garantiert sind. Wozu benötigen wir also dieses Meta-Modell und das Criteria-API? Dazu möchten wir zunächst den folgenden kleinen Programmausschnitt aus Kapitel 7 wiederholen und diskutieren.

```
EntityManager em = emf.createEntityManager();
TypedQuery<Kunde>  tq = em.createQuery("select k from Kunde where "
        + " k.nachname = 'Meier'", Kunde.class);
List<Kunde> tl = tq.getResultList();
for (Kunde kunde : tl) {
  System.out.println(kunde.getNachname() + " " +kunde.getVorname());
}
em.close();
```

Die Klasse mit diesem Ausschnitt kompiliert ohne Probleme, und dennoch kommt es zu einem Laufzeitfehler. Der JPQL-String enthält einen Syntaxfehler. Die Definition der Variablen `k` für das Entity `Kunde` in der `from`-Klausel fehlt, die Variable wird aber trotzdem verwendet. Diesen Fehler kann das System erst bei der Ausführung des Programms erkennen. Für den Compiler ist es zunächst nur ein einfacher, aus Java-Sicht korrekter String.

Um Fehler wie diesen zu vermeiden, sieht die JPA die Generierung eines statischen Meta-Modells zur Compile-Zeit vor. Darüber hinaus lässt sich auch zur Laufzeit ein dynamisches Modell aus den konfigurierten Annotationen auslesen.

8.1 Der statische Zugriff

Annotationen gehören seit Java 5.0 zum offiziellen Sprachumfang. Es handelt sich um weiterführende Informationen, die an zentraler Stelle im Quellcode abgelegt sind. JPA benutzt diese Annotationen, dem Leser sind sie aus den vorangegangenen Kapiteln bekannt. Um diese Annotationen auszuwerten und ggf. weitere Dateien bzw. Informationen zu generieren, wird ein *Annotation Processor* benötigt. Im Fall der Criteria-API erzeugt der Annotation-Processor Java-Klassen, mit denen man typsichere Abfragen generieren kann.

8.1.1 Grundsätze der Generierung

Die JPA-Spezifikation legt fest, wie die statischen Meta-Klassen zu generieren sind [URL-JPA2]:

- Für jede verwaltete Klasse `X` einer Persistenzeinheit wird eine äquivalente Meta-Klasse `X_` generiert. Der Klassenname wird also mit einem Unterstrich ergänzt. Der Paketname bleibt erhalten.
- Die Meta-Klasse wird mit der Annotation `javax.persistence.metamodel.StaticMetamodel` gekennzeichnet.
- Erbt eine Klasse `X` von einer Oberklasse `O`, muss dies auch im statischen Meta-Modell erfolgen. `X_` muss also auch von `O_` erben.
- Für jedes persistente Attribut `t` einer Klasse `X`, das nicht vom Typ einer Collection ist, wird ein statisches Attribut vom Typ `SingularAttribute` erzeugt:

  ```
  public static volatile SingularAttribute<X,T> t;
  ```

  Der Datentyp `T` der Variablen wird dabei aus der *Originalklasse* übernommen.
- Basiert der Datentyp der Variablen `y` von Klasse `X` auf einer Collection, wird je nach Typ der Collection das entsprechende Attribut generiert. Unterschieden wird zwischen den Interfaces `Collection`, `Set`, `List` und `Map` aus dem Paket `java.util`:

  ```
  public static volatile CollectionAttribute<X,E> e;
  public static volatile SetAttribute<X,E> e;
  public static volatile ListAttribute<X,E> e;
  public static volatile MapAttribute<X,K,V> m;
  ```

Im Klassenmodell der Abbildung 8.1 haben wir die Zusammenhänge der vorhandenen `Attribute`-Objekte noch einmal zusammengefasst.

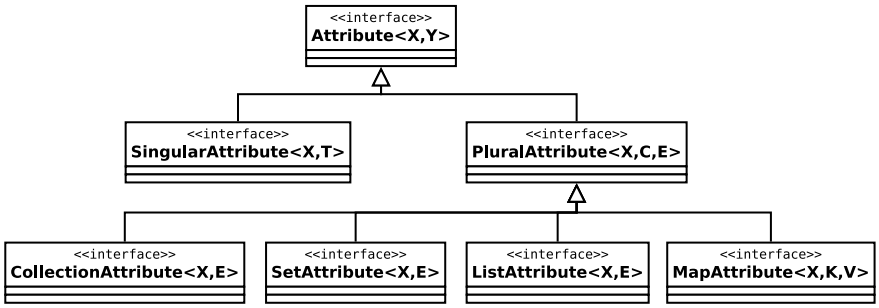


ABBILDUNG 8.1 Klassenhierarchie der Attribute-Interfaces aus JPA 2.0

8.1.2 Manuelle und automatische Generierung

In unseren Projekten greifen wir dabei auf den Generator von OpenJPA zurück, aber auch EclipseLink und Hibernate haben solche Generatoren, um ein statisches Meta-Modell zu generieren. Der Leser ist aufgefordert, auch diese bei Interesse auszuprobieren.

In der Java-Version 5.0 musste zur Verarbeitung der Annotationen noch das externe Kommandozeilenprogramm `apt` verwendet werden. Ab Java-Version 6.0 wird die Verarbeitung direkt vom Compiler durchgeführt. Zur Compile-Zeit wird im Klassenpfad im Verzeichnis `/META-INF/services` die Datei `/META-INF/services/javax.annotation.processing.Processor` gesucht. Enthält diese Datei einen Klassennamen, der auf einen Annotation Processor verweist, findet die Verarbeitung automatisch statt.

Unsere JPA-Implementierungen stellen jeweils die folgenden Klassen bereit:

```
org.apache.openjpa.persistence.meta.AnnotationProcessor6
org.eclipse.persistence.internal.jpa.modelgen.CanonicalModelProcessor
org.hibernate.jpamodelgen.JPAMetaModelEntityProcessor
```

Wie der Klassenname von EclipseLink zeigt, spricht die JPA an dieser Stelle auch von einem *kanonischen* Modell, wenn es analog zu den Regeln auf Seite 208 gebaut wurde. Die Implementierungen von OpenJPA und EclipseLink benötigen beide spezifische Kommandozeilenparameter. In unserem Projektverzeichnis enthält die Datei `common-targets.xml` die Task, die die Generierung übernimmt.

```
<target name="generateModel" description="Generiert das Meta-Model">
  <delete dir="${metamodel.home}" />
  <mkdir dir="${metamodel.home}" />
  <javac srcdir="${src.home}" destdir="${metamodel.home}"
      failonerror="false" fork="true">
    <classpath refid="gen-classpath" />
    <compilerarg value="-proc:only" />
    <compilerarg value="-Aopenjpa.metamodel=true" />
  </javac>
</target>
```

Nach Ausführen der Task liegen die Dateien im jeweiligen Projektverzeichnis im Ordner `metamodel`. Das Argument `-proc:only` weist den Compiler an, nur die Annotationen auszuwerten. Der restliche Quellcode wird nicht betrachtet. Die weiteren Kommandozeilenoptionen sind obligatorische Argumente für OpenJPA und EclipseLink.

Projekt:

Unser Eclipse-Projekt `criteria-api` bietet dem Leser die Möglichkeit neben der statischen Generierung auch alle weiteren Beispiele dieses Kapitels nachzuvollziehen. Unser `ANT`-Target generiert dazu die statischen Klassen in das Verzeichnis `metamodel` des Projektes. Im Projekt selbst ist das Verzeichnis zugleich Teil des Klassenpfads, da die generierten Klassen in den einzelnen Klassen referenziert werden.

Eine komfortablere Möglichkeit ist die Generierung der Meta-Klassen über die verwendete grafische Entwicklungsumgebung. Fast jede Entwicklungsumgebung bietet heute die Möglichkeit, entsprechende *Annotation Processors* einzubinden – dann erfolgt im Fall von JPA die Generierung der Meta-Klassen immer zeitgleich mit der Entwicklung der Entities. Wir wollen dem Leser dies am Beispiel von Eclipse verdeutlichen. Wie in Abbildung 8.2 zu sehen ist, lässt sich in den Eigenschaften eines Java-Projektes das *Annotation Processing* aktivieren. Kommandozeilenparameter (s. oben beim Ant-Task) sind ebenfalls zu ergänzen. Der Vorteil

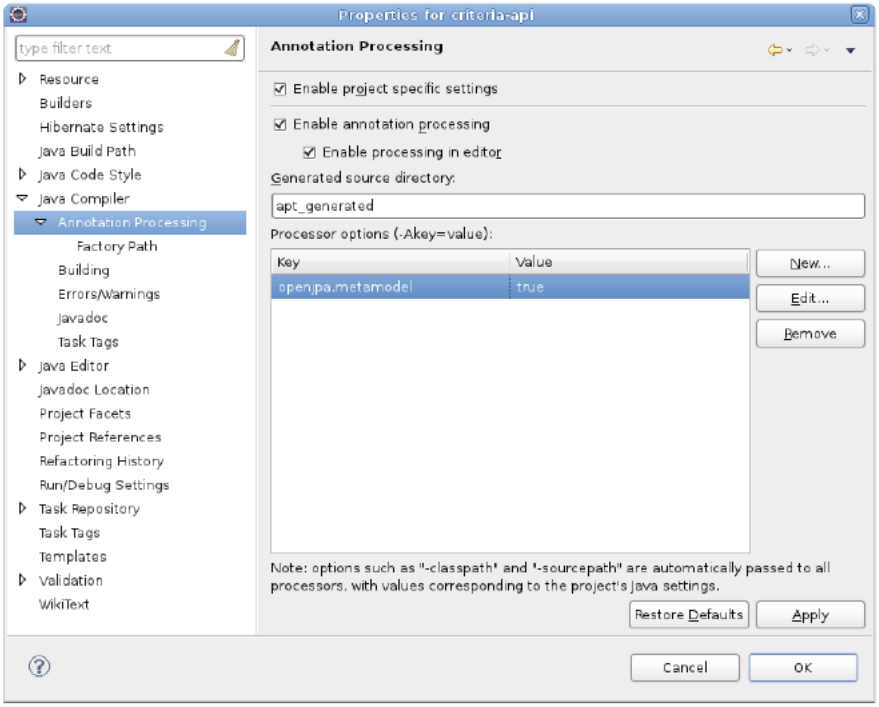


ABBILDUNG 8.2 Aktivierung des Annotation Processing in Eclipse

dieses Vorgehens: der Ordner, in den die Klassen generiert werden, wird automatisch in den Build-Pfad des Projektes aufgenommen. Alle Meta-Klassen und deren Attribute sind über die Autovervollständigung während der Entwicklungszeit verfügbar. Der Pfad zur Factory-Klasse muss im Menü `Factory Path` angegeben werden. Hier reicht der Verweis auf eine der JAR-Dateien unserer Implementierungen. Danach wird das Projekt neu kompiliert, und das generierte, statische Modell steht zur Verfügung.

Das Ergebnis dieser Generierung schauen wir uns in der Klasse `Kunde_` an.

```
package de.jpainfo.model;

import java.util.Date;
import javax.persistence.metamodel.SetAttribute;
import javax.persistence.metamodel.SingularAttribute;

@javax.persistence.metamodel.StaticMetamodel
(value=de.jpainfo.model.Kunde.class)
@javax.annotation.Generated
( value="org.apache.openjpa.persistence.meta.AnnotationProcessor6",
  date="Tue Aug 23 14:34:47 CEST 2011")
public class Kunde_ {
    public static volatile SingularAttribute<Kunde,Date> geburtsdatum
        ;
    public static volatile SingularAttribute<Kunde,Character>
        geschlecht;
    public static volatile SingularAttribute<Kunde,Integer> id;
    public static volatile SetAttribute<Kunde,Konto> konten;
    public static volatile SingularAttribute<Kunde,Integer>
        kundennummer;
    public static volatile SingularAttribute<Kunde,String> nachname;
    public static volatile SingularAttribute<Kunde,String> vorname;
    public static volatile SingularAttribute<Kunde,Adresse> wohnort;
}
```

OpenJPA und EclipseLink fügen neben der Annotation `@StaticMetaModel` auch noch die Annotation `@Generated` mit ein. Hibernate verzichtet darauf. Die Spezifikation ist an dieser Stelle leider etwas ungenau. Unserer generierten Klasse liegt das Entity `Kunde` zugrunde. Zum besseren Vergleich führen wir sie hier noch einmal auf:

```
package de.jpainfo.model;

@Entity
public class Kunde {
  @Id
  @GeneratedValue
  private Integer id;
  private Integer kundennummer;
  private String vorname;
  private String nachname;
  @Temporal(TemporalType.DATE)
  private Date geburtsdatum;
  private Character geschlecht;
  @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
  @JoinColumn(name = "WOHNORT_ID")
  private Adresse wohnort;
  @ManyToMany(cascade = CascadeType.ALL)
  private Set<Konto> konten = new HashSet<Konto>();

  // weitere Getter und Setter Methoden ...
}
```

Wie genau diese generierten Klassen verwendet werden können, zeigen wir in Abschnitt 8.2. Der Leser wird aber schon hier erkennen, dass sich mit dem Zugriff auf die generierten Attribute typsichere Abfragen realisieren lassen.

Hinweis:

In unserem Eclipse-Projekt haben wir die generierten Klassen in unser CVS mit übernommen und stellen sie auch dem Leser im Download bereit. Damit kann das Projekt sofort als Ausgangspunkt für weitere Übungen benutzt werden. Die Frage, ob und an welcher Stelle solche generierten Klassen abzulegen sind, wollen wir an diesem Punkt nicht diskutieren.

8.2 Abfragen mit der Criteria-API

Ausgangspunkt einer `Criteria`-Abfrage ist wie auch in der JPQL im Abschnitt 7.1 auf Seite 178 der `EntityManager` einer Persistenzeinheit. Mit der Methode `getCriteriaBuilder()` bekommt man Zugriff auf eine Instanz der Klasse `CriteriaBuilder`. Sie ist Ausgangspunkt, um Objekte vom Typ `CriteriaQuery` zu erstellen. Darüber hinaus besitzt auch die Klasse `EntityManagerFactory` diese Methode.

```
CriteriaQuery<Object> createQuery();
<T> CriteriaQuery<T> createQuery(Class<T> resultClass);
CriteriaQuery<Tuple> createTupleQuery();
```

Wie wir im Abschnitt 7.5.2 gezeigt haben, werden bei einer SQL-Abfrage zunächst die `From`- und `Where`-Bedingung im Datenbanksystem ausgewertet, daran anschließend die Ergebnismengen selektiert. Auch in der Criteria-API ist der Ausgangspunkt die `From`-Anweisung. Sie wird in Java durch den Typen `javax.persistence.criteria.Root` (die Wurzel) repräsentiert. Basierend auf dieser Wurzel, lassen sich dann Einschränkungen und Selektionen durchführen. Das `Root`-Objekt ist der Einstieg einer Query, über den sich die anderen Entities über die bereits bekannte Navigation erreichen lassen.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Kunde> query = cb.createQuery(Kunde.class);
Root<Kunde> kunde = query.from(Kunde.class);
query.select(kunde);
TypedQuery<Kunde> q = em.createQuery(query);
```

Im oben dargestellten Beispiel erhält unsere Abfrage genau ein `Root`-Element, das Entity `Kunde`. Dieses `Root`-Objekt wird durch die Instanzmethode `from` der Query hinzugefügt. Es muss nicht das einzige `Root`-Objekt bleiben. Weitere Entities ließen sich hinzufügen, was im Ergebnis einem kartesischen Produkt der beteiligten Entities entspricht, da die entsprechenden Join-Bedingungen fehlen. Die obige Abfrage liefert alle Einträge unserer Kunden-Tabelle zurück.

8.2.1 Join-Bedingungen in der Criteria-API

Im Rahmen der JPQL präsentierten wir dem Leser im Abschnitt 7.3.1 die impliziten Joins. Sie entstehen durch Navigation über die Attribute eines Entity zu einem anderen Entity über eine 1:1-Beziehung oder über eine `Collection` zu einer Menge von anderen Entities (1:n-Beziehung). Diese impliziten Joins sind auch in der Criteria-API möglich. Verwendet werden

sie bei der direkten Referenzierung (`SingularAttribute`) oder über Attribute, denen eine `Collection` zugrunde liegt (`Collection`-, `Set`-, `List`- oder `MapAttribute`).

Ein `Join` wird mit der Methode `join` auf dem bereits bekannten `Root`-Objekt ausgeführt. Darüber hinaus lässt es sich auf allen `Join`-Typen aus dem Package `javax.persistence.criteria` ausführen. Diese Methode ist im Interface `From` definiert. Einen Ausschnitt aus der relevanten API haben wir in Abbildung 8.3 in einem Klassenmodell abgebildet.

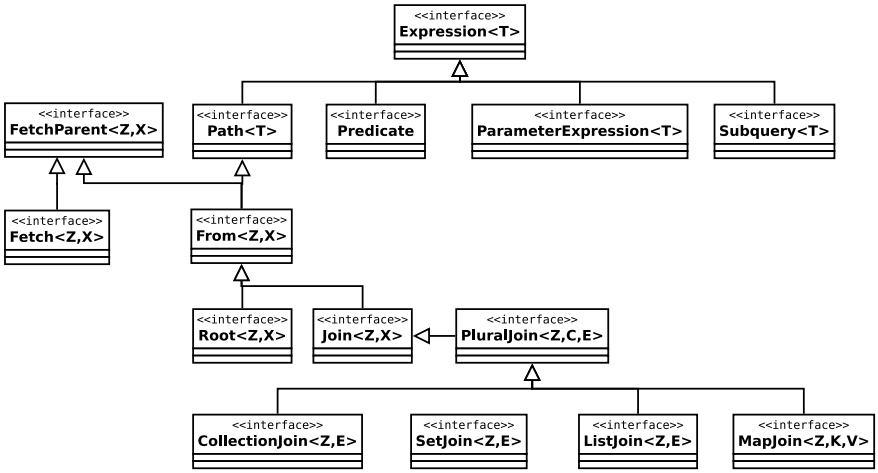


ABBILDUNG 8.3 Klassenhierarchie der Query-Interfaces aus JPA 2.0

Das nächste Programmbeispiel verdeutlicht diesen Join, wenngleich die Selektion noch wenig Sinn ergibt. Zunächst wird der Join zwischen `Kunde` und `Konto` hergestellt, selektiert wird aber nur der Kunde. Er tritt in der Ergebnismenge so oft auf, wie das Kunde-Objekt in Beziehungen zu Konten-Objekten steht.

```
CriteriaQuery<Kunde> query = cb.createQuery(Kunde.class);
Root<Kunde> kunde = query.from(Kunde.class);
Join<Kunde, Konto> konto = kunde.join(Kunde_.konten);
query.select(kunde);
```

Um hier auch schon vor der Einführung der `Select`-Anweisung zu sinnvollen Ergebnissen zu kommen, greifen wir dem Abschnitt 8.2.3 kurz vor und demonstrieren die Selektion einiger Attribute unserer Entity:

```
CriteriaQuery<Object[]> query = cb.createQuery(Object[].class);
Root<Kunde> kunde = query.from(Kunde.class);
Join<Kunde, Konto> konto = kunde.join(Kunde_.konten);
query.select(cb.array(kunde.get(Kunde_.kundennummer),
                      konto.get(Konto_.nummer),
                      konto.get(Konto_.kontostand)));
```

Die obige Criteria-Abfrage entspricht dem folgenden JPQL-Statement:

```
select k.kundennummer, kto.nummer, kto.kontostand
from Kunde k inner join k.konten kto
```

Der Rückgabewert der `join`-Methode ist ein `Join`-Objekt. Dessen Typparameter sind an erster Stelle (Z) der linke Teil des Joins, an zweiter Stelle (X) steht der rechte Teil der Join-Beziehung. Das API spricht in diesem Zusammenhang auch von `Source` und `Target`, die Buchstaben wurden von dort übernommen. Bei 1:n-Beziehungen steht an zweiter Stelle der Typparameter der Collection.

Wir wollen uns außerdem noch einmal die Parameter der `join`-Methode ansehen. Die Methode existiert im Interface `From` insgesamt 12 mal. Übergeben werden können die bereits aus Kapitel 8.1.1 definierten Objekte vom Typ `Attribute`, also `SingularAttribute` bzw. einer der `PluralAttribute`-Typen, sowie darüber hinaus ein Join-Typ, den wir uns im nächsten Kapitel näher ansehen.

`Left-` und `Fetch-Joins`

Wird in der `join`-Methode der Parameter des `JoinType` nicht angegeben, wird automatisch ein *Inner Join* ausgeführt. Zur Erinnerung: Diese Join-Beziehung liefert immer nur die Entities, die tatsächlich eine Beziehung zu mindestens einem Entity der anderen Seite der Beziehung unterhalten. In unserem Modell bedeutet dies, dass nur Kunden in der Ergebnismenge auftauchen, die auch ein Konto besitzen.

Um einen Join zu konstruieren, der *alle* Kunden und Konten liefert, auch für Kunden, die kein aktives Konto besitzen, ist ein `LeftJoin` erforderlich:

```
CriteriaQuery<Object[]> query = cb.createQuery(Object[].class);
Root<Kunde> kunde = query.from(Kunde.class);
Join<Kunde, Konto> konto = kunde.join(Kunde_.konten, JoinType.LEFT);
query.select(cb.array(kunde.get(Kunde_.kundennummer),
                      konto.get(Konto_.nummer),
                      konto.get(Konto_.kontostand)));
```

Attribute eines Entity, die per Definition der JPA bzw. durch Konfiguration des Attributes `LAZY` geladen werden, lassen sich auch in der Criteria-API mit so genannten `Fetch`-Joins zusätzlich zum eigentlichen Entity lesen. Dafür stehen die überlagerten Methoden `fetch()` aus dem Interface `FetchParent` bereit. Wie in der Abbildung 8.3 auf der vorherigen Seite zu sehen ist, sind diese Methoden in den bereits bekannten Objekten `Join` und `Root` aufrufbar. Analog zu den `Fetch`-Joins in der JPQL wollen wir die Verwendung am Beispiel der Adresse unserer Kunden-Klasse demonstrieren. Das Attribut `wohnort` ist im Entity `Kunde` als `LAZY` konfiguriert. Wird ein `Kunde` gelesen, ist das Attribut `wohnort` nur ein Platzhalter, der erst beim Zugriff auf dieses Attribut mit dem tatsächlichen Wert gefüllt wird.

```
@Entity
public class Kunde {
  @Id
  @GeneratedValue
  private Integer id;
  private Integer kundennummer;
  private String vorname;
  private String nachname;
  @Temporal(TemporalType.DATE)
  private Date geburtsdatum;
  private Character geschlecht;
  @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
  @JoinColumn(name = "WOHNORT_ID")
  private Adresse wohnort;
```

```
    @ManyToMany(cascade = CascadeType.ALL)
    private Set<Konto> konten = new HashSet<Konto>();

    // weitere Getter und Setter Methoden ...
  }
```

Um bei der Selektion des Kunden auch gleich dessen Adresse mitzulesen, führen wir auf dem `Root<Kunde>`-Objekt die `fetch()`-Methode aus. Dieser Methode wird das aus dem generierten Meta-Modell zu selektierende `Adresse`-Objekt übergeben:

```
CriteriaQuery<Kunde> query = cb.createQuery(Kunde.class);
Root<Kunde> kunde = query.from(Kunde.class);
kunde.fetch(Kunde_.wohnort, JoinType.LEFT);
query.select(kunde);
Query q = em.createQuery(query);
List<Object[]> result = q.getResultList();
```

Die äquivalente Abfrage in der JPQL würde dazu wie folgt lauten:

```
select k from Kunde k inner join fetch k.wohnort
```

Der Leser ist dazu aufgefordert, das obige Beispiel auszuführen und die Zeile beginnend mit `kunde.fetch...` zu kommentieren. Es kommt zu keinem Join während der Selektion aller Kunden. Wird daran anschließend jedoch die Adresse gelesen, wird die Datenbank für jeden Kunden einzeln nach dessen Adresse abgefragt. Dies ist das so genannte n+1-Select-Problem, dem wir uns bereits in Abschnitt 4.4 gewidmet haben.

8.2.2 Einschränkung der Ergebnismenge

Wie in Standard-SQL und JPQL wird mit dem Schlüsselwort `where` die Ergebnismenge eingeschränkt. Dazu stehen im `CriteriaQuery`-Objekt zwei `where`-Methoden zur Verfügung. Ihnen können als Parameter ein `Expression<Boolean>`- oder ein bzw. mehrere `Predicate`-Instanzen übergeben werden.

Rückgabewert der Methoden ist jeweils das modifizierte `CriteriaQuery`-Objekt selbst. Damit lassen sich auch komplexe Selektionsbedingungen konstruieren, ohne die Lesbarkeit des Quellcodes einzuschränken.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Kunde> query = cb.createQuery(Kunde.class);
Root<Kunde> kunde = query.from(Kunde.class);
query.select(kunde).where(cb.equal(kunde.get(Kunde_.nachname), "Meier
    "));
```

Auch wenn die obige Abfrage zunächst komplizierter aussieht als das äquivalente JPQL-Statement, wird der Leser nach einiger Übung erkennen, dass der Aufbau einer Criteria-Abfrage dem eines normalen `SQL`-Befehls folgt. Einstieg dazu ist das mit der Methode `from()` erzeugte `Root`-Objekt, in unserem Beispiel der Kunde. Dieses Objekt ist auch Bestandteil der Selektion (dazu jedoch im Abschnitt 8.2.3 mehr). Herangezogen wird das Objekt in unserem ersten Beispiel auch für die Konstruktion der `where`-Bedingung. Der `CriteriaBuilder` implementiert Methoden wie `equal()`, `between()`, `concat()` und `count()`, kurz gesagt: alle Sprachelemente, die der Leser bereits aus der JPQL kennt.

Um die Attribute entsprechend verwenden zu können, wird die bereits bekannte Pfad-Navigation benutzt. Der Punkt (`kunde.nachname`) aus der JPQL wird in der Criteria-API über die Methode `get()` realisiert. Der Methode `get()` kann man Parameter vom Typ `Attribute` übergeben. Der Leser kennt diese bereits aus dem Abschnitt 8.1.1. Es sind die generierten Meta-Model-Attribute unserer Entities. Der Zugriff auf die Postleitzahl des Wohnortes unserer Kunden sieht demnach wie folgt aus:

```
// JPQL
select k.wohnort.plz from Kunde k

// Criteria-API
query.select(kunde.get(Kunde_.wohnort).get(Adresse_.plz));
```

Hinweis:

Die seit Java 5.0 implementierte variable Parameterangabe `Varargs` ermöglicht es, beliebig viele Bedingungen der `where`-Bedingung zu übergeben. Die Parameter müssen vom Typ `Predicate` oder `Expression` sein. Darüber hinaus sind fast alle Interfaces so genannte *Fluent Interfaces*, das heißt, ihre Methoden liefern entsprechende Objekte wieder zurück, um so effiziente Methoden-Ketten zu ermöglichen.

Die Spezifikation spricht von sogenannten einfachen `Predicates` bei den Methoden `isNull`, `isNotNull` und `in` aus dem `Expression`-Interface. *Compound Interfaces* lassen sich mit den Methoden `and`, `or` bzw. `not` konstruieren.

Analog zur JPQL sind auch hier Parameterübergaben erlaubt. Mit Hilfe der `ParameterExpression<T>` werden der Query erst zur Laufzeit der Abfrage die entsprechenden Bindings übergeben:

```
CriteriaQuery<Object> query = cb.createQuery(Object.class);
Root<Kunde> kunde = query.from(Kunde.class);
ParameterExpression<String> param = cb.parameter(String.class, "
    nachname");
query.where(cb.equal(kunde.get(Kunde_.nachname), param));
query.select(kunde.get(Kunde_.nachname));
Query q = em.createQuery(query);
q.setParameter(param, "Beck");
```

8.2.3 Selektion der Ausgabe

Kommen wir nun zur dritten wichtigen Methode, die uns ein `CriteriaQuery`-Objekt bereitstellt. Es handelt sich um die Anweisung `select()`, mit der die *Spalten* des Ergebnisses selektiert werden. Parameter dieser Methode sind vom Typ `Selection<T>`.

Damit das Bild komplett wird, müssen wir unser Klassenmodell von Seite 213 noch ein wenig ergänzen bzw. die für dieses Kapitel unwichtigen Interfaces weglassen. Die gesamte Interface-Typ-Hierarchie der Criteria-API beginnt mit dem Element `TupleElement`. Abb. 8.4 auf der nächsten Seite zeigt den jetzt für uns relevanten Ausschnitt. Die `select`-Methode nimmt als Parameter ein `Selection<T>`-Objekt entgegen.Die Klassenhierarchie zeigt, dass es sich dabei um eine `Expression<T>` handeln kann. Aber auch `Path<T>`-Objekte, also die Attribute, die wir über die bereits gezeigte `get`-Methode erhalten, sind erlaubt. Die nachfol-

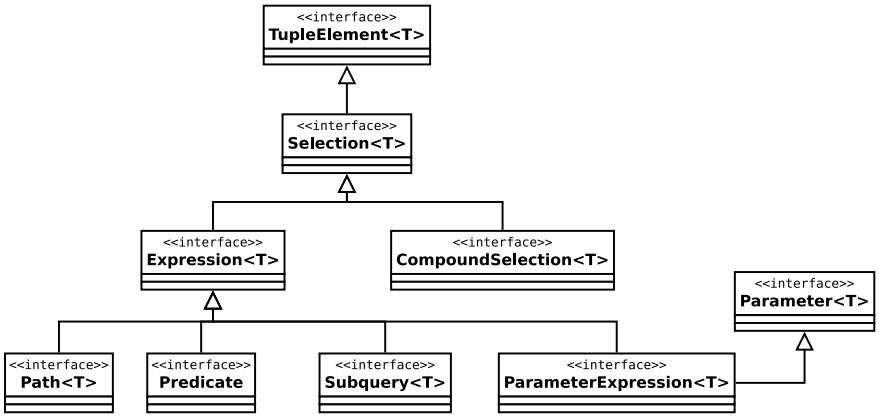


ABBILDUNG 8.4 Klassenhierarchie der `Selection`-Objekte

gende Aufstellung zeigt die wichtigsten Methoden aller Criteria-Interfaces, gegliedert nach dem Ort der Implementierung und dem Typ des Rückgabewertes der Methoden:

- `CriteriaBuilder`
  - `CriteriaQuery<T>`
    `createQuery`, `createTupleQuery`
  - `Expression<T>`
    `abs`, `all`, `any`, `avg`, `coalesce`, `concat`, `count`, `countDistinct`, `currentDate`, `currentTime`, `currentTimestamp`, `diff`, `function`, `keys`, `least`, `length`, `literal`, `locate`, `lower`, `max`, `mode`, `neg`, `nullif`, `nullLiteral`, `prod`, `quot`, `size`, `some`, `sqrt`, `substring`, `sum`, `sumAsDouble`, `sumAsLong`, `toBigDecimal`, `toBigInteger`, `toDouble`. `toFLoat`,`toInteger`, `toLong`, `toString`, `trim`, `upper`
  - `ParameterExpression<T>`
    `parameter`
  - `Predicate`
    `and`, `between`, `conjunction`, `disjunction`, `equal`, `exists`, `ge`, `greatherThan`, `greaterThanOrEqualTo`, `greatest`, `gt`, `isEmpty`, `isFalse`, `isMember`, `isNot-Empty`, `isNotMember`, `isNotNull`, `isNull`, `isTrue`, `le`, `least`, `lessThan`, `less ThanOrEqualTo`, `like`, `lt`, `not`, `notEqual`, `notLike`, `or`
- `CriteriaQuery`
  - `CriteriaQuery<T>`
    `distinct`, `groupBy`, `having`, `multiselect`, `orderBy`, `select`, `where`
  - `Root<X>`
    `from`
  - `Subquery<U>`
    `subquery`
- `Expression<T>`
  - `Expression<T>`
    `as`

- Predicate
  in, isNotNull, isNull

Der Typ der Selection<T> muss kompatibel zum Rückgabe-Typ der bereits erstellten CriteriaQuery sein. Andernfalls kommt es bereits beim Kompilieren zu einer Fehlermeldung, was als einer der wesentlichen Vorteile der Criteria-API anzusehen ist.

Die einfachste Form der Selektion haben wir bereits in unseren eingehenden Beispielen gezeigt. Das von der From-Methode erhaltene Root-Objekt erbt über die Interfaces From und Path von Expression und ist somit ein gültiger Parameter der select-Methode:

```
CriteriaQuery<Kunde> query = cb.createQuery(Kunde.class);
Root<Kunde> kunde = query.from(Kunde.class);
query.select(kunde)
```

Die Ergebnismenge enthält also eine Liste von Kunden-Objekten unseres Entity Kunde.

Um mehr als ein Element mit der select-Bedingung zu lesen, stehen die Methoden multiselect() bzw. tuple() zur Verfügung. Wir demonstrieren zuerst die Verwendung der tuple()-Methode. Dieser Methode können beliebig viele Selektionsparameter übergeben werden, der Typ des Ergebnisses ist vom Typ des CriteriaQuery-Objekts abhängig:

```
CriteriaQuery<Tuple> query = cb.createTupleQuery();
Root<Kunde> kunde = query.from(Kunde.class);
query.select(cb.tuple(kunde,
                      kunde.get(Kunde_.nachname),
                      kunde.get(Kunde_.vorname).alias("VORNAME")));
```

Die Ergebnismenge besitzt in diesem Fall eine Liste von Tuple-Objekten. Auf einzelne Elemente lässt sich mit den get()-Methoden über die Index-Position oder über den Namen zugreifen, sofern der Selektion ein Alias zugewiesen wurde. In unserer Ausgabe greifen wir auf die beiden ersten Elemente mit einem Index zu, das dritte Element bekommen wir über den Alias-Namen, der mit der Methode alias() einem Selektionsparameter zugewiesen wurde. An erster Stelle steht das Kunde-Objekt selbst, die zweite und dritte Position wird mit den String-Objekten der Nachnamen und Vornamen belegt.

```
List<Tuple> result = q.getResultList();
for (Tuple tuple : result) {
    // Zugriff ueber den Index
    System.out.print(tuple.get(0) + " " + tuple.get(1) + " ");
    // Zugriff ueber den Alias
    System.out.println(tuple.get("VORNAME"));
}

// Beispiel-Ausgabe dazu:
de.jpainfo.model.Kunde@539f0d Schmidt Angelina
```

Neben der Methode tuple() lässt sich die Ergebnismenge auch mit array() und der Methode construct() bestimmen. Mit Hilfe der construct-Methode lassen sich sowohl Entities neu instanziieren als auch nicht verwaltete Klassen-Instanzen erstellen. Wir greifen zur Demonstration auf das im Abschnitt 7.5.4 auf Seite 202 gezeigte Beispiel zur Verwendung von Konstruktoren in der JPQL zurück:

```
select new de.jpainfo.model.auswertung.KundeStadt(
k.wohnort.plz, k.wohnort.ort, count(k))
from Kunde k
group by k.wohnort.plz, k.wohnort.ort
```

Dieses JPQL-Statement soll jetzt mit Hilfe der Criteria-API konstruiert werden. Dazu verwenden wir die `groupBy()`-Methode aus Abschnitt 8.2.5. Die Ergebnisliste wird Objekte vom Typ der Klasse `KundeGesamt` enthalten. Die Parameter für den Konstruktor selektieren wir analog zum dargestellten JPQL-Statement über das `Root`-Element:

```
CriteriaQuery<KundeStadt> query = cb.createQuery(KundeStadt.class);
Root<Kunde> kunde = query.from(Kunde.class);
query.select(cb.construct(KundeStadt.class,
        // Navigation zu Adresse
        kunde.get(Kunde_.wohnort).get(Adresse_.plz),
        kunde.get(Kunde_.wohnort).get(Adresse_.ort),
        cb.count(kunde))); // Gezaehlt wird der Kunde
query.groupBy(kunde.get(Kunde_.wohnort).get(Adresse_.plz),
              kunde.get(Kunde_.wohnort).get(Adresse_.ort));
TypedQuery<KundeStadt> q = em.createQuery(query);
List<KundeStadt> result = q.getResultList();
```

Der Leser wird spätestens an diesem Beispiel erkennen, dass die Verwendung der Criteria-API die Lesbarkeit der Java-Programme nicht unbedingt erhöht. Die Typsicherheit hat auch ihre Nachteile.

Mehrere Elemente lassen sich auch mit der Methode `multiselect()` der `CriteriaQuery` abfragen. Die Methode nimmt eine beliebige Menge von Parametern entgegen. Das Ergebnis der Abfrage wird dabei im Wesentlichen vom Typ der zugrunde liegenden Abfrage bestimmt:

- Ist die Abfrage vom Typ `CriteriaQuery<Tuple>` bzw. mit der Methode `createTupleQuery()` erzeugt, wird das Ergebnis `Tuple`-Objekte enthalten. Die Reihenfolge der `Tuple`-Attribute wird durch die Reihenfolge der Parameter der Methode bestimmt.
- Liegt der Query ein beliebiger Typ `X` zugrunde, enthält die Ergebnisliste Elemente vom Typ `X`. Die Elemente der Ergebnismenge werden dem Konstruktor von `X` zur Laufzeit übergeben.
- Typisierte Arrays einer Abfrage `CriteriaQuery<X[]>` werden auch als Ergebnis entsprechend zurückgegeben. Die Reihenfolge der Parameter der `multiselect`-Methode bestimmt dabei auch die Reihenfolge im Array.
- Ist der Typ der Abfrage unspezifiziert, wird die Ergebnisliste `Object`-Instanzen enthalten. Voraussetzung dafür ist jedoch, dass nur ein Element selektiert wurde.
- Werden bei einer untypisierten Abfrage mehrere Elemente selektiert, enthält die Ergebnisliste Zeilen vom Typ `Object[]`. Das `Object[]`-Array beinhaltet die Selektionsergebnisse entsprechend der Reihenfolge der `multiselect`-Methode.

```
CriteriaQuery<Object[]> query = cb.createQuery(Object[].class);
Root<Kunde> kunde = query.from(Kunde.class);
query.multiselect(kunde.get(Kunde_.nachname),
                  kunde.get(Kunde_.vorname),
                  kunde.get(Kunde_.kundennummer));
TypedQuery<Object[]> q = em.createQuery(query);
// Zeilentyp der Liste
//[java.lang.String, java.lang.String, java.lang.Integer]
List<Object[]> result = q.getResultList();
```

8.2.4 Unterabfragen

Unterabfragen können Bezug auf die Abfrage nehmen, in der sie verwendet werden, oder unabhängig davon sein. Man spricht von so genannten *Correlated Subqueries*, wenn die Unterabfrage z.B. in ihrer `Where`-Bedingung auf Werte der äußeren Abfrage zurückgreift. Haben die Abfragen keinen Bezug zueinander, handelt es sich um eine *Non-correlated Subquery*. Die Unterschiede zwischen beiden Arten werden wir dem Leser in unseren nächsten Beispielen verdeutlichen.

Im Abschnitt 7.4.1 im Rahmen der JPQL sind wir auf diese Begriffe bewusst nicht eingegangen. In der Criteria-API muss der Leser die Unterschiede jedoch kennen, um Unterabfragen mit der Methode `correlate` in Bezug zur eigentlichen Abfrage zu setzen.

```
CriteriaQuery<Object> query = cb.createQuery(Object.class);
Root<Konto> kto = query.from(Konto.class);
// Subquery unabhaengig von Root<Konto>
Subquery<Double> sq = query.subquery(Double.class);
Root<Konto> kto2 = sq.from(Konto.class);
sq.select(cb.avg(kto2.get(Konto_.kontostand)));
// Was soll selektiert werden ?
query.multiselect(kto, kto.get(Konto_.kontostand));
// Vergleich mit dem Durchschnitt der Subquery
query.where(cb.gt(kto.get(Konto_.kontostand), sq));
Query q = em.createQuery(query);
```

Wie bei anderen Criteria-Abfragen bilden wir zunächst das Ausgangsobjekt der eigentlichen Abfrage: das `Root<Konto>`-Objekt. Basierend auf derselben Query wird eine Subquery instanziiert, ihr Rückgabewert wird vom Typ `Double` sein. Dieser Typ wird durch die Methode `avg()` bestimmt. Die Unterabfrage greift nicht auf Werte der eigentlichen Abfrage zurück. Sie kann unabhängig von ihr konstruiert werden. Das `Root`-Objekt namens `kto2` ist Ausgangspunkt, um den Durchschnitt aller Salden unserer Konten zu berechnen. Das Ergebnis dieser Unterabfrage wird in der `Where`-Bedingung der Oberabfrage verwendet. Die Abfrage ermittelt alle Konten, deren Saldo größer als der Durchschnitt der Salden aller Konten ist.

Dem Leser dürfte das Ergebnis der obigen Abfrage bekannt vorkommen. Wir haben das gleiche Ergebnis mit der JPQL-Abfrage erhalten, die im Abschnitt 7.4.1 bereits entwickelt wurde. Um die Vergleichbarkeit zu erhöhen, wurden in der Criteria-Abfrage die Java-Objekte analog zu den Aliasen der JPQL-Abfrage benannt.

```
select kto, kto.kontostand from Konto kto
where kto.kontostand >
  (select avg(kto2.kontostand) from Konto kto2)
```

Greift die Subquery auf Elemente der Oberabfrage zurück, korreliert sie mit dieser. Diese Korrelation wird im einfachsten Fall mit der Referenzierung des `Root`-Objektes aus der eigentlichen Abfrage durchgeführt. Die folgende – zugegeben, nicht sehr sinnvolle – Abfrage liefert uns alle Kunden, deren Geburtsdatum mit mindestens einem weiteren Kunden übereinstimmt.

```
CriteriaQuery<Object> query = cb.createQuery(Object.class);
Root<Kunde> kunde = query.from(Kunde.class);
// Subquery referenziert dieses Root<Kunde> in der Where-Bedingung
Subquery<Kunde> sq = query.subquery(Kunde.class);
Root<Kunde> kunde_gem = sq.from(Kunde.class);
```

```
sq.where(cb.and(
  // Kunden mit gleichem Geburtsdatum
  cb.equal(kunde.get(Kunde_.geburtsdatum),
           kunde_gem.get(Kunde_.geburtsdatum))),
  // aber nicht er selbst
  cb.notEqual(kunde, kunde_gem));
sq.select(kunde_gem);
// Was soll selektiert werden ?
query.multiselect(kunde.get(Kunde_.kundennummer), kunde.get(Kunde_.
    geburtsdatum));
// Test ob die Subquery ein Ergebnis hat
query.where(cb.exists(sq));
```

Die umgebende Abfrage berücksichtigt einen Kunden nur dann, wenn die Unterabfrage mindestens einen weiteren Kunden zurückgibt, dessen Geburtstag das gleiche Datum hat wie der soeben betrachtete Kunde. Aber: Es darf sich nicht um den gleichen Kunden handeln, sonst wären alle Kunden in unserer Ergebnismenge vorhanden. Die äquivalente JPQL-Abfrage dazu lautet:

```
select kunde.kundennummer, kunde.geburtsdatum from Kunde kunde
where exists (
  select kunde_geb from Kunde kunde_geb
  where
   kunde.geburtsdatum = kunde_geb.geburtsdatum and
   and kunde != kunde_geb
)
```

Um in Subqueries auf unidirektionale Beziehungen zwischen Entities zurückzugreifen, reicht es oft nicht aus, nur die entsprechenden Objekte zu referenzieren. Vielmehr muss mit der Methode `correlate()` ein neues `Root`-Objekt erzeugt werden. Mit diesem `Root`-Objekt, das sich nun auf die Subquery bezieht, lassen sich erneut Bedingungen formulieren. Wir wollen mit dem folgenden JPQL-Beispiel den Abschnitt zu den Unterabfragen abschließen. Gesucht sind alle Kunden, die mehr als zwei Konten unterhalten.

Diese Abfrage formulieren wir zunächst in JPQL:

```
select kunde from Kunde kunde
where (
  select count(konten) from kunde.konten konten
  ) > 2
```

Das JPQL-Statement zeigt, dass wir einen Bezug zwischen dem Kunde-Objekt der äußeren Statements und der inneren Abfrage benötigen. Darüber hinaus benötigen wir diese Referenz, um einen weiteren Join in der Subquery zu realisieren. Die Methode `correlate()` liefert uns dieses weitere Objekt zurück.

```
CriteriaQuery<Object> query = cb.createQuery(Object.class);
Root<Kunde> kunde = query.from(Kunde.class);
// Die Subquery liefert ein SQL-Count
Subquery<Long> sq = query.subquery(Long.class);
// Subquery mit correlate fuer die Join-Beziehung
Root<Kunde> kundeSub = sq.correlate(kunde);
// Join in der Subquery, mit Bezug zur Oberabfrage
Join<Kunde, Konto> konten = kundeSub.join(Kunde_.konten);
query.where(cb.gt(sq.select(cb.count(konten)), 2));
query.select(kunde);
Query q = em.createQuery(query);
```

Die so erstellte Subquery wird Bestandteil der `Where`-Bedingung der äußeren Abfrage und führt uns zum gewünschten Ergebnis.

8.2.5 Gruppierungen mit `Group By` und `Having` und Sortierungen mit `Order By`

Bei der Selektion einer Abfrage, die eine `Group By`-Anweisung enthält, müssen alle Elemente, die nicht in einer Aggregatsfunktion enthalten sind, analog zu den Elementen sein, die zu der eigentlichen Gruppierung geführt haben, also den Elementen, die nach dem `Group By`-Befehl stehen. Die theoretische Betrachtung dazu haben wir bereits in Abschnitt 7.5.2 durchgeführt.

Zur Demonstration suchen wir die Orte, an denen mehr als fünf unserer Kunden wohnen, für Marketingzwecke ein geeigneter Ausgangspunkt. Dazu betrachten wir zunächst die leichter lesbare JPQL-Abfrage.

```
select k.wohnort.ort, count(k)
from Kunde k
group by k.wohnort.ort
having count(k) > 5
```

Noch einmal zum Verständnis dieser Gruppierung: `Group By` teilt das Ergebnis unserer Selektion in einzelne Gruppen ein, die übereinstimmende Orte haben. Mittels `Having` werden nur solche Gruppen gewählt, in denen mehr als fünf Kunden wohnen. Andere Gruppen werden mit dieser Selektion verworfen.

```
CriteriaQuery<Object> query = cb.createQuery(Object.class);
Root<Kunde> kunde = query.from(Kunde.class);
// GroupBy-Elemente muessen auch in der Selektion erscheinen!
query.groupBy(kunde.get(Kunde_.wohnort).get(Adresse_.ort));
query.having(cb.gt(cb.count(kunde), 5));
// Ort muss selektiert werden, da in GroupBy-Bedingung
query.multiselect(kunde.get(Kunde_.wohnort).get(Adresse_.ort),
                  cb.count(kunde));
Query q = em.createQuery(query);
List<Object[]> result = q.getResultList();
```

Zur Sortierung der Ausgabe übergeben wir der `CriteriaQuery`-Instanz mit der Methode `orderBy` ein oder mehrere `Order`-Instanzen. Diese Objekte liefert uns wie bereits erwartet der `CriteriaBuilder`. Bei aufsteigender Reihenfolge muss die Methode `asc()` benutzt werden, für die umgekehrte Sortierung entsprechend `desc()`.

```
query.orderBy(cb.desc(cb.count(kunde)));
```

Elemente des Befehls `OrderBy` müssen auch in der entsprechenden Selektionsbedingung der Abfrage vorkommen. Wird in der Selektionsbedingung ein Entity angegeben, lassen sich in der `OrderBy`-Klausel alle persistenten Attribute des Entity verwenden, selbst wenn diese nicht einzeln in der Selektion vorkommen. Darüber hinaus kann jedes weitere `Expression`-Element benutzt werden, das auch in der Selektion enthalten ist. In unserem Beispiel ist das zum Beispiel das Objekt `Expression<Long>`, das wir mit dem Ausdruck `cb.count(kunde)` erhalten.

8.2.6 Benutzung von datenbankspezifischen Funktionen

Die Möglichkeit, datenbankspezifische Funktionen in der JPQL abzusetzen, sind nur mit nativem SQL möglich. Das Criteria-API bietet diese Möglichkeit auch, hat aber den Vorteil, typsicher zu sein. Der jeweilige Rückgabetyp wird der Methode `function()` mit übergeben. Darüber hinaus nimmt diese Methode eine beliebige Anzahl von weiteren Parametern entgegen, die die jeweilige Datenbankfunktion zur Ausführung benötigt:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Object> query = cb.createQuery();
Root<Kunde> kunde = query.from(Kunde.class);
query.multiselect(
        cb.function("lpad",
                    String.class,
                    kunde.get(Kunde_.nachname), cb.literal(20), cb.
                        literal("-")),
        cb.function("last_day",
                    Date.class,
                    kunde.get(Kunde_.geburtsdatum))
);
```

Die Methode `function()` besitzt die folgende Signatur:

```
<T> Expression<T> function(java.lang.String name,
                           java.lang.Class<T> type,
                           Expression<?>... args)
```

In unserem obigen Beispiel demonstrieren wir die Verwendung der Oracle-Funktionen `lpad` und `last_day`. Die Funktion `lpad` füllt eine Selektion um eine Anzahl von Zeichen an der linken Stelle auf. Die Funktion `last_day` liefert den letzten Monat eines Datums, in unserem Beispiel den letzten Tag eines Monats des Geburtsdatums der Kunden.

Hinweis:

Mit Verwendung von spezifischen Datenbankfunktionen verliert der Anwender die Möglichkeit zur Portabilität von Anwendungen über die Datenbank hinweg. Nichtsdestotrotz bietet die `function()`-Methode die Möglichkeit, bereits vorhandene Datenbank-Entwicklungen in Form von kundeneigenen Funktionen (Stored Procedures) weiter zu benutzen. In vielen Unternehmen ist die Wahl einer Datenbank eine langfristige Unternehmensentscheidung, die Frage eines Wechsels bzw. der Portabilität also oft eine rhetorische Frage.

8.2.7 Modifikation von bestehenden Abfragen

`CriteriaQuery`-Objekte können im Programmablauf auch wiederverwendet werden. Die Methoden `select()`, `where()`, `groupBy()`, `having()` und `orderBy()` ersetzen jeweils zuvor hinzugefügte Elemente der zugrunde liegenden Abfrage. Diese Logik ermöglicht es dem Anwender, bei Bedarf einzelne Elemente der Abfrage zu modifizieren, ohne eine gesamte Abfrage neu aufbauen zu müssen. So können z.B. Bedingungen in Instanzvariablen vorbelegt und bei Bedarf der Abfrage zugewiesen werden.

```
CriteriaQuery<Object> query = cb.createQuery(Object.class);
Root<Kunde> kunde = query.from(Kunde.class);
query.multiselect(kunde.get(Kunde_.nachname),
                  kunde.get(Kunde_.geschlecht));
Predicate p1 = cb.equal(kunde.get(Kunde_.geschlecht), 'm');
Predicate p2 = cb.equal(kunde.get(Kunde_.geschlecht), 'w');
query.where(p1);
Query q = em.createQuery(query);
```

Die Methode `createQuery()` erzeugt aus einer `CriteriaQuery` eine Instanz von`Typed Query`. Dieses Statement überführt die Criteria-Abfrage in eine Abfrage des Packages `javax.persistence`. Die Spezifikation verlangt von einer JPA-Implementierung, dass bei jeder Ausführung von `createQuery()` eine neue Instanz erzeugt wird – selbst dann, wenn die `CriteriaQuery`-Instanz bereits verwendet wurde. Bestehende Abfragen werden nicht mehr verändert.

Mit unseren definierten Bedingungen können wir also nacheinander unterschiedliche Abfragen erzeugen, die keinen Einfluss aufeinander haben.

```
Predicate p1 = cb.equal(kunde.get(Kunde_.geschlecht), 'm');
Predicate p2 = cb.equal(kunde.get(Kunde_.geschlecht), 'w');
query.where(p1);
// q1 mit Predicate p1
Query q1 = em.createQuery(query);
// Aenderung der query ohne Auswirkung auf q1
query.where(p2);
Query q2 = em.createQuery(query);
```

Hinweis:

Elemente einer Abfrage sind nicht portabel über mehrere Abfragen hinweg. So können z.B. unsere oben definierten `Predicate`-Instanzen mehrfach verwendet werden. Sie haben jedoch einen eindeutigen Bezug zu genau einer `CriteriaQuery`-Instanz.

8.3 Der dynamische Zugriff

Auch ohne Generierung der in Abschnitt 8.1.1 beschriebenen statischen Meta-Klassen unseres Modells ist ein typsicherer Zugriff möglich. Dazu wurde in der JPA-Spezifikation eine weitere Ebene der JPA-Modellsicht eingeführt. Begriffe wie *Entity*, *Assoziation* oder *Attribut* sind hier als Java-Klassen definiert und erlauben eine alternative Art des typsicheren Zugriffs auf die hinterlegten Daten.

Im Paket `javax.persistence.metamodel` findet der Leser Interface-Definitionen, die ihm an dieser Stelle des Buches mehr als bekannt vorkommen werden. Ein Entity wird z.B. im Interface `javax.persistence.model.EntityType` abgebildet. Ein Attribut des Entity findet sein Äquivalent in `Attribute` bzw. einem der Sub-Interfaces des gleichen Paketes. In Abbildung 8.5 versuchen wir, die Beziehungen zwischen den Interfaces abzubilden. Uns ist bewusst, dass Interfaces keine Assoziationen untereinander aufweisen, eine mögliche Implementierung von `EntityType` muss uns aber entsprechend der API über

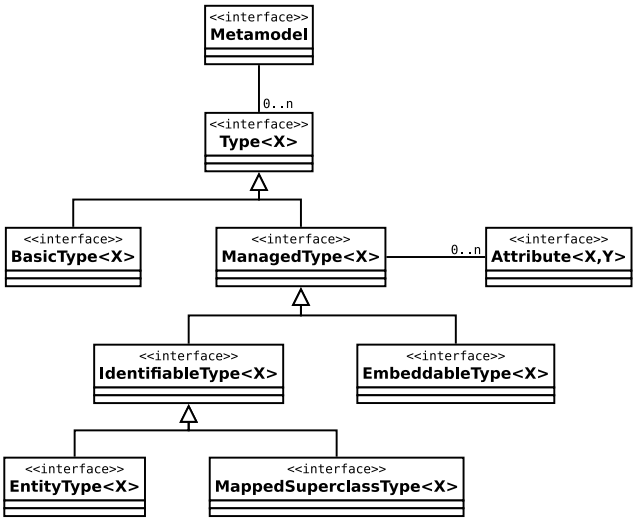


ABBILDUNG 8.5 Hierarchie des Packages `javax.persistence.metamodel`

die Methode `getAttributes()` alle Attribute liefern. Die Methode `getMetamodel()` im `EntityManager` bzw. `EntityManagerFactory` liefert den Einstiegspunkt für unsere Abbildung: eine Instanz von `javax.persistence.metamodel.Metamodel`. Jeder EntityManager kennt *seine* Entities, die Java-Klassen also, die er zu verwalten hat. In unserem nachstehenden Beispielprogramm lesen wir alle zu verwaltenden Entities und deren Attribute aus.

```
EntityManager em = emf.createEntityManager();
Metamodel metamodel = em.getMetamodel();
Set<EntityType<?>> entities = metamodel.getEntities();
for (Iterator i1 = entities.iterator(); i1.hasNext();) {
    EntityType<?> entityType = (EntityType<?>) i1.next();
    System.out.println("======= Entity: " + entityType.getName()+ "
        ====== ");
    Set<?> attributes = entityType.getAttributes();
    for (Iterator i2 = attributes.iterator(); i2.hasNext();) {
        Attribute<?,?> attribute = (Attribute<?, ?>) i2.next();
        System.out.printf("  - %-15s %-15s %-20s %n",
                attribute.getName(),
                attribute.getPersistentAttributeType(),
                attribute.getJavaType());
    }
}
```

Die Ausgabe für unser Entity `Kunde` lautet wie folgt:

```
======= Entity:de.jpainfo.model.Kunde ======
  - vorname          BASIC          class java.lang.String
  - id               BASIC          class java.lang.Integer
  - wohnort          ONE_TO_ONE     class de.jpainfo.model.Adresse
  - geburtsdatum     BASIC          class java.util.Date
  - kundennummer     BASIC          class java.lang.Integer
  - nachname         BASIC          class java.lang.String
```

```
- konten          MANY_TO_MANY     interface java.util.Set
- geschlecht      BASIC            class java.lang.Character
```

Auf die erneute Darstellung der Beziehung zwischen den unterschiedlichen Attribut-Interfaces verzichten wir an dieser Stelle. Das UML-Klassenmodell dazu findet der Leser in Abbildung 8.1 auf Seite 209.

Doch wie können wir die Meta-Model-Elemente in unseren Criteria-Abfragen benutzen? Wir greifen dazu unser einführendes Beispiel zur Bildung einer Abfrage mit `Tuple`-Elementen von Seite 218 wieder auf. Anstelle der direkten Referenz unserer generierten Meta-Klassen und deren Attribute liefert uns die Meta-Model-Instanz ganz ohne Generierung das gewünschte Ergebnis. Zur besseren Vergleichbarkeit haben wir die Instanzvariable unseres Entity-Typen `Kunde` auch mit einem Unterstrich versehen. Es handelt sich hierbei aber nicht um eine generierte Meta-Klasse!

```
CriteriaBuilder cb = em.getCriteriaBuilder();
Metamodel metamodel = em.getMetamodel();
CriteriaQuery<Tuple> query = cb.createTupleQuery();
Root<Kunde> kunde = query.from(Kunde.class);
EntityType<Kunde> kunde_ = metamodel.entity(Kunde.class);
query.select(cb.tuple(
        kunde,
        // zum Vergleich mit generierten Meta-Klassen:
        // kunde.get(Kunde_.nachname),
        kunde.get(kunde_.getSingularAttribute("nachname")),
        // kunde.get(Kunde_.vorname).alias("VORNAME"))
        kunde.get(kunde_.getSingularAttribute("vorname")).alias("
            VORNAME")));
TypedQuery<Tuple> q = em.createQuery(query);
```

Die JPA-Spezifikation erlaubt im Rahmen der Criteria-Query auch Abfragen, die letzten Endes wieder in einer schwach typisierten Abfrage resultieren. Die überladenen Methoden wie `join()` oder `get()` erlauben auch Strings als Übergabeparameter. Das Ergebnis bzw. potenzielle Fehler sind dann erst zur Laufzeit zu erkennen.

9 Bean-Validierung mit JSR 303

Java-EE 6 führte neben JPA 2.0 auch *Bean Validation 1.0* ein. Dieses Validierungs-Framework ist im JSR 303 [URL-BV] definiert und wird häufig mit BV abgekürzt. Es stellt eine Reihe sogenannter Constraints zur Verfügung, die als Annotationen die erlaubten Werte von Bean-Properties einschränken. Ein einfaches Beispiel ist die Längenbegrenzung eines Strings etwa für einen Nachnamen:

```
@Size(min = 3, max = 30)
private String nachname;
```

Bean-Validation kann allgemein für JavaBeans und somit auch für JPA-Entities verwendet werden. Die JPA-Spezifikation realisiert die nahtlose Integration von Bean-Validation in das JPA-Programmiermodell: Ein JPA-Provider muss – falls eine BV-Implementierung im Klassenpfad verfügbar ist – diese automatisch verwenden.

Neben der deklarativen Beschränkung erlaubter Werte ist eine weitere Eigenschaft der Bean-Validation die Möglichkeit, Redundanzen im Java-EE-Stack zu vermeiden. Auch JSF 2.0 integriert Bean-Validation nach dem oben beschriebenen Automatismus, so dass sowohl für JSF als auch für JPA dieselben Constraints zur Validierung verwendet werden und so das DRY-Prinzip (Don't Repeat Yourself) Beachtung findet.

Bean-Validation ist unverzichtbarer Bestandteil von Java-EE 6, aber auch in Java-SE verwendbar. Wir verwenden für unsere Beispiele *Hibernate Validator* [URL-HV], die Referenzimplementierung von Bean-Validation. Hibernate Validator wird auch in den populären Application-Servern GlassFish 3 und JBoss-AS 7 verwendet.

9.1 Einfache Validierungen

In Abschnitt 3.4.4 haben wir einen Entity-Listener implementiert, der beim Persistieren eines Entity dessen Vornamen validiert. Eine sehr viel einfachere und elegantere Möglichkeit ist die Verwendung der von Bean-Validation bereitgestellten Annotationen. Das im Folgenden dargestellte Entity `Kunde` verwendet die BV-Annotationen `@NotNull`, `@Size`, `@Past` und `@Pattern`, die wir später noch erläutern. Diese und alle weiteren BV-Annotationen dürfen nur auf Instanz-, nicht jedoch auf Klassen-Properties angewendet werden.

LISTING 9.1 Entity `Kunde` mit Bean-Validation-Constraints

```
1  @Entity
2  public class Kunde {
3
4    @Id @GeneratedValue
5    private Integer id;
6
7    @NotNull
8    @Size(min = 3, max = 30)
9    @Column(length = 30, nullable = false)
10   private String vorname;
11
12   @NotNull
13   @Size(min = 3, max = 30)
14   @Column(length = 30, nullable = false)
15   private String nachname;
16
17   @NotNull
18   @Past
19   @Temporal(TemporalType.DATE)
20   private Date geburtsdatum;
21
22   @NotNull
23   @Size(min = 3, max = 30)
24   @Pattern(regexp = ".+@.+\\..+")
25   @Column(length = 30, nullable = false)
26   private String email;
27
28   @NotNull
29   @Column(nullable = false)
30   private Anrede anrede;
31
32   @NotNull
33   private Adresse adresse;
34
35   ...
36 }
```

Das Beispiel zeigt neben den BV-Annotationen auch die JPA-Annotationen `@Temporal` und `@Column`. Während `@Temporal` benötigt wird, um die zeitliche Detaillierung festzulegen, ist `@Column` optional. Wir haben die Annotation `@Column` hinzugenommen, um hier noch einmal auf deren Relevanz bezüglich der Validierung von Werten einzugehen. Die `@Column`-Annotation validiert *nicht*. Wenn man die Provider-Werkzeuge zur Schema-Generierung nutzt, werden die Informationen entsprechend verwendet. Im Beispiel erzeugen wir für die String-Properties Spalten mit `VARCHAR(30)` und Not-Null-Constraints. Falls ein bereits existierendes Schema verwendet wird, das längere Strings oder Null-Werte zulässt, erzeugt die Verwendung der `@Column`-Annotation zur Laufzeit keinen Fehler.

Zurück zum Beispiel. Das Entity `Kunde` in Listing 9.1 verwendet die Constraints `@NotNull`, `@Size`, `@Past` (Zeile 18) und `@Pattern` (Zeile 24). Diese prüfen das annotierte Property auf Existenz, Größe, zeitliche Lage in der Vergangenheit und auf ein Textmuster. Eine Übersicht aller verfügbaren Constraints gibt Tabelle 9.1. Die Constraints befinden sich im Package `javax.validation.constraints`. Das originäre Package für Bean-Validation ist

`javax.validation`. Mehrere Unter-Packages enthalten weitere Funktionalität, wie etwa eine Service-Provider-Schnittstelle.

TABELLE 9.1 Constraint-Annotationen des JSR 303

Annotation	Beschreibung
`@AssertFalse`	Property muss `false` sein
`@AssertTrue`	Property muss `true` sein
`@DecimalMax`	Property kleiner oder gleich Wert von `value`
`@DecimalMin`	Property größer oder gleich Wert von `value`
`@Digits`	Property darf nicht mehr Vor- und Nachkommastellen haben, als in `integer` und `fraction` angegeben
`@Future`	Property muss Datum in der Zukunft sein
`@Max`	Property kleiner oder gleich Wert von `value`
`@Min`	Property größer oder gleich Wert von `value`
`@NotNull`	Property darf nicht `null` sein
`@Null`	Property muss `null` sein
`@Past`	Property muss Datum in der Vergangenheit sein
`@Pattern`	Property muss Pattern in `regexp` entsprechen
`@Size`	Größe des Property muss zwischen `min` und `max` liegen

Die Annotation `@Future` verlangt, dass das annotierte Datum in der Zukunft liegt und ist damit komplementär zu `@Past`. Für die Bereichsprüfung ganzer Zahlen stehen `@Min` und `@Max`, für gebrochene Zahlen `@DecimalMin` und `@DecimalMax` zur Verfügung. Der Constraint `@Size` beschränkt die Länge von Strings, wie im Beispiel gesehen, aber auch die Größe von Collections, wofür wir später noch ein Beispiel entwickeln werden.

Die Verwendung der Beispielklasse mit Bean-Validierung erfolgt wie in vorangegangenen Beispielen:

```
...
Kunde kunde = new Kunde("Max", "Mustermann", "29.07.1962",
                        "max@mustermann.de", Anrede.HERR);
kunde.setAdresse(new Adresse("Am kurzen Weg", "17 a",
                             "12345", "Berlin"));
em.persist(kunde);
...
```

Im Falle einer Constraint-Verletzung wird die Runtime-Exception `ConstraintViolationException` geworfen. Diese Exception ist ein Wrapper über die Menge aller Constraint-Verletzungen für die validierte Bean. Die Constraints einer Bean werden alle gemeinsam geprüft. Im Fehlerfall können daher mehrere Constraint-Verletzungen existieren, die alle in eine `ConstraintViolationException` gepackt werden. Zusätzlich können Beans in Beziehungen zu anderen Beans stehen, deren Properties ebenfalls validiert werden. Wir gehen hierauf im Abschnitt 9.2 ein. Auch diese Constraint-Verletzungen werden über die `ConstraintViolationException` als Hülle weitergereicht.

Es ist dem Entwickler überlassen, wie detailliert die Fehlermeldungen für den Endanwender auszusehen haben. In der Regel genügt aber nicht *„Die Kundendaten sind teil-*

weise nicht erlaubt", sondern eher „*Fehlender Nachname*" oder „*Vorname zu lang*". Wir wollen daher kurz darauf eingehen, wie die entsprechenden Fehlermeldungen aus der `ConstraintViolationException` zu extrahieren sind. Der folgende Code-Ausschnitt zeigt den entsprechenden Catch-Block.

```
...
try {
  ...
  em.persist(kunde);
  ...
} catch (ConstraintViolationException cve) {
  Set<ConstraintViolation<?>> violations =
      cve.getConstraintViolations();
  for (ConstraintViolation<?> violation : violations) {
    System.out.println("ConstraintDescriptor: "
                       + violation.getConstraintDescriptor());
    System.out.println("PropertyPath: "
                       + violation.getPropertyPath());
    System.out.println("Message: "
                       + violation.getMessage());
  }
}
...
```

Die Methode `getConstraintViolations()` liefert eine Menge von `ConstraintViolation`-Objekten, über die iteriert werden kann. Das Beispiel in Listing 9.1 verlangt, dass ein Nachname nicht `null` ist und dass die E-Mail eines Kunden zwischen 3 und 30 Zeichen lang ist sowie einem gewissen Textmuster folgt. Falls diese drei Bedingungen nicht erfüllt sind, liefert der Rumpf obiger For-Schleife die folgende Ausgabe:

```
ConstraintDescriptor: ConstraintDescriptorImpl{...
PropertyPath: nachname
Message: kann nicht null sein

ConstraintDescriptor: ConstraintDescriptorImpl{...
PropertyPath: email
Message: muss zwischen 3 und 30 liegen

ConstraintDescriptor: ConstraintDescriptorImpl{...
PropertyPath: email
Message: muss auf Ausdruck ".+@.+\..+" passen
```

Der Constraint-Descriptor enthält Informationen über die zu validierende Bean und deren Properties. Der interessierte Leser sollte dieses Objekt als Ausgangspunkt für detailliertere Informationen verwenden. Man erkennt weiter, dass für jedes Property die entsprechende Fehlermeldung in lokalisierter Form über die Methode `getMessage()` verfügbar ist. Der von uns verwendete Provider Hibernate Validator enthält Lokalisierungen für Deutsch, Englisch, Französich, Spanisch, Portugiesisch, Ungarisch und Türkisch sowie Mongolisch und Chinesisch. Es ist jedoch auch möglich, individuelle Fehlermeldungen zu erzeugen. Alle in Tabelle 9.1 aufgezählten Annotationen besitzen das Attribut `message`, dessen String-Wert als Fehlermeldung fungiert und mit den anderen Attributen der Annotation parametrisiert ist. Die überarbeitete Anwendung der Annotation für den Nachnamen verwendet dieses Attribut:

```
@NotNull
@Size(min = 3, max = 30, message =
```

```
    "Der Nachname muss zwischen {min} und {max} Zeichen haben.")
@Column(length = 30, nullable = false)
private String nachname;
```

Die Fehlermeldung stellt sich nun wie erwartet dar:

```
ConstraintDescriptor: ConstraintDescriptorImpl{...
PropertyPath: nachname
Message: Der Nachname muss zwischen 3 und 30 Zeichen haben.
```

Tipp:

Bean-Validation erlaubt die Verwendung von Constraints auch in Interfaces und sowohl auf Properties als auch auf Gettern. Da JPA ausschließlich mit Klassen arbeitet, gehen wir auf Interfaces bei der Verwendung von Bean-Validation nicht ein. Bei der Wahlmöglichkeit zur Platzierung der BV-Constraints raten wir dazu, konsistent mit der Verwendung der JPA-Annotationen zu sein.

9.2 Beziehungen und eingebettete Objekte

Für das Property `adresse` vom Typ `Adresse` (Zeile 33 in Listing 9.1) wurde lediglich `@NotNull` als Constraint definiert. Wenn die Klasse `Adresse` serialisierbar ist, wird das Property von JPA als Byte-Array behandelt, und Bean-Validation überprüft lediglich den `@NotNull`-Constraint. Der folgende Code zeigt die Klasse `Adresse`, deren Properties ebenfalls mit BV-Annotationen versehen sind.

```
public class Adresse implements Serializable {

  @NotNull
  @Size(min = 3, max = 30)
  private String strasse;

  @NotNull
  @Size(max = 4)
  private String hausnummer;

  @NotNull
  @Size(min = 5, max = 5)
  private String plz;

  @NotNull
  @Size(min = 3, max = 30)
  private String ort;

  ...
}
```

Bean-Validation ist allgemein für JavaBeans verwendbar und nicht nur für Entity-Beans. Die Bean `Adresse` kann daher prinzipiell validiert werden. Ohne weitere Konfiguration werden assoziierte Beans jedoch nicht validiert. Die Validierung assoziierter Beans kann man mit der

Annotation `@Valid` explizit fordern. Im Beispiel des Kunden gilt es, das Property `adresse` entsprechend zu verändern.

```
@Valid
@NotNull
private Adresse adresse;
```

Nun werden alle Constraints der Klasse `Adresse` bei der Validierung einer Kundeninstanz ebenfalls validiert.

Falls die binäre Darstellung der Adresse in der Datenbank unerwünscht ist, kann die Adresse in den Kunden eingebettet werden, wie wir dies in den Kapiteln 2 und 4 bereits dargestellt haben. Dazu wird das Property `adresse` in der Klasse `Kunde` als `@Embedded` annotiert. Damit die eingebettete Adresse validiert wird, muss ebenfalls mit `@Valid` annotiert werden.

```
@Entity
public class Kunde {

  ...
  @NotNull
  @Valid
  @Embedded
  private Adresse adresse;
  ...
}
```

Die Klasse `Adresse` muss einbettbar sein, also mit `@Embeddable` annotiert werden.

```
@Embeddable
public class Adresse {

  @NotNull
  @Size(min = 3, max = 30)
  private String strasse;
  ...
}
```

Bei der Validierung allgemeiner Beziehungen verhält es sich anders. Derartig in Beziehung stehende Objekte (mit `@OneToOne`, `@OneToMany`, `@ManyToOne` oder `@ManyToMany` annotierte Properties) sind beide JPA-Entities, so dass jedes Entity einzeln validiert wird. Weil die Validierung nicht nur vor dem Persistieren, sondern auch vor dem Aktualisieren und Löschen von Entities durchgeführt wird, kann hier sogar das Problem entstehen, „zu viel“ zu validieren: Wenn ein Beziehungsziel wegen eines deklarierten Lazy-Loadings nicht geladen ist, würde dessen Validierung das Beziehungsziel nachladen und damit die Datenbank mit einem, eventuell mehreren zusätzlichen Select-Anweisungen belasten. Um dies zu vermeiden, definiert JPA das Interface `TraversableResolver` im Package `javax.validation`, das definiert, ob eine Validierung zum Beziehungsziel kaskadieren darf oder nicht. Die Spezifikation verlangt ausdrücklich, dass die Validierung für alle vier Beziehungsarten nicht kaskadiert.

Das in Abschnitt 2.2.5 vorgestellte späte Laden von Properties über die `@Basic`-Annotation mit

```
@Basic(fetch = FetchType.LAZY)
```

fällt ebenfalls in diese Kategorie. Derartige Properties dürfen in Folge einer Validierung nicht nachgeladen werden, falls sie noch nicht geladen sind. Das Listing 11.4 auf Seite 283 enthält ein Beispiel zur Verwendung der Annotation `@Basic` mit spätem Laden.

Als letzten Punkt dieses Abschnitts wollen wir noch einmal die Verwendung des Constraints `@Size` diskutieren. Bisher haben wir `@Size` für die Beschränkung der Länge eines Strings verwendet, aber bereits die Anwendbarkeit für Collections erwähnt. Collections können in Form von Beziehungen (Abschnitte 4.1 und 4.2) und Wertemengen (Abschnitt 4.3) in JPA verwendet werden. Die Annotation `@Size` erlaubt für beide Alternativen, die Größe der Collection einzuschränken. Als Beispiel verwenden wir eine Liste von TANs (Transaktionsnummern). Der Kunde kann diese selbst anfordern und generieren lassen. Pro Anforderung werden 10 TANs generiert, insgesamt darf ein Kunde nur maximal 50 TANs vorrätig haben. In einer realen Anwendung würde dies wahrscheinlich funktional bei der Generierung überprüft werden, wir wollen dies jedoch zu Demonstrationszwecken auf JPA-Ebene tun. Der hierfür benötigte Code reduziert sich auf die einfache Verwendung von `@Size`:

```
@Size(max = 50)
@ElementCollection
private List<String> tans;
```

Werden mehr als 50 TANs gespeichert, wird eine `ConstraintViolationException` geworfen.

9.3 Konfiguration

Die automatische Verwendung der Constraints kann per Konfiguration ausgeschaltet werden. Zudem kann der Zeitpunkt der Validierung bestimmt werden. Wir beginnen zunächst mit der Frage, ob validiert werden soll oder nicht. Diese sowie alle weiteren Konfigurationen können sowohl in der Datei `persistence.xml` als auch programmatisch in Java vorgenommen werden. Die Datei `persistence.xml` ist die zentrale JPA-Konfigurationsdatei und wurde bereits in den Abschnitten 3.2 und 3.3 eingeführt. Pro Persistenzeinheit kann mit dem Element `<validation-mode>` der Validierungsmodus definiert werden. Das folgende Beispiel zeigt die Verwendung von `AUTO` als Validierungsmodus. Dies ist der Default, so dass das Beispiel identisch zur Nichtverwendung des `<validation-mode>`-Elements ist.

```
<persistence ... >
  <persistence-unit name="bank">
    ...
    <validation-mode>AUTO</validation-mode>
    ...
  </persistence-unit>
</persistence>
```

Die erlaubten Werte des Validierungsmodus entsprechen den Werten der Enumeration `ValidationMode` im Package `javax.persistence`. Dies sind

- `AUTO`
  Der JPA-Provider muss bei Verfügbarkeit eines BV-Providers validieren.

- `CALLBACK`
  Der JPA-Provider muss validieren.
- `NONE`
  Der JPA-Provider darf nicht validieren.

Tipp:

Falls Ihre Anwendung mit Hilfe von Bean-Validation nachweisbar validieren soll, müssen Sie `CALLBACK` verwenden. Falls Sie `AUTO` verwenden und beim Packaging oder Deployment ein Fehler passiert und der BV-Provider nicht eingebunden wird, bemerkt dies die Anwendung nicht, da die Constraint-Annotationen einfach ignoriert werden. Bei der Verwendung von `CALLBACK` wird in diesem Fall eine Exception geworfen.

Die bereits angesprochene programmatische Konfiguration von Bean-Validation erfolgt über eine Map von Properties. Das für Bean-Validation maßgebliche Property ist `javax.persistence.validation.mode`. Die Methode `createEntityManagerFactory()` der Klasse `Persistence` erlaubt im zweiten Parameter die Nutzung einer Map zur Konfiguration der Persistenzeinheit. Der folgende Code-Ausschnitt zeigt beispielhaft, wie die Validierung ausgeschaltet werden kann.

```
Map<String, String> props = new HashMap<String, String>();
props.put("javax.persistence.validation.mode", "NONE");
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("bank", props);
EntityManager em = emf.createEntityManager();
```

Weil ein Application-Server die Erzeugung der `EntityManagerFactory` implizit vornimmt, kann man in Java-EE nur die Alternative über die Datei `persistence.xml` anwenden, oder muss auf container-spezifische Konfigurationsmöglichkeiten zurückgreifen.

9.4 Explizite Validierung

Die einfache Verwendung und der automatische Einsatz von BV-Constraints machen gerade den Charme dieses Ansatzes aus. Wenn der Alles-oder-nichts-Ansatz jedoch nicht gewünscht ist und nur Instanzen einer bestimmten Klasse validiert werden sollen, ist auch dies möglich. Die Entscheidung, ob validiert werden soll, ist bis auf Ebene der einzelnen Instanz möglich. Das entsprechende API enthält eine Methode, die sogar eine Was-wäre-wenn-Validierung erlaubt. Obwohl in der Praxis derartige Validierungen eher selten vorkommen, stellen wir diese Alternative der Vollständigkeit halber dar.

Das API zur expliziten Validierung wird durch das Interface `Validator` im Package `javax.validation` definiert. In Java-SE kann über die Klasse `Validation` und das Interface `ValidatorFactory` eine Instanz von `Validator` erzeugt werden:

```
ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
Validator validator = factory.getValidator();
```

In Java-EE wird die `Validator`-Instanz mit der `@Resource`-Annotation injiziert:

```
@Stateless
public class MySLSB {

  @Resource
  Validator validator;
  ...
}
```

Wenn die `Validator`-Instanz existiert, kann mit den Methoden `validate()`, `validateProperty()` und `validateValue()` eine Bean, ein Property einer Bean oder ein Wert für ein Property (was wäre wenn?) validiert werden. Alle drei Methoden enthalten als jeweils letzten Parameter ein Varargs-Argument zur Angabe von Validierungsgruppen, auf die wir in Abschnitt 9.5 eingehen.

```
<T> java.util.Set<ConstraintViolation<T>>
        validate(T object, java.lang.Class<?>... groups)

<T> java.util.Set<ConstraintViolation<T>>
        validateProperty(T object,
                         java.lang.String propertyName,
                         java.lang.Class<?>... groups)

<T> java.util.Set<ConstraintViolation<T>>
        validateValue(java.lang.Class<T> beanType,
                      java.lang.String propertyName,
                      java.lang.Object value,
                      java.lang.Class<?>... groups)
```

Wir beginnen mit einem Beispiel zur Validierung einer Bean mit der Methode `validate()` und verwenden das bereits bekannte Kunde-Entity.

```
ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
Validator validator = factory.getValidator();

Kunde kunde = new Kunde("Max", "Mustermann", "29.07.1962",
                        "max@mustermann.de", Anrede.HERR);
kunde.setAdresse(...);

Set<ConstraintViolation<Kunde>> violations =
        validator.validate(kunde);
for (ConstraintViolation<?> violation : violations) {
  ...
}
```

Der Aufruf von `validator.validate()` liefert eine Menge von `ConstraintViolations` zurück, die identisch zum Exception-Handling des Beispiels auf Seite 230 weiterverwendet werden können. Der Unterschied zu unseren bisherigen BV-Beispielen ist die Unabhängigkeit von JPA. Das Entity `Kunde` muss nicht persistiert werden.

Nach demselben Schema kann ein einzelnes Property mit der Methode `validateProperty()` validiert werden. Das zu validierende Property oder der Getter ist in String-Form anzugeben.

```
violations = validator.validateProperty(kunde, "nachname");
```

Die dritte Validierungsmethode benötigt nicht einmal eine konkrete Instanz der zu validierenden Bean, sondern lediglich die Klasse, das Property und den (potenziellen) Wert. Die Methode `validateValue()` realisiert damit sozusagen eine Was-wäre-wenn-Validierung.

```
violations = validator.validateValue(Kunde.class, "nachname", "A");
```

9.5 Validierungsgruppen

In allen bisherigen Beispielen fand die Validierung vor dem eigentlichen Persistieren des Entity statt, wenn man von der expliziten Validierung im letzten Abschnitt einmal absieht. Der genaue Zeitpunkt der Validierung ist über den Lebenszyklus eines Entity definiert und somit über die in Abschnitt 3.4 bereits angesprochenen Ereignisarten. Von den dort genannten sieben Ereignisarten sind für eine Validierung immer nur die Zeitpunkte *vor* einer Operation sinnvoll, so dass drei Ereignisarten übrig bleiben: Pre-Persist, Pre-Update und Pre-Remove. Die Validierungen finden statt, wenn alle entsprechend mit `@PrePersist`, `@PreUpdate` und `@PreRemove` annotierten Callback-Methoden ausgeführt und beendet wurden. Um das Verhalten genauer spezifizieren und vor allem ändern zu können, wurden entsprechende *Validierungsgruppen* eingeführt, die über Properties definiert werden können. Die Definition der Properties erfolgt dabei entweder über die Datei `persistence.xml` oder über die Map der Entity-Manager-Fabrik. Die drei Properties sind:

- `javax.persistence.validation.group.pre-persist`
- `javax.persistence.validation.group.pre-update`
- `javax.persistence.validation.group.pre-remove`

Die Werte der drei Properties sind die jeweils zu verwendenden Validierungsgruppen. Eine Validierungsgruppe ist ein Interface, dessen voll qualifizierter Klassenname angegeben werden muss. Bei mehreren Validierungsgruppen sind diese durch Komma zu trennen.

Bean-Validation definiert eine Validierungsgruppe: die Default-Gruppe. Diese wird durch das Interface `Default` im Package `javax.validation.groups` repräsentiert. Diese Gruppe ist für Pre-Persist und Pre-Update, nicht jedoch für Pre-Remove vordefiniert. JPAs Default für Validierungsgruppen entspricht also der folgenden Verwendung der Properties:

```
<persistence ... >
  <persistence-unit name="bank">
    ...
    <properties>
      <property name="javax.persistence.validation.group.pre-persist"
                value="javax.validation.groups.Default"/>
      <property name="javax.persistence.validation.group.pre-update"
                value="javax.validation.groups.Default"/>
      <property name="javax.persistence.validation.group.pre-remove"
                value=""/>
    </properties>

  </persistence-unit>
</persistence>
```

Um auch beim Löschen eines Entity zu validieren, muss das dritte Property ebenfalls den Wert `javax.validation.groups.Default` bekommen.

Die Möglichkeit, anwendungsdefinierte Validierungsgruppen zu definieren, erlaubt eine sehr flexible Verwendung von Bean-Validation. Wir entwickeln zunächst ein Beispiel, das die Erlaubnis zum Löschen eines Entity überwacht und dann ein Beispiel, das alternative Dateneingaben für ein Entity realisiert.

Für das erste Beispiel soll ein Kunde nur gelöscht werden können, falls er keine Verbindlichkeiten gegenüber der Firma oder Bank mehr hat. Als anwendungsdefinierte Validierungsgruppe definieren wir das leere Interface `Remove`:

```
public interface Remove {
}
```

Dieses Interface kann nun über das `groups`-Attribut als Validierungsgruppe angegeben werden. Das `groups`-Attribut ist für alle vordefinierten (Tabelle 9.1) und anwendungsdefinierten Constraints verfügbar. Im Beispiel wählen wir ein boolesches Attribut, um die Existenz von Verbindlichkeiten anzuzeigen, so dass sich folgender Code ergibt:

```
@Entity
public class Kunde {
  ...
  @AssertFalse(groups = Remove.class)
  private boolean hatVerbindlichkeiten;
  ...
}
```

Wird nun dem Property `javax.persistence.validation.group.pre-remove` der voll qualifizierte Klassenname (`de.jpainfo.model.Remove`) als Wert zugewiesen, so wird beim Löschen des Entity der Constraint überprüft, beim Einfügen oder Aktualisieren jedoch nicht.

Als zweites Beispiel modellieren wir Zahlungsdaten für eine Online-Bestellung. Dabei soll zwischen den Alternativen *Bankeinzug* und *Kreditkarte* gewählt werden können. Dies könnte eventuell mit mehreren Klassen und objektorientierter Vererbung realisiert werden, wir wollen hier jedoch alle Daten in einer Klasse vereinen, um die Möglichkeit einer alternativen Validierung zu demonstrieren. Die Klasse `Zahlungsdaten` in Listing 9.2 zeigt die entsprechenden BV-Constraints.

LISTING 9.2 Entity `Zahlungsdaten` mit alternativ zu validierenden Daten

```
1  @Entity
2  public class Zahlungsdaten {
3
4    @Id @GeneratedValue
5    private Integer id;
6
7    @NotNull
8    private String name;
9
10   @Null(groups = Kreditkarte.class)
11   @NotNull(groups = Bankeinzug.class)
12   private String bank;
13   @Null(groups = Kreditkarte.class)
```

```
14     @NotNull(groups = Bankeinzug.class)
15     private String blz;
16     @Null(groups = Kreditkarte.class)
17     @NotNull(groups = Bankeinzug.class)
18     private String kontonummer;
19
20     @Null(groups = Bankeinzug.class)
21     @NotNull(groups = Kreditkarte.class)
22     private String institut;
23     @Null(groups = Bankeinzug.class)
24     @NotNull(groups = Kreditkarte.class)
25     private String nummer;
26     @Null(groups = Bankeinzug.class)
27     @NotNull(groups = Kreditkarte.class)
28     private String gueltigBis;
29     ...
30 }
```

Der Name des Kunden wird mit `@NotNull` versehen, so dass dieses Property beim Einfügen und Ändern validiert wird. Die Properties `bank`, `blz` und `kontonummer` in den Zeilen 10 bis 18 müssen für die Alternative *Bankeinzug* vorhanden sein, so dass die Constraints `@Null` und `@NotNull` jeweils mit der entsprechenden Validierungsgruppe versehen sind. Bei den Properties `institut`, `nummer` und `gueltigBis` in den Zeilen 20 bis 28 werden die Validierungsgruppen für die beiden Constraints jeweils vertauscht. Auf weitere Constraints, wie etwa `@Size`, haben wir der Übersichtlichkeit halber verzichtet.

Wird nun `javax.persistence.validation.group.pre-persist` auf eine der beiden (leeren) Interfaces `Bankeinzug` oder `Kreditkarte` und, durch Komma getrennt, auf `Default` gesetzt, wird das Vorhandensein der jeweiligen Daten geprüft. Der Name des Kunden muss bei beiden Alternativen existieren. Um auf die aufwendige Angabe mehrerer Validierungsgruppen verzichten zu können, können Validierungsgruppen erben. Wenn im Beispiel das Interface `Bankeinzug` von `Default` erbt, genügt die Angabe von `Bankeinzug`, um beide Validierungen vorzunehmen:

```
public interface Bankeinzug extends Default { }
```

Die geschilderte Flexibilität über alternative Validierungen wird in JPA dadurch erheblich eingeschränkt, dass die Validierungsgruppe bei der Erzeugung der Entity-Manager-Fabrik angegeben werden muss und damit für die gesamte Persistenzeinheit gültig ist. Für eine sinnvolle Verwendung des Beispiels muss daher mit zwei Persistenzeinheiten gearbeitet werden. Eine zukünftige Version von JPA löst dies hoffentlich eleganter.

Ein Beispiel für eine sehr gelungene Integration von Validierungsgruppen demonstriert JavaServer Faces. In der Version 2.0 von JSF wurde das Tag `<f:validateBean>` eingeführt, dessen Attribut `validationGroups` die Angabe der zu verwendenden Validierungsgruppen erlaubt.

Der folgende Ausschnitt einer JSF-Seite zeigt die Verwendung des Tags `<f:validateBean>` mit dem Attribut `validationGroups`.

```
<f:validateBean validationGroups="de.jpainfo.model.Kreditkarte">
  ...
  <h:inputText value="#{...zahlungsdaten.bank}" />
  ...
```

```
    <h:inputText value="#{...zahlungsdaten.institut}" />
    ...
  </f:validateBean>
```

Ohne auf JSF-Details einzugehen, sei hier nur erwähnt, dass die Angabe des Wertes für `validationGroups` nicht literal zu erfolgen hat. Es kann ein allgemeiner Expression-Language-Ausdruck verwendet werden, dessen Auswertung zur Laufzeit in Java erfolgt, so dass die Validierungsgruppe dynamisch bestimmt wird. Für eine Einführung in JSF empfehlen wir unser Buch zu JSF 2.0 [Mü10].

9.6 Anwendungsdefinierte Validierungs-Constraints

Sollten die in Tabelle 9.1 auf Seite 229 dargestellten Constraints nicht ausreichen, erlaubt Bean-Validation die Erstellung anwendungsdefinierter Constraints. Man benötigt hierzu die den Constraint repräsentierende Annotation sowie den eigentlichen Constraint-Validierer.

Als Beispiel für einen benutzerdefinierten Constraint wollen wir die Volljährigkeit eines Kunden prüfen. Bei der Code-Darstellung beginnen wir mit der Annotation. Wir definieren die Annotation `@Volljaehrig`, die anstatt der Annotation `@Past` für den Geburtstag verwendet werden kann.

```
@Constraint(validatedBy = VolljaehrigValidator.class)
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Volljaehrig {

    String message() default "Nicht volljährig";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};

}
```

Die Annotationen `@Target` und `Retention` sind die Standard-JDK-Annotationen zur Definition von Annotationen. `@Constraint` ist die JPA-spezifische Meta-Annotation, die aus der Annotation `@Volljaehrig` einen BV-Constraint macht. Im Attribut `validatedBy` wird der Validierer `VolljaehrigValidator` an diesen Constraint gebunden. Man erkennt die beiden bereits bekannten Attribute `message` und `groups` und das von uns nicht weiter erläuterte Attribut `payload`.

Beim Attribut `message` bleibt anzumerken, dass eine internationalisierte Fehlermeldung deutlich mehr Aufwand benötigt, aber nicht Ziel dieses Buchs ist. Das Attribut `groups` erlaubt die Verwendung anwendungsdefinierter Constraints in Validierungsgruppen, wie in Abschnitt 9.5 gezeigt.

Die Implementierung des Constraints erfolgt in der Klasse `VolljaehrigValidator`, die das Interface `ConstraintValidator` implementieren muss.

```
public class VolljaehrigValidator implements ConstraintValidator<
    Volljaehrig, Date>  {
```

```
    @Override
    public void initialize(Volljaehrig constraintAnnotation) {
      // TODO Auto-generated method stub
    }

    @Override
    public boolean isValid(Date date, ConstraintValidatorContext cvc) {
      // heute:
      Calendar cal = new GregorianCalendar();
      // heute vor 18 Jahren:
      cal.add(Calendar.YEAR, -18);
      return date.before(cal.getTime());
    }

  }
```

Die Methode `initialize()` erlaubt den Zugriff auf Attribute der Annotation. Unsere Annotation `@Volljaehrig` besitzt keine anwendungsdefinierten Attribute, so dass der Methodenrumpf leer bleiben kann.

Die Methode `isValid()` realisiert die Validierungslogik. Der erste Parameter enthält das annotierte Property, der zweite den Kontext, in dem die Validierung stattfindet. Dieser wird im Beispiel nicht weiter verwendet.

Nach der Definition der Annotation und des Constraint-Validierers kann nun die Annotation verwendet werden, um die Volljährigkeit eines Kunden zu garantieren:

```
  @Volljaehrig
  private Date geburtstag;
```

Wie bereits erwähnt, stehen auch für anwendungsdefinierte Constraints die Attribute `message` und `groups` zur Verfügung.

9.7 Validierung mit XML

Der mit Java-EE 5 eingeschlagene und mit Java-EE 6 fortgeführte Weg, bestimmte konfigurierende und definierende Eigenschaften sowohl mit Annotationen als auch mit Hilfe von XML spezifizieren zu können, wird auch von Bean-Validation realisiert. Wir skizzieren ansatzweise das Vorgehen, um den Umfang dieses Kapitels nicht zu sprengen.

Für die Konfiguration von Bean-Validation wird die Datei `META-INF/validation.xml` verwendet. In ihr kann z. B. der BV-Provider, der in Abschnitt 9.2 erwähnte Traversable-Resolver oder die Implementierung des Interfaces `ValidatorFactory` aus Abschnitt 9.4 angegeben werden. Da Bean-Validation ebenfalls die Maxime *Convention over Configuration* befolgt, genügt für unser Beispiel die Angabe der Datei, die die Constraints enthält.

```
<validation-config
  xmlns="http://jboss.org/xml/ns/javax/validation/configuration"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
      "http://jboss.org/xml/ns/javax/validation/configuration">
```

```
  <constraint-mapping>
      /META-INF/kunde-constraints.xml
  </constraint-mapping>

</validation-config>
```

In der Datei `kunde-constraints.xml` wird für jede zu validierende Klasse die XML-Entsprechung der Constraints aus Tabelle 9.1 und der anwendungsdefinierten Constraints angegeben. Wir wollen uns hier auf den Vornamen des bekannten Beispiels beschränken, das folgendermaßen definiert war:

```
@NotNull
@Size(min = 3, max = 30)
private String vorname;
```

Die Umsetzung in XML ist im Folgenden dargestellt. Die Tags und Attribute sind selbsterklärend. Erwähnenswert ist das Attribut `ignore-annotations`. Ist es auf `true` gesetzt, werden sämtliche BV-Annotationen in der Bean ignoriert. Ist es auf `false` gesetzt, werden die Constraints in Java *und* XML verwendet. Sollte ein Constraint sowohl in Java als auch in XML definiert worden sein, hat die XML-Version die höhere Präferenz und wird zur Validierung genutzt.

```
<constraint-mappings
  xmlns="http://jboss.org/xml/ns/javax/validation/mapping"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
      "http://jboss.org/xml/ns/javax/validation/mapping
          validation-mapping-1.0.xsd">

  <bean class="de.jpainfo.model.KundeXML" ignore-annotations="true">
    <field name="vorname">
      <constraint annotation="javax.validation.constraints.NotNull"/>
      <constraint annotation="javax.validation.constraints.Size">
        <element name="min">3</element>
        <element name="max">30</element>
      </constraint>
    </field>
  </bean>

  <bean
  ...

</constraint-mappings>
```

Wir beenden hiermit unsere Ausführungen zur Bean-Validation. Dem interessierten Leser legen wir die Lektüre der Spezifikation [URL-BV] und der Dokumentation des Hibernate Validators [URL-HV] nahe. Hier findet er weitere Themen, wie die Zusammenfassung mehrerer Constraints zu einem neuen Constraint (Constraint Composition) oder die Definition der Ausführungsreihenfolge mehrerer Constraints über die Annotation `@GroupSequence`.

Projekt:

Das Projekt *bean-validation* enthält den Code für die Beispiele dieses Kapitels mit Ausnahme des JSF-Beispiels in Abschnitt 9.5. Das JSF-Beispiel ist im Projekt *bean-validation-ee* enthalten und benötigt GlassFish als Ablaufumgebung.

10 JPA-Konfiguration

In unseren bisherigen Beispielen haben wir fast ausschließlich die JPA-Annotationen benutzt. Wir sind davon überzeugt, dass es in den meisten Anwendungsfällen sinnvoll ist, die Meta-Informationen zur Überführung der objektorientierten Welt in eine relationale Datenbank direkt in der jeweiligen Java-Klasse durchzuführen. Darüber hinaus kann es jedoch Anforderungen bzw. betriebliche Erfordernisse geben, die das Mapping und die Konfiguration in separaten Dateien erforderlich machen. Dies kann z. B. auch dann der Fall sein, wenn der Java-Quellcode gar nicht zur Verfügung steht, diese Klassen aber in einer Datenbank persistiert werden sollen. In diesem Kapitel wollen wir uns nun ausführlich den Konfigurationsmöglichkeiten der Datei `persistence.xml` widmen. Darüber hinaus werden wir auf die zur Verfügung stehenden Elemente der XML-Mapping-Dateien im JPA-Standard eingehen und deren äquivalenten Java-Annotationen gegenüberstellen.

10.1 Persistenzeinheiten

Eine Persistenzeinheit, dem Leser bereits durch das XML-Element `<persistence-unit>` der Datei `persistence.xml` bekannt, setzt sich aus folgenden Elementen zusammen:

- einer Factory-Klasse, die eine Implementierung des Interfaces `EntityManager` bereitstellt, sowie deren Konfigurationen;
- einer Menge von Klassen, die im Rahmen dieser Einheit als persistent definiert werden;
- den Meta-Informationen, die darüber Auskunft geben, in welcher Form eine Klasse in der Datenbank abgebildet wird. Meta-Informationen können sowohl als Annotation als auch in XML definiert werden.

Eine Persistenzeinheit kann innerhalb einer JAR- oder einer WAR-Datei definiert sein. Die Persistenzeinheit ist dann im Bereich dieser Datei gültig. In diesem Bereich muss der Name einer Persistenzeinheit eindeutig sein.

Da EAR-Dateien sowohl JAR- als auch WAR-Dateien enthalten können, sind die Sichtbarkeitsregeln für Persistenzeinheiten innerhalb einer EAR-Datei etwas komplizierter. Wird eine Persistenzeinheit innerhalb eines EAR-Projektes definiert, so ist sie für alle Komponenten der EAR-Anwendung sichtbar. Kommt es zu einem Namenskonflikt zwischen einer Persistenzeinheit der EAR-Datei und einer Persistenzeinheit innerhalb einer JAR- bzw. WAR-Datei,

die in der EAR-Datei liegt, ist die Persistenzeinheit der EAR-Ebene nicht sichtbar. Sie kann jedoch mit Hilfe einer Pfadangabe und dem Zeichen # explizit angesprochen werden. Details hierzu finden sich in der JPA-Spezifikation im Abschnitt *Persistence Unit Scope* [URL-JPA2].

Persistenzeinheiten werden in der Datei `persistence.xml` definiert. Diese Datei muss im `META-INF`-Verzeichnis liegen. Das Verzeichnis befindet sich stets auf höchster Ebene innerhalb der Archivdatei bzw. des Verzeichnisses, für das die Persistenzeinheit gültig ist.

Das Wurzel-Element der Datei heißt `<persistence>` und kann aus einem oder mehreren `<persistence-unit>`-Elementen bestehen. Der Namensraum ist im Listing 10.1 dargestellt. Er ist erforderlich, da die EJB-Spezifikation eine Schemavalidierung verlangt.

LISTING 10.1 Grundgerüst der Datei `persistence.xml`

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
     http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  xmlns="http://java.sun.com/xml/ns/persistence" version="2.0">
  <persistence-unit name="..." transaction-type="JTA">
    <description>...</description>
    <provider>...</provider>
    <jta-data-source>...</jta-data-source>
    <non-jta-data-source>...</non-jta-data-source>
    <mapping-file>...</mapping-file>
    <jar-file>...</jar-file>
    <class>...</class>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>
    <shared-cache-mode>ALL</shared-cache-mode>
    <validation-mode>AUTO</validation-mode>
    <properties>
      <property name="..." value="..." />
    </properties>
  </persistence-unit>
</persistence>
```

Das XML-Element `<persistence-unit>` konfiguriert die eingangs erwähnte Persistenzeinheit. Es besitzt die Attribute `name` und `transaction-type`:

- `name`: Der eindeutige Bezeichner für die Einheit. Er wird in Java-SE-Anwendungen in der Methode `Persistence.createEntityManagerFactory(String unitName)` referenziert, um die entsprechende `EntityManagerFactory` zu initialisieren. In einer Java-EE-Umgebung referenzieren die Annotationen `@PersistenceUnit` bzw. `@PersistenceContext` diesen Namen. Damit kann der Application-Server den EJB-Komponenten zur Laufzeit den zur Persistenzeinheit gehörenden Entity-Manager injizieren.
- `transaction-type`: Dieses Attribut kann die Werte `RESOURCE_LOCAL` oder `JTA` annehmen. Wenn das Attribut nicht angegeben ist, wird in einer Java-SE-Umgebung der Wert `RESOURCE_LOCAL` angenommen, bei Java-EE der Wert `JTA`.

Folgende Sub-Elemente des Elements `<persistence-unit>` sind erlaubt:

- Das Element `<description>` ist optional und erlaubt eine erweiterte textuelle Beschreibung der Persistenzeinheit.

- Stehen in einer Anwendung mehrere JPA-Implementierungen zur Verfügung, kann innerhalb von `<provider>` eine Implementierung für die Persistenzeinheit angegeben werden. Diese Angabe sollte immer dann erfolgen, wenn die Applikation Erweiterungen einer bestimmten JPA-Implementierung voraussetzt. Der genannte Provider muss das Interface `javax.persistence.spi.PersistenceProvider` implementieren.
- Die Elemente `<jta-data-source>` bzw. `<non-jta-data-source>` kennzeichnen im Java-EE-Umfeld den globalen JNDI-Namen der Datasource, die vom Provider zu verwenden ist.
- Neben den in diesem Buch verwendeten Annotationen lässt JPA die Konfiguration der zu persistierenden Klassen in XML-Mapping-Dateien zu. Das Element `<mapping-file>` kann innerhalb einer Persistenzeinheit mehrmals auftreten und referenziert jeweils eine Datei, deren Format wir im Abschnitt 10.2 noch näher beschreiben werden. Existiert im Verzeichnis `META-INF` die Mapping-Datei `orm.xml`, wird diese Datei verwendet, selbst dann, wenn ihr Name nicht explizit in der Konfiguration angegeben wurde. Wir haben hierfür bereits in Abschnitt 3.7 ein Beispiel entwickelt.
- Das Element `<jar-file>` erlaubt die Angabe weiterer JAR-Dateien, in denen die JPA-Implementierung Klassen mit Persistenzinformationen finden kann. Die Pfadangabe erfolgt wie beim Element `<mapping-file>` relativ zum Wurzelverzeichnis der Persistenzeinheit, also oberhalb des Verzeichnisses `META-INF`.
- Klassen, die persistent sein sollen, werden mit dem Element `<class>` angegeben. Dieses Element kann zusätzlich zu den beiden vorhergehenden Elementen benutzt werden. In einer Java-SE-Umgebung sollten alle Klassen, die mit der JPA-Implementierung in der Datenbank abgelegt werden, hier aufgeführt sein. Die Spezifikation warnt ausdrücklich davor, dass eine Anwendung nicht portabel ist, wenn nicht alle Klassen mit dem `<class>`-Element in der Konfigurationsdatei aufgeführt sind.
- Das Element `<exclude-unlisted-classes>` kann die Werte `true` oder `false` annehmen und ist zur Verwendung im Java-EE-Umfeld bestimmt. Ist es auf `true` gesetzt, bleiben die Klassen vom JPA-Provider unberücksichtigt, die nicht im `<class>`-Element aufgeführt sind.
- Das `<properties>`-Element ist zur Konfiguration von implementierungsspezifischen Details zu verwenden. Es kann sich aus mehreren `<property>`-Elementen zusammensetzen. Die eigentliche Konfiguration wird innerhalb der Sub-Elemente mit den Attributen `name` und `value` durchgeführt. Die JPA-Spezifikation empfiehlt allen Implementierungen, ein spezielles Präfix zu benutzen, um Überschneidungen zu vermeiden. Properties, die eine Implementierung nicht kennt, sind von dieser zu ignorieren. Unsere Beispielprojekte nutzen dies, um die drei Provider Hibernate, EclipseLink und OpenJPA zu konfigurieren.
- Die Benutzung des Second-Level-Cache wird mit dem Element `<shared-cache-mode>` konfiguriert. Mögliche Werte des Elements sind `ALL`, `NONE`, `ENABLE_SELECTIVE`, `DISABLE_SELECTIVE` und `UNSPECIFIED`. Die Beschreibung findet der Leser im Abschnitt 6.3.5.
- Zur Validierung der Entity-Werte steht das Element `<validation-mode>` mit den Ausprägungen `AUTO`, `CALLBACK` und `NONE` zur Verfügung. Eine Einführung in die Bean-Validierung ist in Abschnitt 9.3 zu finden.

10.2 Mapping-Dateien und Annotationen

Wir haben uns, bis auf ein kurzes Beispiel am Anfang des Buches, bei der Konfiguration der Persistenzeinheit komplett auf Annotationen gestützt und sind nicht den Umweg über die Deklaration der einzelnen Elemente im XML-Format gegangen. Annotationen haben gegenüber der Mapping-Datei den Vorteil, dass sie dort hinterlegt werden, wo man sie auch verwendet. Sie sind lesbarer als lange Konfigurationsdateien und besser in die jeweiligen Entwicklungsumgebungen integriert.

Gegen Annotationen spricht, dass die Installation einer Anwendung in einer anderen Umgebung und dadurch erzwungen in einer anderen Konfiguration ohne erneutes Kompilieren nicht durchgeführt werden kann. Befinden sich die Konfiguration bzw. Teile von ihr in einer einfachen Textdatei, kann z. B. die Abbildung auf ein anderes Datenbankschema ohne Veränderung des Quellcodes durchgeführt werden.

JPA bietet die Möglichkeit, Mapping-Dateien *und* Annotationen zu benutzen. Teile der Mapping-Datei überschreiben dabei die per Annotation vorgenommene Konfiguration. Damit ist der Entwickler in der Lage, sowohl die Vorteile von Annotationen als auch die Flexibilität einer Konfiguration in einer separaten Textdatei zu nutzen.

Dieses Kapitel soll dem Leser neben einer Einführung in die Konfiguration auch als eine Art Referenz für das Java-Persistence-API dienen. Es enthält neben Erklärungen der einzelnen Elemente der Mapping-Datei die äquivalenten Annotationen im Quell-Code. Auf Besonderheiten, so sie denn noch nicht in unseren bisherigen Beispielen aufgeführt sind, werden wir ebenfalls aufmerksam machen.

Auf einen vollständigen Abdruck des XML-Schemas, mit dem die Mapping-Dateien validiert werden, haben wir bewusst verzichtet. Diese Datei findet der Leser in der JAR-Datei des JPA-Providers bzw. der Spezifikation und auch abgedruckt in der Dokumentation zur JPA selbst [URL-JPA2].

Wir haben stattdessen aus dem Schema eine nahezu vollständige XML-Mapping-Datei inklusive aller optionalen Attribute und Sub-Elemente generiert und wollen anhand dieser Datei unsere Referenz aufbauen. Dem Leser empfehlen wir zur optimalen Nutzung dieser Referenz auch die elektronische Version des Buches. In der PDF-Datei ist ein leichtes Navigieren per Mausklick zwischen den einzelnen XML-Elementen möglich.

Aus Gründen der Übersichtlichkeit verwenden wir in unseren Code-Fragmenten nicht immer syntaktisch korrektes XML. Auf diese Fälle wird der Leser immer dann stoßen, wenn wir ein XML-Sub-Element nicht mehrfach an den Stellen erklären, an denen es verwendet wird. Wir verweisen in diesen Fällen auf eine zentrale Stelle, an der dieses Element dann komplett beschrieben ist. Dazu ein Beispiel. Das Element `<entity-mappings>` besitzt das Sub-Element `<table-generator>`, das wir wie folgt darstellen:

```
<entity-mappings version="1.0">
  ...
  <table-generator>
    <!-- Abschnitt 10.2.5 Seite 250 -->
  </table-generator>
  ...
</entity-mappings>
```

Wir verschweigen nur, dass das Element `<table-generator>` zwingend ein Attribut `name` besitzen muss. Die richtige Definition des Elements findet sich erst im entsprechenden Abschnitt, auf den verwiesen wird. Wir sind der Meinung, dass die Referenz damit besser lesbar ist. Auf entsprechende Schema-Verletzungen einer Mapping-Datei wird der Entwickler durch seine Entwicklungsumgebung hingewiesen, so dass mögliche Fehler früh erkannt werden.

Den Elementen gegenübergestellt haben wir jeweils die entsprechenden Annotationen. Diesem Quellcode ist dann leicht zu entnehmen, welchen Datentyp die Elemente haben müssen, ob die Angabe zwingend erforderlich ist und welche Default-Werte JPA vorsieht.

10.2.1 Das Element `<entity-mappings>`

Das Wurzel-Element einer jeden Mapping-Datei ist das Element `<entity-mappings>`.

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings version="2.0"
  xmlns="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
      orm_2_0.xsd">
  <description>...</description>
  <persistence-unit-metadata>
    <!-- Abschnitt 10.2.2 Seite 248 -->
  </persistence-unit-metadata>
  <package>package</package>
  <schema>schema</schema>
  <catalog>catalog</catalog>
  <access>PROPERTY
    <!-- Abschnitt 10.2.3 Seite 249 -->
  </access>
  <sequence-generator>
    <!-- Abschnitt 10.2.4 Seite 249 -->
  </sequence-generator>
  <table-generator>
    <!-- Abschnitt 10.2.5 Seite 250 -->
  </table-generator>
  <named-query>
    <!-- Abschnitt 10.2.6 Seite 251 -->
  </named-query>
  <named-native-query>
    <!-- Abschnitt 10.2.7 Seite 251 -->
  </named-native-query>
  <sql-result-set-mapping>
    <!-- Abschnitt 10.2.8 Seite 252 -->
  </sql-result-set-mapping>
  <mapped-superclass>
    <!-- Abschnitt 10.2.33 Seite 270 -->
  </mapped-superclass>
  <entity>
    <!-- Abschnitt 10.2.9 Seite 252 -->
  </entity>
  <embeddable>
    <!-- Abschnitt 10.2.34 Seite 270 -->
  </embeddable>
</entity-mappings>
```

Das Element `package` kann den Namen eines Java-Packages enthalten. Dieses Package wird allen Klassen ohne qualifizierenden Namen vorangestellt. Die Konfigurationen von `package`, `schema`, `catalog` und `access` besitzen ihre Gültigkeit für alle Elemente in derselben Mapping-Datei. Konfigurationen für die gesamte Persistenzeinheit werden im Sub-Element `<persistence-unit-metadata>` vorgenommen.

10.2.2 Das Element `<persistence-unit-metadata>`

Das Element `<persistent-unit-metadata>` definiert eine Reihe von Meta-Daten, die für die gesamte Persistenzeinheit gelten. Wird das Sub-Element `<xml-mapping-metadata-complete>` angegeben, ist das komplette Mapping in den XML-Dateien enthalten. Annotationen in den Klassen werden dann vollständig ignoriert.

```
<persistence-unit-metadata>
  <description>...</description>
  <xml-mapping-metadata-complete />
  <persistence-unit-defaults>
    <description>...</description>
    <schema>...</schema>
    <catalog>...</catalog>
    <delimited-identifiers/>
    <access>
      <!-- Abschnitt 10.2.3 Seite 249 -->
    </access>
    <cascade-persist/>

    <entity-listeners>
      <entity-listener>
        <!-- Abschnitt 10.2.35 Seite 271 -->
      </entity-listener>
    </entity-listeners>
  </persistence-unit-defaults>
</persistence-unit-metadata>
```

Die Elemente `catalog` und `schema` konfigurieren für die gesamte Persistenzeinheit die optionalen Namen für den Datenbankkatalog sowie das Datenbankschema. Sie lassen sich durch direkte Angabe im jeweiligen Element überschreiben.

Allen Beziehungen, die zwischen Entities existieren, kann, ohne dass dies jeweils einzeln anzugeben ist, die Kaskadierungsart `PERSIST` der Enum `CascadeType` zugewiesen werden. Diese Einstellung führt das Element `<cascade-persist/>` durch. Ein Überschreiben dieser Einstellung je Assoziation lässt die Spezifikation nicht zu, ist aber für eine Umsetzung in späteren Versionen der JPA vorgesehen. Ohne die Angabe lassen sich die Assoziationen auch individuell konfigurieren.

Im SQL-Standard 92 (SQL-92) wurde erstmals der Begriff des *Delimited Identifiers* eingeführt. Damit waren z.B. Tabellennamen mit Leerzeichen möglich. Unterschieden wird dann auch nach Groß- und Kleinschreibung, sofern die Objektnamen in Anführungszeichen stehen. Das Element `<delimited-identifiers>` aktiviert dieses Verhalten für die Persistenzeinheit. Es kann nicht überschrieben werden.

10.2.3 Das Element <access>

Das Element <access> konfiguriert die Art und Weise, wie die Implementierung den Zugriff auf die Werte des Instanz-Objekts durchführen soll. Ist PROPERTY eingestellt, erfolgt der Zugriff über die Getter- und Setter-Methoden. Mit der Angabe FIELD wird direkt auf die Instanzvariablen zugegriffen.

```
<access>
  PROPERTY <!-- PROPERTY|FIELD -->
</access>
```

Das Element hat Gültigkeit für alle Entities der Persistenzeinheit bei denen **keine** Annotation @Access vorhanden ist. Das Element kann überschrieben werden. In der JPA-Version 1.0 gab es noch keine korrespondierende Annotation für dieses Element, in der aktuellen Version kann es an der Klasse, Methode oder am Attribut verwendet werden:

```
public @interface Access {
  AccessType value();
}
```

10.2.4 Das Element <sequence-generator>

Mit dem Element <sequence-generator> werden Primärschlüsselgeneratoren konfiguriert, die datenbankseitig eindeutige Ids für Entity-Instanzen erzeugen. Das Attribut name ist erforderlich, alle anderen sind optional. Referenziert wird der Generator im Element <generated-value>. Dieses Element benutzt einen Generator über dessen eindeutigen Namen.

```
<sequence-generator name="name" allocation-size="" catalog=""
  initial-value="1" schema="db-schema"
  sequence-name="db-sequence-name" />
```

Die äquivalente Annotation hat das folgende Aussehen:

```
public @interface SequenceGenerator {
  String name();
  String sequenceName() default "";
  String catalog() default "";
  String schema() default "";
  int initialValue() default (int) 1;
  int allocationSize() default (int) 50;
}
```

Der Name eines Sequence-Generators ist über die gesamte Persistenzeinheit gültig. Die Verwendung des gleichen Namens in mehreren Mapping-Dateien ist nicht erlaubt. Das XML-Element überschreibt bei Namensgleichheit einen Generator, der mit Hilfe der Annotation definiert wurde. Das Attribut initialValue gibt an, bei welchem Wert der Generator zu zählen beginnt. Das Attribut allocationSize gibt die Schrittweite je generierter Id an. Ist die Zahl größer als 1, muss die Implementierung nicht bei jedem Insert die Sequence nach einem neuen Wert fragen, sondern legt sich beim ersten Abfragen einen Vorrat von Ids an. Die Verwendung dieses Generators ist nur bei Datenbanken sinnvoll, die Sequences unter-

stützen. Ist dies nicht der Fall, kann auf andere Generatoren, z. B. den Table-Generator, ausgewichen werden.

10.2.5 Das Element `<table-generator>`

Mit Hilfe eines Table-Generators werden eindeutige Schlüssel für ein Entity erzeugt, die nicht auf einer Datenbank-Sequenz beruhen. Die Schlüssel werden in einer Tabelle abgelegt und von der JPA-Implementierung direkt verwaltet.

```
<table-generator allocation-size="" catalog="" initial-value=""
  name="" pk-column-name="" pk-column-value="" schema="" table=""
  value-column-name="">
  <unique-constraint>
    <!-- Abschnitt 10.2.11 Seite 255 -->
  </unique-constraint>
</table-generator>

public @interface TableGenerator {
  String name();
  int allocationSize() default 50;
  String catalog() default "";
  int initialValue() default 0;
  String pkColumnName() default "";
  String pkColumnValue() default "";
  String schema() default "";
  String table() default "";
  UniqueConstraint[] uniqueConstraints() default {};
  String valueColumnName() default "";
}
```

Bis auf das Attribut `name` sind alle Attribute optional. Sie werden, soweit nicht vorhanden, entsprechend sinnvoll vorbelegt.

Der Name eines Table-Generators sollte über die gesamte Persistenzeinheit eindeutig sein. Ein Table-Generator, der in einer XML-Mapping-Datei definiert ist, überschreibt einen Generator gleichen Namens, der per Annotation konfiguriert wurde.

Jeder Table-Generator legt seine erzeugten Ids in genau einer Tabelle ab. Das Attribut `pkColumnValue` unterscheidet dabei, für welches Entity der Wert registriert wurde. Wird das Attribut weggelassen, ist der Klassenname das Unterscheidungskriterium. Sollen mehrere Entities denselben Generator benutzen, kann dies nur über die Definition in den Mapping-Dateien realisiert werden. JPA lässt es nicht zu, dass z. B. das Entity `Adresse` einen Table-Generator benutzt, der im Entity `Kunde` mittels Annotation definiert ist. Gleichwohl lassen sich Generatoren mit unterschiedlichem Namen per Annotation definieren, die auf eine Tabelle zugreifen. Diese Möglichkeit lässt der Sequence-Generator ebenfalls zu. Abschnitt 2.1.2 zeigt die Verwendung an einem Beispiel.

10.2.6 Das Element <named-query>

Das Element <named-query> definiert eine Abfrage, auf die man sich mit Hilfe des eindeutigen Namens im Quellcode beziehen kann. Damit können eine Reihe von Abfragen an zentraler Stelle definiert werden.

```
<named-query name="query-name">
  <description>...</description>
  <query>...</query>
  <lock-mode>NONE
    <!--  READ|WRITE|OPTIMISTIC|OPTIMISTIC_FORCE_INCREMENT|
          PESSIMISTIC_READ|PESSIMISTIC_WRITE|
          PESSIMISTIC_FORCE_INCREMENT|NONE -->
  </lock-mode>
  <hint name="hint-name" value="hint-value"/>
</named-query>

public @interface NamedQuery {
  String name();
  String query();
  QueryHint[] hints() default {};
  LockModeType lockMode() default LockModeType.NONE
}
```

Der Name einer Abfrage muss in der Persistenzeinheit eindeutig sein. XML-Definitionen überschreiben eventuell vorhandene Annotationen. Die Elemente `name` und `query` müssen immer angegeben werden. Beispiele für die Annotation `@NamedQuery` sind ab Seite 181 zu finden.

10.2.7 Das Element <named-native-query>

Mit dem Element <named-native-query> werden direkte SQL-Abfragen definiert. Angegeben werden müssen das Attribut `name` sowie das Element `query`. Die anderen Elemente und Attribute sind optional.

```
<named-native-query name="native-query-name" result-class=""
  result-set-mapping="" >
    <description>...</description>
    <query>...</query>
    <hint name="hint-name" value="hint-value"/>
</named-native-query>

public @interface NamedNativeQuery {
  String name();
  String query();
  QueryHint[] hints() default {};
  Class resultClass() default void.class;
  String resultSetMapping() default "";
}
```

In XML definierte Abfragen überschreiben auch hier Abfragen aus den Annotationen. Beispiele für entsprechende Abfragen finden sich im Abschnitt 7.1.3.

10.2.8 Das Element <sql-result-set-mapping>

Mit dem Element <sql-result-set-mapping> lässt sich das Ergebnis einer nativen SQL-Abfrage direkt in Java-Objekte abbilden. Neben dem Sub-Element <entity-result>, das definierte Entities referenziert, bezieht sich das Element <column-result> auf ein skalares Ergebnis einer Abfrage ohne Entity-Bezug.

```
<sql-result-set-mapping name="">
  <description>...</description>
  <entity-result discriminator-column="" entity-class="">
    <field-result column="" name=""/>
  </entity-result>
  <column-result name=""/>
</sql-result-set-mapping>

public @interface SqlResultSetMapping {
  String name();
  ColumnResult[] columns() default {};
  EntityResult[] entities() default {};
}

public @interface ColumnResult {
  String name();
}

public @interface EntityResult {
  Class entityClass();
  FieldResult[] fields() default {};
  String discriminatorColumn() default "";
}

public @interface FieldResult {
  String column();
  String name();
}
```

Weil die Sub-Elemente nur im Element <sql-result-set-mapping> verwendet werden, sind sie in vollem Umfang hier aufgeführt. Beispiele zur Verwendung findet der Leser im Abschnitt 7.1.3.

10.2.9 Das Element <entity>

Während das XML-Element <entity> mit seinen möglichen Sub-Elementen sehr groß werden kann, ist die entsprechende Annotation auf das Schlüsselwort `@Entity` und einen optionalen Namen beschränkt. Alle anderen Angaben finden sich an den jeweiligen Attributen oder Getter-Methoden oder sind eigene Annotationen, die an der Klasse konfiguriert werden.

Angaben in der Mapping-Datei ergänzen bzw. überschreiben die im Quellcode verankerten Annotationen. Besitzt das Element <entity> das Attribut `metadata-complete="true"`, werden sämtliche Annotationen in der Java-Klasse ignoriert. Die JPA-Implementierung verwendet dann nur die Definitionen der XML-Datei. Dieses Attribut kann auch verwendet werden, um das für die Persistenzeinheit gültige Attribut <xml-mapping-metadata-

complete> für einzelne Entities auszuschalten. Die globale Aktivierung dieses Flags finden Sie im Element <persistence-unit-metadata>, beschrieben auf Seite 248.

Das Attribut class ist beim <entity>-Element zwingend vorgeschrieben. Mit ihm wird die Verbindung zwischen dem XML-Element und der zu konfigurierenden Klasse hergestellt. Das Attribut name vergibt einen optionalen Namen für das Entity, der z. B. in JPA-QL-Abfragen verwendet werden kann. Er ist wie in der @Entity-Annotation optional und besitzt im Standardfall den unqualifizierten Namen der konfigurierten Klasse.

```
<entity class="" access="PROPERTY" cacheable="true"
  metadata-complete="true" name="">
  <description>...</description>
  <table>
    <!-- Abschnitt 10.2.10 Seite 254 -->
  </table>
  <secondary-table>
    <!-- Abschnitt 10.2.12 Seite 255 -->
  </secondary-table>
  <primary-key-join-column>
    <!-- Abschnitt 10.2.13 Seite 256 -->
  </primary-key-join-column>
  <id-class>
    <!-- Abschnitt 10.2.36 Seite 271 -->
  </id-class>
  <inheritance>
    <!-- Abschnitt 10.2.37 Seite 272 -->
  </inheritance>
  <discriminator-value>...</discriminator-value>
  <discriminator-column>
    <!-- Abschnitt 10.2.14 Seite 256 -->
  </discriminator-column>
  <sequence-generator>
    <!-- Abschnitt 10.2.4 Seite 249 -->
  </sequence-generator>
  <table-generator>
    <!-- Abschnitt 10.2.5 Seite 250 -->
  </table-generator>
  <named-query>
    <!-- Abschnitt 10.2.6 Seite 251 -->
  </named-query>
  <named-native-query>
    <!-- Abschnitt 10.2.7 Seite 251 -->
  </named-native-query>
  <sql-result-set-mapping>
    <!-- Abschnitt 10.2.8 Seite 252 -->
  </sql-result-set-mapping>
  <exclude-default-listeners/>
  <exclude-superclass-listeners/>
  <entity-listeners>
    <entity-listener>
      <!-- Abschnitt 10.2.35 Seite 271 -->
    </entity-listener>
  </entity-listeners>
  <pre-persist method-name="..." />
  <post-persist method-name="..." />
  <pre-remove method-name="..." />
  <post-remove method-name="..." />
  <pre-update method-name="..." />
```

```
  <post-update method-name="..." />
  <post-load method-name="..." />
  <attribute-override>
    <!-- Abschnitt 10.2.15 Seite 257 -->
  </attribute-override>
  <association-override>
    <!-- Abschnitt 10.2.16 Seite 257 -->
  </association-override>
  <attributes>
    <!-- Abschnitt 10.2.17 Seite 258 -->
  </attributes>
</entity>

public @interface Entity {
  String name() default "";
}
```

Die Elemente `<exclude-default-listeners>` und `<exclude-superclass-listeners>` verhindern die Ausführung der entsprechenden Listener der Persistenzeinheit bzw. der Oberklasse. Für das Entity selbst lassen sich eigene Listener definieren. Implementiert das Entity selbst Aktionen für einzelne Ereignisse, sind die Elemente `<pre-persist>`, `<post-persist>` und folgende aus obigem Quellcode zu verwenden. Beispiele dazu findet der Leser ab Seite 79.

Das Element `<discriminator-value>` wird im Rahmen der Vererbung benutzt. Werden sämtliche Entities einer Vererbungshierarchie in einer Tabelle abgelegt, wird der Wert des XML-Elements bzw. seiner äquivalenten Annotation `@DiscriminatorValue` verwendet, um die Klassen in der Hierarchie voneinander zu unterscheiden. Wird auf eine explizite Angabe verzichtet, zieht die JPA-Implementierung automatisch den Namen des Entities zur Unterscheidung heran.

10.2.10 Das Element `<table>`

Das Element `<table>` erlaubt die Konfiguration der Tabelle, in der ein Entity abgelegt werden soll. Alle Attribute sind optional. Im Standardfall entspricht der Name einer Tabelle dem Namen des Entities.

```
<table catalog="" name="" schema="">
  <unique-constraint>
    <!-- Abschnitt 10.2.11 Seite 255 -->
  </unique-constraint>
</table>

public @interface Table {
  String name() default "";
  String catalog() default "";
  String schema() default "";
  UniqueConstraint[] uniqueConstraints() default {};
```

Soll ein Entity auf mehrere Tabellen aufgeteilt werden, muss das Element `<secondary-table>` verwendet werden. Es kann im Gegensatz zum aktuellen Element auch mehrfach in der Konfiguration zu einem Entity auftreten.

Hinweis:

Das Mapping eines Entity-Properties in eine bestimmte Spalte wird nicht innerhalb des `<table>`-Elements vorgenommen. Diese Mapping-Information findet sich im jeweiligen Attribut im Element `<column>`, beschrieben in Abschnitt 10.2.18 auf Seite 259.

10.2.11 Das Element `<unique-constraint>`

Werden auf einer Tabelle zusätzliche Unique-Constraints benötigt, kann dieses Element verwendet werden. Es erlaubt die Angabe mehrerer `<column-name>`-Elemente, die dem Constraint angehören sollen.

```
<unique-constraint name="">
  <column-name>...</column-name>
</unique-constraint>

public @interface UniqueConstraint {
  public String name() default "";
  public String[] columnNames();
}
```

Die Elemente `<table>`, `<secondary-table>`, `<table-generator>` und `<join-table>` lassen beliebig viele Constraint-Definitionen zu. Im Bereich der Annotationen werden mehrere einzelne Definitionen als Array in der Variablen `uniqueConstraints` der entsprechenden Annotation zusammengefasst.

10.2.12 Das Element `<secondary-table>`

Für die Aufteilung eines Entity in zwei oder mehr Tabellen wird das Element `<secondary-table>` benutzt. Der Name der Tabelle kann sich nicht wie beim `<table>`-Element aus dem Namen des Entity ergeben. Das Attribut `name` muss daher zwingend angegeben werden.

```
<secondary-table catalog="" name="" schema="">
  <primary-key-join-colum>
    <!-- Abschnitt 10.2.13 Seite 256 -->
  </primary-key-join-colum>
  <unique-constraint>
    <!-- Abschnitt 10.2.11 Seite 255 -->
  </unique-constraint>
</secondary-table>

public @interface SecondaryTable {
  String name();
  String catalog() default "";
  String schema() default "";
  PrimaryKeyJoinColumn[] pkJoinColumns() default {};
  UniqueConstraint[] uniqueConstraints() default {};
}
```

Die beiden Sub-Elemente können beliebig oft auftreten. In der Annotation werden sie jeweils mit Hilfe von Arrays der Konfiguration hinzugefügt.

Das Element `<secondary-table>` besitzt als einziges mögliches Elternelement `<entity>`. Als Annotation steht für mehrere Tabellen das Element`@SecondaryTables` zur Verfügung.

10.2.13 Das Element `<primary-key-join-column>`

Das XML-Element `<primary-key-join-column>` kann in den folgenden Situationen verwendet werden:

- Wird in einer Vererbungshierarchie die Strategie`JOINED` ausgewählt, wird jede Subklasse, die spezifische Attribute zur Oberklasse besitzt, in einer eigenen Tabelle gespeichert. Die Angabe des Elements erfolgt in XML unterhalb von `<entity>` für die erbenden Klassen.
- Wird ein Entity mit Hilfe von `<secondary-table>` in mehreren Tabellen gespeichert, kann der Join zwischen diesen Tabellen mit `<primary-key-join-column>` konfiguriert werden.
- Das Element kann ebenfalls in einem `<one-to-one>`-Element verwendet werden. Dabei wird die Assoziation zwischen den beteiligten Entities konfiguriert.

```
<primary-key-join-column column-definition="" name=""
  referenced-column-name="" />

public @interface PrimaryKeyJoinColumn {
  String name() default "";
  String referencedColumnName() default "";
  String columnDefinition() default "";
}
```

Das Attribut `name` bezieht sich auf den Namen des Primärschlüssels in der aktuellen Tabelle. Mit `referenced-column-name` wird der Primärschlüssel jener Tabelle angegeben, mit der verbunden wird. `column-definition` erlaubt zusätzliche SQL-Fragmente, die bei Erzeugung der Datenbank-Objekte verwendet werden können. Bei `<one-to-one>`-Beziehungen sollte darauf verzichtet werden.

10.2.14 Das Element `<discriminator-column>`

Werden Klassen einer Klassenhierarchie mit den Strategien`SINGLE_TABLE` komplett oder mit `JOINED` zumindest teilweise in einer Tabelle abgelegt, benötigt JPA ein Kriterium, mit dem sich die einzelnen Zeilen unterscheiden und einer konkreten Klasse innerhalb der Hierarchie zuordnen lassen. Den Wert des Kriteriums legt jede Klasse selbst mit dem Element`<discriminator-value>` fest. Die Spalte, die diesen Wert aufnimmt, wird in der obersten Klasse der Vererbungshierarchie festgelegt bzw. dort, wo sich die Art der Strategie vom Wurzelelement unterscheidet.

```
<discriminator-column column-definition=""
  discriminator-type="STRING" length="31" name=""/>
```

```
public @interface DiscriminatorColumn {
  String name() default "DTYPE";
  DiscriminatorType discriminatorType() default DiscriminatorType.
      STRING;
  String columnDefinition() default "";
  int length() default (int) 31;
}
```

Ist das Element nicht angegeben, obwohl die gewählte Vererbungsstrategie dies verlangt, wird als Spaltenname `DTYPE` gewählt. Der Datentyp ist per `DefaultDiscriminatorType.STRING`. Außerdem können `CHAR` oder `INTEGER` gewählt werden. Bei Auswahl von `STRING` kann mit dem Attribut `length` dessen Länge konfiguriert werden. `<column-definition>` erlaubt bei Erstellung der Tabelle verwendete SQL-Fragmente.

10.2.15 Das Element `<attribute-override>`

Das Element `<attribute-override>` wird benutzt, um Property-Definitionen einer Oberklasse zu überschreiben. Das Element kann ebenfalls benutzt werden, um die Konfiguration von Properties einer eingebetteten Klasse zu ändern.

Sowohl das Attribut `name` als auch das Sub-Element `<column>` müssen angegeben werden.

```
<attribute-override name="">
  <description>...</description>
  <column>
    <!-- Abschnitt 10.2.18 Seite 259 -->
  </column>
</attribute-override>

public @interface AttributeOverride {
  String name();
  Column column();
}
```

Der Name bezieht sich auf das Feld der Oberklasse bzw. der eingebetteten Klasse, deren Spaltendefinition geändert werden soll.

10.2.16 Das Element `<association-override>`

Das Element `<association-override>` hat die gleiche Aufgabe wie das vorherige Element, es bezieht sich allerdings auf Assoziationen einer Oberklasse, die mit `<mapped-superclass>` konfiguriert ist. Es können `ManyToOne`- und `OneToOne`-Beziehungen überschrieben werden.

```
<association-override name="">
  <description>...</description>
  <join-table>
    <!-- Abschnitt 10.2.32 Seite 269 -->
  </join-table>
  <join-column>
    <!-- Abschnitt 10.2.25 Seite 264 -->
  </join-column>
```

```
</association-override>

public @interface AssociationOverride {
  String name();
  JoinColumn[] joinColumns() default {};
  JoinTable joinTable() default @JoinTable;
}
```

Das Attribut name muss angegeben werden. Das Element muss mindestens ein <join-column>-Element enthalten oder aber das an dieser Stelle seit Version 2.0 erlaubte Element <join-table>.

10.2.17 Das Element <attributes>

Das Element <attributes> besitzt keine äquivalente Annotation. Die den Sub-Elementen entsprechenden Annotationen werden direkt an den Attributen bzw. Methoden konfiguriert. Deshalb besitzen die Annotationen kein name-Element, während das name-Attribut bei den XML-Annotationen obligatorisch sein muss, um das Entity-Property richtig zu referenzieren.

Die Elemente <id> oder <embedded-id> kennzeichnen das identifizierende Attribut des Entity. Zur Identifizierung benutzt werden kann nur eine der beiden Alternativen.

```
<attributes>
  <description>...</description>
  <element-collection>
    <!-- Abschnitt 10.2.19 Seite 259 -->
  </element-collection>
  <!-- Achtung Auswahl: Entweder Element <id> oder <embedded-id>, pro
    Entity ist nur eine Angabe erlaubt -->
  <embedded-id>
    <!-- Abschnitt 10.2.22 Seite 262 -->
  </embedded-id>
  <id>
    <!-- Abschnitt 10.2.21 Seite 261 -->
  </id>
  <basic>
    <!-- Abschnitt 10.2.23 Seite 262 -->
  </basic>
  <version>
    <!-- Abschnitt 10.2.24 Seite 263 -->
  </version>
  <many-to-one>
    <!-- Abschnitt 10.2.28 Seite 266 -->
  </many-to-one>
  <one-to-many>
    <!-- Abschnitt 10.2.28 Seite 266 -->
  </one-to-many>
  <one-to-one>
    <!-- Abschnitt 10.2.27 Seite 265 -->
  </one-to-one>
  <many-to-many>
    <!-- Abschnitt 10.2.29 Seite 267 -->
  </many-to-many>
  <embedded>
    <!-- Abschnitt 10.2.31 Seite 269 -->
```

```
    </embedded>
    <transient name="" />
    <version>
      <!-- Abschnitt 10.2.24 Seite 263 -->
    </version>
  </attributes>
```

Das Element `<attributes>` kann nur Sub-Element von `<entity>` oder `<mapped-superclass>` sein. Das Element `transient` kennzeichnet ein Property, dass nicht zu persistieren ist.

10.2.18 Das Element `<column>`

Das Element `<column>` konfiguriert eine Tabellenspalte für ein Attribut eines Entity. Alle Attribute sind optional. Der Name der Spalte wird, wenn nicht angegeben, aus dem jeweiligen Property-Namen des Entity abgeleitet.

```
    <column column-definition="" insertable="true" length="0"
      name="" nullable="true" precision="0" scale="0" table=""
      unique="true" updatable="true"  />

  public @interface Column {
    String name() default "";
    boolean unique() default false;
    boolean nullable() default true;
    boolean insertable() default true;
    boolean updatable() default true;
    String columnDefinition() default "";
    String table() default "";
    int length() default (int) 255;
    int precision() default (int) 0;
    int scale() default (int) 0;
  }
```

Das Attribut `unique` generiert einen Unique-Constraint auf der Spalte. Er kann zusätzlich zu anderen Bedingungen angelegt werden, die bereits im Zuge des Primärschlüssels oder auf Ebene der Tabelle definiert sind.

Die Attribute `insertable` und `updatable` schließen die betroffene Spalte in einem Insert- bzw. Update-Statement aus, wenn sie als `false` definiert sind.

10.2.19 Das Element `<element-collection>`

Seit JPA 2.0 ist der Entwickler in der Lage, Elemente einfacher Datentypen in Collections mit dem Element `<element-collection>` in einem Property des Entity zu referenzieren. Aufgrund der Komplexität dieses XML-Elements verzichten wir an dieser Stelle auf die Darstellung des XML-Quellcodes. Abbildung 10.1 stellt den Schema-Auszug mit den möglichen Sub-Elementen grafisch dar.

```
  public @interface ElementCollection  {
    Class targetClass() default void.class;
    FetchType fetch() default FetchType.LAZY;
  }
```

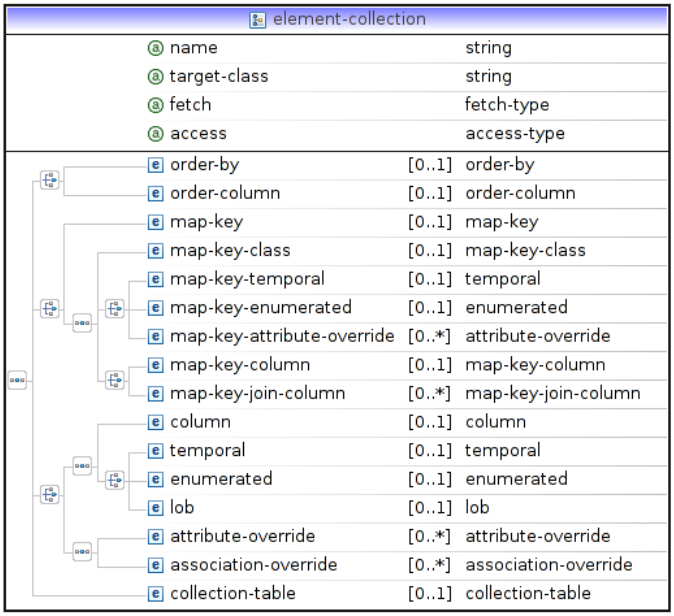


ABBILDUNG 10.1 `<element-collection>`

Die eingebetteten Elemente des referenzierten Objektes (kein Entity) sind nicht in der Tabelle des referenzierenden Entity enthalten. Daher verhält sich die `<element-collection>` fast wie eine One-To-Many-Beziehung. Der entscheidende Unterschied ist aber: Das Ziel-Objekt ist kein mit `@Entity` annotierter Typ. Beispiele dazu findet der Leser in Abschnitt 4.3.

10.2.20 Das Element `<collection-table>`

Die Tabelle mit einfachen Datentypen bzw. Entity-Typen, die in einer anderen Klasse als Collection eingebettet werden, wird mit dem Element `<collection-table>` definiert.

```
<collection-table catalog="" name="" schema="">
  <join-column>
    <!-- Abschnitt 10.2.25 Seite 264 -->
  </join-column>
  <unique-constraint>
    <!-- Abschnitt 10.2.11 Seite 255 -->
  </unique-constraint>
</collection-table>

public @interface CollectionTable {
  String name() default "";
  String catalog() default "";
  String schema() default "";
  JoinColumn[] joinColumns() default {};
  UniqueConstraint[] uniqueConstraints() default {};
}
```

Wird der Tabellenname nicht angegeben, nimmt die Spezifikation den Entity-Namen mit Unterstrich, gefolgt vom Property-Namen als Default-Einstellung, an. Ohne Angabe einer `<join-column>` muss jede Implementierung die Referenz zum Entity über die Fremdschlüsselspalte suchen. Der Name der Spalte setzt sich dann aus dem Entity-Namen mit Unterstrich und dem Primärschlüssel-Namen zusammen. Diese sinnvollen Annahmen ermöglichen es dem Anwender, beim Aufbau eines neuen Datenbank-Schemas eine Vielzahl optionaler Angaben wegzulassen. Die Verwendung eines bestehenden Datenbank-Schemas ist mit Ausprägung der einzelnen Attribute ebenfalls möglich.

10.2.21 Das Element `<id>`

Das Element `<id>` kennzeichnet das Property eines Entity, das den Primärschlüssel darstellt. Das Attribut `name` muss angegeben sein.

```
<id name="" access="PROPERTY" >
  <column>
    <!-- Abschnitt 10.2.18 Seite 259 -->
  </column>
  <generated-value generator="" strategy="TABLE" />
  <temporal>DATE</temporal>
  <table-generator>
    <!-- Abschnitt 10.2.5 Seite 250 -->
  </table-generator>
  <sequence-generator>
    <!-- Abschnitt 10.2.4 Seite 249 -->
  </sequence-generator>
</id>

public @interface Id {
}

public @interface GeneratedValue {
  GenerationType strategy() default GenerationType.AUTO;
  String generator() default "";
}

public @interface Temporal {
  TemporalType value();
}
```

Das Element `<generated-value>` konfiguriert zum einen die Strategie der Primärschlüsselerzeugung und zum anderen den zu verwendenden Generator. Zur Auswahl stehen folgende Strategien:

- `AUTO`: die JPA-Implementierung wählt eine geeignete Methode zur Erzeugung der Primärschlüssel selbst aus,
- `IDENTITY`: verwendet eine Schlüsselspalte, die bei bestimmten Datenbanksystemen selbstinkrementierend ist,
- `SEQUENCE`: verwendet eine Sequence in der Datenbank,
- `TABLE`: benutzt eine eigene Tabelle, die den zuletzt generierten Schlüssel beinhaltet.

Das Element `<temporal>` gibt an, dass die Schlüsselspalte ein temporaler Datentyp ist. An dieser Stelle sind `DATE`, `TIME` oder `TIMESTAMP` möglich.

10.2.22 Das Element <embedded-id>

<embedded-id> weist auf eine Instanzvariable, deren Datentyp einen oder mehrere Primärschlüssel in einer separaten Klasse kapselt. Im Ergebnis findet sich für das Entity ein zusammengesetzter Primärschlüssel in der Datenbank.

```
<embedded-id name="" access="PROPERTY">
  <attribute-override>
    <!-- Abschnitt 10.2.15 Seite 257 -->
  </attribute-override>
</embedded-id>

public @interface EmbeddedId {
}
```

Das Element <attribute-override> erlaubt es, einzelne Attribute aus der eingebetteten Klasse zu überschreiben und das Mapping für sie dementsprechend anzupassen.

10.2.23 Das Element <basic>

Das Element <basic> wird zur Konfiguration des Mappings für die folgenden Typen verwendet:

- primitive Datentypen und ihre Wrapper-Klassen
- String
- java.math.BigInteger
- java.math.BigDecimal
- java.util.Date
- java.util.Calendar
- java.sql.Date
- java.sql.Time
- java.sql.Timestamp
- byte[],Byte[]
- char[],Character[]
- Enumerationen

Auf das Element bzw. die Annotation kann bei Properties mit diesen Typen auch verzichtet werden. Sie werden dann automatisch abgebildet, falls sie nicht als transient gekennzeichnet sind. Wird das XML-Element verwendet, muss das Attribut name angegeben werden. Mit optional="false" können Sie null-Werte im Attribut verhindern.

```
<basic fetch="EAGER" name="" optional="true" access="PROPERTY">
 <column>
   <!-- Abschnitt 10.2.18 Seite 259 -->
 </column>
 <!-- Achtung Auswahl: <lob/>, <temporal>...</temporal> oder
 <enumerated>...</enumerated -->
 <lob />
 <!-- ORDINAL|STRING -->
```

```
  <enumerated>ORDINAL</enumerated>
  <!-- DATE|TIME|TIMESTAMP -->
  <temporal>DATE</temporal>
</basic>

public @interface Basic {
  FetchType fetch() default FetchType.EAGER;
  boolean optional() default true;
}

public @interface Lob {
}

public @interface Enumerated {
  EnumType value() default EnumType.ORDINAL;
}

public enum EnumType {
  ORDINAL,
  STRING
}

public @interface Temporal {
  TemporalType value();
}

public enum TemporalType {
  DATE, // - java.sql.Date
  TIME, // - java.sql.Time
  TIMESTAMP // - java.sql.Timestamp
}
```

Das Element `Lob` wird für so genannte „Large Objects“ verwendet. Das Element `<temporal>` erlaubt die Werte `DATE`, `TIME` oder `TIMESTAMP`, analog zur Annotation `@Temporal`. Mit `<enumerated>` kann entweder der Wert `STRING` oder der Wert `ORDINAL` angegeben werden, um eine Aufzählung in der Datenbank abzulegen. Im Default-Fall (`ORDINAL`) werden die Enum-Elemente auf Integer-Werte abgebildet.

Mit dem Attribut `fetch` wird die Art und Weise definiert, wie die Daten aus der Datenbank gelesen werden. Mit `EAGER` wird das konfigurierte Property sofort geladen. `LAZY` weist die JPA-Implementierung an, die Daten erst zu laden, wenn darauf zugegriffen wird.

10.2.24 Das Element `<version>`

Das Element `<version>` kennzeichnet das Feld, das die Werte für das optimistische Locking aufnehmen soll. Damit kann die JPA-Implementierung feststellen, ob ein anderer Prozess in der Zwischenzeit Änderungen auf dem betrachteten Datensatz durchgeführt hat.

```
<version name="" access="PROPERTY" >
  <column>
    <!-- Abschnitt 10.2.18 Seite 259 -->
  </column>
  <temporal>DATE</temporal><!-- oder TIME oder TIMESTAMP -->
</version>
```

```
public @interface Version {
}
```

Obwohl die Schemadatei mehrere `version`-Elemente pro Entity zulässt, empfiehlt die Spezifikation nur eine Angabe je Entity. Andernfalls ist die Applikation nicht portabel, und es kann beim Wechsel der JPA-Implementierung zu einem Mehraufwand kommen. Des Weiteren sollte das `version`-Attribut immer in der Haupttabelle des Entity abgelegt sein. Hält sich eine Anwendung nicht daran, kann es auch zu Problemen mit einer anderen JPA-Implementierung kommen. Eine JPA-Anwendung sollte das zugrunde liegende Property nicht selbst verändern. Dies obliegt im Rahmen des Locking der JPA-Implementierung selbst. Eine Ausnahme davon können Batch-Updates sein. Diese Massenänderungen verändern Entities im aktiven Persistenzkontext nicht, eine Anpassung des `version`-Attributes muss in diesem Fall durchgeführt werden. Anmerkungen dazu findet der Leser in Abschnitt 7.6.

10.2.25 Das Element `<join-column>`

Das Element `<join-column>` konfiguriert die Spalte, die den eigentlichen Join zwischen den an einer Assoziation beteiligten Entities herstellt.

```
<join-column column-definition="" insertable="true"
  name="" nullable="true" referenced-column-name="" table=""
  unique="true" updatable="true" />

public @interface JoinColumn {
  String name() default "";
  String referencedColumnName() default "";
  boolean unique() default false;
  boolean nullable() default true;
  boolean insertable() default true;
  boolean updatable() default true;
  String columnDefinition() default "";
  String table() default "";
}
```

10.2.26 Das Element `<cascade>`

Das Element `<cascade>` gibt an, welche Operationen an die Entities der anderen Seite einer Assoziation weitergegeben werden sollen. Ist `ALL` angegeben, werden alle durchzuführenden Operationen kaskadiert.

```
<cascade>
  <cascade-all />
  <cascade-detach /> <!-- neu in JPA 2.0 -->
  <cascade-merge />
  <cascade-persist />
  <cascade-refresh />
  <cascade-remove />
</cascade>
```

```
public enum CascadeType {
  ALL, PERSIST, MERGE, REMOVE, REFRESH, DETACH
};
```

CascadeType ist keine Annotation, sondern eine Enumeration. Ein oder mehrere Elemente dieser Aufzählung können in den cascade-Parametern der Annotationen @ManyToOne, @ManyToMany, @OneToOne und @OneToMany verwendet werden.

10.2.27 Das Element <one-to-one>

Mit dem <one-to-one>-Element wird eine Beziehung zu einem anderen Entity gekennzeichnet. Diese Beziehung ist einwertig, d.h. es kann an beiden Seiten maximal ein Entity beteiligt sein. Für das Verständnis der Konfiguration dieser und der nachfolgenden Beziehungsarten ist es hilfreich, jeweils den Eigentümer der Beziehung zu identifizieren. Bei einer unidirektionalen Beziehung ist der Eigentümer einer Beziehung eindeutig zu erkennen. Es ist die Seite, die eine alleinige Referenz zum Gegenüber besitzt. Bei einer bidirektionalen Beziehung enthält der Nicht-Eigentümer der Beziehung das Attribut <mapped-by>. Das Attribut beinhaltet den Namen der Instanzvariablen der Beziehung auf der Eigentümerseite.

Das Attribut target-entity ist bei einer 1:1-Beziehung in den meisten Fällen optional, da der Typ des referenzierten Entity über den Typ der Instanz-Variablen ermittelt werden kann.

```
<one-to-one fetch="LAZY" mapped-by="" name="" optional="true"
  access="PROPERTY" target-entity="" id="true" maps-id=""
  orphan-removal="true">
  <!-- Achtung Auswahl: Entweder Element <join-column>, <
      primary-key-join-column>
    oder <join-table> -->
  <join-column>
    <!-- Abschnitt 10.2.25 Seite 264 -->
  </join-column>
  <primary-key-join-column>
    <!-- Abschnitt 10.2.13 Seite 256 -->
  </primary-key-join-column>
  <join-table>
    <!-- Abschnitt 10.2.32 Seite 269 -->
  </join-table>
  <cascade>
    <!-- Abschnitt 10.2.26 Seite 264 -->
  </cascade>
</one-to-one>

public @interface OneToOne {
  Class targetEntity() default void.class;
  CascadeType[] cascade() default {};
  FetchType fetch() default FetchType.EAGER;
  boolean optional() default true;
  String mappedBy() default "";
  boolean orphanRemoval() default false;
}
```

Eine 1:1-Beziehung wird, solange ein Entity nicht komplett in die Tabelle des besitzenden Entity integriert wird, mit zwei Tabellen und einer Fremdschlüsselbeziehung realisiert. Ohne weitere Konfiguration gelangen die Entities in eine eigene Tabelle. Die Tabelle der besitzen-

den Entities erhält außerdem eine Fremdschlüsselspalte auf die andere Tabelle. Der Join lässt sich über das optionale Element `<join-column>` noch wesentlich detaillierter konfigurieren. Die genaue Erläuterung der Elemente findet der Leser im Abschnitt 4.1.

10.2.28 Die Elemente `<many-to-one>` und `<one-to-many>`

In einer 1:n-Beziehung muss die `Many`-Seite der Eigentümer der Beziehung sein. Das Attribut `mapped-by` ist nur auf der `One`-Seite zugelassen. Dazu ein Beispiel:

- Entity X referenziert eine Instanz von Entity Y.
- Y referenziert mehrere Instanzen von X, die Referenzen auf X werden in einer `Set` gehalten.

Dies ist eine bidirektionale Beziehung. Entity X ist der Eigentümer der Beziehung. Entity X wird in Tabelle X abgebildet, Entity Y in Tabelle Y. Da X der Eigentümer ist, bekommt Tabelle X eine Foreign-Key-Spalte, mit der die Einträge in Tabelle Y referenziert werden. Damit die JPA-Implementierung für diese Beziehung nur zwei Tabellen verwendet, ist die Angabe des Attributs `mapped-by` erforderlich.

Bei einer unidirektionalen Many-To-One-Beziehung verwendet die JPA-Spezifikation automatisch nur zwei Tabellen. Die Angabe von `mapped-by` unterbleibt, da sie nur auf der `One`-Seite zugelassen ist. Würde die Angabe erfolgen, wäre die Beziehung bidirektional.

Bei einer unidirektionalen One-To-Many-Beziehung werden drei Tabellen verwendet.

```
<many-to-one access="PROPERTY" fetch="LAZY" name=""
  id="true" maps-id="" optional="true" target-entity="">
  <!-- Achtung Auswahl: Entweder Element <join-column> oder <
      join-table> -->
  <join-column>
    <!-- Abschnitt 10.2.25 Seite 264 -->
  </join-column>
  <join-table>
    <!-- Abschnitt 10.2.32 Seite 269 -->
  </join-table>
  <cascade>
    <!-- Abschnitt 10.2.26 Seite 264 -->
  </cascade>
</many-to-one>
```

Weil das `<one-to-many>`-Element eine Reihe von Sub-Elementen enthält, die sich gegenseitig ausschließen, verzichten wir auf eine Darstellung des XML-Codes, da dieser syntaktisch nicht korrekt wäre. Wir benutzen zur Darstellung des Elements Abbildung 10.2 unseres Schema-Editors.

```
public @interface ManyToOne {
  Class targetEntity() default void.class;
  CascadeType[] cascade() default {};
  FetchType fetch() default FetchType.EAGER;
  boolean optional() default true;
}

public @interface OneToMany {
  Class targetEntity() default void.class;
  CascadeType[] cascade() default {};
```

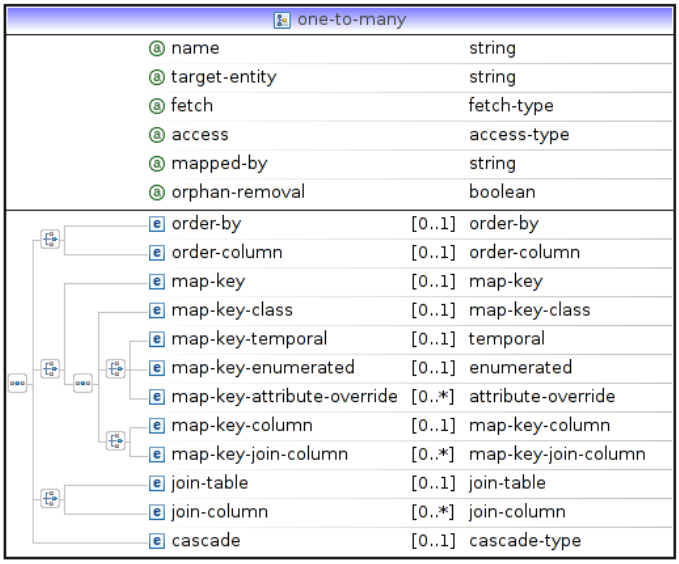


ABBILDUNG 10.2 `<one-to-many>`-Element

```
    FetchType fetch() default FetchType.LAZY;
    String mappedBy() default "";
  }
```

Das Attribut `optional` kennzeichnet, ob die Beziehung vorhanden sein muss. Ist sie als `false` konfiguriert, darf das Attribut des Entity nicht `null` sein.

Das Entity, bei dem die `<one-to-many>`-Beziehung konfiguriert ist, muss mit dem Attribut `targetEntity` die Klasse kennzeichnen, die auf der anderen Seite der Beziehung steht. Ist die mengenwertige Seite mit einem generischen Typ spezifiziert, kann darauf verzichtet werden. Auf der `Many`-Seite kann in der Regel immer auf die Angabe verzichtet werden, da der Typ mit Hilfe der Reflection-API vom JPA leicht zu ermitteln ist.

10.2.29 Das Element `<many-to-many>`

Eine Beziehung zwischen Entities, in der auf beiden Seiten mehrere Instanzen einer Klasse teilnehmen können, wird mit dem Element `<many-to-many>` konfiguriert. Bei einer bidirektionalen Beziehung bleibt es dem Entwickler überlassen, welche Seite der Eigentümer der Beziehung sein soll. Das Attribut `mapped-by` muss dann angegeben werden. In einer unidirektionalen Beziehung ist automatisch das Entity Eigentümer, welches die `many-to-many`-Deklaration besitzt, genauer gesagt, das Entity-Property, auf das sich das XML-Element mit dem Attribut `name` bezieht.

Das Attribut `target-entity` muss dann angegeben werden, wenn bei der Definition der Endpunkte keine generischen Typen verwendet werden.

```
<many-to-many access="PROPERTY" fetch="LAZY"
  mapped-by="" name="" target-entity="">
```

```
  <!-- order-by oder order-column -->
  <order-by>
    <!-- Abschnitt 10.2.30 Seite 268 -->
  </order-by>
  <!-- map-key, map-key-class, map-key-temporal, map-key-enumerated,
       map-key-attribute-override, map-key-column
           map-key-join-column -->
  <map-key>
    <!-- Abschnitt 10.2.39 Seite 272 -->
  </map-key>
  <join-table>
    <!-- Abschnitt 10.2.32 Seite 269 -->
  </join-table>
  <cascade>
    <!-- Abschnitt 10.2.26 Seite 264 -->
  </cascade>
</many-to-many>

public @interface ManyToMany {
  Class targetEntity() default void.class;
  CascadeType[] cascade() default {};
  FetchType fetch() default FetchType.LAZY;
  String mappedBy() default "";
}
```

Diese Form der Beziehung wird in drei Tabellen abgelegt. Die Datenbankbeziehungen werden mit dem Element `<join-table>` konfiguriert.

10.2.30 Das Element `<order-by>`

Das Element `<order-by>` konfiguriert die Reihenfolge der durch eine Beziehung referenzierten Entities. Ist das Element bzw. die Annotation nicht angegeben, wird der Primärschlüssel in aufsteigender Reihenfolge zur Sortierung herangezogen. Im Element werden die Property-Namen des referenzierten Entity durch Komma getrennt angegeben. Zusätzlich können die Schlüsselwörter `ASC` für eine aufsteigende bzw. `DESC` für eine absteigende Sortierung benutzt werden.

```
<order-by>
   property1, property2 ASC|DESC
</order-by>

public @interface OrderBy {
  String value() default "";
}

public @interface OrderBy {
  String value() default "";
}
```

Das `<order-by>`-Element kann auch im Element `<element-collection>` verwendet werden. Die auf- oder absteigende Sortierung erfolgt dann anhand der einzelnen Werte der Collection, wenn es sich um einfache Java-Typen handelt. Die Angabe der Properties wird dabei ignoriert.

10.2.31 Das Element <embedded>

Das Element <embedded> verweist mit dem XML-Attribut name auf ein Property, dessen Typ eine Klasse ist, die als Embeddable (vgl. Kapitel 10.2.34 auf Seite 270) konfiguriert ist. Instanzen dieser eingebetteten Klasse benutzen automatisch das identifizierende Attribut des besitzenden Entity mit. Die Properties werden in das besitzende Entity „eingebettet“, das Mapping dieser Eigenschaften kann jedoch mit dem Element <attribute-override> überschrieben werden.

```
<embedded access="PROPERTY" name="">
  <attribute-override>
    <!-- Abschnitt 10.2.15 Seite 257 -->
  </attribute-override>
  <association-override>
    <!-- Abschnitt 10.2.16 Seite 257 -->
  </association-override>
</embedded>

public @interface Embedded {
}
```

10.2.32 Das Element <join-table>

Das Mapping von Assoziationen lässt sich mit dem Element <join-table> beeinflussen.

```
<join-table catalog="" name="" schema="">
  <join-column>
    <!-- Abschnitt 10.2.25 Seite 264 -->
  </join-column>
  <inverse-join-column>
    <!-- Abschnitt 10.2.25 Seite 264 -->
  </inverse-join-column>
  <unique-constraint>
    <!-- Abschnitt 10.2.11 Seite 255 -->
  </unique-constraint>
</join-table>

public @interface JoinTable {
  String name() default "";
  String catalog() default "";
  String schema() default "";
  JoinColumn[] joinColumns() default {};
  JoinColumn[] inverseJoinColumns() default {};
  UniqueConstraint[] uniqueConstraints() default {};
}
```

Fehlt die Angabe des Tabellennamens, wird dieser von JPA automatisch aus den Namen der beiden beteiligten Entities zusammengesetzt. Der Eigentümer der Beziehung wird zuerst genannt, die beiden Namen durch einen Unterstrich verbunden. Mit den Elementen <join-column> und <inverse-join-column> lassen sich die Join-Bedingungen zwischen den Tabellen entsprechend konfigurieren. Letzteres Element konfiguriert die Fremdschlüsselspalten, die auf die Tabelle des Entity zeigen, das nicht Eigentümer der Beziehung

ist. Das Element `<join-column>` konfiguriert die Spalten, die die Tabelle des Eigentümer-Entity referenzieren.

10.2.33 Das Element `<mapped-superclass>`

Eine mit dem Element `<mapped-superclass>` konfigurierte Klasse besitzt im Gegensatz zum Element `<entity>` keine eigenen Tabellen. Sämtliche Attribute werden an die Sub-Klassen vererbt. Entsprechende Konfigurationen finden ihre Auswirkungen erst in den Tabellen der erbenden Klassen.

```
<mapped-superclass access="PROPERTY" class=""
  metadata-complete="true">
  <description>...</description>
  <id-class>
    <!-- Abschnitt 10.2.36 Seite 271 -->
  </id-class>
  <exclude-default-listeners />
  <exclude-superclass-listeners />
  <entity-listeners>
    <entity-listener>
      <!-- Abschnitt 10.2.35 Seite 271 -->
    </entity-listener>
  </entity-listeners>
  <pre-persist method-name="" />
  <post-persist method-name="" />
  <pre-remove method-name="" />
  <post-remove method-name="" />
  <pre-update method-name="" />
  <post-update method-name="" />
  <post-load method-name="" />
  <attributes>
    <!-- Abschnitt 10.2.17 Seite 258 -->
  </attributes>
</mapped-superclass>

public @interface MappedSuperclass {}
```

Mit den Elementen `<association-override>` und `<attribute-override>` können die Sub-Klassen das Verhalten entsprechend anpassen, genauer gesagt: das Standardverhalten der Oberklasse überschreiben.

10.2.34 Das Element `<embeddable>`

Mit Hilfe des Elements `<embeddable>` werden Instanzen einer Klasse in anderen Klassen eingebettet. Sie benutzen deren Primärschlüssel zur Identifizierung mit.

```
<embeddable access="PROPERTY" class="" metadata-complete="true">
  <description>...</description>
  <attributes>
    <basic>
      <!-- Abschnitt 10.2.23 Seite 262 -->
    </basic>
```

```
    <transient name="" />
  </attributes>
</embeddable>

public @interface Embeddable {
}
```

10.2.35 Das Element <entity-listener>

Das Element <entity-listener> definiert eine Listener-Klasse, die auf bestimmte Persistenzereignisse reagieren soll. Bei Eintritt des Ereignisses wird die im jeweiligen Sub-Element genannte Methode ausgeführt. Die einzelnen Sub-Elemente sind optional und können je Listener-Element maximal einmal auftreten.

```
<entity-listener class="">
  <description>...</description>
  <post-load method-name="" />
  <post-persist method-name="" />
  <post-remove method-name="" />
  <post-update method-name="" />
  <pre-persist method-name="" />
  <pre-remove method-name="" />
  <pre-update method-name="" />
</entity-listener>
```

Analog zum Element <entity-listeners> existiert die Annotation @EntityListeners. Mit dieser Annotation können mehrere Listener-Klassen in einem Array definiert werden. Die in Frage kommenden Methoden werden analog zu obigen Elementnamen mit folgenden Annotationen gekennzeichnet:

- @PrePersist
- @PostPersist
- @PreRemove
- @PostRemove
- @PreUpdate
- @PostUpdate
- @PostLoad

10.2.36 Das Element <id-class>

Soll ein Entity einen zusammengesetzten Primärschlüssel benutzen, kann das Element <id-class> auf die entsprechende Schlüsselklasse verweisen. Die Namen und Datentypen der Attribute in der Entity-Klasse sowie der Id-Klasse müssen übereinstimmen.

```
<id-class class=""/>

public @interface IdClass {
  Class value();
}
```

10.2.37 Das Element <inheritance>

Soll eine Vererbungshierarchie in einer relationalen Datenbank abgebildet werden, bieten sich die folgenden Möglichkeiten an:

- Mit der Einstellung SINGLE_TABLE wird für die gesamte Hierarchie eine Tabelle benutzt. Die Tabelle kann viele Spalten mit Null-Werten enthalten, wenn die Sub-Klassen viele unterschiedliche Eigenschaften besitzen.
- TABLE_PER_CLASS legt für jede konkrete Klasse eine eigene Tabelle an.
- Im Falle von JOINED werden nur die von der Oberklasse unterschiedlichen Attribute in eine eigene Tabelle abgebildet. Ein Tabellen-Join stellt die Verbindung zwischen den Tabellen her.

```
<!-- Zur Auswahl stehen: SINGLE_TABLE|JOINED|TABLE_PER_CLASS -->
<inheritance strategy="SINGLE_TABLE" />

public @interface Inheritance {
  InheritanceType strategy() default InheritanceType.SINGLE_TABLE;
}

public enum InheritanceType {
  SINGLE_TABLE, JOINED, TABLE_PER_CLASS
};
```

In Kapitel 5 beschreiben wir die verschiedenen Alternativen ausführlich und führen auf Seite 143 einen Vergleich der Strategien durch.

10.2.38 Das Element <order-column>

Um eine Sortierung von Entities auch über den Persistenzkontext hinaus in der Datenbank zu gewährleisten, wurde mit JPA-Version 2.0 das Element <order-column> eingeführt. Es wird bei mehrwertigen Beziehungen verwendet. Die JPA-Implementierung stellt mit Hilfe einer zusätzlichen Spalte die Sortierung der Entities auch in der Datenbank sicher. Dies bedeutet, dass selbst nach Insert-, Delete- bzw. Update-Operationen die Sortierung erhalten bleibt.

```
<order-column column-definition="" insertable="true"
  name="" nullable="true" updatable="true"/>

public @interface OrderColumn {
  String name() default "";
  boolean nullable() default true;
  boolean insertable() default true;
  boolean updatable() default true;
  String columnDefinition() default "";
}
```

10.2.39 Das Element <map-key>

Eine mehrwertige Beziehung wird häufig in einer java.util.Map abgelegt. Über ein eindeutiges Property der referenzierten Entities als Map-Key lässt sich wesentlich schneller das

gesuchte Entity finden, als über eine Liste zu iterieren. Der Schlüssel kann der Primärschlüssel oder ein anderes Property des Entity sein. Ist das Attribut nicht der Primärschlüssel, muss ein `Unique-Constraint` einen eindeutigen Zugriff gewährleisten.

```
  <map-key name="" />

public @interface MapKey {
  String name() default "";
}
```

Das Attribut `name` ist optional. Wird es nicht angegeben, legt die JPA-Implementierung das jeweilige Entity unter seinem Primärschlüssel in die Map.

10.2.40 Die weiteren `<map-key-...>`-Elemente

Für eine bessere Abbildung von Beziehungen zwischen Entities in einer `java.util.Map` wurden in der JPA-Version 2.0 weitere XML-Elemente eingeführt. Der Key, unter dem das referenzierte Entity in der Map abgelegt wird, ist dabei nicht notwendigerweise Bestandteil des Entity.

Das Element `map-key-class` spezifiziert ein Entity, das mit `@Embeddable` annotiert ist. Unter Instanzen dieses Entity als Schlüssel werden in der Map die referenzierten Objekte des eigentlichen Ziel-Objektes abgelegt.

```
<map-key-class class=""/>

public @interface MapKeyClass {
  Class value();
}
```

Wird die Map parametrisiert, kann die Angabe von `<map-key-class>` bzw. der Annotation auch entfallen, da der Typ des Schlüssels und auch des Ziel-Objektes feststehen.

Zur Abbildung einer Beziehung über eine Map mit Hilfe eines einfachen Datentyps steht das Element `<map-key-column>` zur Verfügung. Darüber hinaus kann auch `<map-key-enumerated>` bei Aufzählungen bzw. `<map-key-temporal>` bei `Calendar`-Typen verwendet werden.

```
<map-key-column column-definition="" insertable="true" length="255"
  name="" nullable="true" precision="0" scale="0" table=""
  unique="true" updatable="true"/>

<!-- ORDINAL|STRING -->
<map-key-enumerated>ORDINAL</map-key-enumerated>

<!-- DATE|TIME|TIMESTAMP -->
<map-key-temporal>DATE</map-key-temporal>

public @interface MapKeyColumn {
  String columnDefinition() default "";
  boolean insertable() default true;
  int length() default 255;
  String name() default "";
  boolean nullable() default false;
```

```
  int precision() default 0;
  int scale() default 0;
  String table() default "";
  boolean unique() default false;
  boolean updatable() default true;
}

public @interface MapKeyEnumerated {
  EnumType value() default EnumType.ORDINAL;
}

public @interface MapKeyTemporal {
  TemporalType value();
}
```

Ein Tabellen-Join über drei Tabellen lässt sich ebenfalls in einer Map abbilden. Benutzt wird dazu das Element `map-key-join-column`. Bei einer Many-To-Many-Beziehung wird der Schlüssel in der `Join`-Table abgelegt. Eine One-To-Many-Beziehung legt den Schlüssel in der Entity-Tabelle des Zielobjektes ab, wenn nicht anders definiert. Dabei ist es nicht erforderlich, dass das Ziel-Objekt den Schlüssel selbst als Property abbildet. Dies ist ein wesentlicher Fortschritt in JPA 2.0.

```
<map-key-join-column column-definition=""
  insertable="true" name="" nullable="true"
  referenced-column-name="" table="" unique="true"
  updatable="true" />

public @interface MapKeyJoinColumn {
  String columnDefinition() default "";
  boolean insertable() default true;
  String name() default "";
  boolean nullable() default false;
  String referencedColumnName() default "";
  String table() default "";
  boolean unique() default false;
  boolean updatable() default true;
}
```

Auch hier ist das Überschreiben von bereits existierenden Definitionen erlaubt. Das Element `<map-key-attribute-override>` verweist mit dem entsprechenden Namen auf eine andere Spaltenkonfiguration. Die entsprechende Annotation ist `@Attribute-Override` aus Abschnitt 10.2.15.

```
<map-key-attribute-override name="">
  <description>...</description>
  <column>
    <!-- Abschnitt 10.2.18 Seite 259 -->
  </column>
</map-key-attribute-override>
```

11 Hibernate-Erweiterungen

Wir beginnen in diesem Kapitel unsere Einführung in provider-spezifische Erweiterungen und starten mit Hibernate. Kapitel 12 stellt Erweiterungen von EclipseLink, Kapitel 13 Erweiterungen von OpenJPA vor.

Alle großen Datenbankhersteller erweitern die standardisierte Sprache SQL um eigene Sprachkonstrukte mit dem Ziel, gewisse Alleinstellungsmerkmale anbieten zu können und sich somit von den Mitbewerbern abzuheben. Dies führt zu einem Konkurrenzkampf um die besten Ideen und deren Umsetzung und ist somit begrüßenswert. Leider führt es aber auch zu einer Herstellerabhängigkeit und somit zu großen Portierungsaufwänden. Ganz analog wetteifern die JPA-Provider ihrerseits um derartige Alleinstellungsmerkmale mit denselben Resultaten: gute neue Ideen auf der einen, Herstellerabhängigkeit und Portierungsaufwand auf der anderen Seite. Falls Sie die freie Wahl des Application-Servers oder des JPA-Providers haben, nutzen Sie diese, und informieren Sie sich. Falls in Ihrem Projekt Application-Server oder JPA-Provider festgelegt sind, was sehr wahrscheinlich ist, schadet ein Blick über den Tellerrand nicht.

In diesem und den beiden nächsten Kapiteln arbeiten Sie in der Regel mit der provider-spezifischen Klasse, die das Interface `EntityManager` implementiert. Diese Klasse kann über die provider-spezifische Implementierung der Entity-Manager-Fabrik erzeugt werden, doch auch über ein offizielles JPA-API: Die Methode `getDelegate()` liefert die implementierende Klasse des Interface `EntityManager`. Für unsere drei Provider sind dies die Klassen:

- `org.hibernate.impl.SessionImp`
- `org.eclipse.persistence.internal.jpa.EntityManagerImpl`
- `org.apache.openjpa.persistence.EntityManagerImpl`

Diese Klassen implementieren auch das provider-spezifische Interface, das für die Erweiterungen verwendet wird. Wir geben kurz die entsprechenden Code-Ausschnitte wieder und führen explizit den Import des Interface an, um Ihnen Ihre eigenen Recherchen bezüglich der Provider-APIs zu erleichtern.

- Hibernates `Session`:

```
import org.hibernate.Session;
...
Session session = (Session) em.getDelegate();
```

- EclipseLinks `JpaEntityManager`:

```
import org.eclipse.persistence.jpa.JpaEntityManager;
...
JpaEntityManager jem = (JpaEntityManager) em.getDelegate();
```

- OpenJPAs `OpenJPAEntityManager`:

```
import org.apache.openjpa.persistence.OpenJPAEntityManager;
...
OpenJPAEntityManager oem = (OpenJPAEntityManager) em.getDelegate();
```

Mit JPA 2.0 wurde die generische Methode `unwrap()` eingeführt, die den Type-Cast überflüssig macht:

```
<T> T unwrap(java.lang.Class<T> cls)
```

Die Beispiele lassen sich dann schreiben als:

```
Session session = em.unwrap(Session.class);
JpaEntityManager jem2 = em.unwrap(JpaEntityManager.class);
OpenJPAEntityManager oem2 = em.unwrap(OpenJPAEntityManager.class);
```

Die in diesem und den beiden nächsten Kapiteln vorgestellten provider-spezifischen Erweiterungen stellen nur einen Teil der tatsächlich existierenden Erweiterungen dar. Wir haben uns bemüht, die wichtigsten und interessantesten Erweiterungen für Sie zusammenzutragen. Die ersten beiden Abschnitte der drei Kapitel sind jeweils demselben Thema gewidmet. Die Abschnitte 11.1, 12.1 und 13.1 gehen auf das Caching der Provider ein und vervollständigen damit das mit JPA 2.0 eingeführte und in Abschnitt 6.3.5 vorgestellte offizielle JPA-Caching. Die Abschnitte 11.2, 12.2 und 13.2 stellen die Möglichkeiten der Byte-Code-Instrumentierung vor. Die Abschnittsüberschriften entsprechen den von den Providern gewählten Bezeichnungen und unterscheiden sich daher, obwohl sie thematisch denselben Inhalt haben.

11.1 Caching

Hibernate setzt schon seit vielen Jahren auf den Einsatz externer Cache-Anbieter. Mit der Implementierung `org.hibernate.testing.cache.CachingRegionFactory` gibt es zwar eine eigene Lösung im Paket, diese wird jedoch laut Dokumentation nicht für den Betrieb produktiver Anwendungen empfohlen. Tabelle 11.1 gibt einen Überblick über die derzeitigen Cache-Implementierungen von Hibernate 4.0. Die Dokumentation erwähnt an einigen Stellen noch die alten Cache-Implementierungen über das Interface `CacheProvider`, die inzwischen veraltet sind. Ein Blick in die aktuelleren API-Dokumentationen des Packages `org.hibernate.cache` verweist auf die neueren Implementierungen über das Interface `RegionFactory` im gleichen Paket.

TABELLE 11.1 Implementierungen des `RegionFactory`-Interfaces

Cache-Name	Implementierende Klasse
HashMap(Testing)	`org.hibernate.testing.cache.CachingRegionFactory`
EHCache	`org.hibernate.cache.ehcache.EhCacheRegionFactory`
Infinispan	`org.hibernate.cache.infinispan.InfinispanRegionFactory`

Hinweis:

Wir beziehen uns in diesem Abschnitt auf die derzeit aktuellste Hibernate-Version 4.0. Gerade im Bereich Caching wurden hier Überarbeitungen vorgenommen, Klassen entfernt bzw. in andere Pakete verschoben. Dem Leser sollte beim Studium der Hibernate-APIs und der dazugehörigen Versionen immer bewusst sein, in welcher Version er sich befindet bzw. welche Version er gerade verwendet.

Die Grundkonfiguration der Persistenzeinheit haben wir für Hibernate bereits in Kapitel 6.3.5 vorgestellt. Um Wiederholungen zu vermeiden, benutzen wir in diesem Abschnitt die Cache-Implementierung *Infinispan* [URL-INF]. Infinispan ist wie Hibernate eine Entwicklung der JBoss-Community und Bestandteil des JBoss-Application-Servers ab Version 6. Unsere Datei `persistence.xml` sieht nun wie folgt aus:

```
<property name="hibernate.cache.use_second_level_cache"
          value="true" />
<property name="hibernate.cache.use_query_cache" value="true" />
<property name="hibernate.generate_statistics" value="true" />

<property name="hibernate.cache.region.factory_class"
          value="org.hibernate.cache.infinispan.
              InfinispanRegionFactory" />
<property name="hibernate.transaction.manager_lookup_class"
          value="de.jpainfo.hibernate.
              TestDummyTransactionManagerLookup" />
```

Der erste Block von Properties kommt dem Leser mit Sicherheit bekannt vor. Er aktiviert den Objekt- und Query-Cache bei Hibernate. Darüber hinaus wird Hibernate angewiesen, entsprechende Statistiken zu generieren, mit denen wir später die Cache-Benutzung dokumentieren wollen. Die Factory-Klasse wird ebenfalls analog zu unserem Beispiel des JPA-Cachings gesetzt, allerdings unter Benutzung von Infinispan.

Die danach folgende Lookup-Klasse ist eine Adapterklasse von uns und dient nur zu Demonstrationszwecken. Für den produktiven Einsatz ist sie nicht zu empfehlen. Infinispan ist als Cache-Lösung in der Lage, mit der JTA-Transaktion der Anwendung synchron zu agieren. Das bedeutet: Cache und Datenbank benutzen die gleiche Transaktion, um Daten zu schreiben (Commit), oder die Änderungen werden gemeinsam rückgängig gemacht (Rollback). Für unsere Beispiele, die nicht in einem Java-EE-Container laufen, haben wir den Transaktions-Typ `RESOURCE_LOCAL` hinterlegt. Die Anwendung muss sich also selbst um die Bereitstellung und Verwaltung der Transaktionen kümmern. JTA ist auch in lokalen Anwendungen nutzbar. Die Darstellung der Konfiguration würde jedoch an dieser Stelle zu komplex werden. Bei Bedarf findet der Leser eine sehr ausführliche Anleitung zur Integration von Hibernate, Infinispan und JTA außerhalb einer Java-EE-Umgebung in der

Dokumentation von Infinispan [URL-INF]. Wir begnügen uns in unseren Beispielen mit dem `DummyTransactionManager` von Infinispan, den unsere Lookup-Klasse in Listing 11.1 bereitstellt.

LISTING 11.1 Lookup-Klasse zur Benutzung des `Dummy-Transaction-Managers`

```
public class TestDummyTransactionManagerLookup
  // Dummy-Manager von Infinispan
  extends DummyTransactionManagerLookup
  // Hibernates Interface zum Laden der Lookup-Klasse ueber das
  // Property hibernate.transaction.manager_lookup_class
  implements TransactionManagerLookup{

  public TransactionManager getTransactionManager(Properties props)
      throws HibernateException {
    try {
      // Dummy-Manager von Infinispans Lookup-Klasse
      return super.getTransactionManager();
    } catch (Exception e) {
      e.printStackTrace();
      return null;
    }
  }

  public String getUserTransactionName() {
    return null;
  }

  public Object getTransactionIdentifier(Transaction transaction) {
    return transaction;
  }
}
```

Zur genaueren Steuerung des Cache-Verhaltens pro Entity stellt Hibernate eine eigene Annotation bereit: `org.hibernate.annotations.Cache`. Die Erklärung der Attribute dieser Annotation findet der Leser in Tabelle 11.2. Diese Annotation ist das Gegenstück zum JPA-Standard `@Cacheable`. Wir empfehlen, nur dann auf diese provider-spezifischen Elemente auszuweichen, wenn deren Funktionsumfang für den Entwickler einen tatsächlichen Mehrwert zum JPA-Standard bereitstellt.

TABELLE 11.2 `@Cache`-Attribute

Element	Bedeutung
`usage`	Dieses Element nimmt einen der Werte `NONE`, `READ_ONLY`, `NONSTRICT_READ_WRITE`, `READ_WRITE` oder `TRANSACTIONAL` an. Mit der Einstellung wird gesteuert, ob ein gleichzeitiger schreibender Zugriff auf Werte des Caches auftreten wird bzw. ob eine JTA-Transaktion zwingend zu benutzen ist. Achtung: Nicht alle Cache-Anbieter unterstützen alle zur Verfügung stehenden Verfahren!
`include`	Entscheidet darüber, ob auch LAZY-Attribute eines Entity in den Cache gebracht werden sollen. Die Default-Einstellung ist `all`, alternativ kann `non-lazy` benutzt werden.
`region`	Der Bereich des Cache, in dem die Entities abgelegt werden sollen.

11.1.1 Objekt-Cache

Zugriff auf den Hibernate-Cache bzw. dessen Statistiken erhalten wir nur über Hibernate-spezifische Objekte und Methoden. Das nachfolgende Listing zeigt dem Leser den Weg von der JPA-`EntityManagerFactory` bis zur Cache-Implementierung von Hibernate.

```
// JPA-Instanz
EntityManagerFactory emf = Persistence.createEntityManagerFactory(
  "bank", DB.oracle);
// Hibernate-Instanz er Factory
HibernateEntityManagerFactory hemf = (HibernateEntityManagerFactory)
    emf;
// Hibernates SessionFactory
SessionFactory factory = hemf.getSessionFactory();
// Hibernates Statistiken
Statistics st = factory.getStatistics();
System.out.println("MissCount: "+st.getSecondLevelCacheMissCount());
System.out.println("HitCount: "+st.getSecondLevelCacheHitCount());
System.out.println("PutCount: "+st.getSecondLevelCachePutCount());
// Hibernates-Cache-Interface
Cache hCache = factory.getCache();
```

Cache-Regions

Die im letzten Code-Beispiel gelesenen Statistiken sind eine aggregierte Sicht auf einen Second-Level-Cache, der in Wahrheit wesentlich feiner gegliedert ist, als es uns die Statistiken vorgaukeln. Hibernate richtet ohne weitere Konfiguration für jedes unserer Entities einen eigenen Cache, sprich *Region* ein. Einstellungen, wie z.B. Cache-Namen, Ablaufzeit, Größe sowie das Verhalten der Cache-Leerung können individuell beeinflusst werden.

Infinispan stellt für die einzelnen Cache-Arten ein Set von Standard-Konfigurationen bereit. Die Datei `infinispan-configs.xml` findet der Leser in der JAR-Datei `hibernate-infinispan.jar`. Wir zeigen daraus exemplarisch die Einstellungen für den Entity-Cache in Listing 11.2, um dem Leser einen Eindruck von den zahlreichen Einstellungsmöglichkeiten zu geben.

LISTING 11.2 Hibernates Infinispan-Konfigurationen

```
<namedCache name="entity">
   <clustering mode="invalidation">
      <stateRetrieval fetchInMemoryState="false" timeout="20000"/>
      <sync replTimeout="20000"/>
   </clustering>
   <locking isolationLevel="READ_COMMITTED"
            concurrencyLevel="1000"
            lockAcquisitionTimeout="15000"
            useLockStriping="false"/>
   <!-- Eviction configuration.  WakeupInterval defines how often
        the eviction thread runs, in milliseconds.  0 means the
        eviction thread will never run.  A separate executor is
        used for eviction in each cache. -->
   <eviction wakeUpInterval="5000"
             maxEntries="10000"
             strategy="LRU"/>
```

```
    <expiration maxIdle="100000"/>
    <lazyDeserialization enabled="true"/>
  </namedCache>
```

Sind davon abweichende Konfigurationen erforderlich, lassen sich diese entweder global für alle Entities und Gruppen oder für einzelne Klassen hinterlegen. Das nachstehende Beispiel erhöht für unser `Kunde`-Entity die Cache-Größe auf 20000 Entity-Instanzen:

```
<property name="hibernate.cache.infinispan.de.jpainfo.model.Kunde.
                eviction.max_entries"
          value= "20000"/>
```

Wir wollen nun die Benutzung des Objekt-Cache mit den Methoden `find()`, `refresh()` und einer Query nachvollziehen. Um den Cache-Zugriff und die Statistiken richtig zu lesen, muss der Leser bei der Entwicklung verschiedene `EntityManager`-Instanzen benutzen. Bei Zugriff über dieselbe Instanz kommt der Second-Level-Cache sonst nicht zum Einsatz, es wird der First-Level-Cache der `EntityManager`-Instanz benutzt. Aus diesem Grund haben wir in Listing 11.3 einzelne Methoden mit unterschiedlichen Instanzen implementiert und zum Verständnis den kompletten Quellcode der Java-Klasse abgebildet.

LISTING 11.3 Second-Level-Cache und Statistiken

```
public class SecondLevelDataCache {
  public static EntityManagerFactory emf;
  public static HibernateEntityManagerFactory hemf;

  public static void main(String[] args) throws Exception {
    emf = Persistence.createEntityManagerFactory("bank", DB.oracle);
    hemf = (HibernateEntityManagerFactory)emf;
    readKundeQuery();
    readKundeFind();
    refreshKunde();
    MonitorCache.printStatistics(hemf);
    emf.close();
  }

  public static void readKundeQuery() {
    EntityManager em = emf.createEntityManager();
    // Query - Variante 1
    Query q = em.createQuery("select k from Kunde k where k.id = 1");
    q.getSingleResult();
    em.close();
  }

  private static void readKundeFind() {
    EntityManager em = emf.createEntityManager();
    // Find - Variante 2
    Kunde k = em.find(Kunde.class, new Integer(1));
    em.close();
  }

  private static void refreshKunde() {
    EntityManager em = emf.createEntityManager();
    // Refresh -Variante 3
```

```
    Kunde k = em.find(Kunde.class, 1);
    Map map = new HashMap<String, Object>() {{
        put("javax.persistence.cache.storeMode", CacheStoreMode.
            REFRESH);
      }};
    em.refresh(k, map);
    em.close();
  }
}
```

Die Methode `readKundeQuery()` selektiert einen Kunden über eine JPA-Query. Weil wir in der Query-Selektion ein ganzes Objekt abfragen und nicht einzelne Attribute des Entity, wird das abgefragte Objekt in den Second-Level-Cache überführt. Danach wird im gleichen Kontext aber einem anderen Entity-Manager mit der `find()`-Methode ebenfalls der `Kunde` gelesen. Die Datenbank wird nicht abgefragt. Das abgefragte Entity wird direkt aus dem Cache gelesen. Die dritte Variante liest mit `find()` ebenfalls aus dem Cache. Die Methode `refresh()` mit explizit gesetztem Property liest aus der Datenbank und aktualisiert den Cache danach.

Die von uns gekürzte Ausgabe der Cache-Statistiken nach der letzten `refresh()`-Methode sieht wie folgt aus:

```
...
===== Cache name: de.jpainfo.model.Kunde
HitCount: 2
PutCount: 1
===== Cache name: de.jpainfo.model.Konto
HitCount: 0
PutCount: 2
===== Cache name: de.jpainfo.model.Adresse
HitCount: 0
PutCount: 1
...
```

Nach dem Ende aller Methoden wurde ein Kunde-Objekt genau zweimal im Cache gefunden: verursacht durch die `find()`-Anweisungen von Methode zwei und drei. Ein Kunde-Objekt (und dessen assoziierte Objekte) wurde dem Cache hinzugefügt. Im Unterschied zu OpenJPA führt das `refresh()` erst dann zu einem erneuten Cache-Update, wenn sich Werte der Instanz tatsächlich geändert haben. Der Leser möge dies mit der Änderung eines Properties nach der Methode `find()` aber vor `refresh()` selbst nachvollziehen.

11.1.2 Query-Cache

Mit Aktivierung des Query-Cache stellt Hibernate die weiteren Cache-Typen `StandardQueryCache` und `UpdateTimestampsCache` bereit. Während der erste Cache die Query-Ergebnisse aufnimmt, wird mit dem zweiten überprüft, ob die Ergebnisse beim Lesen aus Ersterem noch gültig sind oder nicht. Hier werden die Timestamps der letzten Updates hinterlegt, die bereits gespeicherte Queries potenziell betreffen könnten.

Hinweis:

Im `UpdateTimestamps`-Cache speichert Hibernate die Zeitpunkte der letzten Datenbank-Updates für bestimmte Tabellen. Timeouts dieses Cache sollten unbedingt höher sein als die der Query-Caches. Ansonsten läuft der Entwickler Gefahr, dass der Query-Cache veraltete Daten zurückliefert. Hibernate empfiehlt, für diesen Cache keine Eviction-Policy zu setzen. In der Standard-Konfiguration ist sie dementsprechend auch deaktiviert: `<eviction wakeUpInterval="0" strategy="NONE"/>`

Zusätzlich zum Property-Eintrag in der Datei `persistence.xml` muss einer Query noch gesagt werden, dass ihre Ergebnisse im Zwischenspeicher abzulegen sind. Wir machen dies wie gewohnt über den JPA-Standard und einen entsprechenden provider-spezifischen Hint. Ohne die Konfiguration der Persistenzeinheit führt Hibernate kein Query-Caching durch, selbst wenn eine Query direkt über den Hint konfiguriert wurde.

```
Query q = em.createQuery("select k from Kunde k where k.id <= :
    upperRange");
q.setHint(QueryHints.HINT_CACHEABLE, true);
```

Der Query-Cache speichert die Entity-Zustände nicht in seinem Cache, sondern nur Referenzen auf Objekte im Entity-Cache. Aus diesem Grund wird empfohlen, den Query-Cache immer in Verbindung mit dem Objekt-Cache zu benutzen. Eine Deaktivierung des Objekt-Caches wird das Zwischenspeichern der Query-Ergebnisse nahezu überflüssig machen.

Projekt:

Unsere Beispiele zum Caching von Hibernate finden Sie im Eclipse-Projekt `hibernatecache`.

11.2 Instrumentation

Um ein Maximum an Performanz zu erreichen, manipulieren JPA-Provider die Entity-Klassen. Durch sogenannte *Byte-Code-Manipulation* werden Änderungen am Compilat eines Entity vorgenommen, um z. B. das Automatic Dirty Checking effizienter oder das späte Laden von Beziehungen (Lazy Loading) überhaupt erst zu realisieren. Die Manipulation von Byte-Code wird in den jeweiligen Dokumentationen der drei Provider unterschiedlich bezeichnet. Hibernate nennt es *Instrumentation*, EclipseLink *Weaving* und OpenJPA *Enhancement*. Wir stellen hier zunächst Hibernates Instrumentation vor. Falls Sie Eclipselink als Provider verwenden, finden Sie in Abschnitt 12.2 die entsprechenden Informationen zu EclipseLinks Weaving. Im Falle von OpenJPA finden Sie Details zu OpenJPAs Enhancement in Abschnitt 13.2.

Als Beispiel verwenden wir ein Kunden-Entity, das ein Byte-Array für ein Passbild sowie eine 1:1-Beziehung zu einem Adresse-Entity besitzt. Standardmäßig werden alle Properties geladen. Bei einer 1:1-Beziehung ist das Standardverhalten das frühe Laden (Eager Loading) der Beziehung. Das Ladeverhalten sowohl des Property als auch der Beziehung kann über das `fetch`-Attribut der `@Basic`- bzw. die `@OneToOne`-Annotation explizit gesetzt werden.

Im folgenden Beispiel (siehe Listing 11.4) sollen beide Properties spät geladen werden (Lazy Loading).

LISTING 11.4 Entity `Kunde` mit `LAZY`-Property und -Beziehung

```
@Entity
public class Kunde {

  @Id @GeneratedValue
  private Integer id;
  private String vorname;
  private String nachname;

  @Basic(fetch = FetchType.LAZY)
  private byte[] passbild;

  @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
  private Adresse adresse;

  ...
}
```

Laut Spezifikation ist die Auswahl von `LAZY` ein Hinweis (Hint) an den JPA-Provider, den er nicht befolgen muss. Hibernate beherzigt den Hinweis ohne Instrumentation für die Beziehung, nicht jedoch für das Passbild. Wir führen dies gleich detaillierter aus.

Hinweis:

Untersuchungen des Java-Enterprise-Performance-Teams von Sun haben ergeben, dass sich der Durchsatz (Transaktionen pro Sekunde) beim Vergleich EAGER/LAZY einer 1:n-Beziehung um lediglich 10% unterschied [URL-PerfEJB]. Obwohl die Untersuchung sich auf EJB 3.0 und damit JPA 1.0 bezog, kann man von ähnlichen Ergebnissen für JPA 2.0 ausgehen. Für die Entscheidung, welcher der beiden Alternativen der Vorzug zu geben ist, sollten Sie immer Vorher/Nachher-Messungen durchführen.

In Überarbeitung des Beispiels ist zunächst anzumerken, dass das Property `passbild` zusätzlich zur Darstellung in Listing 11.4 die Annotation `@Lob` benötigt, weil die Tabellengenerierung durch Hibernate sonst in einer `VARBINARY`-Spalte der Länge 255 resultiert, was für das Beispielpassbild zu klein ist. Interessanter ist jedoch die Behandlung der beiden Properties bezüglich des Ladezeitpunkts. Betrachten wir zunächst den Fall, dass die in Listing 11.4 verwendeten `fetch`-Attribute nicht existieren, so dass der Default `EAGER` maßgebend ist. Das `Select` auf die Kundentabelle liest alle Tabellenspalten inklusive `passbild`. Die 1:1-Beziehung wird ebenfalls in diesem `Select` mit einem Left-Outer-Join gelesen, so dass lediglich ein `Select` abgesetzt wird.

Bei der in Listing 11.4 dargestellten Variante mit jeweils spätem Ladezeitpunkt und ohne Instrumentierung ergibt sich beim Passbild ein frühes Laden, bei der 1:1-Beziehung jedoch ein spätes Laden zum Zeitpunkt des ersten Zugriffs.

Für die Instrumentierung bietet Hibernate eine Ant-Task an, die wie im folgenden Code-Ausschnitt verwendet werden kann.

```
<target name="instrument" depends="compile">
  <taskdef name="instrument" classname=
        "org.hibernate.tool.instrument.javassist.InstrumentTask">
    <classpath path="${build.home}"/>
    <classpath refid="classpath" />
  </taskdef>
  <instrument verbose="true">
    <fileset dir="${build.home}/de/jpainfo/model/">
      <include name="Kunde.class"/>
    </fileset>
  </instrument>
</target>
```

Durch eine andere Verwendung des `<fileset>`-Elements können alternativ auch alle Compilate instrumentiert werden. Wir haben uns hier auf die Klasse `Kunde` beschränkt.

Nach der Instrumentierung wird das Passbild initial nicht geladen, die entsprechende Spalte ist also im initialen `Select` nicht enthalten. Beim Zugriff auf das Property wird es mit einem gesonderten `Select` explizit nachgeladen.

Projekt:

Das Projekt *hibernate-instrument* enthält das verwendete Beispiel sowie ein Ant-Build-File für die Verwendung der Hibernate-Instrumentierung.

11.3 Hibernate-Search

Kommen wir nun zu einem weiteren Teilprojekt, mit dem Hibernate weit über den JPA-Standard hinausgeht. Mit der Integration von Lucene [URL-LUC], einer vollständig in Java geschriebenen Suchmaschine, bietet Hibernate die Möglichkeit, die im Persistenzkontext vorhandenen Entities zu durchsuchen. Dabei werden die Suchbegriffe nicht in SQL-Konstrukte übersetzt, sondern direkt auf einem eigens angelegten Index außerhalb der Datenbank durchgeführt.

Hibernate kapselt die Zugriffe auf den Index vollständig. Der Anwendungsentwickler hat mit dem eigentlichen Lucene-API nahezu keine Berührungspunkte. Auch die Art und Weise, wie eine Suche durchgeführt wird, orientiert sich am Interface `org.hibernate.Query`, der Schnittstelle, mit der JPQL-Abfragen und native SQL-Abfragen durchgeführt werden.

Das Hinzufügen, Ändern oder Entfernen von Entities aus dem Index erfolgt automatisch mit Hilfe von Hibernates Event-Listenern. In den folgenden Abschnitten wollen wir einen kurzen Einblick in die Konfiguration und Benutzung einer Lucene-basierten Entity-Suche geben.

11.3.1 Lucene

Lucene führt die Suche nach Begriffen zur Laufzeit nicht auf den zu durchsuchenden Objekten, sondern auf einem vorher angelegten Index aus. Mit Objekten sind an dieser Stelle keine Java-Objekte gemeint. Der Begriff umfasst hier sämtliche Dinge, die durchsucht werden

können: Web-Seiten, Word-Dokumente, Datenbanken aber eben auch Instanzen von Java-Klassen. Ein durchsuchbarer Index wird in Lucene mit der Klasse `Directory` modelliert. Ein Directory kann man sich wie einen Behälter vorstellen, der den zu speichernden Index aufnimmt. Folgende Container stellt Lucene bereit, einige davon sind teilweise noch als *experimental* gekennzeichnet:

FileSwitchDirectory ist eine Verzeichnis-Instanz, bei der die Index-Dateien zwischen verschiedenen `Directory`-Instanzen aufgeteilt werden.

FSDirectory stellt die Standard-Implementierung des Indexes im Dateisystem dar und ist Ausgangspunkt für die plattformunabhängige Ablage der Index-Dateien. Subklassen dieser Klasse bieten spezifische Erweiterungen für Linux und Windows, sofern diese benötigt werden.

NRTCachingDirectory bietet eine `Near-real-time`-Implementierung für Anwendungen, bei der die Indizierungshäufigkeit gering ist, der Index aber häufig neu geöffnet wird. Diese Implementierung beugt der Segmentierung des Indexes vor.

RAMDirectory legt den Such-Index im Arbeitsspeicher des Rechners ab.

Ein Index besteht aus einer Menge von sogenannten *Dokumenten*. Jedes Dokument besitzt eine Reihe von Feldern (*Fields*). Ein Feld wiederum besitzt einen Namen sowie eine Menge von *Termen*, die einfache Strings sind. Inhalte indizierter Felder können zusätzlich direkt im Index abgelegt werden. Dies dient dazu, in der Liste der Suchergebnisse eine kurze Beschreibung des jeweiligen Ergebnisses zu liefern. Diese Information ließe sich aus den eigentlichen Ergebnissen nicht mehr rekonstruieren.

Hinweis:

Mit dem Wort Feld wird in der Informatik eine Zusammenfassung von Daten gleichen Datentyps in einer ein- oder mehrdimensionalen Anordnung, also einem Array, verstanden. Wenn wir von einem Array sprechen, benutzen wir nicht das Wort *Feld*, sondern eben *Array*. Wenn wir also in diesem Kapitel das Wort *Feld* verwenden, beziehen wir uns immer auf das *Field* eines Lucene-Indexes.

11.3.2 Konfiguration in Hibernate

Hibernate stellt für die eingangs erwähnten Index-Container Provider-Klassen bereit, die die gesamte Konfiguration und Einrichtung des Lucene-Index übernehmen. Wie die Konzepte von Hibernate und Lucene zusammengeführt werden, zeigt Tabelle 11.3.

TABELLE 11.3 Lucene und Hibernate im Vergleich

Lucene	Hibernate
Index	Entity-Klasse
Document	Instanz eines Entity
Field	Attribut eines Entity
Term	Inhalt des Attributes

Welches Entity in welchem Index landet, wird zum einen über eine entsprechende Annotation im Entity konfiguriert. Zum anderen können über entsprechende Properties in den Konfigurationsdateien generelle Vorgehensweisen festgelegt werden:

```
<property name="hibernate.search.default.directory_provider"
          value="filesystem" />
<property name="hibernate.search.default.indexBase"
          value="/tmp/lucene/" />
<property name="hibernate.search.default.indexName"
          value="index" />
<property name="hibernate.search.Konto.directory_provider"
          value="ram" />
```

Mit Hilfe der ersten Einstellung und des Schlüsselwortes `default` wird festgelegt, dass für Indizes ohne Konfiguration das Verzeichnis im zweiten Property gilt. Diese Einstellung wird auch für die Art des Indexes, in unserem Beispiel ein dateibasierter Index, vorgenommen. Außerdem konfigurieren wir einen RAM-basierten Index im letzten Property namens `Konto`, den die Entities über das Element `indexName` der Annotation `@Indexed` benutzen können.

```
@Entity
@org.hibernate.search.annotations.Indexed
public class Kunde {
  @Id
  @org.hibernate.search.annotations.DocumentId
  @GeneratedValue
  private int id;
  // Attribute und Methoden ...
}

@Entity
@org.hibernate.search.annotations.Indexed(index="Konto")
public abstract class Konto {
  // Attribute und Methoden ...
}
```

Weil der Name des Index beim Entity `Kunde` nicht konfiguriert ist, greifen für diesen Index die Standardeinstellungen. Hibernate wird für den ersten Fall das Lucene-API auf das Verzeichnis `/tmp/lucene/index` umleiten. Für den Index `Konto` wird kein Verzeichnis im Dateisystem benötigt, er wird komplett im Arbeitsspeicher des Rechners gehalten. Weil jedes Lucene-Dokument eine eindeutige Id benötigt, greift das Beispiel idealerweise auf die Id des Entity zurück. Die Annotation `@DocumentId` übernimmt diese Einstellung in den Index, kann aber bei Vorhandensein der JPA-Annotation `@Id` auch entfallen.

Automatisches Indizieren

Werden Hibernates Annotationen benutzt und ist mit dem Schlüsselwort `@Indexed` die Indizierung aktiviert, führt Hibernate automatisch bei jedem Insert, Update und Delete die entsprechenden Änderungen auch auf dem konfigurierten Index aus. Die JAR-Dateien von *Hibernate Search* müssen dazu im Klassenpfad der Anwendung zur Verfügung stehen.

Wie kann man einen Index aufbauen, wenn sich bereits Daten in der Datenbank befinden? Wird mit einer leeren Datenbank begonnen, ist dies kein Problem, da man dann auch mit einem leeren Index beginnen kann. Soll dagegen der Index aus einer bestehenden Datenbank

erstellt werden, hilft die Klasse `FullTextSession`. Das Programm dazu ist in Listing 11.5 dargestellt.

LISTING 11.5 Index-Generierung mit Hibernates Massen-Indizierungstools

```
public class MassenIndizierung {
  public static EntityManagerFactory emf;
  public static HibernateEntityManagerFactory hemf;

  public static void main(String[] args) throws Exception {

    emf = Persistence.createEntityManagerFactory("bank", DB.h2);
    EntityManager em = emf.createEntityManager();
    Session hSession = (Session) em.getDelegate();
    FullTextSession fullTextSession =
      Search.getFullTextSession(hSession);

    fullTextSession.createIndexer(Kunde.class)
    // .batchSizeToLoadObjects( 25 )
    // .cacheMode( CacheMode.NORMAL )
    // .threadsToLoadObjects( 5 )
    // .threadsForIndexWriter( 3 )
    // .idFetchSize( 150 )
    // .threadsForSubsequentFetching( 20 )
    .startAndWait();
  }
}
```

Achtung:

Die Hibernate-Dokumentation weist ausdrücklich darauf hin, dass sich während einer Massen-Indizierung der Index in einem Zustand befindet, in dem er nicht von anderen Programmen benutzt werden sollte. Für eine optimale Massen-Indizierung lässt sich der Vorgang in mehrere parallele Threads auslagern. Weitere Einstellungsmöglichkeiten bieten die von uns dargestellten, auskommentierten Methoden.

Wie sieht der Index nach der Erstellung aus? Mit dem Programm Luke [URL-LUK] lassen sich Lucene-Indizes öffnen und durchsuchen. Abbildung 11.1 zeigt den bisherigen Stand. Der Index besteht momentan aus 990 Dokumenten. Das entspricht der Anzahl von Datenbanksätzen der Tabelle `Kunde`. Jedes Dokument besitzt zwei Felder, `_hibernate_class` und `id`. Das erste Feld ist ein internes Feld, das von Hibernate automatisch in den Index aufgenommen wurde, das zweite Feld entspricht der eindeutigen Id unseres Kunden-Entity. In diesem Zustand ist der Index noch wenig nützlich, wir müssen daher weitere Felder des Kunden-Entity mit aufnehmen.

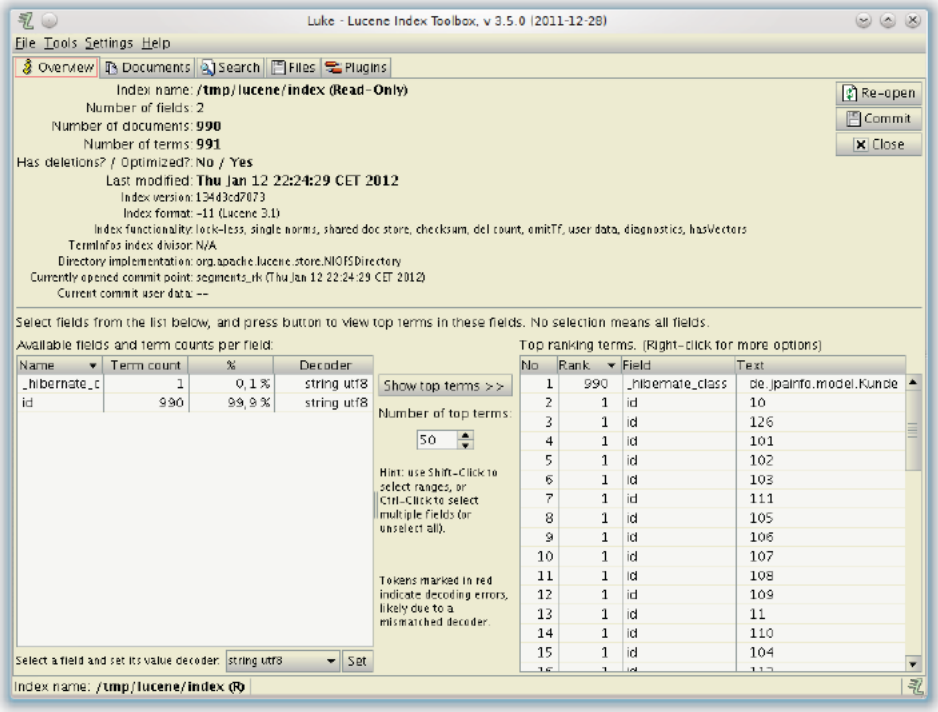


ABBILDUNG 11.1 Lucene-Index der Klasse `Kunde` (1)

Hinweis:

Das Erzeugen eines Index sowie die darauf durchgeführten Abfragen müssen mit dem gleichen Lucene-Analyzer durchgeführt werden. Dieser Analyzer teilt den Text je nach zugrunde liegender Sprache in einzelne Teile. Als Standard-Einstellung wird die Klasse `org.apache.lucene.analysis.standard.StandardAnalyzer` benutzt. Über das Property `hibernate.search.analyzer` lassen sich andere Analyzer einstellen. Darüber hinaus ist eine umfangreiche Konfiguration von Analyzern und Filtern über Annotationen am Entity bzw. einzelnen Attributen möglich.

Index-Erweiterung um Entity-Attribute

Im Gegensatz zu JPA, wo die Persistierung bestimmter Attribute mit der Annotation `@Transient` explizit unterdrückt werden muss, muss an dieser Stelle jedes Attribut, das in den Index aufgenommen werden soll, als solches explizit konfiguriert werden. Die Annotation `@Field` merkt ein Property für die Indizierung vor.

Für Entity-Attribute vom Typ `Double`, `Long`, `Integer` und `Float` steht darüber hinaus die Annotation `@NumericField` zur Verfügung.

TABELLE 11.4 `@Field`-Attribute

Option	Beschreibung
`index`	Mit dem Wert `Index.YES` wird die Indizierung des Feldes aktiviert. Damit kann man nach Werten des Feldes suchen.
`name`	Beschreibt den Namen des Feldes, unter dem das Entity-Attribut in den Index übernommen wird. Erfolgt keine Angabe, so wird der Name des Attributes benutzt.
`store`	Die Einstellung `Store.YES` legt den originalen Inhalt des Entity-Attributs auch im Such-Index ab, und nicht nur den indizierten Wert. Damit lassen sich Suchergebnisse darstellen, indem die Werte des Such-Indexes angezeigt werden und nicht auf die Datenbank zugegriffen werden muss. `Store.COMPRESS` komprimiert die Originalwerte zusätzlich.
`analyze`	Um einen Feldinhalt nach seinen einzelnen Elementen zu durchsuchen, muss das Entity-Attribut analysiert werden. `Analyze.NO` deaktiviert dieses Verhalten – so ergibt die Suche nur dann einen Treffer, wenn die Suchbedingung exakt dem Feldinhalt entspricht.
`boost`	Um ein Feld gegenüber anderen Feldern bei der Suche zu bevorzugen, kann mit diesem Element ein *Boost*-Faktor statisch gesetzt werden. Auch eine dynamische Priorisierung von Feldern, in Abhängigkeit vom Zustand des jeweiligen Entity ist mit der Annotation `@DynamicBoost` auf Klassen-Ebene möglich.

Mit der Annotation `@IndexedEmbedded` können Entities, die über eine Assoziation mit dem zu indizierenden Entity verbunden sind, ebenfalls in den Index aufgenommen werden. Diese Möglichkeit bezieht sich auf alle n:1-Beziehungen sowie auf `@Embedded`-Konfigurationen. Tabelle 11.5 fasst das Element `@IndexedEmbedded` zusammen.

TABELLE 11.5 `@IndexedEmbedded`-Attribute

Option	Beschreibung
`prefix`	Präfix, das den Attributen des eingebetteten Entity vorangestellt werden soll.
`depth`	Tiefe, bis zu der die Indizierung durchgeführt werden soll.
`indexNullAs`	Mit diesem Element werden `null`-Werte der Attribute, in diesem Fall der Beziehung zu anderen beteiligten Entities behandelt. Die Standard-Einstellung ist, dass diese Werte nicht in den Index übernommen werden. Mit `DEFAULT_NULL_TOKEN` kann ein Ausgangswert über das Property `hibernate.search.default_null_token` konfiguriert werden.

Achtung:

Die für eine Indizierung mittels `@IndexedEmbedded` benutzten Beziehungen müssen bidirektional sein. Andernfalls kann Hibernate bei Änderung des referenzierten Entity (in unserem Beispiel das Attribut `wohnort`) nicht den Index des Root-Entity aktualisieren.

11.3.3 Hibernate-Bridges

Lucene arbeitet als Suchmaschine textbasiert, also auf Strings. Es muss daher eine Konvertierung zwischen den im Entity verwendeten Attribut-Typen und der entsprechenden String-

Repräsentation geben. Hibernate stellt für diesen Zweck die folgenden Klassen zur Verfügung:

- BigDecimalBridge,
- BigIntegerBridge,
- BooleanBridge,
- DateBridge,
- DoubleBridge,
- EnumBridge,
- FloatBridge,
- IntegerBridge,
- LongBridge,
- NumberBridge,
- ShortBridge,
- StringBridge.

Ist beim Attribut nichts anderes angegeben, versucht Hibernate automatisch die passende Adapter-Klasse zu finden.

Der Annotation @DateBridge kann über den Parameter resolution mitgeteilt werden, mit welcher Genauigkeit die Konvertierung eines Date-Feldes in einen String durchgeführt werden soll.

Zur Auswahl stehen: DAY, HOUR, MILLISECOND, MINUTE, MONTH, SECOND und YEAR. Für das Geburtsdatum der Kunden wird die Auflösung DAY benutzt:

```
@Temporal(TemporalType.DATE)
@Field(
    index = Index.NO,
    store = Store.YES)
@DateBridge(
    resolution = Resolution.DAY)
private Date geburtsdatum;
```

Implementierung eigener Brücken

Reichen die von Hibernate bereitgestellten Klassen nicht aus, besteht die Möglichkeit, durch Implementierung der nachfolgenden Interfaces die Konvertierung der Attribute in Strings durchzuführen. Alle Interfaces befinden sich im Package org.hibernate.search.bridge:

- StringBridge: Durch Implementierung der Methode objectToString() kann die Umwandlung einer beliebigen Java-Klasse in deren String-Repräsentation durchgeführt werden.
- Das Interface TwoWayStringBridge erbt vom Interface StringBridge. Hier muss zusätzlich zur letztgenannten Methode mit der Methode stringToObject auch die Konvertierung von String in den Objekttyp bereitgestellt werden. Dieses Interface ist erforderlich, wenn eine Brücke auf einem Property angewendet wird, das mit der Annotation @DocumentId deklariert ist. Die oben genannten Hibernate-Bridges implementieren alle dieses Interface.

- Mit dem Interface `ParameterizedBridge` können der selbst entwickelten Adapter-Klasse noch Parameter zur Umwandlung mitgegeben werden.
- Die Interfaces `FieldBridge` und analog dazu `TwoWayFieldBridge` erlauben den direkten Zugriff auf das zugrunde liegende Lucene-`Document`. Damit lassen sich Felder zu einem Dokument hinzufügen, die im entsprechenden Entity nicht vorhanden sind. Aus einem `Date`-Attribut kann man mit diesem Interface z. B. drei einzelne Lucene-Felder Jahr, Monat, Tag anlegen. Weitere Hinweise finden sich auf der Lucene-Homepage [URL-LUC] sowie der Dokumentation zu *Hibernate-Search* [URL-HIB].

Wir wollen nachfolgend die Konvertierung des Property `geburtsdatum` der Klasse `Kunde` anpassen. Dazu implementiert die Klasse die Interfaces `StringBridge` und `ParameterizedBridge`.

```
public class DateFormatterBridge implements StringBridge,
    ParameterizedBridge {
  public static final String DATE_FORMAT = "dateFormat";
  private String simpleDateFormat;

  public String objectToString(Object object) {
    SimpleDateFormat sdf = new SimpleDateFormat(simpleDateFormat);
    return sdf.format(object);
  }

  public void setParameterValues(Map m) {
    Object dateFormat = m.get(DATE_FORMAT);
    if (dateFormat != null) {
      this.simpleDateFormat = (String) dateFormat;
    }
  }
}
```

Die Konfiguration unseres `Kunde`-Entity mit Index-Informationen sieht wie folgt aus. Auf die Darstellung der `Adresse`-Klasse verzichten wir an dieser Stelle. Auch bei ihr haben wir die Properties so konfiguriert, dass sie mit indiziert werden.

```
@Entity
// Default-Index des Properties aus persistence.xml!
@org.hibernate.search.annotations.Indexed
public class Kunde {
  @Id
  @GeneratedValue
  @org.hibernate.search.annotations.DocumentId
  private int id;
  @org.hibernate.search.annotations.Field
  private Integer kundennummer;
  @org.hibernate.search.annotations.Field
  private String vorname;
  @org.hibernate.search.annotations.Field
  private String nachname;
  @Temporal(TemporalType.DATE)
  @org.hibernate.search.annotations.Field(
      index = Index.NO,
      store = Store.YES)
  @org.hibernate.search.annotations.FieldBridge(
      impl = de.jpainfo.bridge.DateFormatterBridge.class,
      params = { @org.hibernate.search.annotations.Parameter(
```

```
            name = "dateFormat", value = "dd.MM.yyyy") })
    private Date geburtsdatum;
    private Character geschlecht;
    @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    @JoinColumn(name = "WOHNORT_ID")
    @IndexedEmbedded(depth = 1)
    private Adresse wohnort;
    @ManyToMany(cascade = CascadeType.ALL)
    private Set<Konto> konten = new HashSet<Konto>();

    // weitere Getter und Setter Methoden ...
}
```

Mit dem Programm aus Listing 11.5 auf Seite 287 wird der Index erneut erstellt. Abbildung 11.2 zeigt das Ergebnis. Wie der Leser erkennen kann, hat unsere Umwandlung des

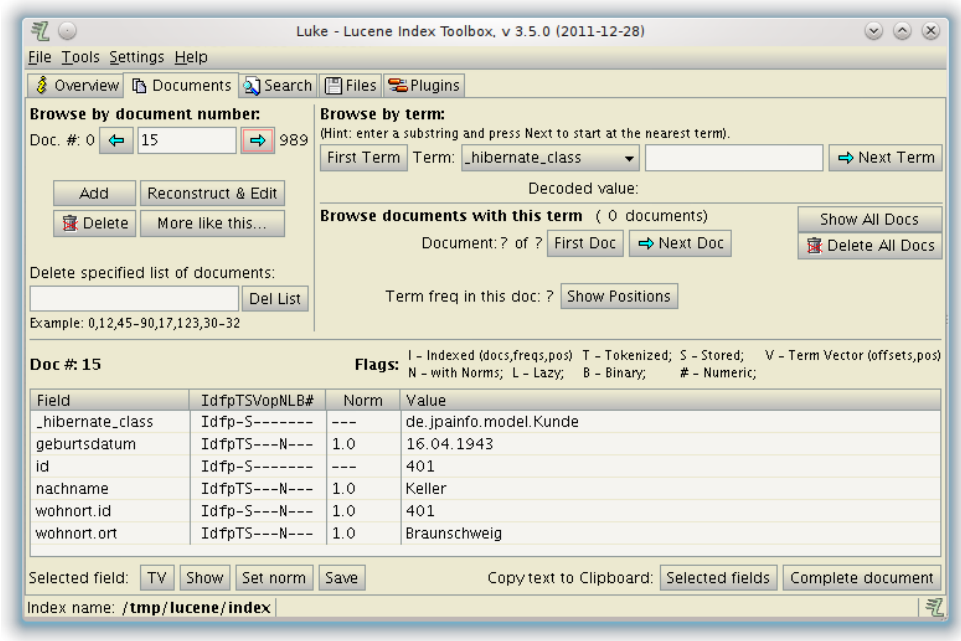


ABBILDUNG 11.2 Lucene-Index der Klasse `Kunde` (2)

Geburtsdatums funktioniert. Des Weiteren finden sich im Index auch die Attribute des zum Kunden gehörenden `Adresse`-Entity. Im dargestellten Beispiel wurden der Nachname, das Geburtsdatum und der Wohnort im Index gespeichert. Dies ist zur Visualisierung von Suchergebnissen ohne erneute Datenbankabfrage ein durchaus sinnvoller Vorgang.

11.3.4 Abfragen mit der Lucene-Syntax

Die Ausführung einer Lucene-Abfrage fügt sich nahtlos in den bekannten JPA-Ablauf ein. Im Listing 11.5 auf Seite 287 wurde bereits gezeigt, wie man mit JPA-konformer Konfiguration zu einer Instanz der Klasse `FullTextSession` gelangt. Wir führen nun eine Abfrage aus:

```
FullTextEntityManager fullTextEntityManager = FullTextEntityManager(
    em);
QueryBuilder b = fullTextEntityManager.getSearchFactory()
    .buildQueryBuilder().forEntity(Kunde.class).get();
//Abfrage mit Wildcards
org.apache.lucene.search.Query luceneQuery = b.keyword()
    .wildcard()
    .onField("nachname")
    .matching("*ei*")
    .createQuery();
//JPA-Query
Query fullTextQuery = fullTextEntityManager.createFullTextQuery(
    luceneQuery);
List result = fullTextQuery.getResultList();
```

In unserem Beispielprogramm haben wir bewusst nicht die Lucene-Klassen in die Import-Anweisungen aufgenommen, sondern weisen den vollen Klassennamen im Programm aus. Damit ist ersichtlich, an welchen Stellen jeweils Lucene-spezifische Klassen zum Einsatz kommen. Auf weitere Möglichkeiten der Lucene-Syntax für Abfragen wollen wir an dieser Stelle nicht eingehen, sondern verweisen auf die Lucene-Dokumentation [URL-LUC] und auch die Dokumentation zu Hibernate-Search [URL-HIB].

11.4 Envers

In vielen betrieblichen Bereichen darf die Aktualisierung von Daten nicht zu einem Überschreiben der alten Daten führen, da diese aus Revisionsgründen erhalten bleiben müssen. Man spricht dann von einer *historischen Datenverarbeitung* oder einer *Auditierung* der (alten) Daten. Es entstehen *Revisionen* von Daten. Mögliche Realisierungsalternativen sind Listener auf Programmiersprachenebene oder Trigger auf Datenbankebene, die gewöhnliche Datenänderungen in entsprechende zusätzliche Auditierungsdaten umwandeln. Mit JPA wären damit Entity-Listener, wie wir sie in Abschnitt 3.4 vorgestellt haben, das Mittel der Wahl. Das Projekt Envers (*En*tity *Ver*sioning) realisiert eine Auditierung von Entities mit Hilfe von Entity-Listenern und nimmt uns damit die entsprechende Eigenentwicklung ab.

Envers ist seit Version 3.6 fester Bestandteil von Hibernate, so dass die entsprechende Bibliothek nicht separat heruntergeladen werden muss. Seit Version 4 ist selbst die Konfiguration der Entity-Listener nicht mehr nötig, so dass ohne zusätzlichen Konfigurationsaufwand sofort mit der Entwicklung einer Envers-basierten Anwendung begonnen werden kann.

11.4.1 Einfache Auditierung

Um Daten zu auditieren, stellt Envers die Annotation `@Audited` aus dem Package `org.hibernate.envers` bereit. Alle weiteren Envers-Artefakte sind ebenfalls in diesem Package und weiteren Subpackages zu finden. Falls alle Properties eines Entity auditiert werden sollen, so ist das Entity selbst zu annotieren. Alternativ sind die jeweils zu auditierenden Properties mit der Annotation zu versehen. Wir beginnen mit einem Beispiel, bei dem alle Daten eines Kunden, in diesem Fall lediglich Vor- und Nachname, auditiert werden sollen.

```
@Entity
@Audited
public class Kunde {

  @Id @GeneratedValue
  private Integer id;
  private String vorname;
  private String nachname;
  ...
}
```

Das zugrunde liegende Tabellenschema kann seit Hibernate 4.0 direkt mit dem entsprechenden Hibernate-Generierungswerkzeug erstellt werden. Eine Verwendung des in früheren Versionen benötigten Envers-spezifischen Generierungswerkzeugs ist nicht mehr notwendig. Die generierte Tabellenstruktur zeigt Abbildung 11.3.

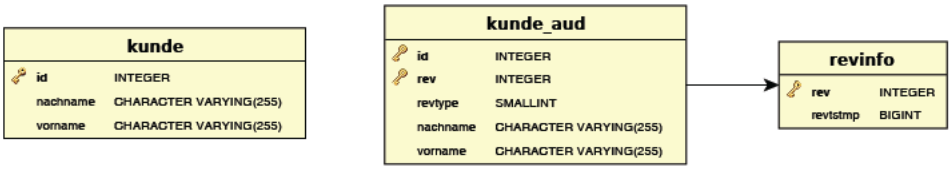


ABBILDUNG 11.3 Tabellenstruktur für die auditierte Kundenklasse

Man erkennt in der Tabelle `kunde` die typische JPA-Struktur. Die Tabelle `kunde_aud` spiegelt die Nutzdaten `vorname` und `nachname` wider und enthält mit `rev` und `revtype` zwei Spalten zur Envers-internen Verwaltung der auditierten Daten. Das Auditieren von Daten geschieht revisionsbasiert. Eine Revision ist der Gesamtzustand aller Daten zu einem bestimmten Zeitpunkt. Der Zeitpunkt ist über eine Datenbanktransaktion definiert, wird als Ganzzahl repräsentiert und pro Transaktion inkrementiert. Diese Revision wird in der Spalte `rev` vermerkt. Die Art der revisionierten Daten, also ob das Entity initial eingefügt, geändert oder gelöscht wurde, wird in der Spalte `revtype` vermerkt. Der Wert entspricht den Werten der Enumeration `RevisionType`: `ADD` (0), `MOD` (1) und `DEL` (2).

Hinweis:

Die Spalte `id` der Tabelle `kunde_aud` ist prinzipiell der Fremdschlüssel in die Tabelle `kunde`. Diese Bedingung lässt sich jedoch nicht in SQL formulieren, da es eine Ausnahme gibt: Wenn ein Kunde gelöscht wird, bestehen seine auditierten Daten weiterhin, wie wir gleich sehen werden.

Die Tabelle `kunde` enthält ausschließlich die (zu einem bestimmten Zeitpunkt) aktuellen Daten. Die Tabelle `kunde_aud` enthält die nach jeder Transaktion gültigen Daten in akkumulierter Form, d. h. die Sammlung aller (irgendwann einmal aktuellen) Daten. Zur Verdeutlichung sind in Tabelle 11.6 die Daten der Tabelle `kunde_aud` für einen einzelnen Kunden angegeben.

Die Tabelle ist so zu interpretieren, dass Vor- und Nachname des Kundenobjekts initial mit `"Albert"` und `"Einstein"` vorbelegt sind. Mit dem Commit der jeweiligen Transaktion für den Java-Ausdruck in der Tabellenspalte `Java` werden die Daten für die Spalten `vorname`, `nachname`, `rev` und `revtype` eingefügt. Die Spalte `id` entspricht der Id des Kunden und

TABELLE 11.6 Revisionsdaten der Tabelle `kunde_aud`

Java	id	vorname	nachname	rev	revtype
`em.persist(kunde)`	4711	Albert	Einstein	22718	0
`kunde.setNachname("Zweistein")`	4711	Albert	Zweistein	23111	1
`kunde.setNachname("Dreistein")`	4711	Albert	Dreistein	45733	1
`em.remove(kunde)`	4711	null	null	61187	2

damit der Spalte `id` der Tabelle `kunde`. Die Daten in der Tabelle `kunde` entsprechen den jeweils neu eingefügten Nutzdaten der jeweiligen Zeile in Tabelle 11.6. Der Leser möge sich jedoch verdeutlichen, dass in der Tabelle `kunde` nur eine einzige Zeile für den Kunden mit id 4711 bzw. nach dem Löschen des Kunden auch diese Zeile nicht mehr existiert.

Die Tabelle `revinfo` ordnet jeder Revisionsnummer einen Zeitstempel zu. Dieser entspricht dem internen Format der Klasse `java.util.Date`, also der Anzahl Millisekunden seit dem 1.1.1970 Mitternacht.

Das Lesen revisionierter Entities gestaltet sich naturgemäß etwas umfangreicher als das Lesen einfacher JPA-Entities, da ein Entity in mehreren Revisionen existieren kann. Envers realisiert dabei sowohl das direkte Lesen von Entities analog zu JPAs `find()`-Methode als auch das Lesen über ein Query-Interface. Der Einstiegspunkt für das Lesen revisionierter Entities ist das Interface `AuditReader`. Eine Instanz des Interfaces erhält man über die `get()`-Methode der `AuditReaderFactory`, die als Parameter einen JPA-Entity-Manager oder eine Hibernate-Session benötigt. In Fortführung des obigen Beispiels kann der Kunde mit Primärschlüssel 4711 zur Revision 22718 wie folgt gelesen werden.

```
EntityManager em = ...
AuditReader auditReader = AuditReaderFactory.get(em);
Kunde k = auditReader.find(Kunde.class, 4711, 22718);
```

Der Nachname zu dieser Revision ist `Einstein`, für die Revisionen 23111 und 45733 entsprechend `Zweistein` und `Dreistein`. Für die Revision 61187 liefert die `find()`-Methode `null` als Kunden zurück. Für Revisionen, an denen das gesuchte Entity nicht aktualisiert worden ist, etwa 22912, ist der Datensatz mit der nächstkleineren Revision gültig. Im Beispiel wäre dies 22718.

Das Lesen aller Revisionen eines Entity, aller Entities einer Revision oder aller Entities, die an einer Revision modifiziert wurden, ist ebenfalls möglich und führt über das schon erwähnte Query-Interface. Envers-Queries werden durch das Interface `AuditQuery` realisiert. Die Methode `createQuery()` des `AuditReaders` erzeugt zunächst eine Instanz von `AuditQueryCreator`, die über die Methoden

- `forEntitiesAtRevision()`
- `forEntitiesModifiedAtRevision()`
- `forRevisionsOfEntity()`

letztendlich eine Instanz von `AuditQuery` zurückgeben. Wir beschränken uns hier auf die Revisionen des Entity `Kunde`:

```
AuditQuery query = auditReader.createQuery()
                   .forRevisionsOfEntity(Kunde.class, true, true);
query.add(AuditEntity.id().eq(4711));
List list = query.getResultList();
```

Die Methode `forRevisionsOfEntity()` liefert *alle* Instanzen für den im ersten Parameter angegebenen Entity-Typ. Falls nicht alle Instanzen gesucht sind, ist die Query über eine Projektion weiter einzuschränken. Hierzu verwendet Envers ein API, das dem Hibernate-Criteria-API nachempfunden ist. Dies ähnelt JPAs Criteria-API, das wir in Kapitel 8 beschrieben haben. Im Beispiel wird über `AuditEntity.id()` die Id als Kriterium bestimmt und 4711 als entsprechender Wert gesetzt. Die Ausführung der Query über `getResultList()` liefert nun eine Menge von Kundenobjekten, die den Revisionen in Tabelle 11.6 entsprechen. Im Gegensatz zu unserem Beispiel mit `find()` wird für die 87347. Revision eine Kundeninstanz geliefert, deren Vor- und Nachname `null` ist.

Hinweis:

Wir haben im Beispiel nicht freiwillig, sondern gezwungenermaßen auf Generizität verzichtet. Die Umstellung des Envers-APIs auf generische Typen ist noch nicht vollständig erfolgt bzw. prinzipiell nicht möglich, so dass eine entsprechende Verwendung, im Beispiel etwa von `List<Kunde>` nur unter Verzicht auf die entsprechende Typsicherheit erfolgen kann.

Ein Grund für das nichtgenerische Query-Interface ist die Möglichkeit, den Rückgabetyp der Methode `forRevisionsOfEntity()` zu steuern. Der zweite Parameter der Methode gibt an, ob der Elementtyp der Rückgabeliste eine Revision des Entity selbst (Parameter = `true`) oder ein dreielementiges Array aller Revisionsdaten (Parameter = `false`) ist. Im Falle des Arrays ist das erste Array-Element das Entity, das zweite eine `DefaultRevisionEntity`-Instanz und das dritte ein Wert der Enumeration `RevisionType`. Das `DefaultRevisionEntity` repräsentiert das JPA-Entity für die Tabelle `revinfo` und enthält somit den Zeitstempel der Revision. Anwendungsdefinierte Revisions-Entities sind möglich, werden von uns aber nicht weiter beschrieben. Der `RevisionType` zeigt an, ob die Revision durch Einfügen, Ändern oder Löschen entstand.

Der dritte Parameter der Methode `forRevisionsOfEntity()` legt fest, ob auch gelöschte Entities in der Antwort enthalten sein sollen. Bei `false` und der Revision 87347 wäre das leere Kunde-Entity nicht in der Ergebnismenge.

11.4.2 Auditieren von Beziehungen

Envers unterstützt auch das Auditieren von Beziehungen. Wir demonstrieren dies am Beispiel einer 1:1-Beziehung zwischen den Entities `Kunde` und `Adresse`, wie wir sie in Abschnitt 4.1 eingeführt haben. Wir annotieren wie im letzten Beispiel das Entity `Kunde` mit `@Audited`, so dass alle Properties – und damit auch das Beziehungs-Property – auditiert werden. Auch das Ziel der Beziehung, in unserem Fall das Entity `Adresse`, ist dann zu auditieren.

```
@Entity
@Audited
public class Kunde {

  @Id @GeneratedValue
  private Integer id;
  private String vorname;
  private String nachname;

  @OneToOne
  private Adresse adresse;
  ...
```

```
@Entity
@Audited
public class Adresse {

  @Id @GeneratedValue
  private Integer id;
  private String strasse;
  private String hausnummer;
  private String plz;
  private String ort;
  ...
```

Soll die Adresse nicht auditiert werden, so ist das Beziehungsziel explizit von der Auditierung auszuschließen, und zwar über das Attribut `targetAuditMode`:

```
@Entity
@Audited
public class Kunde {
  ...
  @OneToOne
  @Audited(targetAuditMode = RelationTargetAuditMode.NOT_AUDITED)
  private Adresse adresse;
  ...
```

Die Tabellenstruktur für auditierte Entities und die auditierte Beziehung ist in Abbildung 11.4 dargestellt.

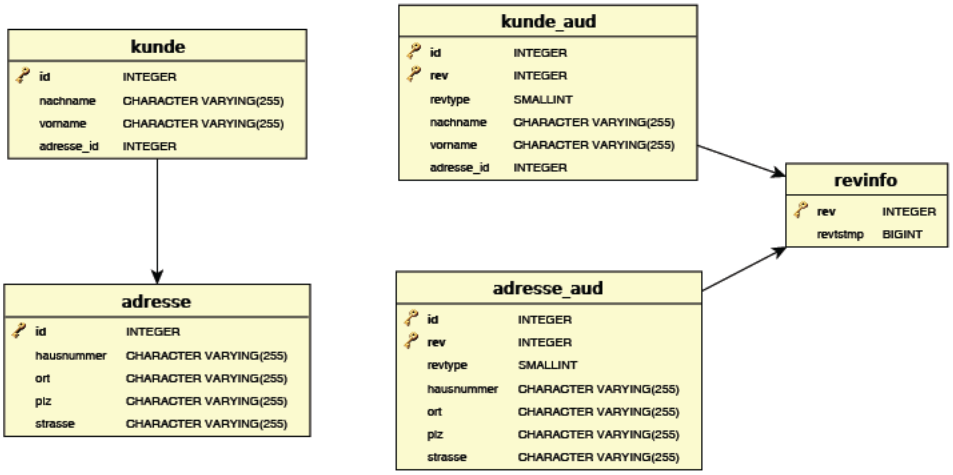


ABBILDUNG 11.4 Tabellenstruktur für auditierte Kunden und Adressen

11.4.3 Typische Anwendungsfälle

In Abschnitt 11.4.1 haben wir gezeigt, wie über eine bekannte Revisionsnummer der zu diesem Zeitpunkt gültige Zustand eines Entity ermittelt werden kann. Dies entspricht jedoch

keinem der typischen Anwendungsfälle, etwa dass ein Anwender Informationen über einen Kunden für einen bestimmten Zeitpunkt (Datum oder Datum mit Uhrzeit) benötigt. Envers stellt über das Interface `AuditReader` Methoden zur Verfügung, die in diesem Zusammenhang nützlich sind:

- `Date getRevisionDate(Number revision)`
- `Number getRevisionNumberForDate(Date date)`
- `List<Number> getRevisions(Class<?> cls, Object primaryKey)`

Die Methode `getRevisionDate()` liefert für eine gegebene Revisionsnummer den zugehörigen Zeitpunkt. Die Methode `getRevisionNumberForDate()` stellt die inverse Operation dar und liefert für einen Zeitpunkt die gegebene Revisionsnummer. Mit dieser Revisionsnummer kann dann analog zum Beispiel aus Abschnitt 11.4.1 das entsprechende Entity gelesen werden:

```
Date date =  ... // vom Benutzer erfragt
Number rev = auditReader.getRevisionNumberForDate(date);
Kunde k = auditReader.find(Kunde.class, 4711, rev);
```

Die Methode `getRevisions()` liefert für einen Entity-Typ und den Primärschlüssel die Liste aller Revisionsnummern, an denen sich der Zustand dieses Entity geändert hat. Mit Hilfe der Methode `getRevisionDate()` können diese Revisionsnummern in Zeitpunkte umgewandelt und dem Benutzer angezeigt werden.

Eine weitere, durchaus realistische Anforderung ist die Frage nach *allen* Änderungen zu einem bestimmten Zeitpunkt. Bei n Entity-Klassen bedeutet dies in der Realisierung auch n Select-Anweisungen der Datenbank. Seit Version 4.0 von Hibernate unterstützt Envers eine derartige Abfrage und kann die Anzahl der Select-Anweisungen auf k begrenzen, wobei k die Anzahl der Entity-Klassen ist, deren Instanzen tatsächlich geändert wurden. Der Schlüssel hierzu ist eine zusätzliche Tabelle `revchanges` mit den Spalten `rev` und `entityname`. In dieser Tabelle werden Revisionsnummern und voll qualifizierte Klassennamen der Entity-Klassen gespeichert, deren Instanzen zur jeweiligen Revisionsnummer verändert wurden. Die konkrete Anzahl von Select-Anweisungen ist damit $k+1$: ein Select über die Tabelle `revchanges` mit anschließenden k Selects über die entsprechenden Entity-Tabellen.

Um die beschriebene Möglichkeit zur Suche aller geänderten Entities nutzen zu können, muss das Property `org.hibernate.envers.track_entities_changed_in_revision` auf `true` gesetzt werden. Die automatische Tabellengenerierung erzeugt dann die Tabelle `revchanges`. Über den `AuditReader` erhält man eine Instanz des Interface `CrossTypeRevisionChangesReader`, das die folgenden Methoden bereitstellt:

- `List<Object> findEntities(Number revision)`
- `List<Object> findEntities(Number revision, RevisionType type)`
- `Map<RevisionType,List<Object>>`
  `findEntitiesGroupByRevisionType(Number revision)`

Diese Methoden erzeugen folgende Rückgabewerte:

- Liste aller in einer Revision geänderten Entities
- Liste aller in einer Revision auf eine bestimmte Art (eingefügt, aktualisiert, gelöscht) geänderten Entities
- Map (gruppiert über Änderungsart) aller in einer Revision geänderten Entities

Wir verwenden im Beispiel die erste Methode und suchen alle Entities, die an einem bestimmten Zeitpunkt geändert wurden:

```
Date date =  ... // vom Benutzer erfragt
Number rev = auditReader.getRevisionNumberForDate(date);
CrossTypeRevisionChangesReader reader =
      auditReader.getCrossTypeRevisionChangesReader();
List<Object> changedEntities = reader.findEntities(rev);
for (Object object : changedEntities) {
  ... // Anzeige/Verarbeitung der geaenderten Entities
```

Achtung:

Als Entwickler sollte man sich bei der Verwendung der Methode `findEntities()` (in den beiden überladenen Versionen) sowie der Methode `findEntitiesGroupByRevisionType()` darüber im Klaren sein, dass diese „recht harmlos" aussehenden Methoden $k+1$ bzw. $3k+1$ Select-Anweisungen in der Datenbank ausführen, was bei einer großen Anzahl geänderter Entities zu Performanz-Problemen führen kann.

11.4.4 Die Auditierungsstrategie

Die bisher beschriebene und in Tabelle 11.6 veranschaulichte Art der Darstellung auditierter Daten ist die Envers-Default-Auditierungsstrategie. Hierbei wird für die Gültigkeit der Daten in der Audit-Tabelle mit der Revisionsnummer der Startzeitpunkt definiert. Das Ende der Gültigkeit wird über die Revisionsnummer des nächsten Datensatzes für dieselbe Id angegeben. Envers-Auditierungsstrategien werden über das Interface `AuditStrategy` im Package `org.hibernate.envers.strategy` definiert. Die beschriebene Default-Strategie ist durch die Klasse `DefaultAuditStrategy` im selben Package realisiert.

Eine alternative Auditierungsstrategie basiert auf Start- und End-Revisionen und ist über die Klasse `ValidityAuditStrategy` realisiert. Die Auditierungstabellen enthalten bei dieser Strategie sowohl eine Startrevisionsnummer als auch eine Endrevisionsnummer. Zur Verdeutlichung ist in Tabelle 11.7 die Tabelle `kunde_aud` für diese Auditierungsstrategie dargestellt. Diese entspricht in ihren Daten der Tabelle 11.6 auf Seite 295 für die Default-Strategie. Die Vornamenspalte wurde aus Platzgründen nicht dargestellt.

Achtung:

Im Package `org.hibernate.envers.strategy` ist noch eine dritte Auditierungsstrategie enthalten, die `ValidTimeAuditStrategy` . Diese ist mit `@Deprecated` gekennzeichnet und sollte nicht mehr benutzt werden.

Eine Analyse und die Lektüre der Envers-Dokumentationen macht folgende Unterschiede deutlich: Bei der `DefaultAuditStrategy` werden bei der Aktualisierung eines Entities eine Zeile in die Audit-Tabelle geschrieben, die die Startrevisionsnummer enthält. Diese Zeile wird niemals aktualisiert. Das Lesen von Audit-Daten benötigt eine Sub-Query, um die Zeile mit der größten Revisionsnummer zu finden, die kleiner oder gleich der gesuchten Revisionsnummer ist. Strukturell ergibt sich für das Beispiel eine Query der Art:

TABELLE 11.7 Revisionsdaten (Tabelle `kunde_aud`) mit `ValidityAuditStrategy`

Java	id	nachname	rev	revend	revtype
`em.persist(kunde)`	4711	Einstein	22718	23111	0
`kunde.setNachname("Zweistein")`	4711	Zweistein	23111	45733	1
`kunde.setNachname("Dreistein")`	4711	Dreistein	45733	61187	1
`em.remove(kunde)`	4711	null	61187	null	2

```
Select ... from kunde_aud kunde_aud0
Where kunde_aud0.rev =
  Select max (kunde_aud1.rev) from kunde_aud kunde_aud1
  Where kunde_aud1.rev <= ? and ...
```

Bei der `ValidityAuditStrategy` werden bei der Aktualisierung eines Entity eine Zeile in die Audit-Tabelle geschrieben, die ebenfalls die Startrevisionsnummer enthält. Zusätzlich wird jedoch die Endrevisionsnummer in der vorhergehenden Revisionszeile für dieses Entity auf die Startrevisionsnummer der eingefügten Zeile aktualisiert. Das Lesen von Audit-Daten erfolgt über eine einfache Query der Art:

```
Select ... from kunde_aud kunde_aud0
Where kunde_aud0.rev <= ?  ...
      and (kunde_aud0.revend > ? or kunde_aud0.revend is null)
```

Vergleichend lässt sich sagen, dass die `ValidityAuditStrategy` gegenüber der `DefaultAuditStrategy` beim Einfügen von Daten etwas langsamer, beim Lesen von Daten deutlich schneller ist. Die Dokumentation spricht von „a lot faster“.

Die Auditierungsstrategie ist ein reines Implementierungsdetail, wenn auch kein unwichtiges. Das API ändert sich dadurch nicht, so dass auf Anwendungsebene keine Code-Änderungen vorgenommen werden müssen. Die Konfiguration der Auditierungsstrategie erfolgt über das Property `org.hibernate.envers.audit_strategy`, das als Wert den qualifizierten Klassennamen der gewählten Strategie erhält.

```
<property name="org.hibernate.envers.audit_strategy"
       value="org.hibernate.envers.strategy.ValidityAuditStrategy" />
```

Projekt:

Das Projekt *envers* enthält den im Zusammenhang mit Hibernate Envers vorgestellen Code.

12 EclipseLink-Erweiterungen

12.1 Caching

EclipseLink bietet mit der Annotation `@org.eclipse.persistence.annotations.Cache` die Möglichkeit, das Cache-Verhalten der Implementierung wesentlich feiner zu steuern, als es die Spezifikation selbst zulässt. Wie bei der `@Cachable`-Annotation wird die `@Cache`-Annotation auf Klassenebene verwendet und hat die nachfolgend aufgeführten Attribute.

`alwaysRefresh`
: Alle Abfragen führen zu einer Aktualisierung des Cache, wenn dieses Attribut auf `true` gesetzt wird.

`coordinationType`
: Um Änderungen von Objekten zu kommunizieren, wenn die Anwendungsarchitektur auf verteilte Caches setzt, können die folgenden Einstellungen der Enum `CacheCoordinationType` benutzt werden:

- `INVALIDATE_CHANGED_OBJECTS`: Die empfangenden Caches werden die enthaltenen Objekte invalidieren.
- `SEND_OBJECT_CHANGES`: Änderungen von Objekten werden kommuniziert und mit den geänderten Daten an die beteiligten Caches verteilt (Default-Einstellung).
- `SEND_NEW_OBJECTS_WITH_CHANGES`: Änderungen und auch neue Objekte werden allen Caches bekannt gegeben.

`disableHits`
: Alle Abfragen gehen zur Datenbank der Anwendung, wenn dieser Schalter aktiviert wird. Die Daten des Entity werden aus der Datenbank geholt. Objektidentität wird mit dem Cache abgeglichen.

`expiry`
: Zeit in Millisekunden, nach der die zwischengespeicherte Instanz aus dem Cache entfernt wird. Abfragen danach gehen zunächst wieder zur Datenbank, um eine aktualisierte Version des Entity zu lesen.

`expiryTimeOfDay`
: Ablauf der zwischengespeicherten Instanz zu einer bestimmten Tageszeit.

`isolation`
Diese Einstellung bestimmt, ob ein Entity im globalen Cache eines Persistenzkontextes gehalten wird oder isoliert von anderen Entity-Instanzen. Zur Verfügung stehen die folgenden Möglichkeiten der Enum `CacheIsolationType`:

- `SHARED`: Ablage im Cache der `EntityManagerFactory`, also der Persistenzeinheit.
- `PROTECTED`: Entity-Instanzen werden nicht im Cache zum gemeinsamen Zugriff zur Verfügung gestellt, die Daten des Entity werden jedoch im Cache gehalten.
- `ISOLATED`: Keine Ablage im globalen Cache der Persistenzeinheit.

`refreshOnlyIfNewer`
Der Cache wird aktualisiert, wenn nach einer Abfrage auf der Datenbank eine neuere Version des Entity vorliegt. Bestimmt wird dies durch den Zähler der Entity-Version (vgl. Seite 157).

`size`
Die Größe des Cache wird nach der Anzahl der Objekte bemessen. Die Grundeinstellung ist 100. Bei den Cache-Typen `FULL` und `WEAK` wird damit die initiale Größe der Map bestimmt, bei den anderen Typen wird die Einstellung für die maximale Größe der Map benutzt.

`type`
Die möglichen `type`-Attribute beeinflussen während der Garbage Collection, ob die Entities aus dem Cache entfernt werden oder nicht. EclipseLink bedient sich dabei der Möglichkeit sogenannter *Weak-* bzw. *Soft-References*. Dies sind besondere Referenzen auf Java-Objekte, die es seit Java 1.2 im Sprachumfang gibt. Die Enum `CacheType` besitzt folgende Ausprägungen:

- `FULL`: Alle Objekte werden im Cache gehalten und werden aus diesem auch nicht mehr entfernt.
- `SOFT`: Verhält sich wie `FULL`. Wenn der Arbeitsspeicher knapp wird, können Entities aus dem Cache über die Garbage Collection entfernt werden.
- `WEAK`: Verhält sich wie `FULL`. Unabhängig vom Zustand des Arbeitsspeichers können Objekte während der Garbage Collection gelöscht werden.
- `SOFT_WEAK`: Verhält sich wie `WEAK`, benutzt aber einen Sub-Cache, der Objekte enthält, die sehr oft benutzt werden. Entfernt werden Objekte erst dann, wenn der Arbeitsspeicher knapp wird. Dies ist der Default, wenn der Typ nicht explizit gesetzt wird.
- `HARD_WEAK`: Verhält sich wie `SOFT_WEAK`, die Objekte werden aber *hart* referenziert (*Hard References*), um ein Löschen durch die Garbage Collection zu verhindern.
- `CACHE`: Diese Einstellung speichert eine feste Anzahl von Objekten im Cache. Objekte werden nach dem Least-Recently-Used-Prinzip aus dem Zwischenspeicher gelöscht. Eindeutige Objekt-Identität ist mit diesem Verfahren nicht sichergestellt.
- `NONE`: Das Caching ist deaktiviert.

Natürlich sind auch globale Einstellungen in der Persistenzeinheit selbst möglich. Die wichtigsten Parameter dazu haben wir in der folgenden Übersicht zusammengestellt. Auf die Erklärung der Parameter verzichten wir an dieser Stelle. Sie lassen sich leicht aus der obigen Übersicht der `@Cache`-Annotation herleiten:

```
eclipselink.cache.type.default
eclipselink.cache.size.default
eclipselink.cache.shared.default
eclipselink.cache.shared.<ENTITY>
eclipselink.cache.type.<ENTITY>
eclipselink.cache.size.<ENTITY>

// Beispiel Cache-Size per Entity:
<property name="eclipselink.cache.size.Kunde" value="100"/>
```

Achtung:

Bei EclipseLink ist die Nutzung des Second-Level-Cache standardmäßig aktiviert! Im Gegensatz dazu erfordern die Implementierungen von Hibernate und OpenJPA für die Cache-Benutzung weitere Konfigurationen. Der L2-Cache kann mit `<shared-cache-mode>NONE</shared-cache-mode>` deaktiviert werden.

12.1.1 Objekt-Cache

Wie bereits in Abschnitt 6.3.5 demonstriert, werden bei der Benutzung der Methoden `find()` bzw. `refresh()` die betroffenen Objekte aus dem Cache abgefragt bzw. bei Nichtvorhandensein in den Cache überführt. EclipseLink bietet darüber hinaus die folgenden Query-Hints an, die wir nach JPA-Standard als herstellerspezifische Parameter einer Abfrage mitgeben können. In vielen Beispielen der Dokumentation wird der Parameter dabei direkt als String übergeben, was manchmal für Verwirrung sorgt. Die beiden folgenden Code-Zeilen sind identisch und sollen dem Leser nur die verschiedenen Wege zum gleichen Ergebnis darstellen.

```
query.setHint("eclipselink.cache-usage", "CheckCacheThenDatabase");
query.setHint(QueryHints.CACHE_USAGE, CacheUsage.
    CheckCacheThenDatabase);
```

Alle Einstellungen gehen von sogenannten Read-Object- bzw. Read-All-Queries aus. EclipseLink benutzt den Cache also nur, wenn *ganze* Objekte selektiert werden. Werden mit einer Query einzelne Attribute selektiert, benutzt EclipseLink intern eine so genannte Report-Query. Der Cache wird dabei nicht benutzt. Die Typisierung der Queries in Read-Object-, Read-All- und Report-Query ist eine Besonderheit von EclipseLink und hat keinen Bezug zum JPA-Standard.

Achtung:

EclipseLink reagiert auf die unterschiedlichen Query-Hints der Cache-Benutzung teilweise überraschend. Benutzt man bei einer Report-Query den Hinweis `CheckCacheOnly`, wird bei Erzeugung der Query eine `QueryException` geworfen. Diese Ausnahme war zu erwarten, da im Cache niemals ein Ergebnis zu einer Report-Query zu finden sein wird. Benutzt man dagegen bei einer Read-All-Query den Parameter `CheckCacheByPrimaryKey`, wird nur ein Objekt zurückgegeben, obwohl die Datenbank eventuell mehrere zu den Selektionskriterieren passende Entities enthält.

TABELLE 12.1 `QueryHints.CACHE_USAGE` (`eclipselink.cache-usage`)

Attribut von `CacheUsage`	Bedeutung
`DoNotCheckCache`	Die Query benutzt immer die Datenbank und nicht den Cache.
`CheckCacheByExactPrimaryKey`	Wenn eine Query als Vergleichsargument nur den Primärschlüssel enthält, kann das Objekt mit diesem Hinweis direkt aus dem Cache gelesen werden. Diese Einstellung kann man nur benutzen, wenn die Query genau ein Ergebnis zurückliefert.
`CheckCacheByPrimaryKey`	Enthält die Query unter anderem den Primärschlüssel als Vergleich, wird der Cache mit diesem Hinweis benutzt. Diese Einstellung kann nur benutzt werden, wenn die Query genau ein Ergebnis zurückliefert.
`CheckCacheThenDatabase`	Wenn gesetzt, wird zunächst der Cache durchsucht, dann wird weiter zur Datenbank gegangen. Diese Einstellung kann nur benutzt werden, wenn die Query genau ein Ergebnis zurückliefert.
`CheckCacheOnly`	Es wird nur der Cache gelesen. Es erfolgt kein Datenbankzugriff!
`ConformResultsInUnitOfWork`	Diese Option ist gültig für Read-Object- und Read-All-Queries: Noch nicht in die Datenbank geschriebene Objekte werden bei Abfragen der gleichen Session ebenfalls in das Ergebnis gestellt und zurückgeliefert.
`UseEntityDefault`	Mit dieser Einstellung wird die Cache-Konfiguration herangezogen, die der EclipseLink-Descriptor-API für das Entity entspricht.

Führt EclipseLink eine Abfrage nach einem einzelnen Objekt aus, wird die Suchbedingung aus der Abfrage extrahiert und mit diesem Schlüssel zunächst der Cache durchsucht. Ist das Objekt im Cache nicht zu finden, geht die Abfrage weiter zur Datenbank. Das Ergebnis wird mit der Abfrage zurückgeliefert und ebenfalls im Cache hinterlegt.

Bei einer Abfrage nach mehreren Objekten, die durch keine Selektions-Einschränkung bzw. durch eine Selektion mit Nicht-Primärschlüsseln erfolgt, wird direkt die Datenbank abgefragt. Da EclipseLink über die Meta-Informationen die Schlüsselattribute kennt, werden die Einträge des Result-Sets mit denen des Caches verglichen. Bei Übereinstimmung wird das Objekt mit den Daten aus dem Cache instanziiert, sonst mit den Werten aus der Datenbank.

Tabelle 12.2 zeigt die spezifischen Query-Hints, um den Typ einer Abfrage entsprechend zu beeinflussen.

TABELLE 12.2 `QueryHints.QUERY_TYPE` (`eclipselink.querytype`)

Attribut von `QueryType`	Bedeutung
`AUTO`	EclipseLink bestimmt den Typ einer Abfrage selbst.
`ReadAll`	Eine Read-All-Query wird benutzt, um eine Collection von Entities entsprechend den Selektionsbedingungen zu lesen.
`ReadObject`	Abfrage der Datenbank, bei der nur das erste Objekt zurückgegeben wird, das den Selektionsbedingungen entspricht.
`Report`	Eine Report-Query liefert bestimmte Attribute der Entities, Aggregationen von Werten, aber auch mehrere Objekte pro zurückgegebener Zeile.

Um die korrekte Cache-Benutzung nachzuvollziehen, rufen wir uns zunächst die JPA-Standard-Konfiguation für den Cache von Seite 166 in Erinnerung. Die zur Verfügung stehenden Parameter

```
CacheRetrieveMode.USE ("javax.persistence.cache.retrieveMode")
CacheStoreMode.USE ("javax.persistence.cache.storeMode")
```

zeigen sowohl beim Lesen der Datenbank bzw. beim Holen der Informationen aus der Datenbank auf die Einstellung `USE`; der Cache wird also benutzt. Mit den EclipseLink-Erweiterungen der auf Seite 301 beschriebenen `@Cache`-Annotation sowie den Query-Hints zur Cache-Benutzung(`CacheUsage`) als auch der Veränderung des Abfrage-Typen über den `QueryType` lässt sich die Cache-Benutzung nun sehr genau steuern.

Ausgangspunkt dafür ist die Klasse `SecondLevelDataCache` unseres aktuellen Projektes.

```
public class SecondLevelDataCache {
  public static EntityManagerFactory emf;
  public static PerformanceMonitor monitor;

  public static void main(String[] args) throws Exception {
    emf = Persistence.createEntityManagerFactory("bank", DB.oracle);
    readKunde(1);
    readKunde(1);

    monitor.dumpResults();
    emf.close();
  }

  public static void readKunde(Integer i) {
    EntityManager em = emf.createEntityManager();
    Query q = em.createNamedQuery("loadKundeById");
    //q.setHint(QueryHints.QUERY_TYPE, QueryType.ReadAll);
    q.setParameter("id", i);
    OutputHelper.output(q.getResultList());
    Session session = ((JpaEntityManager) em.getDelegate()).
        getSession();
    monitor = ((PerformanceMonitor)session.getProfiler());
    em.close();
  }
}
```

In unserem Beispiel verwenden wir ein weiteres – zusätzliches – Tool von EclipseLink: den *Performance-Monitor*. Dieser Monitor gibt dem Entwickler weitere Informationen an die Hand, um das Laufzeitverhalten der Anwendung zu untersuchen. Wir benutzen es hier, um die Cache-Benutzung zu demonstrieren. Die Ausgabe zeigt, dass der Cache benutzt wird. Die erste Abfrage legt die Instanz in den Cache hinein, die zweite verzeichnet einen sogenannten `Hit`, die Instanz wird aus dem Cache zurückgegeben:

```
Performance Monitor:1325247287882
Operation    Value (ns)
Counter:CacheHits    1
Counter:CacheMisses 1
Counter:ClientSessionCreates     2
Counter:ConnectCalls     1
Counter:ReadObjectQuery 2
Counter:ReadObjectQuery:Kunde:loadKundeById 2
Counter:ReadObjectQuery:Kunde:loadKundeById:CacheHits    1
```

```
Counter:ReadObjectQuery:Kunde:loadKundeById:CacheMisses 1
Counter:UnitOfWorkCreates    4
Info:CacheSize:Kunde     1
...
```

Mit der entsprechenden Konfiguration dieses Monitors und einer entsprechenden Ausgabedatei legt EclipseLink automatische Log-Ausgaben auch im Server-Betrieb ab. Wir meinen: eine sinnvolle Ergänzung zur Beobachtung laufender Anwendungen:

```
<property name="eclipselink.profiler" value="PerformanceMonitor"/>
<property name="eclipselink.logging.file" value="output.log"/>
```

12.1.2 Query-Cache

Mit EclipseLink können zu einer bestimmten Query auch deren Ergebnisse in Form von `ResultSets` gespeichert werden. Die Methode `cacheQueryResults()` in der Klasse `ReadQuery` aktiviert den Query-spezifischen Cache der jeweiligen Abfrage. Bei der Standard-Einstellung werden die letzten 100 Abfragen der Queries mit ihren Parametern abgelegt. Bei der wiederholten Abfrage wird ohne Datenbankzugriff das Ergebnis zurückgeliefert. Alternativ dazu kann der Parameter für eine JPA-Standard-Query mit dem Hint `QUERY_RESULTS_CACHE` aktiviert werden.

Unser Ausgangspunkt ist die Abfrage `load`, bei der wir absichtlich nicht über den Primärschlüssel unseres Entity selektieren, sondern eine Like-Bedingung auf den Nachnamen konstruieren:

```
@NamedQuery(name = "loadKundeByName",
            query = "select k from Kunde k where
                     k.nachname like :nachname"
)
```

Unsere Abfrage führen wir nun mehrmals hintereinander aus. Dabei wird der Parameter der Abfrage variiert, um zu kontrollieren, welche Abfrage tatsächlich zur Datenbank geht und welche aus dem Cache bedient wird. Natürlich benutzen wir auch diesmal wieder den EclipseLink-Performance-Monitor, um unsere Aussagen zu überprüfen.

```
emf = Persistence.createEntityManagerFactory("bank", DB.oracle);
EntityManager em = emf.createEntityManager();

Query query = em.createNamedQuery("loadKundeByName");
query.setHint(QueryHints.QUERY_RESULTS_CACHE, HintValues.TRUE);
query.setParameter("nachname", "Bu%");
OutputHelper.output(query.getResultList());

query.setParameter("nachname", "Ba%");
OutputHelper.output(query.getResultList());

query.setParameter("nachname", "Bu%");
OutputHelper.output(query.getResultList());
```

Wenn der Leser die Logging-Einstellung auf den Wert `FINE` belässt, wird er tatsächlich feststellen, dass nur zwei SQL-Befehle auf der Datenbank abgesetzt werden. Die erneute Query-Ausführung mit dem Parameter „Bu%“ geht direkt auf den Cache, ohne die Datenbank er-

neut zu belasten. Darüber hinaus werden die durch die Abfrage gelesenen Entities auch noch in den Objekt-Cache von EclipseLink überführt. Die gekürzte Ausgabe des Monitoring zählt nach Beendigung des Programms 34 `Kunde`-Instanzen im Cache.

```
Performance Monitor:1325158584891
Operation   Value (ns)
Counter:CacheHits   1
Counter:CacheMisses 2
Counter:ClientSessionCreates    1
Counter:ConnectCalls    1
Counter:ReadAllQuery    3
Counter:ReadAllQuery:de.jpainfo.model.Kunde:loadKundeByName 3
Counter:ReadAllQuery:de.jpainfo.model.Kunde:loadKundeByName:CacheHits
        1
Counter:ReadAllQuery:de.jpainfo.model.Kunde:loadKundeByName:
    CacheMisses 2
Counter:UnitOfWorkCreates   1
Info:CacheSize:Kunde    34
```

Obwohl wir in unserem Beispiel-Projekt die 1:1-Beziehung zwischen `Kunde` und `Adresse` mit dem `FetchType.LAZY` konfiguriert haben, lädt EclipseLink die Adresse auch dann, wenn wir nur den Kunden selektieren. Dieses Verfahren ist laut Spezifikation erlaubt. Um die Log-Ausgaben zu minimieren und zunächst nur `Kunde`-Objekte in den Cache zu überführen, aktivieren wir das Lazy-Loading, indem wir das dynamische Weaving aktivieren. Unser ANT-Target bzw. die Ausführung der Test-Klasse rufen wir mit dem Parameter `-javaagent:eclipselink.jar` auf. Ohne diesen Parameter werden bei Ausführung auch die `Adresse`-Instanzen geladen und in den Cache überführt.

Projekt:

Das Projekt zu den in diesem Abschnitt gezeigten Caching-Erweiterungen heißt `eclipselinkcache`.

12.2 Weaving

Auch EclipseLink bietet Werkzeuge an, die Manipulationen am Byte-Code einer Entity-Klasse durchführen. Eine allgemeine Einführung in die Byte-Code-Manipulation findet sich im Abschnitt 11.2 über Hibernates Instrumentation. Die EclipseLink-Dokumentation verwendet im Allgemeinen den Begriff des *Weaving* (Weben, Einweben). Um Code-Duplizierung zu vermeiden und einen direkten Vergleich mit Hibernate und später OpenJPA zu ermöglichen, verwenden wir auch hier das Beispiel eines Kunden, dessen Passbild als Byte-Array und dessen Adresse als 1:1-Beziehung zum Entity Adresse realisiert sind. Listing 11.4 auf Seite 283 zeigt die entsprechend annotierten Properties.

EclipseLinks Weaving kann im Rahmen des Build-Prozesses über eine Ant-Task statisch oder zum Ladezeitpunkt der Entity-Klasse dynamisch über einen Java-Agenten erfolgen. Das herunterladbare Projekt stellt beide Alternativen zur Verfügung.

Das statische Weaving erfolgt entweder über einen Kommandozeilenaufruf der Klasse `org.eclipse.persistence.tools.weaving.jpa.StaticWeave` oder über eine Ant-Task:

```
...
<taskdef name="weave" classname=
    "org.eclipse.persistence.tools.weaving.jpa.StaticWeaveAntTask">
  <classpath refid="classpath" />
</taskdef>

<target name="weaving" description="perform weaving"
      depends="define.task">
   <weave  source="${build.home}" target="${woven.home}">
     <classpath refid="classpath" />
   </weave>
</target>
...
```

Nach dem Aufruf des zweiten Targets befinden sich im Verzeichnis `${woven.home}` die überarbeiteten Klassen des Verzeichnisses `${build.home}`. Wird die überarbeitete Klasse `Kunde` zur Programmausführung verwendet, so enthält die entsprechende `Select`-Anweisung die Spalte `passbild` nicht. Das zweite `Select` für das Adressobjekt wird nicht ausgeführt. Erst wenn auf das Passbild beziehungsweise die Adresse zugegriffen wird, erfolgt ein erneuter Zugriff auf die Datenbank, und die entsprechenden Daten werden nachgeladen.

Hinweis:

Falls Sie sich für die von EclipseLink im Rahmen des Weavings vorgenommenen Code-Änderungen interessieren, raten wir zur Verwendung eines Java-Decompilers. Ein für die persönliche Nutzung freier Decompiler kann unter [URL-JD] heruntergeladen werden. Bei Interesse an weiteren Details finden Sie in [SM12] eine Einführung in das Thema.

Die zweite Alternative zur Manipulation des Byte-Codes ist der Einsatz eines Java-Agenten. Ein Java-Agent ist ein mit Java 5 eingeführtes API, um vor dem Aufruf der `main()`-Methode noch eine andere Methode aufrufen zu können. Diese Methode hat die Signatur

```
public static void premain(String agentArgs, Instrumentation inst)
```

und bekommt im ersten Parameter Argumente des Agenten, im zweiten Parameter eine `Instrumentation`-Instanz. `Instrumentation` ist das zentrale Interface zur Instrumentierung von Java-Klassen und wurde zusammen mit dem Package `java.lang.instrument` ebenfalls in Java 5 eingeführt. Das Interface `ClassFileTransformer` dieses Packages enthält eine einzige Methode `transform()`, deren fünf Parameter unter anderem den Byte-Code der zu ändernden Klasse enthält. Der Rückgabewert dieser Methode ist der neue Byte-Code der Klasse.

Die beschriebene JDK-Schnittstelle wird von JPA übernommen. Das Interface `ClassTransformer` im Package `javax.persistence.spi` enthält als einzige Methode ebenfalls `transform()` mit identischer Signatur. Über diese Methode nimmt EclipseLink die entsprechenden Manipulationen bei Verwendung des Java-Agenten vor.

Die Implementierung von EclipseLink realisiert die in der Dokumentation zum Package `java.lang.instrument` nachzulesenden Anforderungen an das Packaging und das Mani-

fest, so dass das Implementierungs-Jar direkt für die Option `javaagent` verwendbar ist. Die allgemeine Syntax lautet:

```
java -javaagent:<Pfad-zum-Jar> <Main-Klasse>
```

Im herunterladbaren Projekt ist im Ant-Build-File ein Target definiert, das den Aufruf mit dem Java-Agenten vereinfacht:

```
<target name="run KundeTest" depends="compile">
  <java classname="de.jpainfo.test.KundeTest" ... >
    <jvmarg value="-javaagent:./lib/eclipselink.jar" />
    <classpath refid="classpath" />
  </java>
</target>
```

Tipp:

Wird EclipseLink innerhalb eines Application-Servers betrieben, wird automatisch der Java-Agent für das Laden von Entity-Klassen verwendet. Ein statisches Weaving ist daher nicht nötig. Ein statisches Weaving kann jedoch sinnvoll sein, weil das Weaving im Build-Prozess und nicht während des Deployments stattfindet. Die Dauer des Deployments wird damit verkürzt.

Das Verhalten der Beispielanwendung unter Verwendung des Java-Agenten ist identisch zum statischen Weaving: Das `Select` des Kunden findet ohne das Property `passbild` statt. Das Property und die Adresse werden erst nachgeladen, wenn ein Zugriff erfolgt.

Projekt:

Das Projekt *eclipselink-weaving* enthält das verwendete Beispiel sowie ein Ant-Build-File, das über drei Targets die unveränderte, die statisch manipulierte und die über einen Java-Agenten manipulierte Entity-Klasse verwendet.

12.3 Mandantenfähigkeit

Eines der großen Ziele von Java-EE 7 ist der Einzug in die Cloud. Obwohl Unklarheiten bezüglich der exakten Definition einer Cloud existieren, scheinen sich Themen wie Skalierbarkeit, Lokationstransparenz, Sicherheit, Dienste und einige mehr als zentrale Themen zu etablieren. Diese und weitere können, was die Software-*Entwicklung* angeht, als sogenannte *cross-cutting concerns* angesehen werden: Bereiche, die in jeder Anwendung existent sind, aber nicht jeweils wieder neu implementiert werden sollen. Java-EE hat dies bereits in anderen Bereichen gelöst: Transaktionalität, Authentifizierung/Autorisierung, Load-Balancing und Fail-Over werden von Java-EE oder von Application-Servern bereits seit Jahren unterstützt. Man kann davon ausgehen, dass auch die cloud-spezifischen Anforderungen in Java-EE 7, spätestens Java-EE 8 befriedigend realisiert sein werden.

Die Mandantenfähigkeit wird im Zusammenhang mit der Cloud auch in den Zielen von Java-EE 7 genannt. Für JPA 2.1 ist die Mandantenfähigkeit ebenfalls eine definierte, zentrale Anforderung. Da JPA aber auch in der Version 2.1 in Java-SE lauffähig sein wird, müssen Mechanismen *in* JPA vorhanden sein und können nicht (ausschließlich) von einer Cloud-Umgebung bereitgestellt werden. EclipseLink besitzt bereits Möglichkeiten, die Mandantenfähigkeit von Anwendungen zu realisieren. Ob diese Möglichkeiten eine proprietäre EclipseLink-Erweiterung bleiben oder in den offiziellen JPA-Standard einfließen werden, lässt sich im Augenblick noch nicht abschätzen.

Um ein Entity mandantenfähig zu machen, wird die Annotation `@Multitenant` im Package `org.eclipse.persistence.annotations` auf Klassenebene verwendet.

```
@Entity
@Multitenant
public class Kunde {

  @Id @GeneratedValue
  private Integer id;
  private String vorname;
  private String nachname;
  @Temporal(TemporalType.DATE)
  private Date geburtsdatum;
  @OneToMany(mappedBy = "kunde")
  private List<Konto> konten;
  ...
}
```

```
@Entity
@Multitenant
public class Konto {

  @Id @GeneratedValue
  private Integer kontonummer;
  private BigDecimal kontostand;
  @ManyToOne
  @JoinColumn(name = "kunde")
  private Kunde kunde;
  ...
}
```

Das Tabellenschema für die beiden Entities ist in Abbildung 12.1 dargestellt. Es enthält für beide Tabellen eine zusätzliche Spalte, die keine Entsprechung in der Entity-Klasse besitzt.

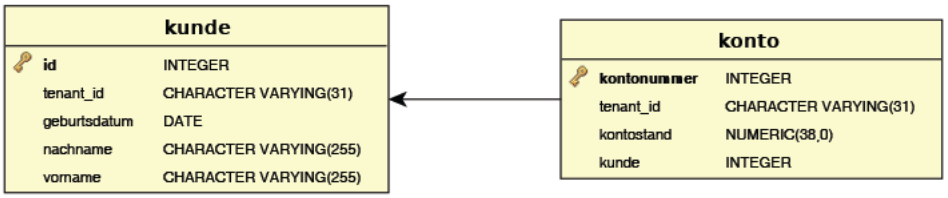


ABBILDUNG 12.1 Entity-Tabellen mit Mandantenspalte

Diese Spalte mit Default-Namen `tenant_id` wird von EclipseLink intern verwendet, um Daten für verschiedene Mandanten zu unterscheiden. Die Werte für die Spalte werden über das Property `eclipselink.tenant-id` entweder in der `persistence.xml` oder innerhalb der Map für die Erzeugung der Entity-Manager-Factory angegeben. Die zweite Alternative ist für diesen Zweck wesentlich sinnvoller, so dass wir das Muster angeben:

```
Map<String, String> props = new HashMap<String, String>();
props.put("eclipselink.tenant-id", "Mandant-1");
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("bank", props);
EntityManager em = emf.createEntityManager();
```

Beim Persistieren von Entities durch diesen Entity-Manager erhält die Spalte `tenant_id` den Wert `"Mandant-1"`. Beim Lesen mit `find()` oder bei der Verwendung von JPQL-Anfragen

fügt EclipseLink den Select-Anweisungen in der Where-Klausel ein „`... and (tenant_id = ?)`“ hinzu, so dass immer nur Daten des entsprechenden Mandanten gelesen werden.

Die Annotation `@Multitenant` besitzt `value` als einziges Attribut, so dass wir auf eine Darstellung in Tabellenform verzichten. Der Wert des Attributs spezifiziert die zu verwendende Strategie zur Implementierung der Mandantenfähigkeit. Der Typ des Attributs ist das Enum `MultitenantType`, ebenfalls im Package `org.eclipse.persistence.annotations`. Die Enum-Werte sind `SINGLE_TABLE`, `TABLE_PER_TENANT`, und `VPD`. `SINGLE_TABLE` ist der Default und bedeutet, dass Daten verschiedener Mandanten in derselben Tabelle enthalten sind. `TABLE_PER_TENANT` realisiert jeweils eigene Tabellen pro Mandant. Im Augenblick (EclipseLink 2.3) wird `TABLE_PER_TENANT` noch nicht unterstützt und führt bei Verwendung zu einer Exception. Der letzte Wert, `VPD`, realisiert die Mandantenfähigkeit durch die Oracle-Datenbankerweiterung *Virtual Private Database* und schließt damit die Verwendung anderer Datenbanken aus.

Eine nicht im Entity enthaltene Spalte zur Unterscheidung von Tabellenzeilen bezüglich eines bestimmten Kritieriums haben wir schon im Kapitel 5 zur Realisierung von Vererbung kennengelernt. Die dort verwendete Annotation `@DiscriminatorColumn` stand sowohl im Namen als auch den möglichen Attributen Pate für die Annotation `@TenantDiscriminatorColumn`, die die Konfiguration dieser Diskriminator-Spalte erlaubt. Die Attribute der Annotation sind in Tabelle 12.3 aufgeführt.

TABELLE 12.3 `@TenantDiscriminatorColumn`-Attribute

`@TenantDiscriminatorColumn(...)`			
Attribut	**Typ**	**Default**	**Beschreibung**
`column-Definition`	`String`	`""`	DDL-Anteil für Spaltendefinition
`context-Property`	`String`	`"eclipselink .tenant-id"`	Name des Spalten-Property
`discriminator-Type`	`Discrimi-natorType`	`String`	Typ des Diskriminators. Mögliche Werte sind `CHAR`, `INTEGER`, `STRING`.
`length`	`int`	31	Länge von String-Diskriminatoren
`name`	`String`	`"TENANT_ID"`	Spaltenname des Diskriminators
`primaryKey`	`boolean`	`false`	Angabe, ob Diskriminator Teil des Primärschlüssels ist
`table`	`String`	`""`	Tabellenname

Falls mehrere Diskriminatorspalten verwendet werden sollen, existiert mit `@TenantDiscriminatorColumns` die entsprechende Pluralform.

Soll die Diskriminatorspalte lediglich `mandant` heißen und die Mandanten mit Ganzzahlen, also z. B. 1, 2, 3, ... unterschieden werden, so kann die Annotation wie folgt verwendet werden.

```
@Entity
@Multitenant
@TenantDiscriminatorColumn(name = "mandant",
                           discriminatorType = INTEGER)
public class Kunde {
  ...
}
```

Die Diskriminatorspalte wird von EclipseLink intern verwendet, um eine Tabellenzeile genau einem Mandanten zuzuordnen. Eine schreibende Verwendung in der Anwendung würde in der Regel zu Fehlern führen und ist daher verboten. Eine lesende Verwendung ist allerdings erlaubt. Hierzu muss das Property gemappt und explizit read-only gemacht werden. Wir hatten Read-only-Properties bereits in Kapitel 2 erwähnt, jedoch kein Beispiel entwickelt. Dies können wir nun nachholen. Der folgende Code mappt die Diskriminatorspalte und macht sie read-only.

```
@Entity
@Multitenant
@TenantDiscriminatorColumn(name = "mandant")
public class KundeMapped {

  @Id @GeneratedValue
  private Integer id;
  private String vorname;
  private String nachname;
  ...
  // Die gemappte Diskriminatorspalte:
  @Column(insertable = false, updatable = false)
  private String mandant;
  ...
}
```

In der Anwendung kann nun mit `getMandant()` lesend auf das Property zugegriffen werden. Die Verwendung des Setters `setMandant()` führt jedoch bei der Synchronisation des Persistenzkontextes mit der Datenbank zu keinen Änderungen in der Datenbank, da EclipseLink durch die Read-only-Definition die Spalte in Insert- und Update-Anweisungen nicht verwendet.

Wir beschließen unsere Ausführungen zu EclipseLink und weisen den Leser darauf hin, dass die OpenJPA-Erweiterung Slice (Abschnitt 13.3) eine andere Art der Mandantenfähigkeit mit separaten Datenbanken pro Mandant ermöglicht.

13 OpenJPA-Erweiterungen

13.1 Caching

Auch OpenJPA erweitert den in der Spezifikation vereinbarten Caching-Standard um eigene Implementierungen. Dabei wird wie bei EclipseLink zwischen einem Objekt-Cache (bei OpenJPA heißt er `DataCache`) und einem Query-Cache unterschieden. Die Cache-Aktivierung erfolgt bei OpenJPA über das nachstehende Property. Im Unterschied zu unserem Beispiel auf Seite 169 aktivieren wir gleich die OpenJPA-Statistiken, um die Cache-Benutzung unserer Beispiele entsprechend zu dokumentieren.

```
<property name="openjpa.DataCache"
          value="true(EnableStatistics=true)"/>
<property name="openjpa.RemoteCommitProvider"
          value="sjvm"/>
```

TABELLE 13.1 Implementierungen des `DataCache`-Interfaces

Parameter	Implementierende Klasse
`false`	–
`true`	`org.apache.openjpa.datacache.ConcurrentDataCache`
`concurrent`	`org.apache.openjpa.datacache.ConcurrentDataCache`
`partitioned`	`org.apache.openjpa.datacache.PartitionedDataCache`

Der Parameter `RemoteCommitProvider` ist optional und wird im Standard auf den Wert `svjm` gesetzt. Bei verteilt eingesetzten Caches lassen sich mit entsprechenden Implementierungen des Interfaces `RemoteCommitProvider` Änderungen am Cache an andere Cache-Instanzen publizieren und weiterverarbeiten. Wir belassen es in unseren Beispielen beim Default-Eintrag. Den interessierten Leser verweisen wir auf die jeweiligen Implementierungen, die OpenJPA mitbringt. Sie können Ausgangspunkt für weitere Eigenentwicklungen sein.

Wie einfach der Plugin-Mechanismus von OpenJPA an dieser Stelle funktioniert, wollen wir mit einer einfachen Cache-Implementierung zeigen, die wir anschließend in der Konfiguration hinterlegen. Unsere Cache-Implementierung verhält sich wie die OpenJPA-Implementierung, nur wird die `put`-Methode überschrieben, um auf der Kommandozeile das Hinzufügen eines Objektes zu signalisieren:

TABELLE 13.2 Implementierungen des `RemoteCommitProvider`-Interfaces

Parameter	Implementierende Klasse
`sjvm`	`org.apache.openjpa.event.SingleJVMRemoteCommitProvider`
`jms`	`org.apache.openjpa.event.JMSRemoteCommitProvider`
`tcp`	`org.apache.openjpa.event.TCPRemoteCommitProvider`

```
package de.jpa.info.openjpa.cache;

import org.apache.openjpa.datacache.ConcurrentDataCache;
import org.apache.openjpa.datacache.DataCachePCData;

public class TestCacheImplementierung extends ConcurrentDataCache {
  @Override
  public DataCachePCData put(DataCachePCData data) {
    System.out.println("Zum Cache hinzu: " + data.getType());
    return super.put(data);
  }
}
```

Mit der Änderung des Property `openjpa.DataCache` wird beim erneuten Aufruf jeder schreibende Data-Cache-Zugriff explizit dokumentiert.

```
<property
  name="openjpa.DataCache"
  value="de.jpa.info.openjpa.cache.TestCacheImplementierung"/>
```

OpenJPA bietet darüber hinaus eine eigene Annotation zur feineren Cache-Steuerung an. Diese Annotation lautet `@DataCache` und befindet sich im Paket `org.apache.openjpa.persistence`. Tabelle 13.3 zeigt die zur Verfügung stehenden Attribute an.

TABELLE 13.3 `@DataCache`-Attribute

Element	Bedeutung
`enabled`	Schaltet das Caching für das Entity ein oder aus; Default: `true`. Auch über das Property `openjpa.DataCache` können bereits beim Aktivieren des Cache Typen vom Caching ausgeschlossen werden. Das Ausschließen überschreibt dann diesen Schalter.
`name`	Name der Partition, in der diese Entity-Instanzen abgelegt werden. Jede Partition wird dabei als eigener Cache angesehen.
`timemout`	Zeit in Millisekunden, bis zu der Instanzen einer Klasse im Cache gültig sind. Nach Ablauf der Zeit erfolgt für diese Entities ein erneuter Datenbankzugriff.

13.1.1 Object-Cache

Objekte werden mit den Methoden `find()`, `refresh()` oder auch mit einer Query in den Second-Level-Cache gebracht. Davon ausgenommen sind in OpenJPA die folgenden Abfragetechniken:

- *Large Result Sets* werden benutzt, um nicht alle Daten auf einmal in den Arbeitsspeicher zu laden, sondern mit rollierenden Result-Sets immer nur einen Teil der Abfragemenge

zu verarbeiten. Diese Technik ist bei OpenJPA deaktiviert, kann aber mit dem Parameter `openjpa.FetchBatchSize` auf einen entsprechenden Wert gesetzt werden.

- Die in OpenJPA mögliche `Extent`-Technologie benutzt den Cache beim Lesen nicht. Bei einem `Extent` wird ähnlich einer Datenbankview auf der `EntityManager`-Implementierung ein Ausschnitt erzeugt.
- Queries, die einzelne Attribute von Entities zurückliefern, aber nicht das Entity selbst, haben ebenfalls keine Auswirkungen auf den Objekt-Cache.

Wir wollen die Cache-Benutzung anhand der Statistiken von OpenJPA verifizieren. Die Statistiken sind Bestandteil des OpenJPA-spezifischen API und nicht über den JPA-Standard zu verwenden. Für den Zugriff darauf wird die `EntityManagerFactory`-Instanz unserer Persistenzeinheit auf die JPA-Implementierung gecastet. Diese Instanz bietet uns dann Zugriff über das Interface `StoreCache`, das die Statistiken bereitstellt. Dieses Interface vereint sowohl das OpenJPA-Cache-Interface als auch unser bekanntes Interface `javax.persistence.Cache` aus der JPA-Spezifikation. Benötigt der Leser darüber hinaus Zugriff auf weitere Methoden von OpenJPA, wird mit `getDelegate()` in das Package `org.apache.openjpa.datacache` und die dortigen Implementierungen verzweigt.

```
// JPA-Instanz
emf = Persistence.createEntityManagerFactory("bank", DB.oracle);
// OpenJPA-Instanz
oemf = OpenJPAPersistence.cast(emf);

// OpenJPA-Cache-Fassade + javax.persistence.Cache
StoreCache cache = oemf.getStoreCache();

// Statistiken des Cache
CacheStatistics st = cache.getStatistics();
System.out.println("ReadCount: " + st.getReadCount());
System.out.println("HitCount: " + st.getHitCount());
System.out.println("WriteCount: " + st.getWriteCount());

// OpenJPA-Cache-Interface
DataCache dataCache = ((StoreCacheImpl) cache).getDelegate();
```

Die Cache-Nutzung demonstrieren wir wieder mit unserem Programm aus Abschnitt 11.1. Im Unterschied zum Quellcode von Seite 280, geben wir die Cache-Statistik am Ende jeder Methode aus.

Die OpenJPA-Statistiken sind kumulierend. Das bedeutet: Die Ausgabe bei Variante drei enthält die gespeicherten Zugriffe der Varianten eins und zwei. Tabelle 13.4 erläutert die Zahlen noch einmal. Alle Werte sind sowohl pro Entity-Klasse als auch als totale Summe nach einem Reset der Statistiken abrufbar.

```
-- nach Variante 1
ReadCount: 1
HitCount: 0
WriteCount: 1

-- nach Variante 2
ReadCount: 2
HitCount: 1
WriteCount: 1
```

```
-- nach Variante 3
ReadCount: 4
HitCount: 2
WriteCount: 2
```

TABELLE 13.4 `CacheStatistics`-Erläuterungen

Name	Bedeutung
`ReadCount`	Zählt die lesenden Zugriffe, die ein Cache nach seiner Instanziierung bzw. dem letzten Reset erfahren hat.
`HitCount`	Lesende Zugriffe auf den Cache, für die auch ein Ergebnis im Cache gefunden werden konnte.
`WriteCount`	Anzahl einzelner Objekte, die dem Cache hinzugefügt wurden. Unsere Query aus Listing 11.3 würde bei einer Selektion von `select k, k.wohnort ...` nicht nur das `Kunde`-Entity, sondern auch das korrespondierende `Adresse`-Entity in den Cache überführen. Der `WriteCount` würde dann nach Variante 1 sofort *zwei* betragen.

Löschen und Pinnen des Cache

Im Abschnitt 6.3.5 sind wir bereits auf die Möglichkeiten eingegangen, im JPA-Standard den Cache zu beeinflussen. OpenJPA erweitert diese Möglichkeiten um einige Methoden im Interface `StoreCache`. Dazu zählen vor allem eine Reihe von `pin()`- bzw. `unpin()`-Methoden. Die `pin()`-Methoden bewahren ein Objekt davor, bei Ausführung der Methode `evictAll()` aus dem Cache entfernt zu werden. Auch nach manueller Entfernung dieses Objektes mit direkter Schlüsselangabe merkt sich OpenJPA diesen Pin. Wird das betroffene Objekt erneut gelesen, wird der Pin wieder wirksam. Wünscht man ihn nicht mehr, muss er explizit mit einer der `unpin()`-Methoden gelöscht werden.

Objekte, die mit `pin()` im Cache festgeschrieben wurden, zählen bei der Ermittlung der Cache-Größe nicht mit. Wird die Größe nicht festgelegt, geht OpenJPA von einer initialen Größe von 1000 Objekten aus.

13.1.2 Query-Cache

Im Gegensatz zu früheren Versionen von OpenJPA ist in der aktuellen Version 2.1 der Query-Cache nicht aktiviert, wenn der einfache Objekt-Cache konfiguriert wird. Das zusätzliche Property

```
<property name="openjpa.QueryCache" value="true"/>
```

muss also explizit auf `true` gesetzt werden.

Der Query-Cache speichert zu einer Abfrage bzw. den Eigenschaften einer Abfrage die zurückgelieferten Ids der Entities. Wird diese Abfrage noch einmal ausgeführt, werden die referenzierten Objekte aus dem Cache zurückgegeben. Der Query-Cache wird nicht benutzt, wenn Instanzen der in der Abfrage referenzierten Entities zwischenzeitlich verändert wurden. Die betroffenen Abfragen werden dann unverzüglich aus dem Query-Cache entfernt.

Zugriff auf den Query-Cache liefert uns wie beim Objekt-Cache die provider-spezifische Implementierung der `EntityManagerFactory`-Instanz. Darüber hinaus erhalten wir leicht abgewandelt zu unseren obigen Beispielen auch Zugriff auf die Query-Statistiken von OpenJPA.

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory(
  "bank", props);
// OpenJPA-Implementierung
OpenJPAEntityManagerFactory oemf = OpenJPAPersistence.cast(emf);
// Query-Cache der Persistenzeinheit
QueryResultCacheImpl qcache = (QueryResultCacheImpl) oemf
    .getQueryResultCache();
QueryStatistics<QueryKey> qstats = qcache.getDelegate().
  getStatistics();
```

Im folgenden Beispiel wollen wir zwei Queries hintereinander ausführen, die gleiche Selektions- und Where-Bedingungen haben. Mit Aktivierung des Query-Cache sollten diese Abfragen nur einmal zur Datenbank gehen. Die zweite Ausführung der Abfrage muss direkt über den Cache erfolgen.

```
public static void readKundeQuery(Integer upperRange) {
  EntityManager em = emf.createEntityManager();
  Query q = em.createQuery("select k from Kunde k where k.id <= :
      upperRange");
  q.setParameter("upperRange", upperRange);
  OutputHelper.output(q.getResultList());
  em.close();
}
```

Die Ausgabe der `dump()`-Methode des OpenJPA-Query-Cache sieht wie folgt aus:

```
Query Statistics starting from Tue Jan 03 17:50:06 CET 2012
Total Query Execution: 2:1(50%)
        Total        Query
1.      2:1(50%)     org.apache.openjpa.datacache.QueryKey@8670f23f[
                       query:[select k from Kunde k where k.id <= 3],
                       access path:[de.jpainfo.model.Kunde],
                       subs:true,
                       ignoreChanges:false,
                       startRange:0,
                       endRange:9223372036854775807,
                       timeout:-1
                     ]
```

Von zwei ausgeführten Queries wurde bei einer ein Treffer im Cache vermerkt. Hierbei handelt es sich offensichtlich um die zweite Abfrage. Die Log-Ausgabe zeigt darüber hinaus, wie der eindeutige Query-Key aussieht, unter dem eine Abfrage in den Cache gelegt wird.

Wir wollen in einem weiteren Beispiel zeigen, dass der Query-Cache auch direkte Auswirkungen auf den normalen Objekt-Cache hat.

```
private static void readKundeFind() {
  EntityManager em = emf.createEntityManager();
  Kunde k = em.find(Kunde.class, new Integer(1));
  System.out.println(k);
  em.close();
}
```

Die selektierten Objekte aus unserem Query-Cache-Beispiel sind auch für die weiteren Zugriffe über `find()` direkt aus dem Cache abrufbar. Die dazugehörige Log-Ausgabe sieht nach Ausführung der Query, `find()`-Methode und nochmaliger Ausführung der Query wie folgt aus:

```
-- Query-Statistics wie oben dargestellt
...
ReadCount: 7
HitCount: 4
WriteCount: 3
```

In unserer Beispiel-Datenbank befinden sich drei Kunden, deren Id kleiner bzw. gleich 3 ist. Insgesamt wird 7 mal auf den Objekt-Cache zugegriffen, und das, obwohl die Queries sich auch in einem eigenen Cache befinden. Der Aufbau der selektierten Objekt-Instanzen erfolgt ressourcensparend nicht mit den Daten der Datenbank, sondern von bereits gecachten Entities. Die `find()`-Methode und die zweite Ausführung unserer Query führen zu insgesamt 4 Hits des Second-Level-Cache.

Projekt:

Der Leser ist an dieser Stelle aufgefordert, weitere Beispiele zur Cache-Benutzung selbst auszuprobieren. Wie verhält sich z.B. der Query-Cache, wenn in unserer `readKundeFind()`-Methode das gelesene Entity verändert wird? Die Antwort darauf haben wir bereits in der Einleitung dieses Kapitels gegeben. Als Ausgangspunkt für weitere Entwicklungen dient das Projekt `openjpacache`.

Weitere Methoden zum Query-Cache wie `evict()`, `pin()` und `unpin()` können analog zum Objekt-Cache angewendet werden. Die API zum Interface `QueryResultCache` beschreibt die Methoden entsprechend.

13.2 Enhancement

Byte-Code-Manipulationen, die von Hibernate Instrumentation (Abschnitt 11.2) und von EcliseLink Weaving (Abschnitt 12.2) genannt werden, bezeichnet die OpenJPA-Dokumentation als *Enhancement*. OpenJPA ermöglich wie EclipseLink sowohl ein statisches Enhancement über eine Main-Klasse und eine Ant-Task als auch ein Enhancement mit Hilfe eines Java-Agenten. Ab Version 2.2.0 enthält OpenJPA außerdem ein Maven-Plugin. Wir gehen hier auf die Ant-Task und die Verwendung des Java-Agenten ein.

Als Beispiel verwenden wir die Kundendefinition aus Listing 11.4, bei dem ein Byte-Array sowie eine 1:1-Beziehung spät geladen werden sollen. Die Verwendung der nicht manipulierten Class-Datei resultiert beim Lesen des Kunden in einem `Select`, das auch die Passbild-Spalte enthält und zugleich über ein Left-Outer-Join die Adresse lädt. Der Hinweis des späten Ladens (`LAZY`) wird also bei beiden Properties ignoriert.

Das statische Enhancement erfolgt über einen Kommandozeilenaufruf der Klasse `PCEnhance` im Package `org.apache.openjpa.enhance` oder alternativ über eine Ant-Task:

```
<target name="enhance" depends="compile">
  <taskdef name="openjpac"
           classname="org.apache.openjpa.ant.PCEnhancerTask">
    <classpath refid="classpath" />
  </taskdef>
  <openjpac>
    <classpath path="${build.home}" />
  </openjpac>
</target>
```

Nach erfolgtem Enhancement beachtet OpenJPA beide Hinweise zum späten Laden, d. h. das Passbild und die Adresse werden zunächst nicht geladen und später beim ersten Zugriff nachgeladen.

Die Möglichkeiten zur Verwendung eines Java-Agenten entspricht den für EclipseLink in Abschnitt 12.2 dargestellten Möglichkeiten. Für eine allgemeine Einführung zum Thema Java-Agenten verweisen wir auch auf Abschnitt 12.2. Das entsprechende Ant-Target entspricht der EclipseLink-Version mit geänderter Jar-Datei:

```
<target name="run KundeTest" depends="compile">
  <java classname="de.jpainfo.test.KundeTest" ... >
    <jvmarg value="-javaagent:./lib/openjpa-all-2.1.1.jar" />
    <classpath refid="classpath" />
  </java>
</target>
```

Tipp:

Unsere Arbeiten mit OpenJPA haben gezeigt, dass ein sinnvolles Arbeiten ohne den Einsatz des Java-Agenten *nicht* möglich ist. OpenJPA benötigt z. B. zwingend ein Enhancement von Entity-Klassen einer Vererbungshierarchie (Kapitel 5). Wir raten dem Leser, *immer* mit OpenJPAs Java-Agenten zu arbeiten. Wie bei EclipseLink ist dies in Java-EE standardmäßig der Fall.

OpenJPA lässt Kommandozeilenparameter und Property-Definitionen für eine weitreichende Steuerung des Enhancements zu, die wir hier nicht wiedergeben wollen Wir verweisen hierzu auf die OpenJPA-Dokumentation [URL-OJPA]. Einzig sei hier erwähnt, dass in der Konfigurationsdatei `persistence.xml` über das Property `RuntimeUnenhancedClasses` das Enhancement von Klassen ohne die beiden von uns vorgestellten Alternativen möglich ist.

```
<property name="openjpa.RuntimeUnenhancedClasses"
          value="supported" />
```

Projekt:

Das Projekt *openjpa-enhancement* enthält die Beispiele zu OpenJPAs Enhancement.

13.3 Slice

Im Kapitel 3 haben wir JPAs Persistenzeinheit eingeführt, die Entities einer Datenbank logisch zusammenfasst. Die OpenJPA-Erweiterung *Slice* hebt die Beschränkung auf eine Datenbank auf und erlaubt die Verwendung mehrerer Datenbanken. Die Erweiterung ist für die JPA-Anwendung in großen Teilen transparent, so dass die Möglichkeiten von Slice, etwa die Verteilung oder Replikation von Daten, (fast) ohne Änderungen am Anwendungs-Code erfolgen. Dies erreicht man dadurch, dass Slice unterhalb der JPA-Anwendung konfiguriert wird. Wir beginnen daher an dieser Stelle – der Konfiguration von Slice.

13.3.1 Konfiguration

Die Entities einer Persistenzeinheit werden von Entity-Managern verwaltet, die alle durch eine Entity-Manager-Factory erzeugt wurden. Diese wiederum ist in einer eindeutigen Beziehung an die Konfigurationsdatei `persistence.xml` bzw. die dort definierte und mit Namen versehene Persistenzeinheit gebunden. Slice erweitert diesen Teil der `persistence.xml` um die Möglichkeit, mehrere Slices (Scheiben/Teile) zu definieren. Ein Slice entspricht dabei der herkömmlichen Definition der Datenbank bwz. Datenbankverbindung. Die Konfiguration erfolgt auf globaler, d. h. alle Slices umfassender Ebene, und spezifisch pro Slice. Zunächst muss Slice aktiviert werden. Dies erfolgt über das Property `openjpa.BrokerFactory` mit dem Wert `slice`, was in Listing 13.1 in Zeile 6 zu erkennen ist. Die weiteren Slice-spezifischen Properties haben das Präfix `openjpa.slice`, auf dessen Nennung wir im Folgenden verzichten. Alle zu verwendenden Slices sind im Property `Names` aufzuzählen (Zeilen 7/8). Das Property `Master` (Zeile 9) gibt an, welcher Slice zur Schlüsselgenerierung bei Sequenzen zu verwenden ist. Das Property `Lenient` (Zeile 10) definiert das Verhalten, falls Slices zur Laufzeit nicht verfügbar sind. Falls das Property `true` ist, wird die Nichtverfügbarkeit ignoriert, ansonsten eine Exception geworfen. Für die Properties gibt es ebenfalls Default-Regeln nach dem Paradigma *Convention over Configuration*, auf die wir jedoch nicht eingehen.

LISTING 13.1 Konfiguration von Slice in der Datei `persistence.xml`

```
1  <persistence xmlns="http://java.sun.com/xml/ns/persistence" ...>
2
3    <persistence-unit name="bank">
4      <class>...</class>
5      <properties>
6        <property name="openjpa.BrokerFactory" value="slice"/>
7        <property name="openjpa.slice.Names"
8                  value="bank1,bank2,bank3,bank4"/>
9        <property name="openjpa.slice.Master" value="bank1"/>
10       <property name="openjpa.slice.Lenient" value="true"/>
11
12       <property name="openjpa.ConnectionDriverName"
13                 value="org.postgresql.Driver"/>
14       <property name="openjpa.ConnectionUserName" value="..."/>
15       <property name="openjpa.ConnectionPassword" value="..."/>
16       <property name="openjpa.slice.bank1.ConnectionURL"
```

```
17                  value="jdbc:postgresql://localhost/bank1"/>
18        <property name="openjpa.slice.bank2.ConnectionURL"
19                  value="jdbc:postgresql://localhost/bank2"/>
20        <property name="openjpa.slice.bank3.ConnectionURL"
21                  value="jdbc:postgresql://localhost/bank3"/>
22        <property name="openjpa.slice.bank4.ConnectionDriverName"
23                  value="com.mysql.jdbc.Driver"/>
24        <property name="openjpa.slice.bank4.ConnectionURL"
25                  value="jdbc:mysql://localhost/bank4"/>
26      </properties>
27    </persistence-unit>
```

Zur Definition der Datenbankverbindung können die mit JPA 2.0 eingeführten Property-Namen `driver`, `url`, `user` und `password` mit Präfix `javax.persistence.jdbc` nicht verwendet werden, so dass die entsprechenden OpenJPA-spezifischen Property-Namen zu verwenden sind. Wir verzichten auf eine Erläuterung der selbsterklärenden Property-Namen und merken lediglich an, dass bei der Verwendung des Präfix `openjpa` der Wert für alle Slices, bei Verwendung des Präfix `openjpa.slice.<slice>` der Wert nur für den entsprechenden Slice gilt. Im Beispiel von Listing 13.1 repräsentieren die Slices `bank1`, `bank2` und `bank3` Postgres-Datenbanken, während der Slice `bank4` eine MySQL-Datenbank repräsentiert. Da wir in den Zeilen 22 bis 25 lediglich `ConnectionDriverName` und `ConnectionURL` für den Slice `bank4` definieren, werden `ConnectionUserName` und `ConnectionPassword` unverändert übernommen.

Die Schema-Generierung von OpenJPA lässt sich auch mit Slice verwenden. Im obigen Fall werden in allen vier Datenbanken die den Entity-Definitionen entsprechenden Tabellen erzeugt.

13.3.2 Verteilungsstrategien

Ohne weitere Konfiguration verteilt Slice zu persistierende Entities automatisch. Bei einem Aufruf von `EntityManager.persist()` wird das Entity randomisiert in eine der verfügbaren, im Beispiel also eine der vier Datenbanken geschrieben. Sind Entities über Beziehungen verbunden, so wird die transitive Hülle, die sich über die Erreichbarkeit dieser Beziehungen definiert, komplett in einer Datenbank gespeichert. Das Handbuch spricht hier von *Collocation*.

Dieses Verhalten der zufälligen Verteilung kann über eine anwendungseigene Verteilungsstrategie überschrieben werden. Grundlage ist das Interface `DistributionPolicy`, das sich wie alle anderen im Folgenden genannten Klassen und Interfaces im Slice-Package `org.apache.openjpa.slice` befindet. Die einzige Methode des Interface ist `distribute()`:

```
String distribute(Object obj, List<String> slices, Object context)
```

Die Methode `distribute()` gibt für ein zu persistierendes Objekt den zu verwendenden Slice als String zurück. Parameter sind das zu persistierende Objekt, die Liste der zur Verfügung stehenden Slices sowie ein OpenJPA-interner Peristenzkontext. Eine implementierende Klasse wird in der `persistence.xml` konfiguriert. Hierzu wird für das Property

`openjpa.slice.DistributionPolicy` der voll qualifizierte Name der implementierenden Klasse angegeben. Wir beginnen mit einem einfachen Beispiel, das die zufällige Verteilung des Standardfalls nachbildet.

```
public class MyDistributionPolicy implements DistributionPolicy{

  private static Random random;

  public MyDistributionPolicy() {
    random = new Random();
  }

  @Override
  public String distribute(Object entity, List<String> slices, Object
       context) {
    return "bank" + (random.nextInt(4) + 1);
  }
}
```

Weil die Verteilungsstrategie beliebig definierbar ist, können auch Anwendungsinformationen als Basis der Verteilung dienen. Im Folgenden wollen wir beispielhaft nach den Anfangsbuchstaben des Nachnamens der Kunden verteilen. Wir geben nur die Methode `distribute()` an:

```
@Override
public String distribute(Object entity, List<String> slices,
                         Object context) {
  if (entity instanceof Kunde) {
    char anfangsbuchstabe =
              ((Kunde) entity).getNachname().toCharArray()[0];
    if ('A' <= anfangsbuchstabe && anfangsbuchstabe < 'I') {
      return "bank1";
    } else if ('I' <= anfangsbuchstabe && anfangsbuchstabe < 'M') {
      return "bank2";
    } else if ('M' <= anfangsbuchstabe && anfangsbuchstabe < 'T') {
      return "bank3";
    } else {
      return "bank4";
    }
  } else {
    return "bank1";
  }
}
```

Eine weitere sinvolle Einsatzmöglichkeit wäre die Realisierung einer Mandantenfähigkeit. Hierzu sind alle Entities mit einem entsprechenden Property, z. B. `mandant`, zu versehen und zur Laufzeit dem entsprechenden Mandanten zuzuordnen:

```
@Entity
public class Kunde {

  @Id
  @GeneratedValue
  private Integer id;
  private String vorname;
  private String nachname;
  @Temporal(TemporalType.DATE)
```

```
    private Date geburtsdatum;
    @Column(nullable = false)
    private String mandant;
    ...
  }
```

Wenn die Daten eines Mandanten jeweils in einer Datenbank gehalten werden, lässt sich die benötigte Verteilungsstrategie leicht realisieren:

```
  @Override
  public String distribute(Object entity, List<String> slices, Object
      context) {
    if (entity instanceof Kunde) {
      return ((Kunde) entity).getMandant();
    } else {
      return "...";
    }
  }
```

Eine andere Alternative zur Realisierung der Mandantenfähigkeit haben wir in Abschnitt 12.3 als EclipseLink-Erweiterung kennengelernt.

Lesende Operationen, also die `find()`-Methode und JPQL-Anfragen, arbeiten auf allen Slices. Mit der Definition von Zielstrategien ist es möglich, dies auf bestimmte Slices einzuschränken. Hierzu werden die beiden Interfaces `FinderTargetPolicy` und `QueryTargetPolicy` mit den Methoden `getTargets()` verwendet:

```
  // Interface FinderTargetPolicy:
  String[] getTargets(Class<?> cls, Object oid,
                      List<String> slices, Object context)

  // Interface QueryTargetPolicy:
  String[] getTargets(String query, Map<Object,Object> params,
                      String language,
                      List<String> slices, Object context)
```

Bei den Parametern der `FinderTargetPolicy`-Methode werden analog zur `find()`-Methode von JPA die Entity-Klasse und der Primärschlüssel angegeben. Bei den Parametern der `QueryTargetPolicy`-Methode werden die JPQL-Anfrage sowie eventuelle Anfrageparameter angegeben. Der Parameter `language` definiert die Sprache (Codierung) der Anfrage. Die beiden letzten Parameter entsprechen den letzten Parametern der `distribute()`-Methode. Rückgabewert der beiden Methoden ist ein Array von Slices, in denen die Suche bzw. die Anfrage durchzuführen ist.

Die Verwendung von Aggregatsfunktionen (Abschnitt 7.5.1) in JPQL-Anfragen ist mit Ausnahme von `AVG` erlaubt. Bei `COUNT`, `SUM`, `MIN` und `MAX` können die jeweiligen Ergebnisse der Slices addiert bzw. das Minimum oder das Maximum gebildet werden. Für die jeweiligen Durchschnittswerte ist ohne die Kenntnis der Größe der Basismengen der Durchschnitt über alle Slices jedoch nicht zu berechnen.

Da Slice Operationen auf mehreren Datenbanken ausführt, stellt sich die Frage nach der Transaktionalität dieser Operationen. Im Default-Fall verwendet Slice das Transaktionsmanagement der jeweiligen Datenbanken und kann damit die ACID-Eigenschaften nicht garantieren. Wenn eine Transaktion in Slice A bereits committed wurde und die Transaktion in Slice B fehlschlägt, kann Slice A nicht mehr zurückgerollt werden.

Alternativ kann der Wert des Property `openjpa.slice.TransactionPolicy` auf `xa` gesetzt werden. Slice verwendet dann einen Two-Phase-Commit-Transaktionsmanager, falls die verwendeten JDBC-Treiber XA-konform sind.

Wir schließen hiermit unsere Ausführungen zu Slice ab. Der Leser findet im OpenJPA-Handbuch noch weitere, von uns nicht erwähnte Möglichkeiten, wie z. B. die Angabe von Hints (`setHint()`) zur Auswahl von Slices für JPQL-Anfragen oder zu Möglichkeiten der Replikation von Daten.

14 Ausblick auf JPA 2.1

Die Entwicklung der JPA-Spezifikation ist keinesfalls abgeschlossen. Die Vielzahl der unterschiedlichen Erweiterungen der einzelnen JPA-Implementierungen gibt einen Anhaltspunkt dafür, welche weiteren speziellen Anforderungen vorhanden sind, um die objektorientierte Welt mit einer relationalen Datenbank in Einklang zu bringen.

Der Umfang der neuen JPA-Version 2.1 wird gegenwärtig diskutiert und soll im dritten Quartal 2012 verabschiedet werden. Wir wollen diesen Abschnitt nutzen, um einige der neuen Merkmale genauer zu betrachten. Weil der abschließende Umfang noch nicht feststeht, kann es in diesem Bereich noch zu Änderungen kommen [URL-JPA2.1].

JPA und Stored-Procedures

Die verschiedenen Datenbank-Implementierungen bieten fast alle die Möglichkeit an, Prozeduren und Funktionen über den SQL-Standard hinaus in sogenannten *Stored Procedures* abzulegen. Die Programmierung dieser Prozeduren erfolgt meist in herstellerspezifischen Programmiersprachen. Bekannte Vertreter dieser Sprachen sind Oracle mit *PL/SQL*, *Transact SQL* für den Microsoft-SQL-Server oder auch *PL/pgSQL* von Postgres.

Bis zur Version 2.0 lassen sich diese Stored Procedures nur mit einer nativen SQL-Abfrage oder mit der in Abschnitt 8.2.6 gezeigten `function()`-Methode der Criteria-API benutzen. Mit JPA 2.1 wird die Annotation `@NamedStoredProcedureQuery` über das bekannte Query-Interface einen einfachen Zugriff auf die erweiterten Datenbankfunktionen liefern.

```
@Entity
@NamedStoredProcedureQuery(
  name = "customerCount",
  procedureName = "count_customers",
  resultClasses = {Kunde.class},
  parameters = {
    @StoredProcedureParameter(
      name="plz",
      mode=ParameterMode.IN,
      type=String.class
    )
  }
)
public class Kunde {
  ...
}
```

```
...
StoredProcedureQuery q =
    em.createNamedStoredProcedureQuery("customerCount");
q.setParameter("plz", "04910");
q.execute();
q.getResultList();
```

In unserem Beispiel definieren wir am Entity `Kunde` eine Stored-Procedure-Query, die eine Datenbankfunktion namens `count_customers` aufruft. Die Query hat einen Parameter – eine Postleitzahl, zu der die Kunden in der Datenbank zu zählen sind. Die Benutzung der Query erfolgt wie bei anderen JPA-Abfragen über die `EntityManager`-Instanz. Eine Instanz der Query wird über die Methode `EntityManager.createNamedStoredProcedureQuery()` erzeugt und liefert das Ergebnis mit den bekannten Methoden wie z.B. `getResultList()` oder `getSingleResult()`.

Parallel dazu wird die JPQL um das Schlüsselwort `FUNCTION()` erweitert. Mit dieser Funktion lassen sich die Prozeduren und Funktionen auch innerhalb der Abfragesprache von JPA benutzen und typsicher wie bei der Criteria-API im Ergebnis verwenden.

Konstruktoren in JPA-Abfragen

Im Beispiel auf Seite 202 haben wir bereits gezeigt, wie sich aus einer Abfrage heraus mit dem Schlüsselwort `NEW` neue Instanzen einer Java-Klasse erzeugen lassen. Mit der neuen Annotation `@ConstructorResult` lassen sich ab Version 2.1 beliebige Konstruktoren mit den entsprechenden Parametern in einem `@SqlResultSetMapping` definieren. Native SQL-Abfragen, die das Result-Mapping entsprechend referenzieren, liefern dann auch ohne das Schlüsselwort `NEW` das gewünschte Ergebnis.

```
@NamedNativeQuery(
  name = "kundeProStadt",
  query = "select a.plz, a.ort, count(k.id) anzahl "
    + " from kunde k, adresse a "
    + " where k.wohnort_id = a.id group by a.ort, a.plz",
  resultSetMapping = "kundeStadtResult"
)
@SqlResultSetMapping(
  name = "kundeStadtResult",
  classes = {
    @ConstructorResult(
      targetClass = KundeStadt.class,
      columns = {
        @ColumnResult(name = "plz"),
        @ColumnResult(name = "ort"),
        @ColumnResult(name = "anzahl", type=Long.class)
      }
    )
  }
)
public class Kunde {
  ...
}
```

Die native SQL-Abfrage `kundeProStadt` zählt die Kunden, gruppiert nach Wohnort und Postleitzahl. Das Ergebnis dieser Abfrage benutzt die im `@SqlResultSetMapping` definierte Struktur, in diesem Fall die neue Annotation `@ConstructorResult` der Klasse

KundeStadt. Natürlich muss die Zielklasse einen entsprechenden Konstruktor mit den angegebenen Parametern besitzen. Ein auf diese Weise instanziiertes Entity befindet sich im Zustand *neu*. Wird die Id mit selektiert, verwaltet es der Entity-Manager als *detached*.

Neue Methoden des Persistenzkontextes

Nachdem in Version 2.0 der EntityManager und die Query bereits mit der Methode unwrap() Zugriff auf die jeweilige implementierende Klasse des Providers ermöglicht haben, werden diese Methoden in 2.1 auch in den JPA-Interfaces javax.persistence.Cache und javax.persistence.EntityManagerFactory definiert.

Das Interface EntityManager wird darüber hinaus um die Methode joinTransaction() erweitert. JPA 2.1 wird es zulassen, einen Persistenzkontext zu erzeugen, der sich zunächst nicht mit der zugrunde liegenden Transaktion synchronisiert. Der Zustand des in einer Java-EE-Anwendung injizierten Persistenzkontextes ist abhängig vom konfigurierten Synchronisationstyp des Kontextes. Mögliche Werte dieser neuen SynchronizationType-Enum sind SYNCHRONIZED und UNSYNCHRONIZED. Die joinTransaction()-Methode bindet den Kontext dann zu einem späteren Zeitpunkt an den Transaktions-Mechanismus.

Auf Basis einer existierenden Query-Instanz werden sich mit der neuen Methode addNamedQuery() zur Laufzeit des Persistenzkontextes weitere Abfragen zur EntityManagerFactory-Instanz hinzufügen lassen. Diese Abfragen müssen also nicht notwendigerweise mit einer entsprechenden Annotation oder einem XML-Element konfiguriert wurden sein.

ON-Conditions bei Left- und Fetch-Joins

In Abschnitt 7.3.3 haben wir mehrfach das Problem adressiert, bei Left-Outer-Joins und zusätzlichen Abfrage-Bedingungen auf der rechten Seite des Join alle Werte zu bekommen. Das neue Schlüsselwort ON in der JPQL wird genau diese Abfragen ermöglichen.

```
select k.nachname, a.plz
from Kunde k left join Adresse a
on a.plz = '04910'
```

Das daraus generierte SQL-Statement hat neben der eigentlichen Join-Bedingung der beiden Entities auch die zusätzliche Bedingung der rechten Seite mit in der ON-Bedingung. Damit werden auch die Kunden ohne Adresse selektiert.

```
select k.nachname, a.plz from kunde k left join adresse a
on (k.wohnort_id = a.id and a.plz = '04910')
```

Das nachstehende JPQL-Statement konnte diese Anforderungen bis jetzt nicht erfüllen. Es liefert nicht die gleichen Ergebnisse wie das obige Statement.

```
select k.nachname, a.plz
from Kunde k left join Adresse a
where a.plz = '04910'
```

Neues in der Criteria-API

Bisher waren Löschvorgänge bzw. Änderungen von Attributen bei einer größeren Menge von Entities nicht mit der Criteria-API möglich. Diese Vorgänge mussten entweder direkt auf der

Datenbank oder aber mit Befehlen der JPQL, beschrieben in Abschnitt 7.6, durchgeführt werden. Die neue Criteria-API in Version 2.1 bietet mit den Methoden `createCriteriaUpdate()` und `createCriteriaDelete()` auch einen typsicheren Weg, entsprechende Massenänderungen durchzuführen.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaDelete<Kunde> deleteQ = cb.createCriteriaDelete(Kunde.class);
Root<Kunde> c = deleteQ.from(Kunde.class);
deleteQ.where(cb.isEmpty(c.get(Kunde_.konten)));
Query q = em.createQuery(deleteQ);
q.executeUpdate();
```

Das dargestellte neue Criteria-Objekt entspricht der folgenden JPQL-Abfrage:

```
delete
from Kunde k
where k.konten is empty
```

Diese Änderungen werden am Persistenzkontext vorbei direkt in die Datenbank geschrieben, gehen damit auch an den potenziell vorhanden optimistischen Lock-Mechanismen vorbei. Das `@Version`-Attribut muss daher vom Entwickler manuell angepasst werden.

Eine weitere Neuerung sind die überladenen `treat()`-Methoden, deren gleichnamiges Schlüsselwort auch in der JPQL zu finden sein wird. Mit Hilfe des `Treat`-Mechanismus ist ein Type-Cast auf den Subtyp einer Oberklasse möglich. Im Ergebnis lassen sich die gecasteten Entities und deren Attribute in einer Abfrage verwenden:

```
CriteriaQuery<Konto> cq = cb.createQuery(Konto.class);
Root<Konto> c = cq.from(Konto.class);
cq.where(cb.or(
  cb.gt(treat(c, Sparkonto.class).get(Sparkonto_.guthabenzinssatz),
        new BigDecimal("2.0")),
  cb.gt(treat(c, Girokonto.class).get(Girokonto_.
      ueberziehungszinssatz),
        new BigDecimal("8.0"))));
```

Der Sprung zur Version 2.1 ist nicht zu vergleichen mit den Änderungen, die zwischen JPA 1.0 und JPA 2.0 stattgefunden haben. Unsere aufgezeigten Neuerungen werden nichtsdestotrotz die Funktionalität und damit einhergehend auch die Akzeptanz der Java Persistence API weiter erhöhen.

Literatur

[ACM03] ALUR, Deepak ; CRUPI, John ; MALKS, Dan: *Core J2EE Patterns: Best Practices and Design Strategies.* 2. Auflage. Prentice Hall, 2003

[Amb03] AMBLER, Scott W.: *Agile Database Techniques.* Wiley, 2003

[HS00] HEUER, Andreas ; SAAKE, Gunter: *Datenbanken: Konzepte und Sprachen.* 2. Auflage. mitp-Verlag, 2000

[Kar10] KARWIN, Bill: *SQL Antipatterns.* Pragmatic Bookshelf, 2010

[KK08] KLINE, Kevin ; KLINE, Daniel: *SQL in a Nutshell.* 2. Auflage. O'Reilly, 2008

[MW07] MÜLLER, Bernd ; WEHR, Harald: *Java Persistence API mit Hibernate.* Addison-Wesley, 2007

[Mü10] MÜLLER, Bernd: *JavaServer Faces 2.0 – Ein Arbeitsbuch für die Praxis.* 2. Auflage. Hanser, 2010

[Oes09] OESTEREICH, Bernd: *Analyse und Design mit UML 2.3.* 9. Auflage. Oldenbourg, 2009

[RB10] RUBINGER, Andrew L. ; BURKE, Bill: *Enterprise JavaBeans 3.1.* 6. Auflage. O'Reilly, 2010

[RQZ07] RUPP, Chris ; QUEINS, Stefan ; ZENGLER, Barbara: *UML 2 glasklar – Praxiswissen für die UML-Modellierung.* 3. Auflage. Hanser, 2007

[SM12] STEFFENS, Marc ; MÜLLER, Bernd: Weaving, Instrumentation, Enhancement: Was ein JPA-Provider so alles macht. In: *Java aktuell* (2012), Nr. 1

URL-Verzeichnis

[URL-ADM] Anemic Domain Model, Martin Fowler
http://martinfowler.com/bliki/AnemicDomainModel.html

[URL-BV] Bean Validation, JSR 303
http://www.jcp.org/en/jsr/summary?id=303

[URL-CDI] Contexts and Dependency Injection for the Java EE platform, JSR 299
http://www.jcp.org/en/jsr/summary?id=299

[URL-DN] DataNucleus Access Platform
http://www.datanucleus.org/

[URL-EECOMP] Java EE Compatibility
http://www.oracle.com/technetwork/java/javaee/overview/compatibility-jsp-136984.html

[URL-EH] Ehcache
http://ehcache.org/

[URL-EL] EclipseLink
http://www.eclipse.org/eclipselink/

[URL-EJB3] Enterprise JavaBeans 3.0, JSR 220
http://www.jcp.org/en/jsr/summary?id=220

[URL-EJB31] Enterprise JavaBeans 3.1, JSR 318
http://www.jcp.org/en/jsr/summary?id=318

[URL-GF] GlassFish Application Server
http://glassfish.java.net

[URL-H2] H2 Database Engine
http://www.h2database.com

[URL-HIB] Hibernate
http://www.hibernate.org

[URL-HSQLDB] HyperSQL
http://www.hsqldb.org

[URL-HV] Hibernate Validator
http://www.hibernate.org/subprojects/validator.html

[URL-INF] Infinispan
http://www.jboss.org/infinispan

[URL-JB] JavaBeans-Spezifikation
http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html

[URL-JBoss] JBoss Application Server
http://www.jboss.org/jbossas

[URL-JD] Java Decompiler
http://java.decompiler.free.fr/

[URL-JPA1] Java Persistence API 1.0, JSR 220, als Teil von EJB 3.0
http://www.jcp.org/en/jsr/summary?id=220

[URL-JPA2] Java Persistence API 2.0, JSR 317
http://www.jcp.org/en/jsr/summary?id=317

[URL-JPA2.1] Java Persistence API 2.1, JSR 338
http://www.jcp.org/en/jsr/detail?id=338

[URL-LUC] Apache Lucene
http://lucene.apache.org/

[URL-LUK] Luke — Lucene Index Toolbox
http://code.google.com/p/luke/

[URL-PerfEJB] Writing performant EJB Beans in the Java EE 5 Platform (EJB 3.0) Using Annotations
http://java.sun.com/developer/technicalArticles/ebeans/ejb_30/

[URL-OJPA] OpenJPA
http://openjpa.apache.org/

[URL-OCA] Open Clip Art Library
http://www.opencliparts.org/

[URL-Serv] Servlet 3.0, JSR 315
http://www.jcp.org/en/jsr/summary?id=315

Stichwortverzeichnis

A

Abfragesprache 188
ABS 196
@Access 37
<access> 248, 249
AccessType 37
ACID 59, 149, 323
Aggregationsfunktion 199
alias() 218
ALL 65, 94
allocationSize 32
@ApplicationScoped 173
array() 218
AS 188
ASC 115, 202, 268
@AssertFalse 229
@AssertTrue 229
<association-override> 257
@AssociationOverride 257
Atomicity 149
<attribute-override> 257
@AttributeOverride 48, 49, 128, 257
@AttributeOverrides 48, 49, 128
<attributes> 258
@Audited 293
Auditierung 293
AuditQuery 295
AuditQueryCreator 295
AuditReader 295
AuditReaderFactory 295
AuditStrategy 299
Aufzählungstyp 41
AUTO 28, 261
Auto-Commit-Modus 149
Automatic Dirty-Checking 61, 75, 282
AVG 199

B

@Basic 38, 39, 232, 262
<basic> 258, 262
Batch
– Delete 202
– Update 202
Bean Validation
– anwendungsdefinierte Constraints 239
begin() 27, 58, 176
Besitzer 90
Beziehung 89
– 1:n- 96, 98
– Besitzer einer 91
– bidirektionale 90
– mit Assoziationsklasse 109
– n:1 96
– n:m 104, 107
– rekursive 113
– Richtung einer 90
– unidirektionale 90
BigDecimal 33, 42, 262
BigInteger 33, 262
BLOB 39
Bridges 290
BYPASS 169
Byte-Code-Manipulation 282

C

Cache
– First-Level- 108, 166
– Second-Level 166
@Cacheable 167, 278, 301
CacheRetrieveMode 169
CacheStoreMode 169
Calendar 33, 38, 262
Callback-Methode 77
Canonical Model 209
<cascade> 264

CascadeType 65, 94, 248
CASE 196
CDI 172
<class> 50, 67, 70, 245
ClassFileTransformer 308
ClassTransformer 308
clear() 56, 63, 65
CLOB 39
close() 64
CMP 16
COALESCE 196
Collection 96, 115
<collection-table> 260
@CollectionTable 124, 127, 260
@Column 42, 82, 119, 185, 259
<column> 259
columnDefinition 42, 43
@ColumnResult 187, 252
<column-result> 252
commit() 58, 153
CONCAT 196
Consistency 149
@Constraint 239
ConstraintValidator 239
ConstraintViolation 230, 235
ConstraintViolationException 229
construct() 218
Container Managed Persistence → CMP
contain() 58
contains() 64, 167
Contexts and Dependency Injection → CDI
Convention over Configuration 34, 138, 143, 240, 320
Conversation → Konversation
Conversation 176
@ConversationScoped 76, 173, 176
Correlated Subquery 220
COUNT 199
createEntityManager() 66, 68
createEntityManagerFactory() 66, 68, 234, 310
createNamedQuery() 178
createNativeQuery() 178, 184
createQuery() 178, 295
CrossTypeRevisionChangesReader 298
CRUD 15, 63
CURRENT_DATE 196
CURRENT_TIME 196
CURRENT_TIMESTAMP 196

D

Data Transfer Object → DTO
Date 33, 38, 262
DateBridge 290
DDL 15
@DecimalMax 229
@DecimalMin 229
Default 236
Default-Listener 80
DefaultAuditStrategy 299
DefaultRevisionEntity 296
<delimited-identifiers> 248
DESC 115, 202, 268
<description> 244
DETACH 65, 94
detach() 63–65, 76
Detached Object 56
@Digits 229
Dirty Reads 150
<discriminator-column> 256
<discriminator-value> 254, 256
@DiscriminatorColumn 135, 256
DiscriminatorType 135, 257
@DiscriminatorValue 136, 254
Diskriminator 135
– -spalte 133
distinct 191
DistributionPolicy 321
DML 15
@DocumentId 286
Domain-Model
– Anemic 101
– Rich 101
Don't Repeat Yourself → DRY
DRY 227
DTO 35
DTYPE 138
Durability 149

E

EAGER 84, 102, 131, 190, 263
eclipselink.read-only 61
@EJB 175
<element-collection> 259
@ElementCollection 122, 123, 259
@Embeddable 48, 50, 270
<embeddable> 270
@Embedded 269
<embedded> 258, 269
<embedded-id> 262
@EmbeddedId 48, 262
end() 27
Enhancement 60, 318
Enterprise JavaBean → EJB

Entity 25
– Serialisierbarkeit 35
@Entity 26, 34, 252
<entity> 252
Entity-Listener 77, 79
<entity-listener> 80, 271
<entity-listeners> 271
Entity-Manager 27, 55
– anwendungsverwaltet 55, 56, 64, 76
– container-verwaltet 55, 68, 76
<entity-mappings> 247
<entity-result> 252
@EntityField 187
@EntityListener 271
@EntityListeners 79, 271
EntityManager 27, 55, 57, 65, 275
EntityManagerFactory 55, 65, 244
EntityNotFoundException 60, 154
@EntityResult 187, 252
EntityTransaction 58, 153
entrySet() 120
enum 41
@Enumerated 41, 262
<enumerated> 262
Enumeration 41
EnumType 41
Envers 293
equals() 25, 46, 108, 111
evict() 167
evictAll() 167
@ExcludeDefaultListeners 82, 101
<exclude-default-listeners> 254
@ExcludeSuperclassListeners 80
<exclude-superclass-listeners> 254
<exclude-unlisted-classes> 245

F

<f:validateBean> 238
fetch 38, 102, 123, 283
Fetch Join 192
FetchType 102, 131, 263
FIELD 37, 249
@Field 289
<field-result> 252
@FieldResult 252
find() 27, 56, 59–61, 169, 177
FinderTargetPolicy 323
First-Level-Cache 166
flush() 58
FlushModeType 204
forEntitiesAtRevision() 295
forEntitiesModifiedAtRevision() 295
forRevisionsOfEntity() 295
from 188
FullTextSession 287, 292
function() 223
@Future 229

G

Gelöschtes Objekt 56
<generated-value> 261
@GeneratedValue 27, 47, 261
GenerationType 28
Geronimo 77
getCache() 167
getConstraintViolations() 230
getDelegate() 77, 275
getPersistenceUnitUtil() 83
getPersistenceUtil() 83
getReference() 56, 59, 60, 177
getResultList() 179
getRevisionDate() 298
getRevisionNumberForDate() 298
getRevisions() 298
getRollbackOnly() 58
getSingleResult() 179
getTransaction() 58, 153
GlassFish 68
Group By 200
groups 237, 239

H

hashCode() 25, 46, 111
Having 200
Hibernate
– -Search 284
Hint 59
Historische Datenverarbeitung 293

I

@Id 26, 36, 46, 81, 261
<id> 258, 261
<id-class> 271
@IdClass 47, 111, 271
Identitätsoperator 25
IDENTITY 28, 32, 78, 261
Impedance Mismatch 16
Implizite Joins 189
IN 195
INDEX 196
@Indexed 286
@IndexedEmbedded 289
@Inheritance 135, 272
@Inheritance-Kombination 146

<inheritance> 272
initialize() 240
@Inject 175
insertable 42–44
Instrumentation 60, 282
Instrumentation 308
InvalidClassException 122
isActive() 58
Isolation 149
Isolation-Level 152
isValid() 240
Iterator 115

J

<jar-file> 245
Java-Agent 318
Java Community Process → JCP
Java Data Objects → JDO
Java-EE 35
Java Persistence API → JPA
Java Specification Request → JSR
Java Transaction API → JTA
java.lang.instrument
– -Package 308
java.persistence.metamodel
– -Package 26
java.sql-Package 39
javaagent 309, 319
javax.enterprise.context
– -Package 173, 176
javax.persistence
– -Package 26
– Hint-Präfix 61
– Präfix für Property-Namen 67
javax.persistence.jdbc.driver 66
javax.persistence.jdbc.password 66
javax.persistence.jdbc.url 66
javax.persistence.jdbc.user 66
javax.persistence.lock.scope 165
javax.persistence.lock.timeout 163, 165
javax.persistence.spi
– -Package 66, 308
javax.persistence.validation.mode-Property 234
javax.validation
– -Package 229
javax.validation.groups-Package 236
JBoss-AS 68
JCP 17
JDBC 15, 149
JDO 16
JNDI 245
JNDI-Name 69
Join 188
Join-Tabelle 105
@JoinColumn 91, 93, 99, 106, 264
<join-column> 264
JOINED 135, 139, 256
@JoinTable 106, 269
<join-table> 269
JpaEntityManager 276
JPQL 145, 188
JTA 70, 244
JTA-Data-Source 69
<jta-data-source> 70, 245

K

Kanonisches Modell 209
Kardinalität 90
Kartesisches Produkt 193
Komposition 121
Konversation 172

L

L2-Cache 166
label 240
LAZY 102, 131, 190, 192, 263, 318
Legacy 51
LENGTH 197
length 42, 43
Link-Tabelle 105
List 96, 115
Loading
– -Lazy 232
– Eager 102
– Lazy 102, 282, 283
@Lob 39, 262, 283
<lob> 262
LOCATE 197
lock() 166
LockModeType 182, 251
– OPTIMISTIC 157
– OPTIMISTIC_FORCE_INCREMENT 160
– PESSIMISTIC_FORCE_INCREMENT 161
– PESSIMISTIC_READ 161
– PESSIMISTIC_WRITE 161
LockTimeoutException 154
Losgelöstes Objekt 56
Lost-Update-Problem 157
LOWER 197
Lucene 284

M

Managed Object 56

@ManagedBean 174
@ManyToMany 104, 105, 265, 267
<many-to-many> 258, 267
@ManyToOne 99, 100, 113, 265, 266
<many-to-one> 258, 266
Map 96, 115
@MapKey 119, 120, 272
<map-key> 272
@MapKeyClass 273
<map-key-class> 273
@MapKeyColumn 273
<map-key-column> 273
@MapKeyEnumerated 273
<map-key-enumerated> 273
MapKeyJoinColumn 129
@MapKeyJoinColumn 130, 273
<map-key-join-column> 273
MapKeyJoinColumns 129
@MapKeyTemporal 273
<map-key-temporal> 273
<mapped-by> 265
<mapped-superclass> 270
mappedBy 95, 107, 113
@MappedSuperclass 145, 270
<mapping-file> 245
MAX 199
@Max 229
Meet-in-the-Middle-Variante 45
MERGE 65, 94
merge() 63, 64, 74
message 230, 239
META-INF 87
META-INF (Verzeichnis) 244
metadata-complete 252
MIN 199
@Min 229
MOD 197
@Multitenant 310

N

n+1-Select-Problem 131
name 42, 43, 99
<named-native-query> 251
<named-query> 251
@NamedNativeQuery 187, 251
NamedParameter 183
@NamedQueries 181
@NamedQuery 181, 182, 251
Non-Correlated Subquery 220
<non-jta-data-source> 245
Non-repeatable Read 150
NonUniqueResultException 154, 179
NoResultException 154, 179
@NotNull 227, 229
@Null 229
nullable 42, 43, 92
NULLIF 197

O

Objektmodell
– feinkörniges 51
@OneToMany 96, 97, 113, 265, 266
<one-to-many> 258, 266
@OneToOne 90, 93, 265
<one-to-one> 258, 265
OpenJPAEntityManager 276
Optimizer-Hint 182
optional 38, 99
OR-Mapping 16
Oracle
– Isolation Level 152
Order By 201
<order-by> 268
<order-column> 272
@OrderBy 115, 268
@OrderColumn 116, 117, 272
org.apache.openjpa.slice 321
org.hibernate.envers
– -Package 293
orm.xml 80, 88, 245
Orphan-Removal 94
orphanRemoval 94, 101
Outer Join 132

P

<package> 248
Paradigm Mismatch 16
ParameterizedBridge 291
@Past 227, 229
@Pattern 227, 229
payload 239
Performanz
– -risiko durch EAGER 132
PERSIST 65, 94, 248
persist() 27, 57, 93
Persistence
– by Reachability 59, 93
Persistence 65, 83, 234
<persistence> 66
Persistence Context 55, → Persistenzkontext
Persistence Unit → Persistenzeinheit
persistence.xml 50, 55, 65, 68, 69, 72, 233, 243, 320
@PersistenceContext 68, 72–74

@PersistenceUnit 68, 73, 74
<persistence-unit> 66, 244
<persistence-unit-metadata> 248
PersistenceUnitTransactionType 66
PersistenceUnitUtil 83
PersistenceUtil 83
Persistenz
– transitive 59, 93
Persistenzeinheit 28, 55, 65, 76, 86, 320
– Name der 66
Persistenzkontext 55, 65, 76
– erweiterter 74, 77
– Synchronisation mit Datenbank 58
– transaktionaler 74, 76
PessimisticLockScope 165
Phantom Reads 151
POJO 32, 33
PositionalParameter 184
@PostLoad 78, 271
<post-load> 271
@PostPersist 78, 271
<post-persist> 271
@PostRemove 78, 271
<post-remove> 271
@PostUpdate 78, 271
<post-update> 271
precision 42, 44, 101
Predicate() 224
@PrePersist 78, 271
<pre-persist> 271
@PreRemove 78, 271
<pre-remove> 271
@PreUpdate 78, 271
<pre-update> 271
Primärschlüssel
– Erzeugung 26
<primary-key-join-column> 256
@PrimaryKeyJoinColumn 140, 256
Projektion 188
<properties> 67, 245
PROPERTY 37, 249
Property
– Definition 33
– read-only 44, 312
<property> 67
<provider> 245

Q

Query 178
Query Language 188
QueryHint 182
@QueryHint 182
QueryTargetPolicy 323
QueryTimeoutException 154

R

Read Committed 152
Read Uncommitted 152
Referenztyp 25
REFRESH 65, 94
refresh() 62, 94, 166
REMOVE 65, 94
remove() 63, 94
removed object 56
Repeatable Read 152
@RequestScoped 173
RESOURCE_LOCAL 66, 244
@Retention 239
Revision 293
RevisionType 294
rollback() 58, 154
Rolle 90

S

scale 42, 44, 101
<schema> 248
Second-Level-Cache 166
<secondary-table> 255
@SecondaryTable 51, 52, 255
@SecondaryTables 53, 256
select 198
select ... for share 164
Selektion 188
SEQUENCE 28, 143, 261
Sequence 28
<sequence-generator> 249
@SequenceGenerator 29, 135, 249
Serializable 152
Serializable 121
serialVersionUID 122
Servlet-Container 56
Session 275
Session-Bean
– Stateless 72
@SessionScoped 173
Set 96
– Collection-Typ 114
setFirstResult() 179
setHint() 170, 324
setLockMode() 166
setMaxResults() 180
setRollbackOnly() 58
SFSB 75
<shared-cache-mode> 167, 245

SINGLE_TABLE 135, 256
SIZE 197
@Size 227, 229
– für Collections 233
Slice 320
SLSB 72
SQL 15
– DDL 15
– DML 15
<sql-result-set-mapping> 252
@SqlResultSetMapping 186, 252
@SqlResultSetMappings 187
SQRT 197
@Stateful 75, 176
@Stateless 72
@StaticMetamodel 26
StringBridge 291
SUBSTRING 197
SUM 199
Surrogatschlüssel 27

T

TABLE 28, 143, 261
@Table 35, 254
<table> 254
TABLE_PER_CLASS 141
TABLE_PER_SUBCLASS 135
@TableGenerator 30, 31, 250
<table-generator> 250
@Target 239
targetAuditMode 297
targetClass 123
targetEntity 97
@Temporal 39, 262
<temporal> 262
TemporalType 184
@TenantDiscrimatorColumn 311
@TenantDiscrimatorColumns 311
Theta-Join 193
Time 33, 262
Timestamp 33, 262
transaction-scoped 64
transaction-type 66
@TransactionAttribute 154, 176
Transaktion
– langlaufende 172
– Resource-Local 58
transaktionsbegrenzt 64
Transaktionsverwaltung 153
– deklarative 154
– Resource-Local 154
transform() 308
@Transient 37, 38
transient 37, 258
Transientes Objekt 56
TraversableResolver 232
TRIM 197
Tuple 218
TYPE 198
TypedQuery 178, 224

U

Union 141
unique 42, 43
Unique-Constraint 34
<unique-constraint> 255
@UniqueConstraint 255
uniqueConstraints 42
unwrap() 276
updatable 42–44
UPPER 198
USE 169
UUID 26

V

@Valid 232
validate() 235
validateProperty() 235
validateValue() 235, 236
Validation 234
<validation-mode> 233, 245
validation.xml 240
ValidationMode 233
Validator 234
ValidatorFactory 234
Validierung
– alternative 237
Validierungsgruppe 235, 236
ValidityAuditStrategy 299
ValidTimeAuditStrategy 299
Verbund 188
Vererbung 133
@Version 145, 157, 263
<version> 258, 263
Verwaltetes Objekt 56
@Volljaehrig 239

W

Weaving 60, 282, 307
WEB-INF 85
WebLogic 77
WebSphere 77
where 193

X

<xml-mapping-metadata-complete> 248

HANSER

Das Arbeitsbuch für Java-Web-Entwickler

Müller
JavaServer Faces 2.0
Ein Arbeitsbuch für die Praxis
515 Seiten.
ISBN 978-3-446-41992-6

JavaServer Faces 2.0 sind ein Framework für die Entwicklung von Benutzerschnittstellen für oder besser als Teil einer Java-Web-Anwendung. Dieses Arbeitsbuch führt Sie Schritt für Schritt in die Programmierung mit JavaServer Faces ein. Sie erfahren, wie Sie damit moderne Benutzerschnittstellen für die Praxis entwickeln. Und natürlich geht es auch darum, wie JavaServer Faces in eine Java-Web-Anwendung zu integrieren sind. Behandelt werden auch Themen wie die Anbindung an eine Datenbank mit JPA und die Verwendung von CDI, das Erzeugen von PDFs und Ajax.

Verfolgen Sie Schritt für Schritt die Entwicklung einer Online-Banking-Anwendung und lernen Sie so anhand realer Aufgabenstellungen alle wichtigen Aspekte von JavaServer Faces 2.0 kennen. Mit Hilfe der Übungen, deren Lösungen Sie sich von der Website zum Buch herunterladen können, können Sie das Gelernte selbst ausprobieren und umsetzen.

Mehr Informationen zu diesem Buch und zu unserem Programm unter **www.hanser.de/computer**

HANSER

Alles wird gut.

Bernd Müller
JBoss Seam
Die Web-Beans-Implementierung
280 Seiten.
ISBN 978-3-446-41190-6

Seam, das innovative und agile Framework im Bereich von Java-EE, integriert JavaServer Faces, Enterprise JavaBeans und Java Persistence-API und macht viele Konfigurations- und Codierungsarbeiten überflüssig.

Mit Seam können Sie Webapplikationen deutlich schneller und einfacher entwickeln und bauen gleichzeitig auf eine skalierbare und standardisierte Java-EE-Architektur auf. Dieses Arbeitsbuch führt Sie in die Funktionsweise von Seam ein. Sie lernen die wichtigsten Konzepte und Komponenten im Detail kennen. Dazu zählen lang laufende Transaktionen, Authentifizierung und Autorisierung, PDF-Erzeugung, Geschäftsprozesse, AJAX und das Generieren und Testen von Seam-Anwendungen.

Mehr Informationen zu diesem Buch und zu unserem Programm unter **www.hanser.de/computer**

HANSER

Programmieren lernen leicht gemacht!

Ratz/Scheffler/Seese/Wiesenberger
Grundkurs Programmieren in Java
711 Seiten
ISBN 978-3-446-42663-4

Dieses Lehrbuch können Sie verwenden, um sowohl Java als auch das Programmieren zu lernen. Es setzt keinerlei Vorkenntnisse aus den Bereichen Programmieren, Programmiersprachen und Informatik voraus. Alle Kapitel sind mit Übungsaufgaben ausgestattet, die Sie zum besseren Verständnis bearbeiten können. Denn: Man lernt eine Sprache nur, wenn man sie auch spricht!

Von den Grundlagen der Programmierung und den elementaren Sprachelementen über die Objektorientierung bis hin zu Threads und Datenströmen und zur Entwicklung von Anwendungen in Netzen lernen Sie alles, was Sie für den erfolgreichen Einstieg in die Programmierung benötigen.

Mehr Informationen zu diesem Buch und zu unserem Programm
unter **www.hanser.de/computer**

HANSER

Java 7: Praxisnah und kompakt

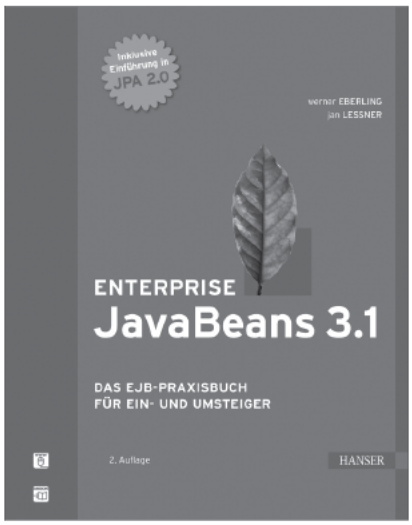
Jobst
Programmieren in Java
405 Seiten
ISBN 973-3-446-41771-7

Sie möchten sich Java von Grund auf aneignen? Dieses Standardwerk hat schon Tausende von Einsteigern zu Java-Profis gemacht. Kompakt, aktuell und präzise bietet es alles, was für die Programmierung in Java wichtig ist. Für die 6. Auflage wurde es grundlegend überarbeitet und konzentriert sich darauf, Ihnen den Einstieg in die Programmierung mit Java 7 möglichst einfach zu machen. Von Anfang an nutzt es dafür die leistungsfähige und komfortable Entwicklungsumgebung Eclipse. Von den elementaren Ausdrucksmöglichkeiten in Java und den Grundlagen der Objektorientierung bis hin zur Nebenläufigkeit, Programmierung in Netzwerken und Anbindung von Datenbanken finden Sie hier alle Themen, die für Einsteiger wichtig sind. Zahlreiche Beispiele und Aufgaben in allen Kapiteln – von elementaren Übungen bis hin zu kleinen Projektarbeiten – helfen Ihnen, Ihr Wissen praktisch umzusetzen und zu festigen.

Mehr Informationen zu diesem Buch und zu unserem Programm unter **www.hanser.de/computer**

HANSER

EJB 3.1 leicht gemacht.



Eberling/Leßner
Enterprise JavaBeans 3.1
Das EJB-Praxisbuch für
Ein- und Umsteiger
2., aktualisierte und
erweiterte Auflage
376 Seiten.
ISBN 978-3-446-42259-9

Wenn Sie sich nicht durch die mehrere Hundert Seiten starke Spezifikation von Enterprise JavaBeans kämpfen und mühsam selbst herausfinden wollen, was nun genau wie in Projekten wirklich funktioniert, dann sind Sie hier genau richtig. Werner Eberling und Jan Leßner stellen in diesem Buch die Standards EJB 3.1 und JPA 2.0 anschaulich vor und reichern diese mit ihrer Praxiserfahrung aus vielen Projekten an.

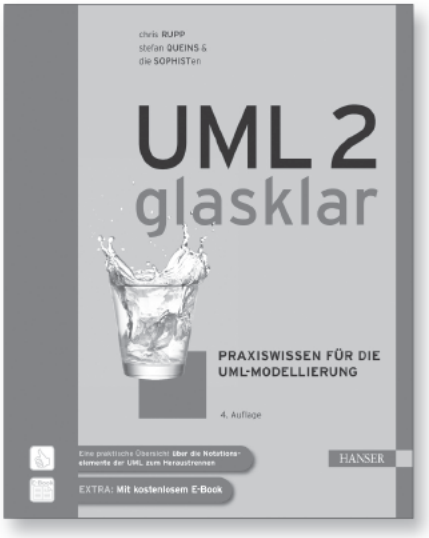
Ein- und Umsteiger in EJB 3.1 erhalten hier eine kompakte Darstellung der grundlegenden Aspekte von komponentenbasierten Mehrschichtsystemen und der Rolle, die EJBs und JPA-Entitäten darin spielen. Alle Aspekte beider Technologien werden anhand zahlreicher Beispiele und einer durchgehenden Applikation detailliert dargestellt und durch wertvolle Praxistipps ergänzt. Im neuen Kochbuchteil finden Sie Lösungen für häufig wiederkehrende Fragestellungen rund um EJBs und Enterprise-Applikationen, die Ihnen unmittelbar bei der Lösung Ihrer Aufgaben im Projekt helfen.

Mehr Informationen zu diesem Buch und zu unserem Programm
unter **www.hanser.de/computer**

HANSER

»Eines der besten auf dem deutschen Markt«

JavaMagazin



Chris Rupp, Stefan Queins,
die SOPHISTen
UML 2 glasklar
4. Auflage · 580 Seiten
ISBN 978-3-446-43057-0
€ 34,90

In diesem Buch lernen Sie die UML auf dem aktuellsten Stand kennen. Hier finden Sie alle Diagramme der UML ausführlich beschrieben und erfahren, wie Sie sie in Ihrem Projekt richtig anwenden. Zahlreiche Tipps, Tricks und anschauliche Beispiele helfen Ihnen dabei.

Als UML-Neuling erhalten Sie eine schrittweise Einführung in die Basiselemente, als erfahrener UML-Anwender können Sie die übersichtliche Darstellung der Notationsmittel der UML 2 als Nachschlagewerk nutzen. Wer sich auf die Prüfung zum Certified Professional für Requirements Engineering – Advanced Level Requirements Modeling des IREB e. V. vorbereiten will, erfährt hier, welche Teile der UML für ihn relevant sind.

Mehr Informationen zu diesem Buch und zu unserem Programm unter **www.hanser-fachbuch.de**

JAVA PERSISTENCE API 2 //

- Hier finden Sie eine aktuelle und fundierte Einführung in JPA.
- Sie lernen die wichtigsten Implementierungen von JPA kennen: Hibernate, EclipseLink und OpenJPA
- Mit einem Ausblick auf JPA 2.1
- Im Internet: Die Beispiele des Buches zum Herunterladen

Wer eine fundierte und praxisorientierte Einführung in die Java Persistence API 2.0 sucht, ist hier richtig. Die Autoren erklären nicht nur die zentralen Konzepte von JPA, sondern gehen auch auf fortgeschrittene Themen wie die Criteria-API und Bean-Validierung ein. Mit diesem Wissen können Sie anspruchsvolle Lösungen mit JPA umsetzen.

Darüber hinaus lernen Sie mit Hibernate, EclipseLink und OpenJPA die drei wichtigsten Implementierungen von JPA kennen. Sie erfahren, welche für die Praxis interessanten Erweiterungen des Standards diese Implementierungen anbieten und wie Sie diese konkret anwenden können.

Alle Themen werden anschaulich am Beispiel real existierender Programme aus dem Bankenbereich vorgestellt. Sie können, nein, Sie sollen diese Programme herunterladen, selbst ausprobieren und als Basis für eigene Entwicklungen verwenden.

Prof. Dr. Bernd **MÜLLER** lehrt Software-Technik an der Fakultät Informatik der Hochschule Braunschweig/Wolfenbüttel. Er ist Autor mehrerer Fachbücher und Zeitschriftenartikel, gefragter Sprecher auf Konferenzen sowie Mitglied der JCP-Expertengruppe für JPA 2.1.
Harald **WEHR** ist IT-Consultant bei der MAN Gruppe in Salzgitter. Neben der Java-Entwicklung ist sein Aufgabenschwerpunkt die SAP HCM System-Betreuung und -Entwicklung für die verschiedenen Unternehmensbereiche.

AUS DEM INHALT //

- Mapping von Objekten
- Objektlebenszyklus
- Objekte und Beziehungen
- Vererbung
- Transaktionen
- JPQL
- Das Criteria-API
- Bean-Validierung mit JSR 303
- JPA-Konfiguration
- Hibernate-Erweiterungen
- EclipseLink-Erweiterungen
- OpenJPA-Erweiterungen
- Ausblick auf JPA 2.1

UNSER BUCHTIPP FÜR SIE //

Müller, Java Server Faces 2.0
2010. 515 Seiten. FlexCover. € 39,90.
ISBN 978-3-446-41992-6

HANSER

€ 39,90 [D] | € 41,10 [A]
ISBN 978-3-446-42693-1
9 783446 426931

www.hanser-fachbuch.de/computer

Systemvoraussetzungen für eBook-Inside: Internet-Verbindung und eBookreader Adobe Digital Editions.