



Das Java Persistence API

Eine Einführung



Cegos Group

inspire
qualify
change



Inhalts- verzeichnis



Datenbank-Zugriff



Das Java Persistence API




Programmieren mit JPA



Datenbank-Zugriff



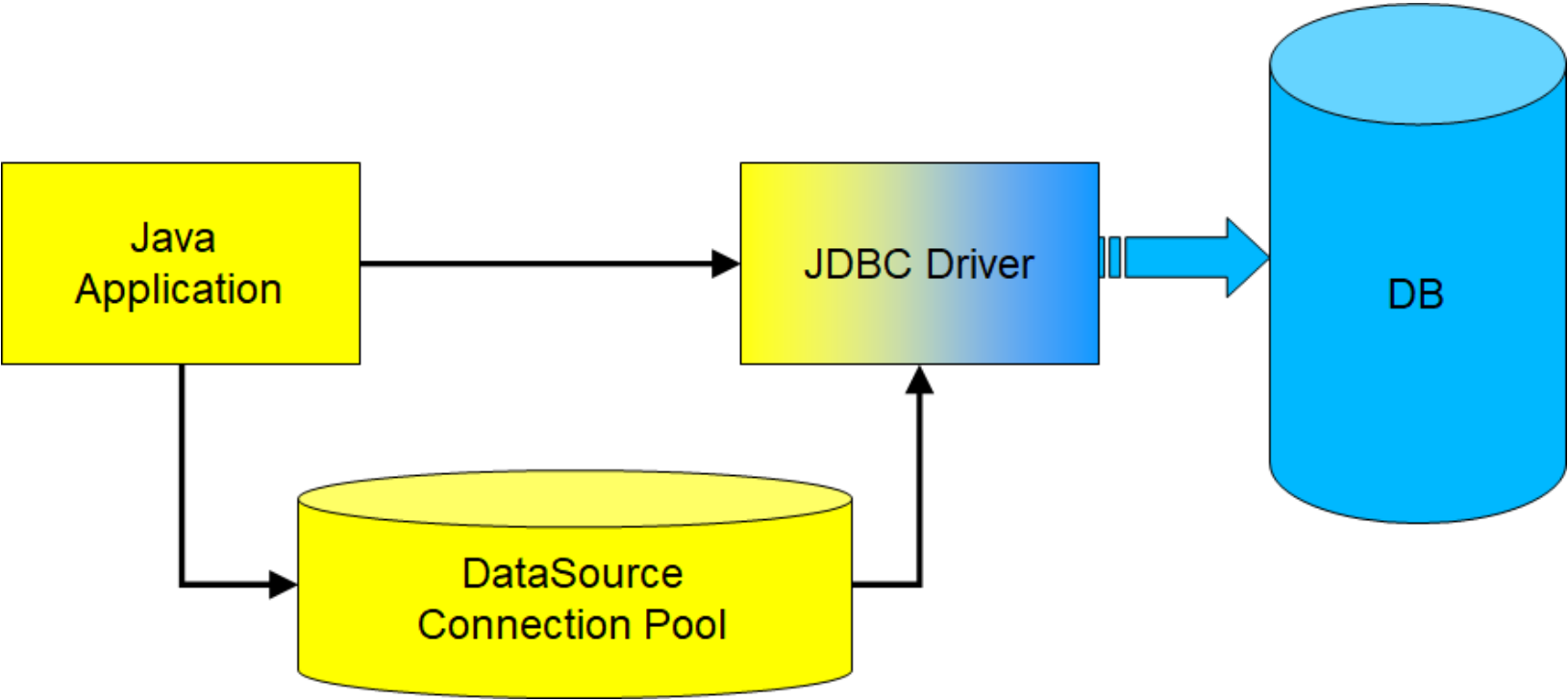
 Nativer SQL-Zugriff

 O/R-Mapper



Nativer SQL-Zugriff

Datenbank, Treiber und DataSource





Ablauf

- Eine konfigurierte `DataSource` enthält einen Connection Pool zur Datenbank
- Die Java-Anwendung
 - holt sich von der `DataSource` eine Connection
 - erzeugt die benötigten Statements
 - setzt diese ab
 - wertet die Ergebnisse aus
 - `ResultSet`
 - `SQLException`
 - schließt die Connection
 - diese wird in den Pool zurückgegeben, nicht real geschlossen!



O/R-Mapper



OO und relationale Datenbanken

- Die Objekte der Objektorientierung leben im Hauptspeicher und sind somit "flüchtig".
 - Die "Objekte" von Datenbanken sind persistent.
- Die Objektorientierung kennt "intelligente" Objekte – Objekte, die ihren Zustand kapseln. Die Klassen dieser Objekte enthalten Methoden, mittels derer der Zustand der Objekte abfragbar und manipulierbar ist.
 - Relationale Datenbanken dagegen enthalten nur "dumme" Daten.
- Relationale Datenbanken kennen andere Typen als objektorientierte Sprachen.
 - Die Datenbank kennt z.B. die Typen CHAR und VARCHAR; Java dagegen kennt den Typ String.
- In der objektorientierten Welt sind Objekte miteinander über Referenzen (also Pointer) verbunden.
 - In relationalen Datenbanken werden die "Objekte" über Fremdschlüssel-Beziehungen miteinander verbunden. Die Objektorientierung kennt aber keine Fremdschlüssel (und auch keine Primärschlüssel).



OO und relationale Datenbanken II

- Die Objektorientierung fokussiert individuelle Objekte; zwischen diesen Objekten kann navigiert werden. (Natürlich lassen sich solche individuellen Objekte auch in Collections zusammenfassen.)
 - Die typische Zugriffsweise von Datenbanken ist dagegen der SELECT in Verbindung mit dem JOIN – eine Zugriffsweise, die grundsätzlich Mengen von Zeilen liefert. Im Gegensatz zur Objektorientierung operiert die Datenbank also mengenorientiert.
- Die Objektorientierung kennt das Vererbungskonzept.
 - Relationale Datenbanken dagegen kennen mit wenigen Ausnahmen keine Vererbung.
- Relationale Datenbanken beruhen wesentlich auf dem Konzept der referenziellen Integrität;
 - in der Objektorientierung ist dieses Konzept von Natur aus unbekannt.



Aufgaben eines O/R-Mappers

- Abbildung von Tabellenzeilen auf Objekte und umgekehrt
 - Ein O/R-Mapper muss eine Zeile einer relationalen Tabelle auf ein Objekt abbilden können – und zwar in beide Richtungen: er muss die Spaltenwerte einer Tabellenzeile lesen und dieses dann den Attributen des Objekts zuweisen können; und er muss umgekehrt die Attributwerte eines Objekts auslesen können und diese den Spalten einer Tabellezeile zuweisen können.
- Abbildung von Primär-/Fremdschlüssel-Beziehungen auf Referenzen
 - Die in der Datenbank enthaltenen Primär-/Fremdschlüsselbeziehungen müssen auf referenzielle Beziehungen der Objekte abgebildet werden.
- Generierung von SQL-Statements
 - Wenn dem OR-Mapping die Abbildung zwischen dem Objektmodell und dem Datenbank-Schema über XML-Dateien bekannt gemacht wird, dann ist es natürlich auch möglich, die für das INSERT, das UPDATE und das DELETE erforderlichen SQL-Statements automatisch zur Laufzeit zu generieren.



Aufgaben eines O/R-Mappers

- Objektorientierte Abfragesprache
 - Für Abfragen sollte ein OR-Mapper eine eigene, objektorientierte Sprache anbieten.
- Identität von Objekten
 - Wird mittels eines OR-Mappers eine Tabellenzeile gelesen und in ein Objekt transformiert, so sollte dieses Objekt das einzige Objekt sein, welches die entsprechende Zeile im Hauptspeicher repräsentiert.
- Natives SQL
 - Für bestimmte Zwecke mag es sinnvoll oder gar notwendig sein, Abfragen oder DML-Befehle direkt in SQL zu formulieren. Der OR-Mapper sollte solche "workarounds" zulassen.



Das Java Persistence API



Architektur



Transaktionssteuerung



Architektur

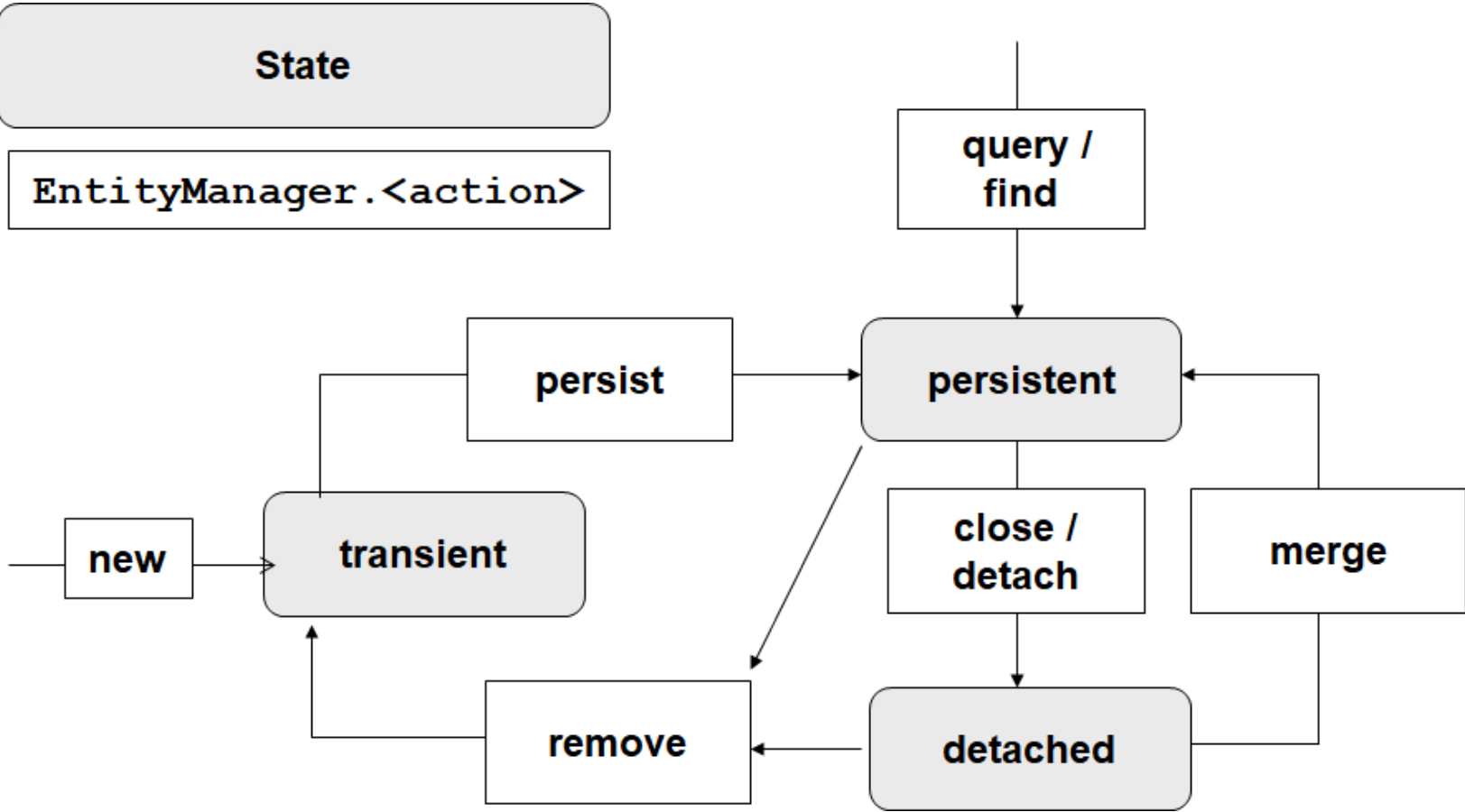


Persistenz mit dem Java Persistence API

- Die Entity ist eine normale Java-Klasse, die spezielle Annotations besitzt
 - Welches Attribut wird in welcher Tabellenspalte abgelegt
 - Optimierung der Datenbankzugriffe mit Lazy Loading, Fetching...
 - Alternativ kann auch eine XML-basierte externen Konfigurationsdatei benutzt werden
- Das Speichern, suchen etc. übernimmt der EntityManager
 - Das Objekt selber ist nicht per se persistent, sondern ist in verschiedenen Zuständen vorhanden:
 - Transient (keine Entsprechung zu einem Datensatz)
 - Persistent (entspricht einem Datensatz, der Entity Manager muss die Objekt-Identität garantieren)
 - Detached (entspricht einem Datensatz, ein detached Objekt ist ein unabhängiger Snapshot des Datenbestandes)
 - Die Zustandswechsel werden vom EntityManager gesteuert

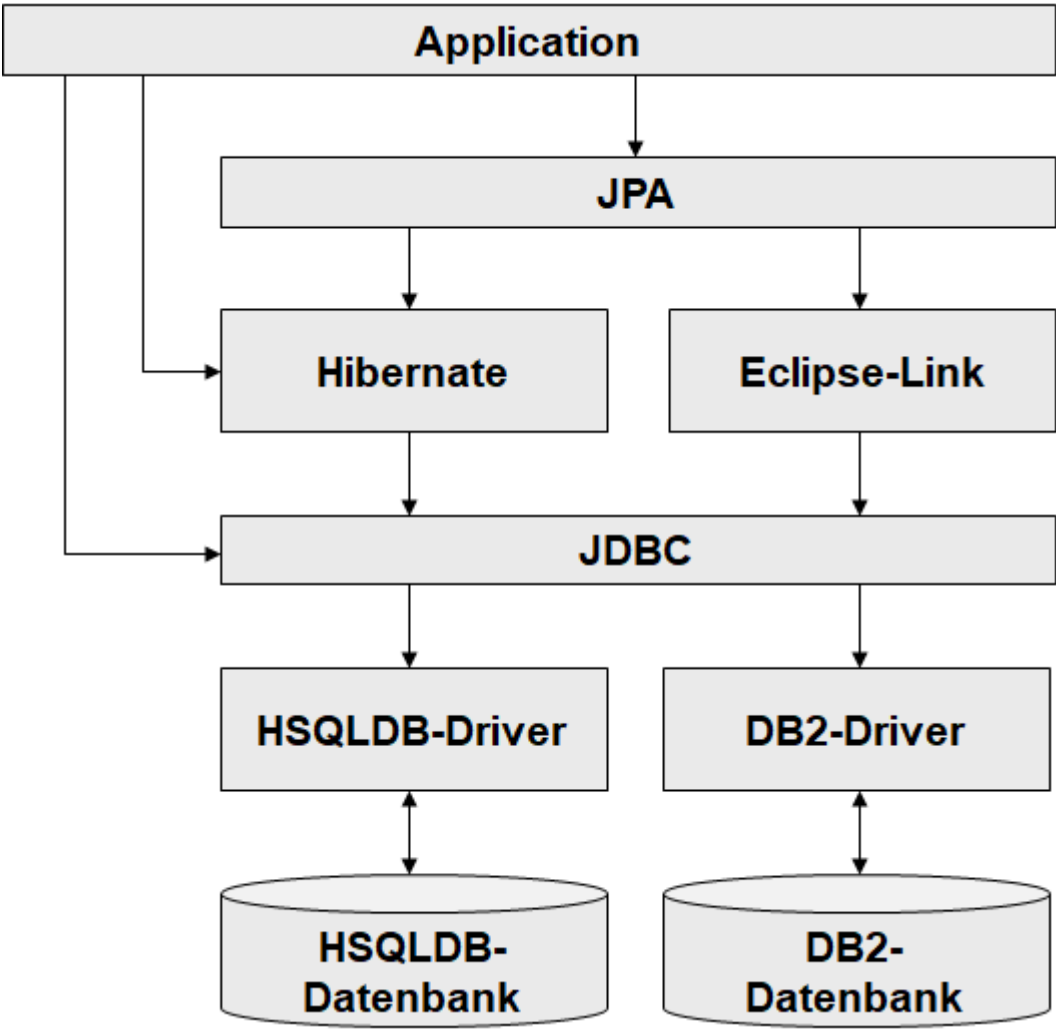


Diagramm





JPA Architektur





Transaktionssteuerung



Transaktions- Management

- Deklaratives Transaktions-Management
 - Container übernimmt die gesamte Transaktionsverwaltung
 - Definition eines Transaktions-Attributes auf Methodenebene
 - Die Propagierung der Transaktion wird vom Transaction Manager des Applikationsservers übernommen
- Die Transaktionssteuerung kann auch explizit vom Programmierer vorgenommen werden
 - Dazu wird der Bean vom Container eine `UserTransaction` übergeben
- Die Zuordnung der Transaktionsattribute zu einer Bean-Methode erfolgt durch Annotations
 - Auch eine externe Konfiguration über einen XML-Deskriptor ist möglich



Deklaratives Transaktions- Management

- Es existieren 6 Transaktions-Attribute
 - Attribut:
 - Not Supported
 - Supports
 - Required
 - Requires New
 - Mandatory
 - Never
- Transaktions-Attribute werden auf Methodenebene vergeben
 - Beim Aufruf einer Methode wird an Hand der Attribute eine neue Transaktion gestartet oder ein vorhandener Kontext übernommen



Required

- Die aufgerufene Methode enthält garantiert einen Transaktionskontext



Requires New

- Die aufgerufene Methode enthält garantiert einen Transaktionskontext



Mandatory

- Der Client muss einen Transaktionskontext besitzen
- Ansonsten: `TransactionRequiredException`



Not supported

- Die aufgerufene Methode läuft ohne Transaktionskontext



Never

- Der Client darf keinen Transaktionskontext besitzen
- Ansonsten: `RemoteException`



Supports

- Die aufgerufene Methode läuft mit dem Transaktionskontext der aufgerufenen Methode



Programmieren mit JPA



Der EntityManager



O/R-Mapping mit Annotations



Relationen



Queries



Der EntityManager



Das Java Persistence API

- Der `javax.persistence.EntityManager` stellt die folgenden Funktionalitäten zur Verfügung:
 - `persist(Object p)` speichert eine Entität
 - Suchen von Entitäten mit `find(Class entityClass, Object primaryKey)`
 - Erzeugen von Query-Objekten
 - Cache-API und Synchronisation mit der Datenbank (refresh, merge, flush)
 - Zugriff auf die aktuell laufende Transaktion
- Ein Applikationsserver muss eine Implementierung des `EntityManagers` zur Verfügung stellen.
 - Dies folgt über die Dependency Injection eines `PersisteneContext`.
 - Solch ein Kontext definiert im Wesentlichen, welche Implementierung verwendet werden soll und wie diese konfiguriert ist.
- Eine Query ist ein komfortables API um Datenbank-Abfragen zu definieren



O/R-Mapping mit Annotations



Annotations des JPA

- AttributeOverride
- AttributeOverrides
- Basic
- Column
- ColumnResult
- DiscriminatorColumn
- Embeddable
- EmbeddableSuperclass
- Embedded
- EmbeddedId
- Entity
- EntityListener
- EntityResult
- Enumerated
- FieldResult
- FlushMode
- GeneratedValue
- Id
- IdClass
- Inheritance
- JoinColumn
- JoinColumns



Flat Book

- Zur Definition einer simplen Entity werden nur zwei Annotations benötigt:

@Entity deklariert die Klasse als Entity-Klasse

@Id definiert die Spalte, die als Primärschlüssel benutzt werden wird

@Entity

```
public class FlatBook implements Serializable {
```

```
    @Id
```

```
    public String getIsbn() {
```

```
        return isbn;
```

```
    }
```



Extended Flat Book

- Mit den Annotations `@Table`, `@Basic`, `@Transient` und `@Column` werden zusätzlich Metadaten eingeführt:
`@Entity`
`@Table(name = "EXTENDED_FLAT_BOOK", uniqueConstraints =
 @UniqueConstraint(columnNames = { "COL_TITLE" })))`
`public class ExtendedFlatBook implements Serializable {`
 `@Transient`
 `public boolean isAvailable() {return available; }`
 `public void setAvailable(boolean available) { this.available =`
 `available;}`
 `@Basic(optional=false)`
 `public String getDescription() {`
 `return description;`
 `}`
 `@Column(length=100, name="COL_TITLE")`
 `public String getTitle() {return title;}`

 `}`



GeneratedKeyFlatBook

- Der Primärschlüssel des Buches kann auch automatisch erzeugt werden. Dazu dient die Annotation `@GeneratedValue`:

`@Entity`

```
public class GeneratedKeyFlatBook implements Serializable {
```

```
    @Id
```

```
    @GeneratedValue(strategy=GenerationType.SEQUENCE)
```

```
    public Long getIsbn() {
```

```
        return isbn;
```

```
    }
```



CompositeFlatKey

```
@IdClass(Isbn.class)
```

```
@Entity
```

```
public class CompositeKeyFlatBook implements Serializable {  
}
```



EmbeddedKeyFlatBo ok

- Der Primärschlüssel kann aber auch als Beispiel für einen "Embedded" Typ verwendet werden.
 - Dabei werden Attribute einer Entity im Objektmodell zu einer eigenen Klasse zusammengefasst, die keine eigene Entity ist, deshalb so im Datenmodell nicht auftritt. Embedded Typen sind somit klar von Relationen zu trennen.
- Dazu dienen die Annotations @Embeddable, @Embedded und @EmbeddedId.



Vererbung

- Entities können im Objektmodell in einer polymorphen Vererbungshierarchie stehen. Im Datenmodell kann dieses durch verschiedene Strategien nachgebildet werden:
 - Eine Typspalte innerhalb einer Tabelle, die alle Attribute aller Subklassen gemeinsam enthält
 - Pro Typ wird eine eigene Tabelle verwendet
 - Basisklassen und Subklassen werden durch eine Relation miteinander verknüpft



Single Table Inheritance

- Book, SchoolBook und SpecialistBook stehen in einer einfachen Vererbungshierarchie, die durch eine einzige Tabelle abgebildet werden soll:
- Um eine abgeleitete Klasse, z.B. SchoolBook definieren zu können, werden die folgenden Annotations benutzt:
 - @Inheritance definiert die Strategie der Umsetzung in das Datenmodell durch die Enumeration `javax.persistence.InheritanceStrategy`. Für das hier gewählte Beispiel wird `SINGLE_TABLE` gewählt:
 - Die zur Typ-Unterscheidung benutzte Spalte wird mit `@DiscriminatorColumn` definiert. Hier erfolgt die Angabe des Spaltennamens sowie des zu verwendende Typs
 - Die Typinformation erfolgt durch die Angabe eines Wertes der Spalte. Dieser wird mit dem `@DiscriminatorValue` angegeben



Table per Class

- Mit dieser Strategie wird pro nicht-abstrakte Klasse der Vererbungshierarchie eine eigene Tabelle angelegt
`@Entity(name="TablePerClassBook")`
`@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)`
`public abstract class Book implements Serializable {`



Vererbung durch Relationen

- Hier wird die Vererbung durch relational verknüpfte Tabellen aufgebaut.

```
@Entity(name="JoinTableBook")
```

```
@Inheritance(strategy = InheritanceType.JOINED)
```

```
public abstract class Book implements Serializable {
```



Mehrere Tabellen pro Entity

- Eine Entity kann auch auf mehrere Tabellen umgesetzt werden. Dazu dient die Annotation `@SecondaryTable`. Im folgenden Beispiel hat jedes Buch eine Verfügbarkeit, die in einer anderen Tabelle (`BOOK_STOCK`)abgelegt ist:

```
@Entity
```

```
@SecondaryTable(name="BOOK_STOCK",
```

```
pkJoinColumns={@PrimaryKeyJoinColumn(name="isbn")})
```




Relationen



One-to-One

- Ein Mapping einer Entity auf mehrere Tabellen ist eine Vorstufe zu einer One-to-One-Relation. Allerdings werden für eine echte Relation zwei Entitäten benötigt. So hat im Folgenden jedes Buch eine Verkaufsstatistik
- In der Klasse Book


```
@OneToOne(optional=false, cascade={CascadeType.ALL})
    @JoinColumn(
        name="BOOK_STATISTICS_ID", unique=true, nullable=false,
        updatable=false)
```
- In der Klasse BookStatistics


```
@OneToOne(optional=false, mappedBy="bookStatistics")
    public Book getBook() {
        return book;
    }
```



One-to-Many

- Die Definition einer One-to-Many-Relation benutzt in der Objektorientierten Welt das Collection-API. Im folgenden Beispiel hat ein Verleger eine Menge von Büchern
- In der Klasse Publisher
 - @ManyToOne
 - @JoinColumn(name="PUBLISHER_ID", nullable=false)
- In der Klasse Book
 - @OneToMany(cascade=CascadeType.ALL, mappedBy="publisher")



Many-to-Many

- Auch Many-to-Many-Relationen benutzen in der Objektorientierten Welt das Collection-API. Im folgenden Beispiel hat jedes Buch eine Liste von Büchern
- In der Klasse Book


```
@ManyToMany(cascade = { CascadeType.ALL })
@JoinTable(name = "BOOKS_AUTHORS", joinColumns = {
    @JoinColumn(name = "ISBN") }, inverseJoinColumns = {
    @JoinColumn(name = "AUTHOR_ID") })
```
- In der Klasse Author


```
@ManyToMany(cascade = { CascadeType.ALL }, mappedBy =
    "authors")
```



Ausblick

- Die Beispiele in den vorherigen Kapiteln zeigen den Datenzugriff mit dem neuen Persistenz-API. Allerdings sind die Konfigurationen noch nicht optimiert eingestellt. Begriffe wie
 - Sortierung
 - Kaskadierung
 - Lazy Loading
 - Fetching-Strategien
 - Massen-Updates und -Deletes
 - Natives SQL
- wurden noch nicht ausführlich gezeigt. Diese Themen werden im Integrata-Seminar 33014: "e-Business-Anwendungen mit J EE" vertieft.



Queries



Abfragen

- So wie die meisten O/R-Mapper definiert auch das Java Persistence API eine eigene Query-Sprache. Die Syntax dieser Sprache ist zwar angelehnt an Standard-SQL, enthält aber auch Elemente der Objektorientierung:
 - Objektorientierte Formulierung der Abfragen
 - Polymorphe Abfragen



Suche aller Verleger

- Die folgende Testmethode sucht alle Verleger:

```

public void searchAllPublishersQuery() {
    Object result = queryTest
        .executeEjbQl("Select publisher from QueryPublisher
as publisher");
    Collection<Publisher> publishers = (Collection<Publisher>) result;
    assertEquals("There must be 4 Publishers!", 4, publishers.size());
    for (Publisher publisher : publishers) {
        System.out.println(publisher.getPublisherName() + " has "
            + publisher.getBooks().size() + " books");
    }
}

```




Suche einen bestimmten Verleger

```
public void searchOnePublisherQuery() {
    Object result = queryTest
        .executeEjbQl("Select publisher from QueryPublisher as publisher where publisher.publisherId = 1");
    Collection<Publisher> publishers = (Collection<Publisher>) result;
    assertSame("There must be 1 Publishers!", 1, publishers.size());
    for (Publisher publisher : publishers) {
        System.out.println(publisher.getPublisherName() + " has "
            + publisher.getBooks().size() + " books");
    }
}
```



Parametrisierte Suche nach einem Verleger

```
public void searchOnePublisherParameterizedQuery() {
    Object result = queryTest
        .executeParemetrizedEjbQl("Select publisher from
QueryPublisher as publisher where publisher.publisherId = ?1",
        new Long(2));
    Collection<Publisher> publishers = (Collection<Publisher>) result;
    assertSame("There must be 1 Publishers!", 1, publishers.size());
    for (Publisher publisher : publishers) {
        System.out.println(publisher.getPublisherName() + " has "
            + publisher.getBooks().size() + " books");
    }
}
```



Bedingungen unter Verwendung der Relationen

```
public void searchBestsellers() {
    Object result = queryTest
        .executeParametrizedEjbql("Select book from QueryBook
        as book where book.bookStatistics.sold > 35");
    Collection<Book> books = (Collection<Book>) result;
    assertSame("There must be 4 matching Books!", 4, books.size());
    for (Book book : books) {
        System.out.println(book.getTitle() + " is sold "
            + book.getBookStatistics().getSold() + " times");
    }
}
```



Verknüpfte Bedingungen

```
public void searchBestsellersWithTitleEndingWith7() {
    Object result = queryTest
        .executeParametrizedEjbql("Select book from QueryBook
as book where book.bookStatistics.sold > 35 and book.title like
'%7'");
    Collection<Book> books = (Collection<Book>) result;
    assertSame("There must be 1 matching Book!", 1, books.size());
    for (Book book : books) {
        System.out.println(book.getTitle() + " is sold "
            + book.getBookStatistics().getSold() + " times");
    }
}
```



Joins

```
public void searchBestsellersByAuthor4() {
    Object result = queryTest
        .executeParametrizedEjbql("Select book from QueryBook
        as book join book.authors as author where
        book.bookStatistics.sold > 35 and author.authorId=4");
    Collection<Book> books = (Collection<Book>) result;
    assertSame("There must be 2 matching Books!", 2, books.size());
    for (Book book : books) {
        System.out.println(book.getTitle() + " by " + book.getAuthors() + "
        is sold "
            + book.getBookStatistics().getSold() + " times");
    }
}
```



Subqueries

```
public void searchBooksWihMoreThan3Authors() {
    Object result = queryTest
        .executeParametrizedEjbQl("Select book from QueryBook
as book where (select count(authors) from book.authors as
authors) >= 3");
    Collection<Book> books = (Collection<Book>) result;
    assertSame("There must be 4 matching Books!", 4, books.size());
    for (Book book : books) {
        System.out.println(book.getTitle() + " has 3 or more authors: " +
            book.getAuthors());
    }
}
```