Zusammenfassung

**Lernziele**

* Datenbanksysteme (DBMS) programmieren können (Stored Procedures, Triggers)
* Programmierschnittstellen für die Applikationsentwicklung vertiefter anwenden (JDBC)
* Werkzeuge zur DB-Anbindung an Applikationen einordnen können
* Grundlagen von objekt- relationalen sowie von objekt-orientierten DBMS kennen
* Semistrukturierte DBMS mit XML kennen
* Information Retrieval sowie Data Warehouse/OLAP einordnen können

**Unterlagen / Bücher**

* Elmasri & Navathe (2009): Fundamentals of Database Systems, Verlag AddisonWesley
* Ramakrishnan & Gehrke (2002): Data Base Management Systems, 3rd Ed., ISBN 0-07-115110-9
* Kemper & Eickler, Datenbanksysteme, Verlag Oldenbourg, ISBN 3-486-25706, 7. Auflage mit Übungsbuch
* Heuer & Saake (2000): Datenbanken: Konzepte und Sprachen, mitp-Verlag, 2. Auflage, 3-8266-0619-1

**Lerninhalte**

* Stored Procedures (PL/SQL)
* Cursors, Triggers, Udateable Views
* Backup und Recovery
* Indexe
* Objektrelationale Datenbanksysteme
* Objektorientierte Datenbanksysteme
* Semistrukturierte Daten (XML) in relationalen Datenbanksystemen
* Daten-Replikation und Synchronisierung
* In-Memory-Datenbanksysteme
* Datenstrukturen in Datenbanksystemen
* NoSQL-Datenbanksysteme
* Verteilte und Cloud Datenbanksysteme

Übersicht des Modules

[OR-Mapping (Einführung) 3](#_Toc475792556)

[Ideen und Lösungsansätze 3](#_Toc475792557)

[Semantic Gap in Persistence 3](#_Toc475792558)

[Object-Relation Mapper 3](#_Toc475792559)

[Technologien 3](#_Toc475792560)

[Custom Mappers 3](#_Toc475792561)

[Im Überblick 3](#_Toc475792562)

[Java Persistence API (JPA) 4](#_Toc475792563)

[Layers 4](#_Toc475792564)

[Provider Auswahl 4](#_Toc475792565)

[Grundgedanke 4](#_Toc475792566)

[JPA Kurz-Tour 4](#_Toc475792567)

[Mapping Regeln 5](#_Toc475792568)

[Regeln für Entites 5](#_Toc475792569)

[Table & Column Mapping 6](#_Toc475792570)

[Einfache Mapping Regeln 6](#_Toc475792571)

[Access Types 6](#_Toc475792572)

[Datentypen 6](#_Toc475792573)

[Mapping in XML 6](#_Toc475792574)

[Entity Relations (Beziehungen) 6](#_Toc475792575)

[1:1 Beziehung 6](#_Toc475792576)

[Bidirektionale 1:1 Beziehung 6](#_Toc475792577)

[Inverse 1:1 Beziehung 6](#_Toc475792578)

[N:1 Beziehung 6](#_Toc475792579)

[Bidirektionale N:1 Beziehung 6](#_Toc475792580)

[1:N Beziehung 6](#_Toc475792581)

[N:M Beziehung 6](#_Toc475792582)

[Bidirektionale N:M Beziehung 6](#_Toc475792583)

[Laden der Relations 6](#_Toc475792584)

[Ladestrategien 6](#_Toc475792585)

[«OR Mapping» mit JPA (Forsetzung) 7](#_Toc475792586)

[Wie kommen Entites und Daten zusammen? 7](#_Toc475792587)

[JPA-Architektur (Begriffe) 7](#_Toc475792588)

[Kontrolle über Verhalten und Zustände der Entities 7](#_Toc475792589)

[Entity-Identität 7](#_Toc475792590)

[Änderungen an Relations 7](#_Toc475792591)

[Transaktionen 7](#_Toc475792592)

[Vererbung 7](#_Toc475792593)

[Anfragen in JPA mit JPQL 7](#_Toc475792594)

# OR-Mapping (Einführung)

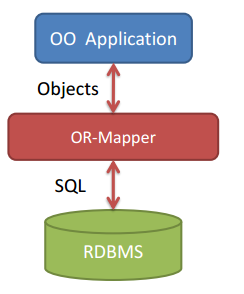
## Ideen und Lösungsansätze

In Software-Programmen möchte man Objekte manipulieren (CRUD), insbesondere speichern (=persistieren) und wieder holen (=abfragen).

Mit reinem JDBC muss ich jede Spalte auf Java Objekte ummappen. Ein OR Mapper soll nun das Objektmodell und das Datenbankmodell verbinden. Dabei möchte ich nichts mit dem Mapping vom Objektmodell zum Datenbankmodell zu tun haben.

## Semantic Gap in Persistence

OR Mapper sollen diesen Semantic Gap zwischen den beiden Modellen schliessen. Die Programme sind meist objekt-orientiert, während die Datenbanken meist relational sind.

Es soll eine «Nahtlose» Abbildung von und zu der darunterliegenden DB sein.

**Funktionen**

* Laden von relationalen Daten als Verbund von Objekten (Joins über mehrere Tabellen)
* Speichern von Änderungen an persistenten Objekten

## Object-Relation Mapper

### Technologien

**Unter Java** Java Persistence API (JPA), Java Data Objects (JDO), Hibernate

**Unter .NET** Entity Framework (EF), NHibternate

### Custom Mappers

In gewissen Fällen sinnvoll

* Bei besonders hoher Abbildungskomplexität (Mappings auf SPs)
* Bei mehreren Datensilos (SQL, NoSQL, Streams, Filesystems)
* Evtl. höhere Performance
* Evtl. besondere Skalierung
* Besondere Transaktionsverwaltung

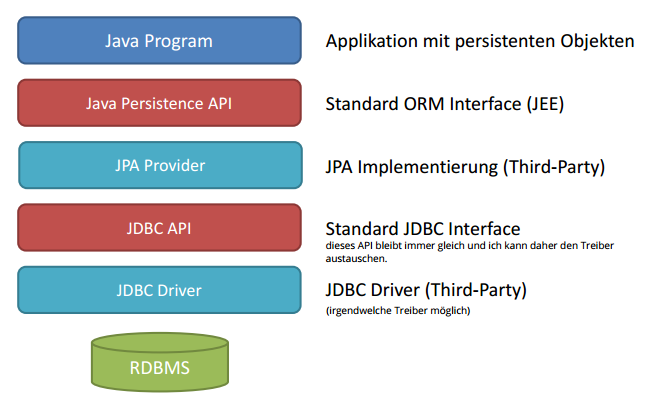
Custom Mappers haben natürlich einen hohen Arbeitsaufwand und die Komplexität ist auch vorhanden.

## Im Überblick

Das **Java Persistence API** (JPA) ist das Standard OR Mapping zu relationalen Datenbanken. Es ist Teil von Enterprise Java Beans, läuft aber auch ohne JEE und ohne JEE Application Server. Es ist nur eine Spezifikation. Diverse Implementierungen vorhanden. Beeinflusst wird es durch Hibernate und Java Data Object.  
**Java Data Object** (sozusagen tod), Standard vor JPA

# Java Persistence API (JPA)

## Layers



## Provider Auswahl

Es gibt verschiedene JPA Implementierungen. Dazu zählen EclipseLink, Hibernate, Apache OpenJPA, DataNucleues,….. In den Beispielen in dieser Vorlesung wird EclipseLink verwendet.

## Grundgedanke

**Entities** als persistente Objekte. Dabei sollen POJO («Plain Old Java Objekte») verwendet werden, also ganz normale Java Objekte. Mit einem ID-Feld für die Abbildung auf den Primarschlüssel in der Datenbank. Zudem ein Parameter losen Konstruktor.

Die **Abbildung** zur Datenbank wird mittels Annotations im Code (nicht ganz POJO) realisiert. Alternativ kann dies auch in separaten Mapping-Files deklariert werden.

Die **Anbindung** an die Datenbank findet im Persistence.XML Konfigurationsfile statt.

## JPA Kurz-Tour

|  |  |
| --- | --- |
| **Entity Klasse** | **Abbildung auf die DB** |
| **JPA Konfiguration** File «persistence.xml » im Folder « META-INF» |  |
| **Laden/Suchen von Entities** | **Explizites Einfügen von Entities** |
| **Update von Entities** | **Explizites Löschen von Entities**  Zum Löschen muss das Objekt zuerst in den Cache geladen werden und kann dann erst gelöscht werden. Mit commit wird dann die Löschoperation erst effektiv ausgeführt wird. |

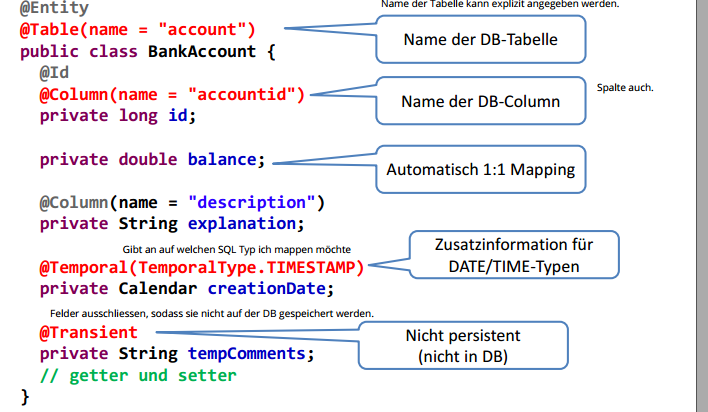
## Mapping Regeln

### Regeln für Entites

Eine Entity ist eine Klasse mit der Annotation @Entity. Diese Klasse kann erben, vererben, kann Interface implementieren oder auch abstrakt sein. Von dieser Seite her gibt es also keine Einschränkungen. Es gibt aber Einschränkungen bezogen auch den Inhalt der Klasse.

* Es muss ein Public oder Protected Konstruktor ohne Argumente vorhanden sein.
* Es ist eine Primary Key Angaben erforderlich (Annotation @id)
* Klasse darf nicht final sein, auch auch keine final Flieds oder Methoden haben
* Die Fileds sollen private oder protected sein und nur über getter/Setter zugreifbar.

### Table & Column Mapping

Mit @Table kann der Name der Tabelle explizit angegeben werden.

Mit einer Spalte ist dies über @Column möglich.

Mit @Temportal kann ich den entsprechenden SQL-Typ angeben.

Über @Transient kann ich Felder ausschliessen.

### Einfache Mapping Regeln

Standardmässig findet das Mapping mit den gleiche Namen statt. Für die Spalten und die Tabellen kann dies explizit angegeben werden. Es ist Case Insensitiv.

Per Default werden alle Attribute in das Mapping einbezogen. Ausnahmen sind die entsprechenden Annotationen.

Bei Calendar/Date Java-Typ braucht es Zusatzinformationen. Dafür lässt sich die Temporal Annotation verwenden. Eine Abbildung ist auch Date, Time oder Timestamp möglich. Aber aufpassen, die Unterstützung ist nicht immer gleich.

### Access Types

|  |  |
| --- | --- |
| **Field Access** DB-Attribute direkt in Fields abbilden | **Property Access** DB-Attribute über Getter/Setter abbilden |

Entweder macht man Field oder Property Access pro Klasse. Die Attribut-Annotation bestimmt den Access Type. Mindestens @Id Annotation ist pro Klasse nötig.

Beim Propery Access die Attribute Annotation beim Getter angeben. Alle Properties müssen Getter und Setter haben. Statt getXYZ() ist auch isXYZ() möglich (passend für den Boolean)

### Datentypen

Es werden folgende **Typen** beim Mapping **unterstützt**:  
Primitive Typen und deren Wrapper-Typen, String, Enums, Byte und Character Array, Date, Calendar und beliebige Serializierbare Klassen (welche nach BLOB abgebildet werden).

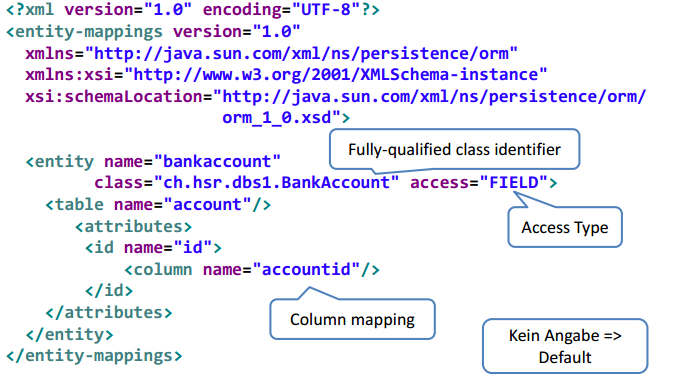
Zu den **unterstützten Relationen** zählen alle Referenzen auf Instanzen mit Entity Klassen. Darunter Collection<>, Set<>,List<> und Maps<> Entities.

Es ist auch eine explizite Detail-Angabe vom DB-Typ möglich. Nützlich für automatische Schema-Erzeugung von JPA.  


### Mapping in XML

Eine Alternative zu den Annotationen in den Java Klassen. Dazu findet die Abbildung in orm.xml im Folder META-INF statt. Die Prioritäten des Mappings sind wie folgt festgelegt.

1. XML-File orm.xml
2. Entity-Annotationen
3. Default-Mapping



## Entity Relations (Beziehungen)

Die Beziehungen können Unidirektional oder bidirektional sein. Es existieren verschiedene Kardinalitäten. In der DB als Tabellen mit FK-Schlüssel Beziehungen. Es ist eine Explizite Annotation oder XML-Mapping nötig.

### 1:1 Beziehung

|  |  |
| --- | --- |
|  |  |

### Bidirektionale 1:1 Beziehung

|  |  |
| --- | --- |
|  |  |

### Inverse 1:1 Beziehung

|  |  |
| --- | --- |
|  |  |

### N:1 Beziehung

|  |  |
| --- | --- |
|  |  |

### Bidirektionale N:1 Beziehung

|  |  |
| --- | --- |
|  |  |

### 1:N Beziehung

|  |  |
| --- | --- |
|  |  |

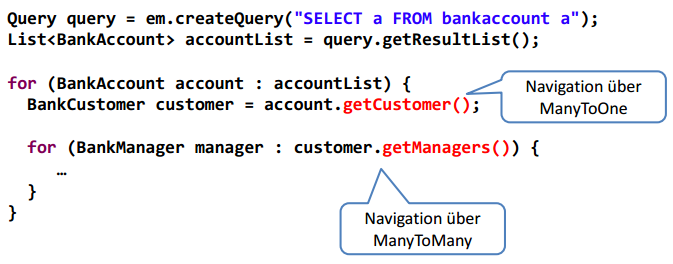
### N:M Beziehung

|  |  |
| --- | --- |
|  |  |

### Bidirektionale N:M Beziehung

|  |  |
| --- | --- |
|  |  |

### Laden der Relations

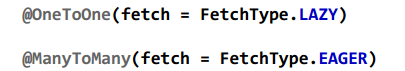
Die Normale Travesierung des Objektgraphen. Objekte laden, welche unter Abhängigkeit von einander stehen.  


### Ladestrategien

**Eager Loading**  
Target Entity wird direkt mit der Beziehung geladen. Default bei @OneToOne und @ManyToOne.

**Lazy Loading**  
Laden der Entity beim ersten Zugriff auf die Beziehung. Default bei @OneToMany und @ManyToMany.

Die Ladestrategien können explizit mit Annotationen gesteuert werden. Es ist ein bisschen doof, da man es bei jeder Beziehung wieder angeben muss. Es gibt andere Entities Frameworks, welche das besser machen.



# «OR Mapping» mit JPA (Forsetzung)

## Wie kommen Entites und Daten zusammen?

**Top down**  Erstelle Business-Modell und erzeuge DB-Schema

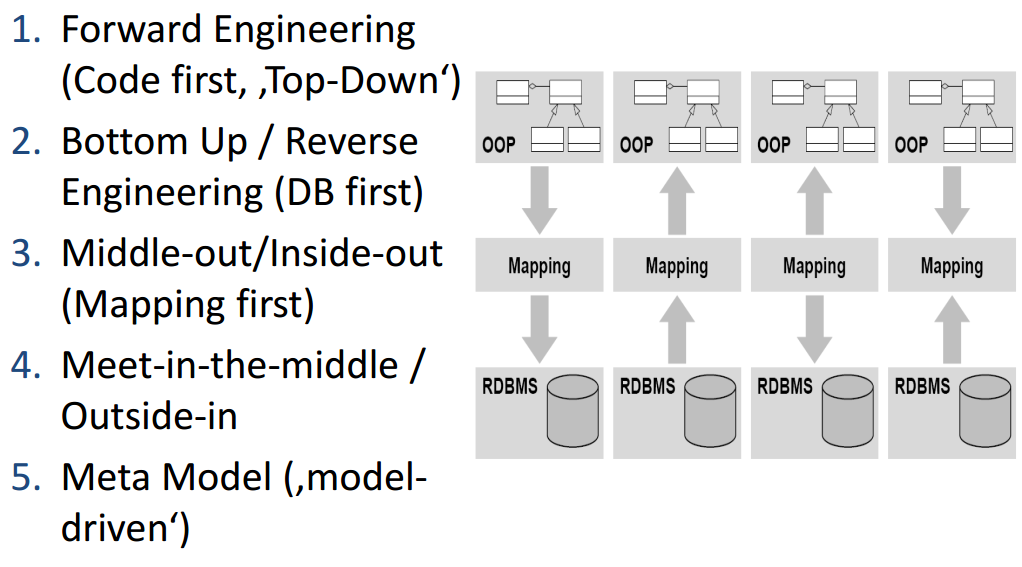
**Bottom up** DB-Schema existiert, erzeuge daraus Business-Modell…

**Middle out** Erstelle Metadaten und generiere Java und DB-Schema

**Meet in the middle** Business-Modell und DB-Schema existieren bereits. Erstelle Metadaten.

**Model Driven** ist für uns nicht wichtig, auch Meta Model genannt.

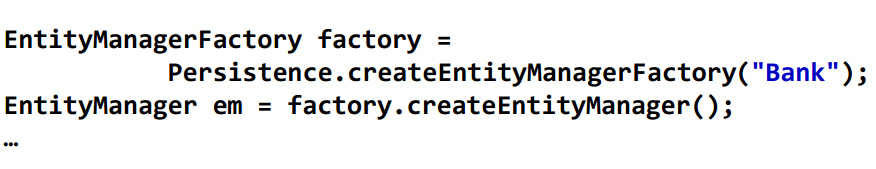
### OR-Mapping Varianten



## JPA-Architektur (Begriffe)

### JPA Persistence Unit

Die Persistence Unit ist die Menge der Entity Klassen und deren Mapping. Bestimmt durch den JPA Provider und die DB-Anbindung. Die EntitiyManagerFactory verwaltet eine PersistenceUnit.



### JPA Persistence Context

Der Persistence Context verwaltet eine Menge von Entity Instanzen zur Laufzeit. Ist eine Entity Instanz managed, dann ist Sie im Persistence Context. Ist eine Entity new oder detached gehört Sie zu keinem Kontext. Der Persistence Context definiert zudem die transaktionelle Session.

Der Enity Manger verwaltet den Persistence Kontext und bietet Lifecycle-Operationen für Entity Instanzen an.

### Entity State

**New** – Entity Instanzen haben keine Persistenz Identity und sind keinem Persistenz Kontext zugeordnet.

**Managed** – Entity Instanzen haben eine Persistenz Identity und sind einem Persistenz Kontext zugeordnet.

**Detached** - Entity Instanzen eine Persistenz Identity und sind aktuell keinem Persistenz Kontext zugeordnet.

**Removed** - Entity Instanzen eine Persistenz Identity und sind einem Persistenz Kontext zugeordnet und sind geschedulted für die Löschung vom Datastore.

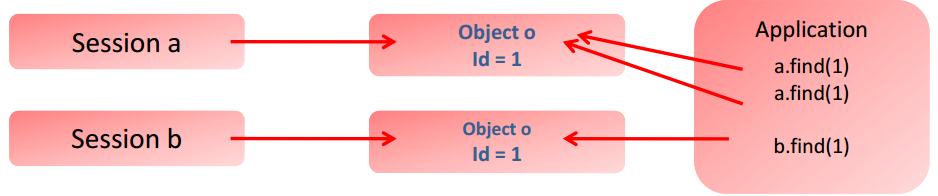
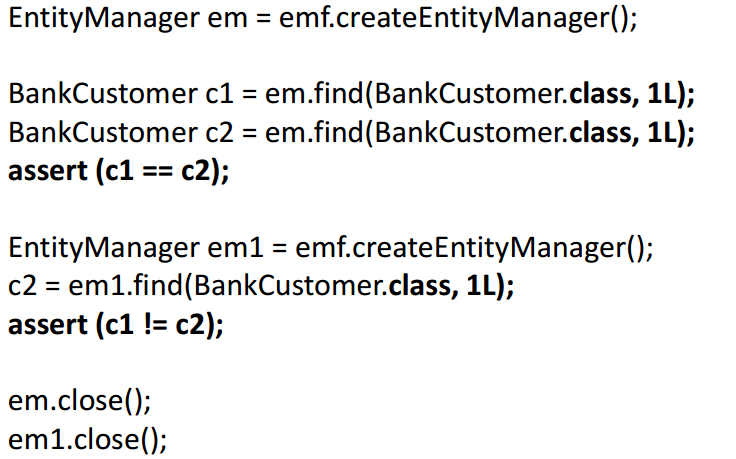
### Operationen auf den Entities

### Persistencte Context (=Sesssion)

Sie werden via dem EntityMangerFactory erzeugt. Der PC kapselt die DB-Session und die Transaktionen und verwaltet einen Cache. Er unterstützt CRUD-Operationen auf Entities. Wenn ein Objekt geladen wird und sich bereits im Cache befindet, dann wird eine Referenz auf den Cache geliefert. Zudem findet Change-Tracking statt. Änderungen auf den Objekten im Cache werden registriert. Geänderte Werte werden beim Commit mit der DB synchronisiert. Beim Close() der EntityManager-Instanz wird der Cache invalidiert.

### Object Identity

Die Session (= Instanz eines EntityMangers = Persistence Context) übersetzt das DB-Konzept des Primärschlüssels in das Konzept von Objekt-Instanzen. Die Session verwaltet daher die Objekt-Identität. Objekte werden über ihre ID im Cache verwaltet. Beim ersten Zugriff auf ein Objekt, wird dessen Zustand von der DB geladen. Bei einem weiteren Zugriff auf das Objekt, wird die Kopie im Cache zurückgegeben. Persistente Objekte werden über ihre Identität identifiziert.

## Entity-Identität

Findet über die Annotation @ID statt. Für jede Entity gibt es ein ID-Property. Es bildet den Primary Key in die DB ab. Die Generierung der Identity findet mit der Annotation @GeneratedValue bei ID-Property statt. Generiert wird sind dann durch die JPA-Instanz beim INSERT neuer ROWS in die DB.

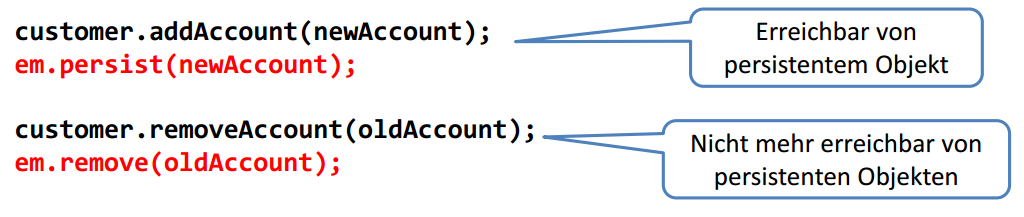
Für die Generierung der ID gibt es verschiedene Strategien. AUTO, IDENTITY, SEQUENCE und TABLE.

|  |  |
| --- | --- |
| Generierungstyp Identity | Generierungstyp Sequence |
| Generierungstyp Table |  |

### Transitive Persistenz

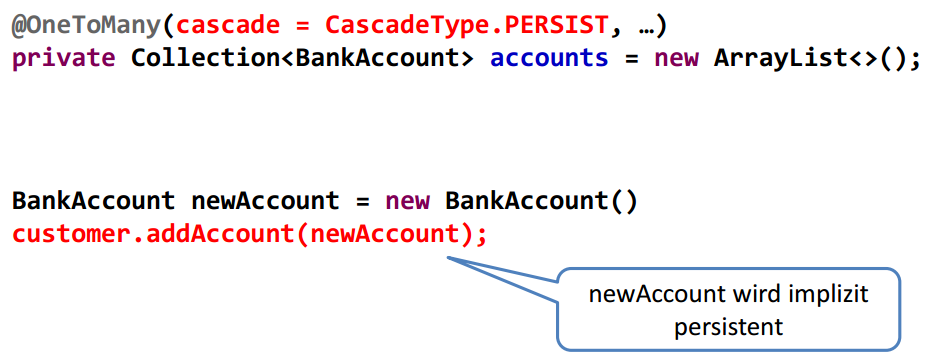
Alle von persistentem Objekt erreichbaren Objekte sollen wiederum persistent sein. Alle anderen Objekte müssen nicht persistent sein. In JPA gibt es keine automatische transitive Persistenz. Entweder explizit in der DB allozieren /deallozieren mit EntityManger.persist(), EntityManger.remove() oder implizit via kaskadierte Relations (Kaskade-Angabe bei Relation Annotation).

### Explizite (De-)Allokation



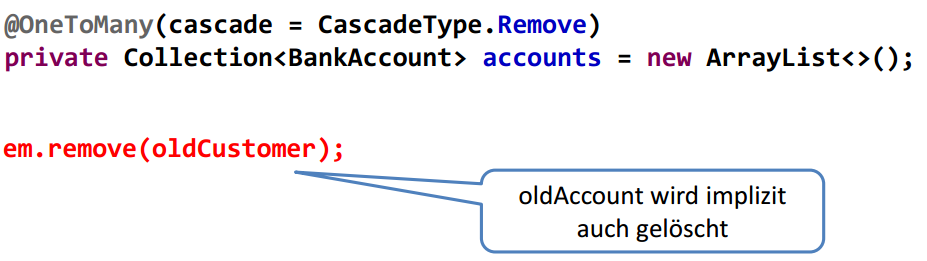
### Kaskadierte Persistenz

Mit der Angabe CascadeType.Persist werden die referenzierten Objekte automatisch persistent.



### Kaskadiertes Löschen

Mit CascadeType.Remove wird beim Löschen des Holder-Objekts auch das referenzierte Objekt entfernt.

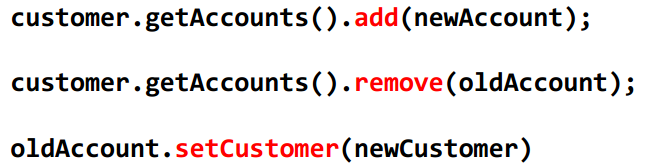


## Änderungen an Relations

### Relations mit Abhängigkeiten

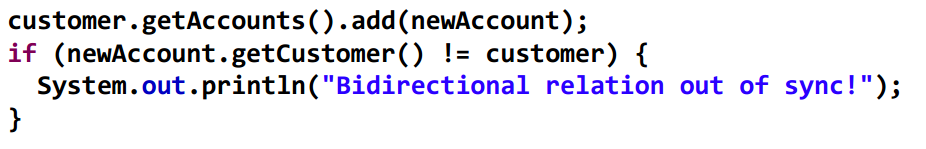
|  |  |
| --- | --- |
| **Aggregation** | **Komposition** |

### Änderungen an Relations

Die Beziehungen lassen sich normal zugreifen. Bei Zugriff findet Eager vs. Lazy Load Verhalten statt. Die Änderungen werden dann bei Commit in die Datenbank gespeichert.

### Inkonsistente Bidirektionalität

Bei den bidirektionalen Relationen muss man aufpassen. Die Änderungen werden durch das JPA nicht synchronisiert und das trotz der Angabe von @OneToMany(mappedBy=»customer»). Es ist nur nach dem Laden konsistent.

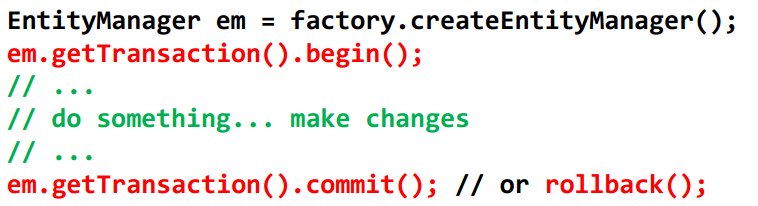


### Bidirectional Sync

|  |  |
| --- | --- |
|  |  |

## JPA Transaktionen

Es sind auch Änderungen in expliziten Transaktionen (gegenseitiger Ausschluss) möglich.



### Transaktionsende

Nach dem **Commit** werden die Änderungen in die DB geschrieben. Die Entity-Zustände werden nicht von der DB refresehed.

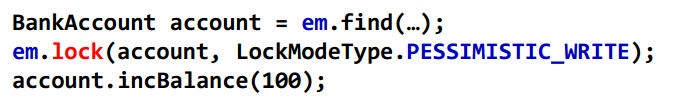
Nach dem **Rollback** werden die Änderungen nicht in die DB gespeichert. Die Entites sind detachated (Sie behalten den Zustand).

### Transaction Isolation

Das JPA basiert auf dem READ COMMITTED Isolation Level. Je nach JPA Provider sind auch andere Levels konfigurierbar. Zum Beispiel bei HIBERNATE im persistence.xml.

### Explizites Locking

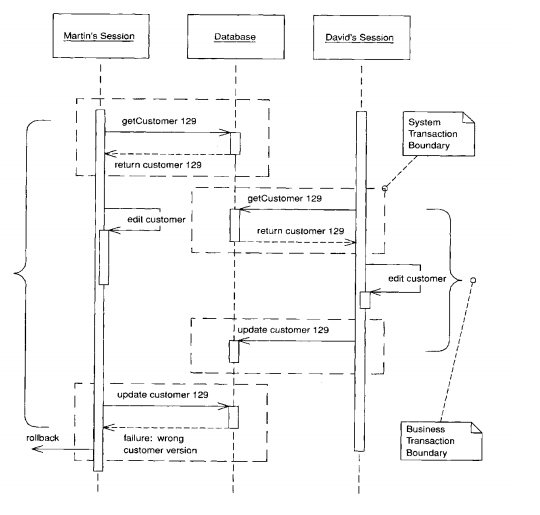
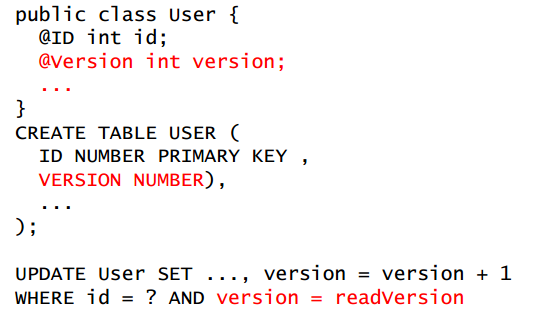
Die Methode lock() beim EntityManager soll Dirty Reads und List Updates ausschliessen.

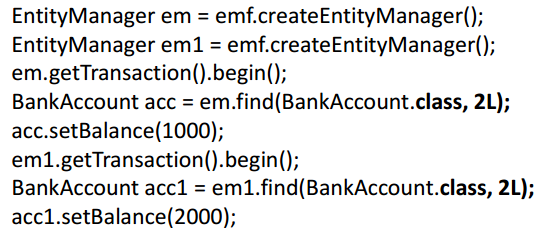
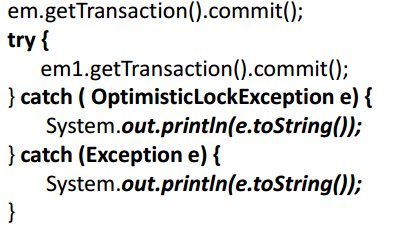


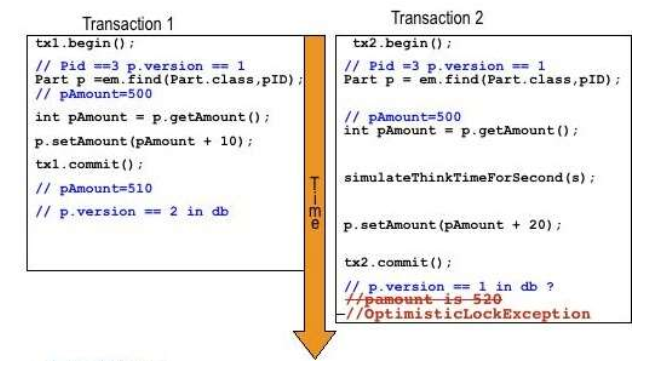
### Optimistic concurrency

« „Prevents conflicts between concurrent business transactions by detecting a conflict and rolling back the transaction“.

**Optimistic**: Transaktionen laufen unbehindert an. Beim Abschliessen wird in einer Validierungsphase überprüft, ob Konflikte aufgetreten sind und gegebenenfalls die Transaktion zurückgesetzt.

Demo  
 



### Zusätzliches Locking mit JPA Entity Locking

Mit JPA 2.0 ist es möglich die Entities zusätzlich zu locken. Dies erlaubt es auch an die Art und Weise des Lockings anzugeben. Bei den Art und Weisen wird zwischen OPTIMISTIC\_FORCE\_INCREMENT und OPTIMISTIC unterschieden.

#### OPTIMISTIC

Der typische Anwendungsfall für den OPTIMISTIC-Sperrmodus ist, wo eine Entität eine intrinsische Abhängigkeit von einer oder mehreren Entitäten hat, um Konsistenz zu gewährleisten, zum Beispiel, wenn es eine Beziehung zwischen zwei Entitäten gibt.

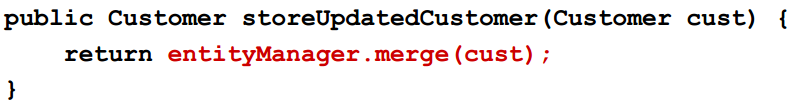
#### OPTIMISTIC\_FORCE\_INCREMENT

OPTIMISTIC\_FORCE\_INCREMENT-Sperrmodus verursacht einen optimistischen Sperrfehler, wenn eine andere Transaktion versucht, die gesperrte Entität zu ändern. Die gemeinsame Verwendung für diese Sperre ist es, die Konsistenz zwischen Entitäten in einer Beziehung zu gewährleisten.

## Kontrolle über Verhalten und Zustände der Entities

### Merge

Verändertes Objekt wird in den Persistenzkontext eingefügt. Zuerst wird überprüft, ob eine Instanz mit derselben Id im Cache ist; falls nicht wird eine Instanz erzeugt und deren Zustand von der DB geladen. Diese Instanz wird mit den Änderungen gemergt. Merge liefert das persistente Objekt zurück.

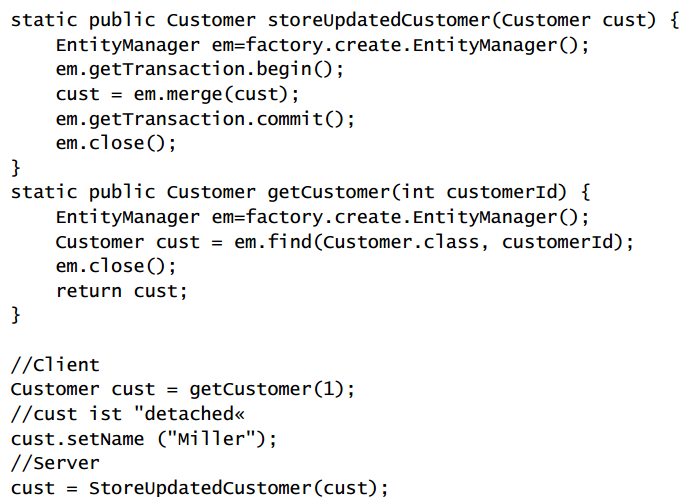


### Refresh

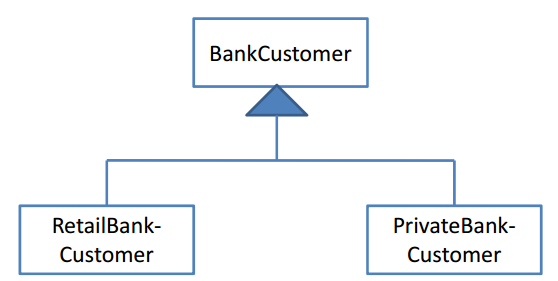
Der Zustand wird neu von der DB geladen.



### Beispiel



## Vererbung

Es gibt drei unterschiedliche Arten eine Vererbung zu implementieren.

Einerseits über Single Table, über Joined Tables oder mit Table Per Class.

|  |  |
| --- | --- |
| Single Table Mapping | |
|  |  |
| Joined Table Mapping | |
|  |  |
| Table Per Class Mapping | |
| **RetailBankCustomer** | **PrivateBankCustomer** |

### Diskussion

Beliebig viele Tabellen können zusammen auf eine Klasse abgebildet werden mit @SecondaryTable. Klassen können zusammen auf eine Tabelle abgebildet werden mit @Embedded. Klassen mit Mapping Informationen der Superklassen können in einer Tabelle abgebildet werden (mit @MappedSuperclass).

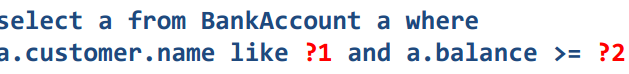
## Anfragen in JPA mit JPQL

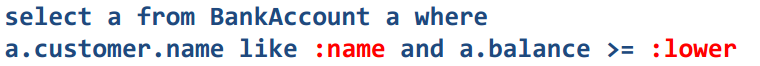
JPQL steht für Java Persistence Querying Language. Es ist eine Anfragesprache in der Analogie zu SQL, es operiert aber auf dem Entity Modell und nicht auf dem DB-Modell. Daher Entities & Fields statt Tables & Columns. Es wird dann vom JPA Provider in SQL übersetzt. Es kennt Path Expressions (=Implizit Joins).



### Beispiele

### Query Parameters

**Positional Parameters  
**

**Named Parameters  
**

### Parameter setzen

Es sind auch Entities als Parameter möglich. Somit wird die SQL-Injection verhindert.

Position   
Named 

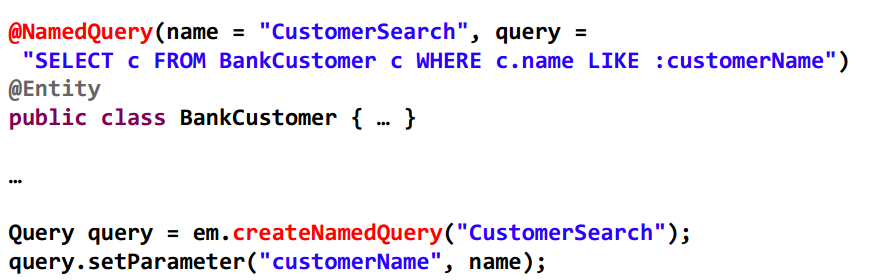
### Dynamic Queries

Eine Query welche zur Laufzeit gebaut und geprüft wird.

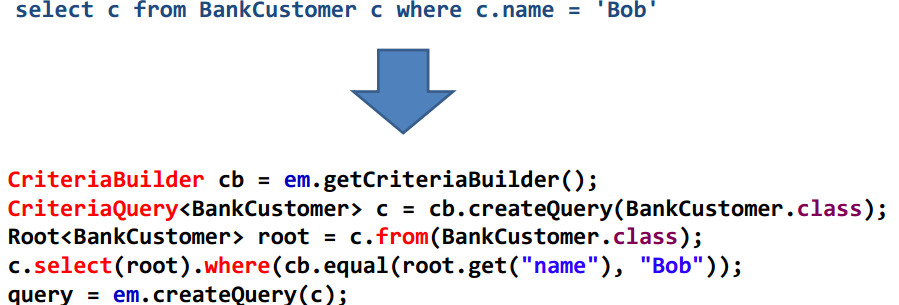


### Named Queries

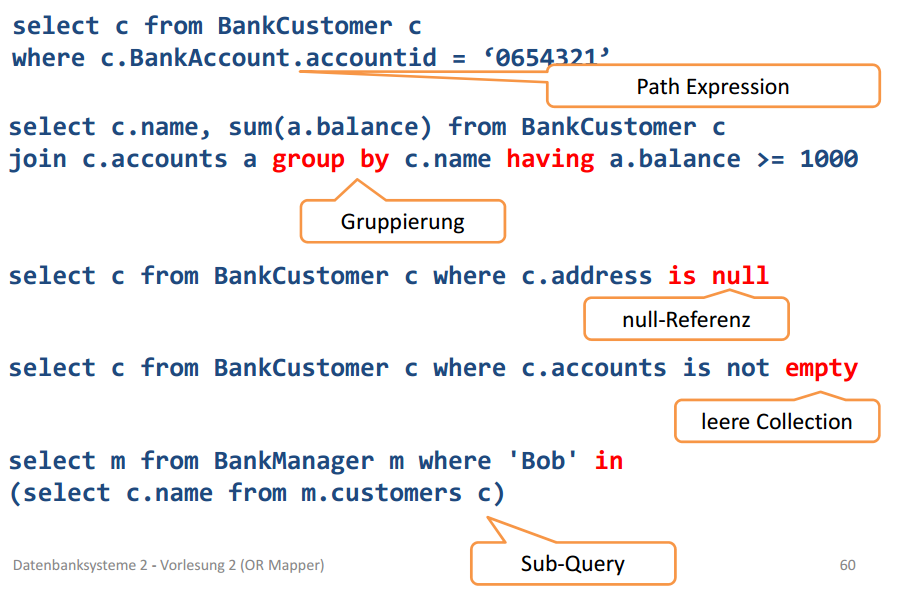
Queries welche statisch bekannt sind. Sie können vom JPA Provider vorgeparst und optimiert werden. Die Annotation @NamedQuery muss bei der Entity-Klasse annotiert werden.



### Criteria API

Queries mit Method Chaining und Generics. Es ist von Vorteil, da es statisch ist (zur Compilezeit geprüft). Es ist aber eine unübersichtliche Weise und wird daher nicht immer verwendet.  


### Weitere Beispiele



## JPA Diskussion und Implementationen

**Zur JPA-Spezifikation**  
Die Query-Sprache und die Annotationen «bindet» die Domain an die Persistenzsicht. RDBMS könnten aber noch abstrahiertet sein. Zudem besteht Bedarf an besserer Konfiguration für Property Access oder auch DB-spezifische Typen.

Es sind mehrere Implemtationen vorhanden, was sehr von Vorteil ist. Die Tools sind aber erst am Kommen und damit ihrem Standard voraus. Das heisst auch, dass sie noch nicht ausgereift sind. (zum Beispiel für benutzerdefinierte Datentypen).

### Fazit

Erfahrungen mit DataNucleus  
JPD bietet besser OO-Kompatibilität als das JPA. Es wird aber kein lazy-loading von grossen Recordsets unterstützt. Man kann sagen eine Two-Man Show.

**Erfahrungen mit Hibernate**  
Es ist konform zu JPA und EJB, hat jedoch auch eine eigene Query-Sprache. Ist ok falls RDBMS gesetzt ist und der Standard nicht im Vordergrund ist. Es ist verbreitet, hat aber mehr Dependencies als EclipseLink. Zudem bietet es eine breite Palette von unterstützten DBMS.

# Stored Procedures (SP)

## Was sind Stored Procedures?

“A SP is a subroutine available to applications that access a RDBMS”. Ein SP kann also mit User Definied Funtion, Subroutine oder Methode bezeichnet werden. Grundsätzlich basiert es auf den Procedure Calls. Die Programme sind so nahe oder gar bei den Daten gespeichert. Zudem bilden sie Grundlage für Triggers.

### Warum Stored Procedures?

* Domain-Logik (Funktionen und Datenkapselung)
* Performanz – Funktionen nahe bei den Daten
* Daten-Konsistenz (Triggers Security)
  + Verfeinerter Zugriffschutz
  + Log Auditing von Zugriffen
* Code-Wiederverwendbarkeit (Anstelle von Subqueries)
* Separation of Duties (Trennung der Aufgaben)

### Wann keine Stored Procedures?

Im Fall wenn Views oder ein Object-relationaler Mapper (ORM) genügen. Zudem macht es den Software-Entwicklungsprozess um ein Element komplexer. Die Wartbarkeit sowie Portierbarkeit darf man nicht ausser Acht lassen.

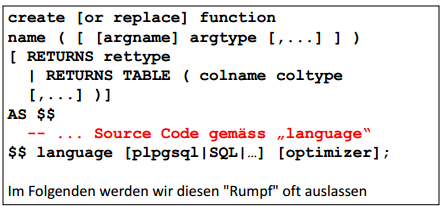
### Einführung

Sie bestehen aus Declarations, Assignments, Loops und sind in der DB gespeichert. Sie können aus anderen SP, Triggern, PL/SQL-Blöcken oder von der Client-Applikationen aufgerufen werden. Es sind Datenbankobjekte (Wie Tabellen). SP laufen innerhalb einer äusseren Transaktion). Zudem haben Sie ein eigenes Recht für die Ausführung.

## PL/SQL

SQL ist eine deklarative Programmiersprache, während PL/SQL eine prozedurale Sprache mit «eingebettetem» SQL ist. Jede Datenbank hat ihre eigene Abwandlung.

### Syntax



### Compiler

Der Code wird beim Aufruf geparsed und als Pseudocode in der DB gespeichert. Erst bei der ersten Ausführung wird die volle Syntax gecheckt. SQL Statements werden vorkompiliert und bei wiederholtem Aufruf wiederverwendet.

|  |  |
| --- | --- |
| **Hello World** | **Beispiel Increment** |

# PL/pqSQL - die Sprache

## Basis

**Deklaration und Block**DECLARE …. sowie BEGIN … END

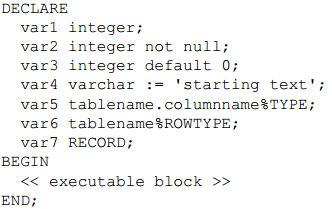
**Assignment**  
variable := expression;

**Rückgabe**  
return value;

**SQL einbetten**  
Direkt mit INSERT, UPDATE, DELETE oder SELECT… INTO…

**SQL ausführen (dynamisches SQL)**  
EXECUTE query\_string;

### Deklarationen



### Block und Kommentare

|  |  |
| --- | --- |
| **Blöcke** | **Kommentare**   * -- Single Line comment * /\* \*/ Multi Line Comments * Hinweis: Jedes Objekt kann auch eine Description haben |

### Conditoinals und Loops

|  |  |
| --- | --- |
| **Conditionals** | **Loops** |

### Exceptions

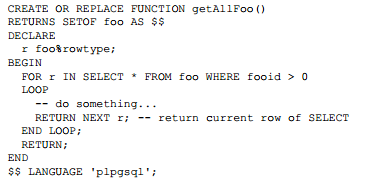
|  |  |
| --- | --- |
| **“Werfen”**   * ***Format***: RAISE level “format” …. * ***Level***: DEBGU, LOG, INFO; NOTICE, WARNING * ***Bsp***. RAISE NOTICE “notice %”, id; | **“Fangen”** |

## Beispiele

|  |  |
| --- | --- |
| SELECT…INTO | INSERT INTO |
| UPDATE | Function that returns void = SP |
| Function that returns a scalar | Function that returns a table |

## SET Returning Functions

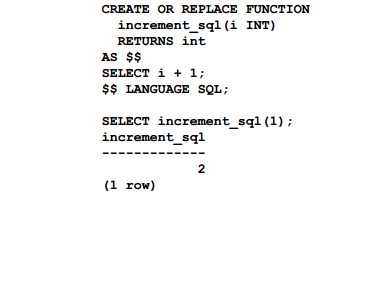
Zu den Datentypen stehen SETOF oder RECORD zur Verfügung. Sowie das Schlüsselwort RETURN NEXT.



# Stored Procedures in SQL und Python (PostgeSQL)

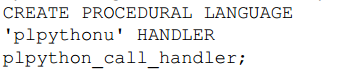
## SP mit SQL (PostgreSQL)

SQL – inline, wie Makros



## PL/Python (PostgreSQL)

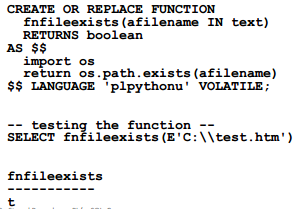
Dabei ist Python in den DB-Server eingebettet. Es nutzt libpq, das C API zu PostgreSQL. Die Python-Binaries müssen installiert werden. Python muss man in PostgreSQL zuerst installieren.



PL/Python ist eine «untrusted» Sprache, darum das «u» in plpythonu». Trusted ist ein «Security Feature».

### Beispiel File Exists

Prüft ob ein File existiert (dazu wird unteranderem “untrusted» benötigt).



## Gemeinsamkeiten Stored Procedures

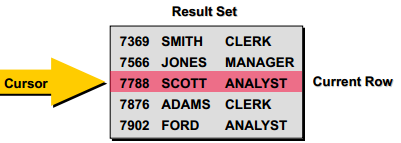
Folgendes ist gemeinsam mit den Store Procedures von PostgreSQL:

* Parameter
  + Namen; IN-Parameter (call by value); OUT(by reference)
* Funtion Overloading
  + HelloWorld(integer), HelloWorld(text)
* Variadic Functions
  + Varargs, «…»
* Set Returing Functions
  + RETURN SET OF….
* Function Polymorphism
  + Anyelement, anyarray

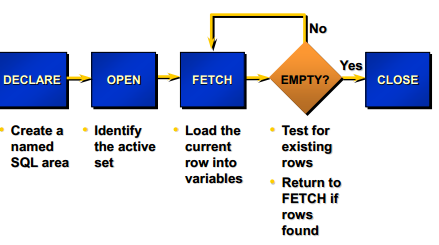
# Cursor

*«handle or name for a “private SQL area” - an area in memory inside DB server in which a parsed statement, procedures or functions are kept”*

Ein Select-Statement liefert in der Regel eine Menge von Tuples (=Result Set). Der Cursor erlaubt den sequentiellen Zugriff auf die einzelnen Tuples des Result Set.



## Verarbeitung

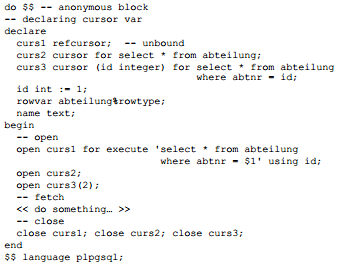


|  |  |
| --- | --- |
|  |  |

## Prinzip

|  |  |  |
| --- | --- | --- |
|  |  |  |

## Beispiel



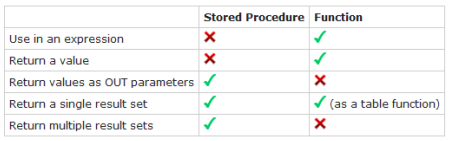
## Cursor mit Parametern

|  |  |
| --- | --- |
|  |  |

## Cursor for Update

|  |  |
| --- | --- |
|  | **Beispiel** |

## Vergleich Prozedur / Funktion



# Vorlesung Woche 4