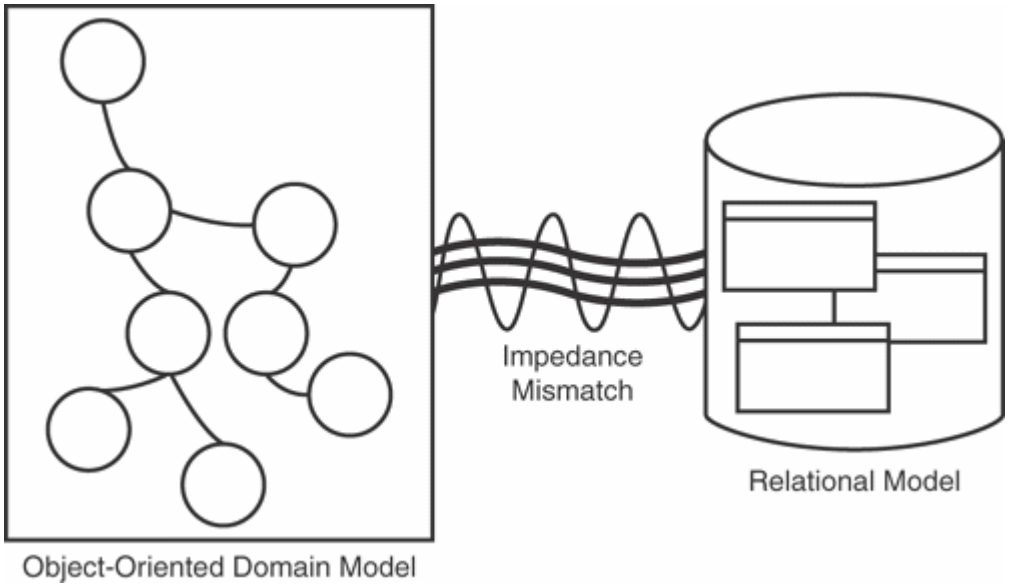
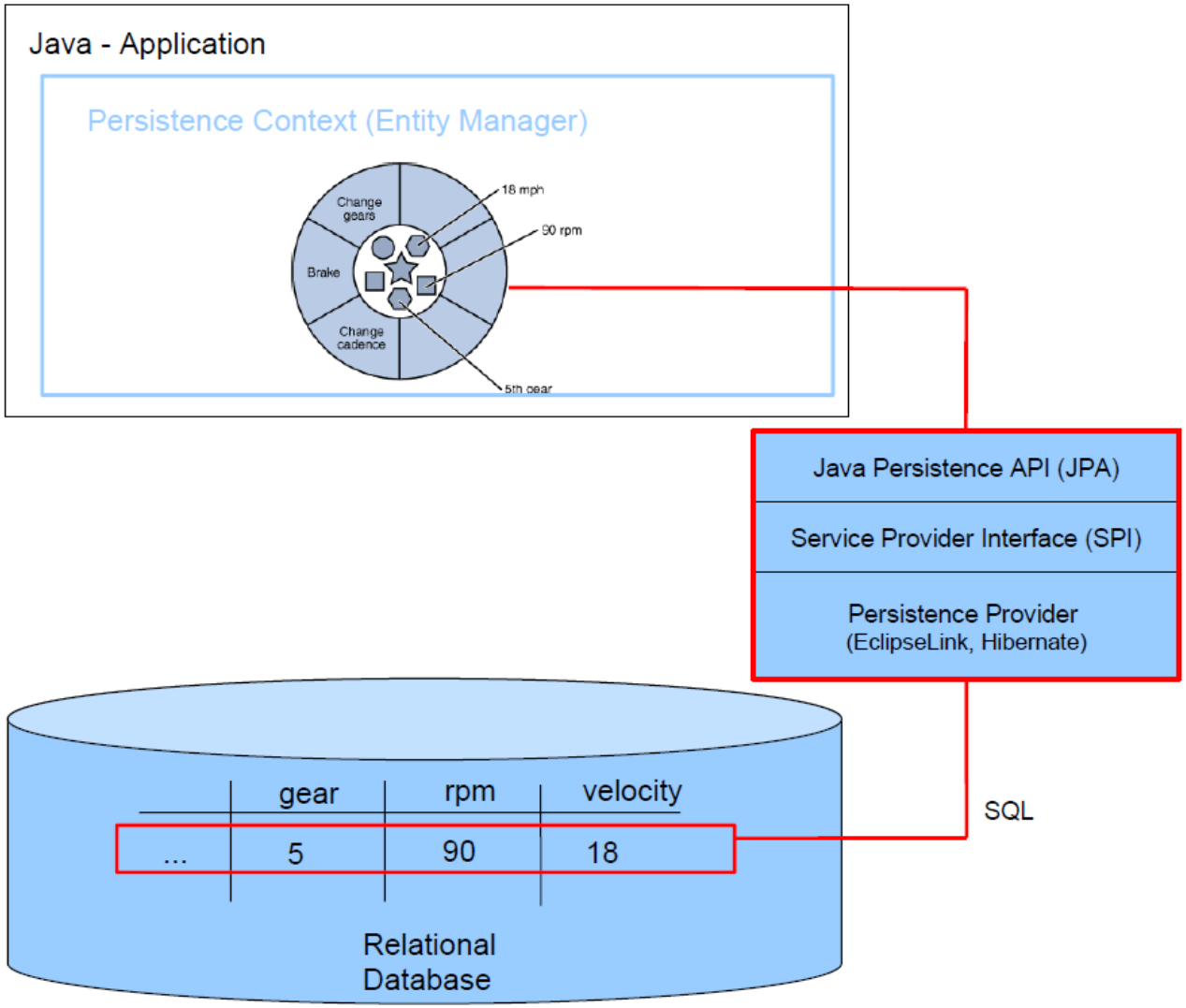
Jakarta Persistence API 3.x

crud + relationships + jp-ql

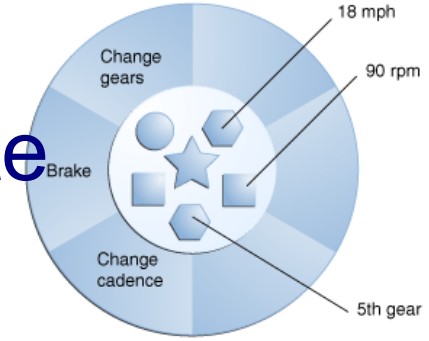


Grundprinzip



Problematik

- Man muss bei der Persistierung immer das Klassenmodell und dessen Umsetzung im Datenmodell (in der DB) berücksichtigen.
- Klassenmodell → objektorientierte Programmiersprache
- Datenmodell → Repräsentation in der Datenbank mit prozeduraler Bearbeitung



	gear	rpm	velocity
...	5	90	18

Relational Table

Persistieren in Java SE

Wie kann man eine Klasse persistieren?

- Annotation **@Entity** bei Klasse
- Annotation **@Id** bei Primärschlüssel
- **Standardkonstruktor** (Default-Konstruktor)
- **Serializable** implementieren

```
@Entity
public class Kfz implements Serializable {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private ~~long~~ id;Long
    private String brand;
    private String type;
```

Mögliche Annotationen

- Annotation **@Table**, zur Benennung der DB-Tabelle
- Annotation **@SequenceGenerator** zum Erstellen einer Sequenz in der DB
- Annotation **@GeneratedValue** für automatische Schlüsselvergabe

Entity mit Autowert und Sequenz

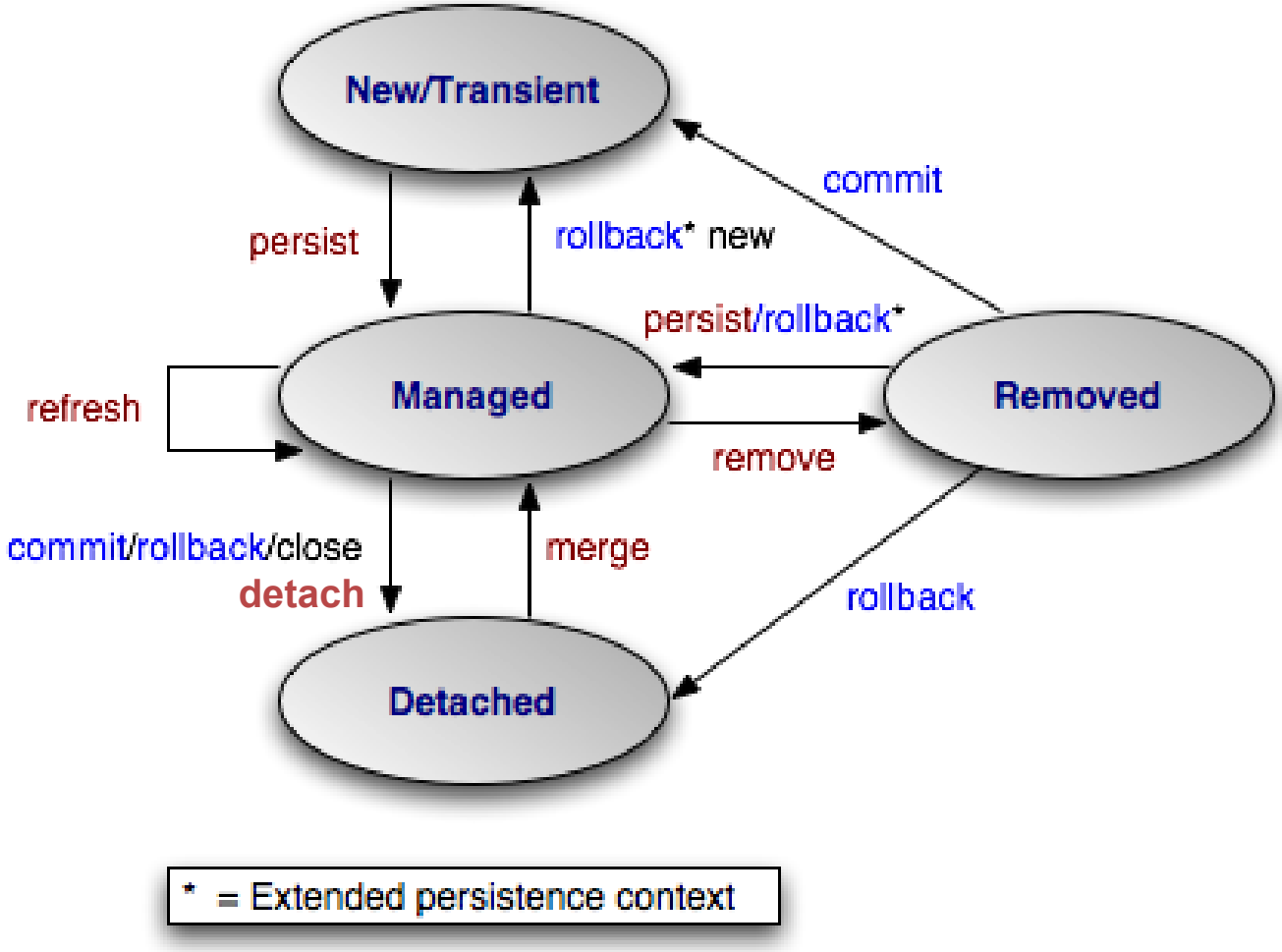
Autowert

```
@Entity
public class Kfz implements Serializable {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String brand;
    private String type;
```

Sequence

```
@Entity
@Table(name="jpademo_kfz")
@SequenceGenerator(name="KfzSeq", sequenceName="JPADEMO_KFZ_SEQ")
public class Kfz implements Serializable {
    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="KfzSeq")
    private Long id;
    private String brand;
    private String type;
```

Zustände eines Objekts



Unterschiede JPA in JavaSE und JakartaEE

- In JakartaEE (bei Verwendung eines Application Servers wie z.B. Glassfish) muss man die zu persistierenden Klassen nicht in die persistence.xml eintragen. Die Annotation mit @Entity genügt.
- Man muss die Transaktionen nicht explizit öffnen und beenden.
- Die Hilfsklasse JpaUtil wird nicht mehr benötigt, da der EntityManager bei JakartaEE injiziert wird (Hollywood-Prinzip)
- Es wird empfohlen, auch bei JakartaEE in der Persistenz-Unit das SQL-Logging zu aktivieren

```
<property name="eclipselink.logging.level" value="FINE"/>
```

Konfiguration von JPA

https://gist.github.com/mortezaadi/8619433

pom.xml mit JavaSE

```xml
<dependencies>
    <dependency>
        <groupId>org.eclipse.persistence</groupId>
        <artifactId>javax.persistence</artifactId>
        <version>2.1.0</version>
    </dependency>
    <dependency>
        <groupId>org.eclipse.persistence</groupId>
        <artifactId>org.eclipse.persistence.jpa</artifactId>
        <version>2.6.0</version>
    </dependency>
    <!-- wird für embedded mode gebraucht (inkl. memory) -->
    <!--
    <dependency>
        <groupId>org.apache.derby</groupId>
        <artifactId>derby</artifactId>
        <version>10.11.1.1</version>
    </dependency>
    -->
    <!-- wird für network mode gebraucht -->
    <dependency>
        <groupId>org.apache.derby</groupId>
        <artifactId>derbyclient</artifactId>
        <version>10.11.1.1</version>
    </dependency>
</dependencies>
```

JPA - API

JPA Implementierung, könnte auch Hibernate sein

Treiber für DerbyDb, hiermit kann auch eine DB-Instanz betrieben werden zB embedded mode

Client-Treiber für DerbyDb

persistence.xml mit JavaSE

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence" version="2.1">
    <persistence-unit name="myPU" transaction-type="RESOURCE_LOCAL">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <class>at.htl.vehicle.entity.Vehicle</class>
        <exclude-unlisted-classes>true</exclude-unlisted-classes>
        <properties>
            <property name="eclipselink.logging.level" value="FINE"/>
            <property name="eclipselink.target-database" value="DERBY"/>
            <!--...-->
            <property name="javax.persistence.jdbc.driver"
                      value="org.apache.derby.jdbc.ClientDriver"/>
            <property name="javax.persistence.jdbc.url"
                      value="jdbc:derby://localhost:1527/myDb;create=true"/>
            <property name="javax.persistence.jdbc.user" value="app"/>
            <property name="javax.persistence.jdbc.password" value="app"/>
            <property name="javax.persistence.schema-generation.database.action"
                      value="drop-and-create"/>
        </properties>
    </persistence-unit>
</persistence>
```

3 Modi der DB

- Network Mode
  - Database and application run in 2 processes
- Embedded Mode
  - Database and application run in 1 process (in the same jvm)
- Memory Mode
  - Similar to embedded mode w/o files

pom.xml mit JakartaEE

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
    <modelVersion>4.0.0</modelVersion>

    <groupId>at.htl.vehicle-jpa</groupId>
    <artifactId>vehicle-jpa</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
    </properties>

    <dependency>
        <groupId>jakarta.platform</groupId>
        <artifactId>jakarta.jakartaee-api</artifactId>
        <version>8.0.0</version>
        <scope>provided</scope>
    </dependency>

    <build>
        <finalName>vehicle</finalName>
    </build>

</project>
```

persistence.xml mit JakartaEE

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence" version="2.1">
    <persistence-unit name="dbPU">
        <jta-data-source>java:jboss/datasources/DbDS</jta-data-source>
        <properties>
            <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
            <property name="hibernate.transaction.flush_before_completion" value="true"/>
            <property name="hibernate.dialect" value="org.hibernate.dialect.DerbyDialect"/>
        </properties>
    </persistence-unit>
</persistence>
```

Konfiguration in Quarkus

- Hinzufügen der Abhängigkeiten zur pom.xml
  - *Hibernate ORM* oder *Hibernate ORM with Panache*
  - Datenbanktreiber
    - JDBC oder
    - reaktiv
- *persistence.xml* nicht notwendig, aber möglich
- Konfiguration über properties in der *application.properties*

https://code.quarkus.io/

CRUD-Operationen

Create – Read – Update - Delete

Persistenzunit

- Die Persistenzunit ist eine zentrale xml-Datei, in der sämtliche Konfigurationen des DB-Zugriffs verwaltet werden.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence" version="2.1">
    <persistence-unit name="dbPU">
        <jta-data-source>java:jboss/datasources/DbDS</jta-data-source>
        <properties>
            <property name="hibernate.hbm2ddl.auto" value="create"/>
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
            <property name="hibernate.transaction.flush_before_completion" value="true"/>
            <property name="hibernate.dialect" value="org.hibernate.dialect.DerbyDialect"/>
        </properties>
    </persistence-unit>
</persistence>
```

https://gist.github.com/ThomasStuetz/9ff41171a3635024b5fc

Einstellungen für Generierung der DB

<property name="javax.persistence.schema-generation.database.action" value="drop-and-create"/>

none	Es werden keine Tabellen generiert. Diese Einstellung ist bei Produktivsystemen
create	Es wird versucht die Tabellen zu erstellen. Existieren diese bereits, wird nichts geändert
drop-and create	Alle Objekte der DB werden gelöscht und vom Provider neu erstellt
drop	Alle Objekte in der DB werden beim Deployment gelöscht

Hibernate-spezifische Einstellungen

```
<property name="hibernate.hbm2ddl.auto" value="create"/>
```

- **validate**: validate that the schema matches, make no changes to the schema of the database, you probably want this for production.
- **update**: update the schema to reflect the entities being persisted
- **create**: creates the schema necessary for your entities, destroying any previous data.
- **create-drop**: create the schema as in **create** above, but also drop the schema at the end of the session. This is great in early development or for testing.

http://antoniogoncalves.org/2014/12/11/generating-database-schemas-with-jpa-2-1/
https://docs.oracle.com/javaee/7/tutorial/persistence-intro005.htm

Transaktionen

- Ein großer Unterschied zwischen JPA in JavaSE und JPA in JavaEE-Umgebungen ist, dass durch die Einbettung der DB-Operationen in EJBs, das Transaktionshandling vom EJB-Container übernommen wird
- Man muss die Transaktionen nicht mehr beginnen und beenden:

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

- sondern „nur“ noch folgendes coden:

  ```
  em.persist(vehicle);
  ```

- Wird die Methode ausgeführt, erfolgt ein Commit(), im Falle eines Abbruchs ein Rollback()

EntityManager

- Eine zentrale Klasse in JPA ist der EntityManager
- Ein entityManager ist mit dem Persistenzkontext assoziert und entspricht einer Menge von Entitäts-Instanzen. Jede Entitäts-Instanz existiert im Persistenzkontext nur einmal und ist daher eindeutig.
- Innerhalb des Persistenzkontexts werden die Entitäten und ihr Lebenszyklus verwaltet.
- Die EntityManager API erzeugt und löscht die persistenten Entitäts-Instanzen und findet Entitäten anhand ihres Schlüssels oder kann Abfragen (Queries) durchführen
- Ein EntityManager darf nur einmal existieren, sonst kann es Datenverlusten kommen.

EntityManager

- Ein EntityManager wird vom Container verwaltet und injiziert.

```
@PersistenceContext
EntityManager em;
```

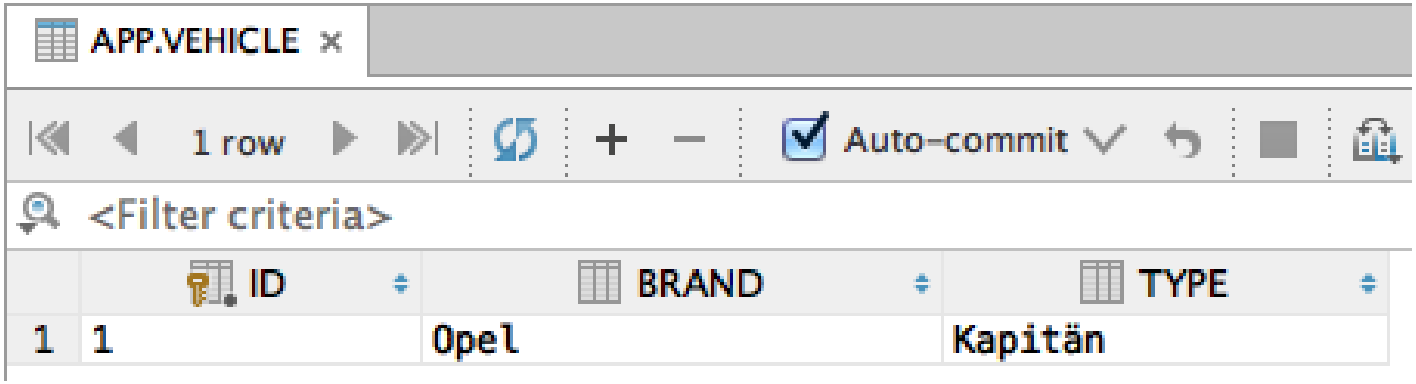
- Gibt es mehr als eine Persistenzunit, muss diese angegeben werden.

```
@PersistenceContext (name = "DbPU")
EntityManager em;
```

This is non-quarkus style

Create

```
Vehicle vehicle = new Vehicle("Opel", "Kapitän");
em.persist(vehicle);
```



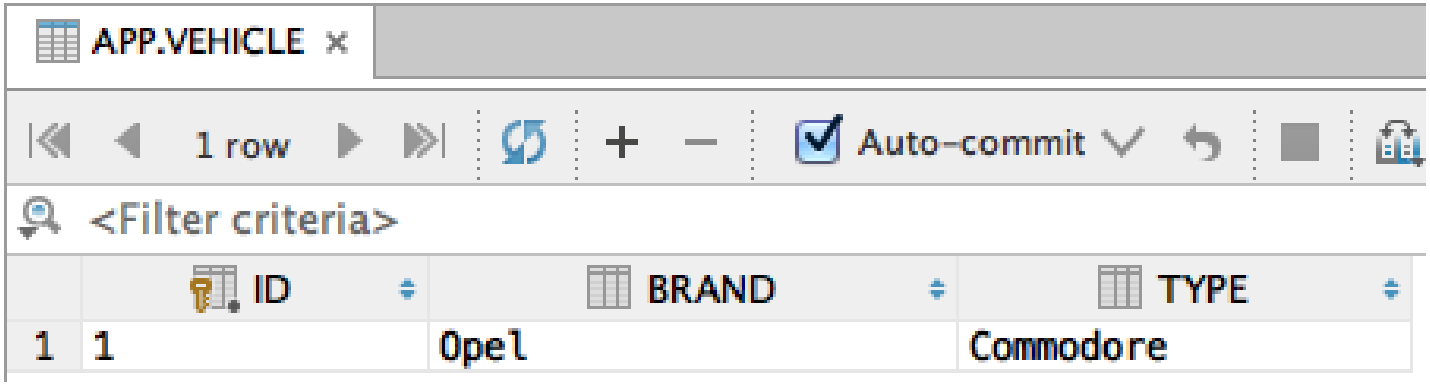
Read

```
vehicle = em.find(Vehicle.class, 1L);
System.out.println(vehicle);
```

```
INFO  [stdout] (ServerService Thread Pool -- 82) Vehicle{id=1, brand='Opel', type='Kapitän'}
```

Update

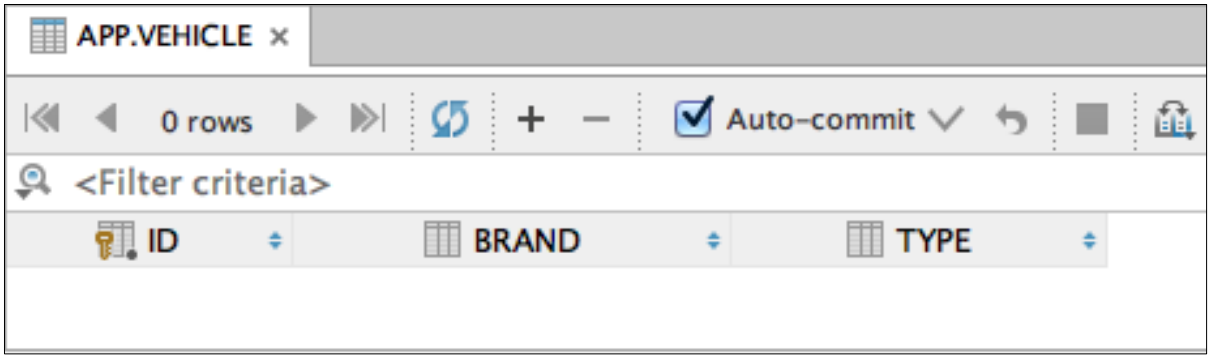
```
vehicle.setType("Commodore");
em.merge(vehicle);
```



	ID	BRAND	TYPE
1	1	Opel	Commodore

Delete

```
em.remove(vehicle);
```



JPA-Converter

http://www.adam-bien.com/roller/abien/entry/new_java_8_date_and

https://weblogs.java.net/blog/montanajava/archive/2014/06/17/using-java-8-datetime-classes-jpa

LocalDate in sql.Date konvertieren 1

JPA-Converter: LocalDate wird transparent als java.sql.Date gespeichert

```
package at.htl.vehicle.entity;

import javax.persistence.*;
import java.io.Serializable;
import java.time.LocalDate;

@Entity
public class VehicleWithDate implements Serializable {

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

    private String brand;
    private String type;
    private LocalDate firstRegistrationDate;

    public VehicleWithDate() {...}

    public VehicleWithDate(String brand, String type, LocalDate firstRegistrationDate) {...}

    Getter and Setter

    @Override
    public String toString() {
        return brand + " "
                + type + " ("
                + firstRegistrationDate.toString() + ")";
    }
}
```

LocalDate in sql.Date konvertieren 2

```
package at.htl.vehicle.entity;

import javax.persistence.AttributeConverter;
import javax.persistence.Converter;
import java.sql.Date;
import java.time.LocalDate;

@Converter(autoApply = true)
public class LocalDateConverter
        implements AttributeConverter<LocalDate, Date> {
    @Override
    public Date convertToDatabaseColumn(LocalDate entityDate) {
        return Date.valueOf(entityDate);
    }

    @Override
    public LocalDate convertToEntityAttribute(Date databaseDate) {
        return databaseDate.toLocalDate();
    }
}
```

Entität mit LocalDate-Attribut

LocalDate in sql.Date konvertieren 3

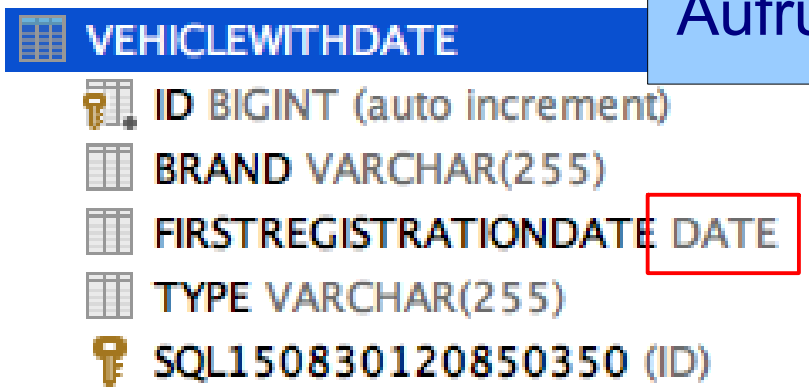
```
import javax.persistence.AttributeConverter;
import javax.persistence.Converter;
import java.sql.Date;
import java.time.LocalDate;

@Converter(autoApply = true)
public class LocalDateConverter implements AttributeConverter<LocalDate, Date> {

    @Override
    public Date convertToDatabaseColumn(LocalDate entityDate) {
        return entityDate == null ? null : Date.valueOf(entityDate);
    }

    @Override
    public LocalDate convertToEntityAttribute(Date databaseDate) {
        return databaseDate == null ? null : databaseDate.toLocalDate();
    }
}
```

Leere Datums-Felder werden berücksichtigt

LocalDate in sql.Date konvertieren 4

```
final DateTimeFormatter dtf = DateTimeFormatter.ofPattern("dd.MM.yyyy");
VehicleWithDate vehicle =
        new VehicleWithDate(
                "Bugatti",
                "Type 57",
                LocalDate.parse("01.01.1936",dtf));
em.persist(vehicle);
```

Transparenter
Aufruf des Converters



	ID	BRAND	FIRSTREGISTRATIONDA...	TYPE
1	1	Bugatti	1936-01-01	Type 57

Beachte bei JavaSE

- Man muss den Converter in die persistence.xml eintragen !!!

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence" version="2.1">
    <persistence-unit name="myTestPU" transaction-type="RESOURCE_LOCAL">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <class>at.htl.roombooker.entity.Booking</class>
        <class>at.htl.roombooker.entity.Form</class>
        <class>at.htl.roombooker.entity.Course</class>
        <class>at.htl.roombooker.entity.Teacher</class>
        <class>at.htl.roombooker.entity.Room</class>
        <class>at.htl.roombooker.entity.converter.LocalDateConverter</class>
        <exclude-unlisted-classes>true</exclude-unlisted-classes>
        <properties>
            <property name="javax.persistence.schema-generation.database.action"
                      value="drop-and-create"/>
            <property name="javax.persistence.jdbc.driver" value="org.apache.derby.jdbc.ClientDriver"/>
            <property name="javax.persistence.jdbc.url" value="jdbc:derby://localhost:1527/db"/>
            <property name="javax.persistence.jdbc.user" value="app"/>
            <property name="javax.persistence.jdbc.password" value="app"/>
        </properties>
    </persistence-unit>

</persistence>
```

JP-QL

Java Persistence Query Language

http://www.objectdb.com/java/jpa/query

http://www.tutorialspoint.com/de/jpa/jpa_jpql.htm

https://en.wikibooks.org/wiki/Java_Persistence/JPQL

https://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/JPQL

Einsatzgebiet

- Ist SQL nachempfunden
- Dient zum
  - Lesen (READ) von Datensätzen: SELECT Query
  - Ändern (UPDATE): UPDATE Query
  - Löschen (DELETE): DELETE Query

  von Datensätzen.
- Arten:
  - Query: Select-Statement direkt im Aufruf
  - NamedQuery: Benannte Query
  - NativeQuery: Natives SQL in der Query
  - NamedNativeQuery

Daten

```
Vehicle[] vehicleArray = {
        new Vehicle("Opel", "Commodore"),
        new Vehicle("Opel", "Kadett"),
        new Vehicle("VW", "Käfer"),
        new Vehicle("BMW", "3"),
        new Vehicle("BMW", "5"),
        new Vehicle("BMW", "7"),
        new Vehicle("Ford", "Taunus"),
        new Vehicle("Ford", "Granada"),
        new Vehicle("Ford", "Capri"),
        new Vehicle("Ford", "Cortina"),
        new Vehicle("NSU", "Prinz")
};

for (Vehicle v : vehicleArray) {
    em.persist(v);
}
```

11 rows | Auto-commit

<Filter criteria>

	ID	BRAND	TYPE
1	2	Opel	Commodore
2	3	Opel	Kadett
3	4	VW	Käfer
4	5	BMW	3
5	6	BMW	5
6	7	BMW	7
7	8	Ford	Taunus
8	9	Ford	Granada
9	10	Ford	Capri
10	11	Ford	Cortina
11	12	NSU	Prinz

Dynamic Query mit Query und TypedQuery

```
Query query = em
        .createQuery("select v from Vehicle v");
List<Vehicle> vehicles = query.getResultList();
System.out.println(vehicles);
```

```
17:28:50,162 INFO  [stdout] (ServerService Thread Pool -- 120) [Opel Commodore, Opel Kadett, VW Käfer, BMW 3,
 BMW 5, BMW 7, Ford Taunus, Ford Granada, Ford Capri, Ford Cortina, NSU Prinz]
```

```
TypedQuery<Vehicle> query = em
        .createQuery("select v from Vehicle v", Vehicle.class);
List<Vehicle> vehicles = query.getResultList();
System.out.println(vehicles);
```

```
17:17:21,004 INFO  [stdout] (ServerService Thread Pool -- 115) [Opel Commodore, Opel Kadett, VW Käfer, BMW 3,
 BMW 5, BMW 7, Ford Taunus, Ford Granada, Ford Capri, Ford Cortina, NSU Prinz]
```

NamedQuery – Definition in Entity

```
@Entity
@NamedQueries({
        @NamedQuery(name = "Vehicle.findAll", query = "select v from Vehicle v"),
        @NamedQuery(name = "Vehicle.findByBrand",
                query = "select v from Vehicle v where v.brand like :BRAND"),
        @NamedQuery(name = "Vehicle.findByBrandAndType",
                query = "select v from Vehicle v where v.brand like ?1 and v.type like ?2")
})
public class Vehicle implements Serializable {

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

    private String brand;
    private String type;

    public Vehicle() {...}

    public Vehicle(String brand, String type) {...}

    Getter and Setter

    @Override
    public String toString() { return brand + " " + type; }
}
```

NamedQuery - Aufruf

```
TypedQuery<Vehicle> query = em
        .createNamedQuery("Vehicle.findAll", Vehicle.class);
List<Vehicle> vehicles = query.getResultList();
System.out.println(vehicles);
```

```
17:45:29,015 INFO  [stdout] (ServerService Thread Pool -- 128) [Opel Commodore, Opel Kadett, VW Käfer, BMW 3,
 BMW 5, BMW 7, Ford Taunus, Ford Granada, Ford Capri, Ford Cortina, NSU Prinz]
```

NamedQuery mit Named Parameter

```
TypedQuery<Vehicle> query = em
        .createNamedQuery("Vehicle.findByBrand", Vehicle.class)
        .setParameter("BRAND","Ford");
List<Vehicle> vehicles = query.getResultList();
System.out.println(vehicles);
```

```
17:48:25,967 INFO  [stdout] (ServerService Thread Pool -- 132) [Ford Taunus, Ford Granada, Ford Capri, Ford
 Cortina]
```

NamedQuery mit Positional Parameters

```
TypedQuery<Vehicle> query = em
        .createNamedQuery("Vehicle.findByBrandAndType"
                , Vehicle.class)
        .setParameter(1,"Ford")
        .setParameter(2,"Cortina");
List<Vehicle> vehicles = query.getResultList();
System.out.println(vehicles);
```

```
17:51:16,958 INFO  [stdout] (ServerService Thread Pool -- 137) [Ford Cortina]
```

NativeQuery (NamedNativeQuery)

```
String sql = "SELECT id, brand, type FROM vehicle WHERE brand LIKE ?1 and type LIKE ?2";
Query query = em
        .createNativeQuery(sql,Vehicle.class)
        .setParameter(1,"Ford")
        .setParameter(2,"Cortina");
List<Vehicle> vehicles = query.getResultList();
System.out.println("Native Query: " + vehicles.get(0).toString());
```

Alle Spalten der Tabelle werden in eine Liste mit Vehicle Objekten geladen

---

```
String sql = "SELECT id, brand, type FROM vehicle WHERE brand LIKE ?1 and type LIKE ?2";
Query query = em
        .createNativeQuery(sql,Vehicle.class)
        .setParameter(1,"Ford")
        .setParameter(2,"Cortina");
Vehicle vehicle = (Vehicle) query.getSingleResult();
System.out.println("Native Query 3: " + vehicle);
```

Alle Spalten der Tabelle werden in ein Vehicle Objekt geladen

---

```
String sql = "SELECT brand, type FROM vehicle WHERE brand LIKE ?1 and type LIKE ?2";
Query query = em
        .createNativeQuery(sql)
        .setParameter(1,"Ford")
        .setParameter(2,"Cortina");
List<Object[]> result = query.getResultList();
System.out.println("Native Query: " + result.get(0)[0] + " " + result.get(0)[1]);
```

Nur gewisse Spalten der Tabelle werden in ein Objekt-Array geladen

```
10:39:03,041 INFO  [stdout] (ServerService Thread Pool -- 90) Native Query: Ford Cortina
```

Aggregationen

```
TypedQuery<Long> query = em.createQuery("SELECT COUNT(v) FROM Vehicle v", Long.class);
long noOfVehicles = query.getSingleResult();
System.out.println("Anzahl Fahrzeuge (TypedQuery): " + noOfVehicles);
```

```
11:11:54,473 INFO  [stdout] (ServerService Thread Pool -- 155) Anzahl Fahrzeuge (TypedQuery): 11
```

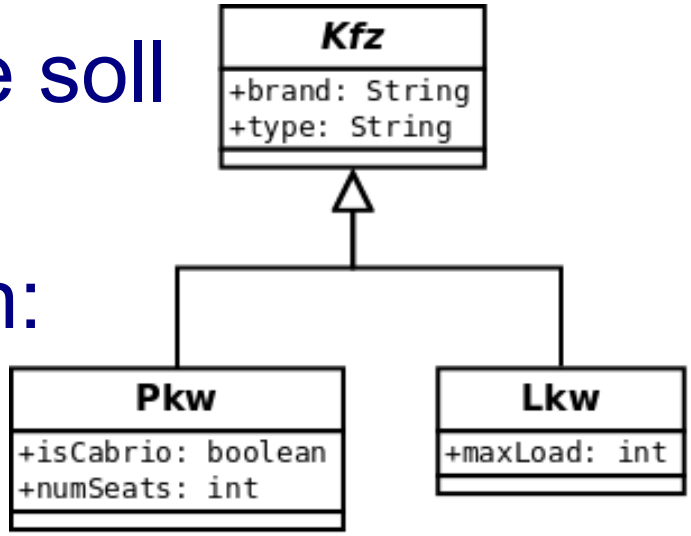
```
Query query = em.createNativeQuery("SELECT COUNT(*) FROM vehicle");
Integer noOfVehicles = (Integer) query.getSingleResult();
System.out.println("Anzahl Fahrzeuge (NativeQuery): " + noOfVehicles);
```

```
11:11:54,477 INFO  [stdout] (ServerService Thread Pool -- 155) Anzahl Fahrzeuge (NativeQuery): 11
```

Beziehungen zwischen Entitäten

Vererbung

- Eine Vererbungshierarchie soll persistiert werden
- Hier gibt es drei Strategien:
  - **JOINED**
  - SINGLE_TABLE
  - TABLE_PER_CLASS



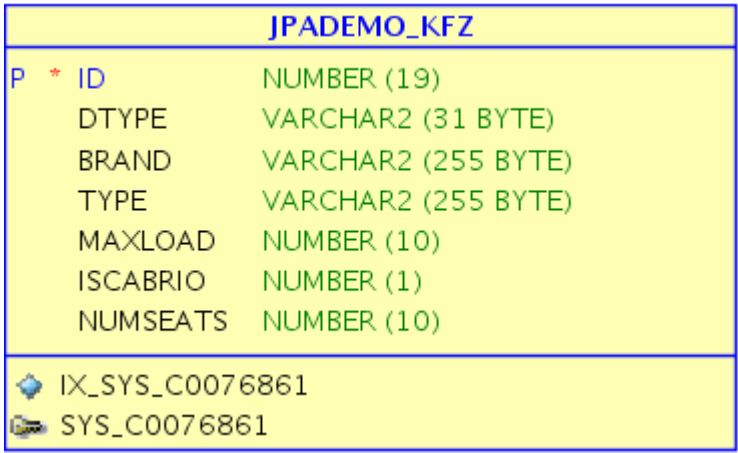

```
@Entity
@Table(name="jpademo_kfz")
@Inheritance(strategy=InheritanceType.JOINED)
@NamedQueries ({
    @NamedQuery(name="Kfz.findAll",
                query="Select k FRO
    @NamedQuery(name="Kfz.findByBra
                query="Select k FROM Kfz k WHERE k.brand LIKE :brand")
})

@SequenceGenerator(name="KfzSeq", sequenceName="JPADEMO_KFZ_SEQ",initia
public abstract class Kfz implements Serializable {
```

```
JOINED          InheritanceType
SINGLE_TABLE    InheritanceType
TABLE_PER_CLASS InheritanceType
class
```

InheritanceType.SINGLE_TABLE

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@Table(name="jpademo_kfz")
public abstract class Kfz implements Serializable {
    @Id
    private Long id;
    private String brand;
    private String type;

@Entity
@Table(name="jpademo_pkw")
public class Pkw extends Kfz implements Serializable {
    private boolean isCabrio;
    private int seats;
```

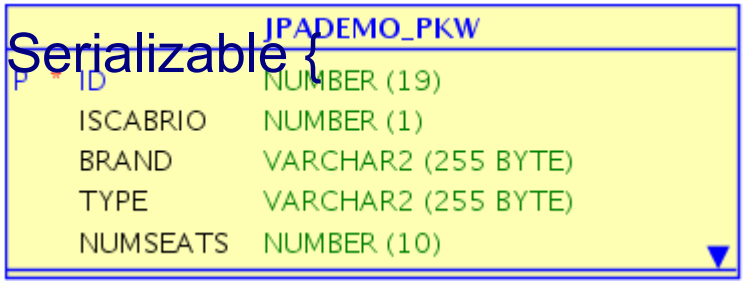
Es wird **eine** Tabelle mit **allen** Attributen für **sämtliche** Klassen angelegt (mit Diskriminator DTYPE)

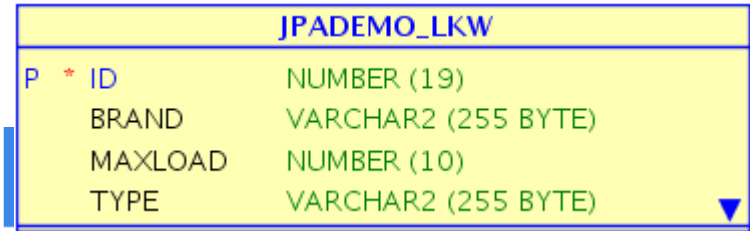


InheritanceType.TABLE_PER_CLASS

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
@Table(name="jpademo_kfz")
public abstract class Kfz implements Serializable {
    @Id
    private Long id;
    private String brand;
    private String type;

@Entity
@Table(name="jpademo_pkw")
public class Pkw extends Kfz implements Serializable {
    private boolean isCabrio;
    private int seats;
```

Es wird **je konkreter** Klasse **eine** Tabelle mit allen Attributen der Klasse angelegt

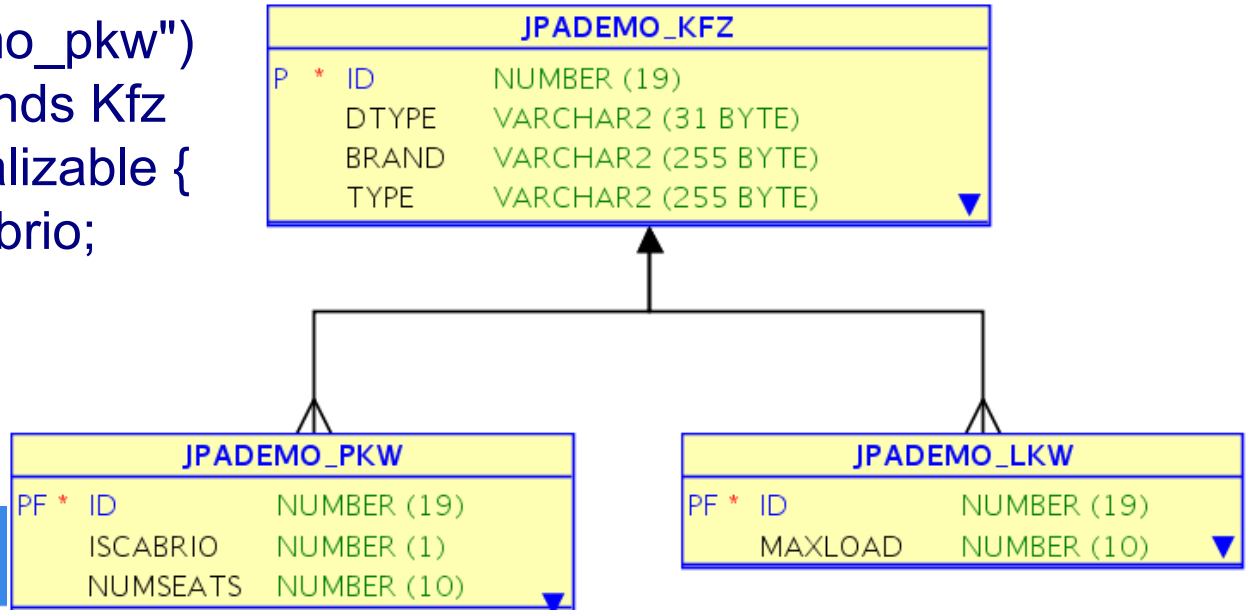




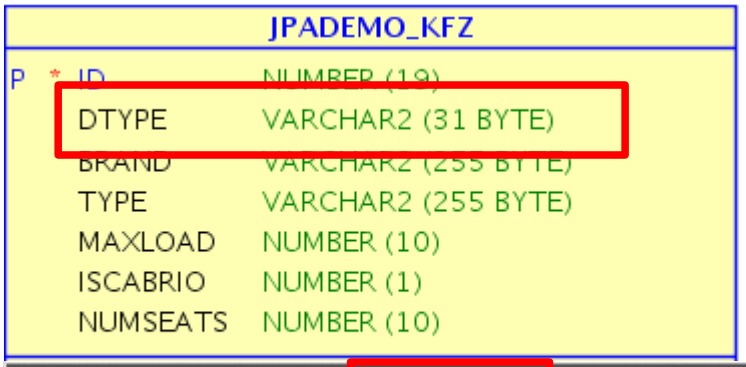
InheritanceType.JOINED

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@Table(name="jpademo_kfz")
public abstract class Kfz implements Serializable {
    @Id
    private Long id;
    private String brand;
    private String type;

@Entity
@Table(name="jpademo_pkw")
public class Pkw extends Kfz
        implements Serializable {
    private boolean isCabrio;
    private int seats;
```

Die Attribute der Basisklasse in einer Tabelle und je eine Tabelle pro abgeleiteter Klasse (mit Diskriminator DTYPE)



Diskriminator



Dies ist ein Beispiel für

„Convention over Configuration“

#	KFZ_ID	DTYPE	KFZ_BRAND	KFZ_TYPE	KFZ_REG_ID
1	1	Pkw	BMW	X3	1
2	2	Lkw	Opel	Blitz	<NULL>

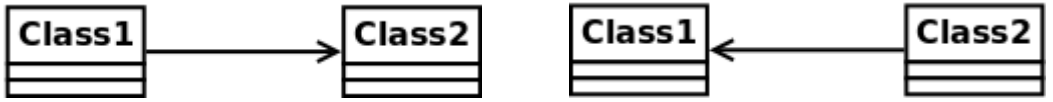
- lat. discriminare = trennen, scheiden
- Der Persistence Provider kann so unterscheiden, zu welcher Klasse die Zeile gehört
- Mit der Annotation `@DiskriminatorValue` kann für jede Tabelle ein eigener Wert (anstelle des Klassennamens vergeben werden.

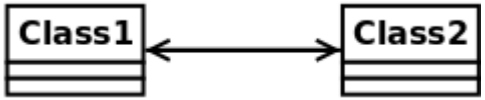
Zugriff auf den Diskriminator

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@DiscriminatorColumn
public abstract class Kfz {

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

    private String brand;

    @Column(name = "DTYPE", insertable = false, updatable = false)
    private String dType;
```

Um auf den Diskriminator (-Wert) zugreifen zu können, verwendet man ein schreibgeschütztes Attribut

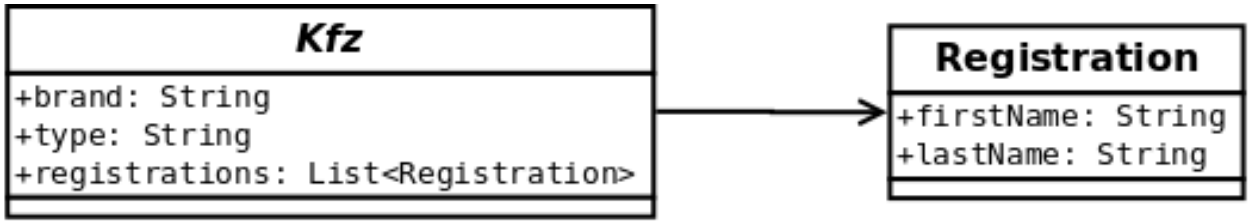
Beziehungen zw. Tabellen (Joins)

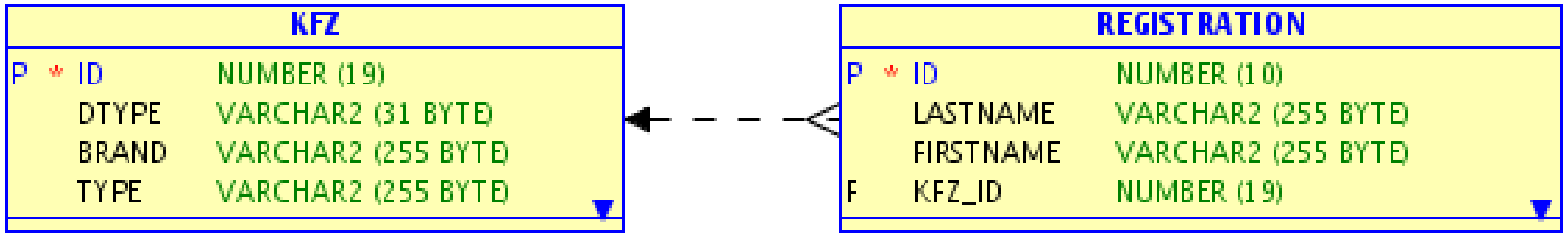
- 3 Arten (Multiplizität / Kardinalität)
  - 1:1
  - 1:n
  - n:m (assoziative Tabelle notwendig)
- Möglichkeiten zur Navigation im Klassenmodell
  - Unidirektional 

  - Bidirektional 

- Ausgestaltung im Datenmodell
  - Mit assoziativer Tabelle (1:n, n:m)
  - Ohne assoziative Tabelle (nur bei 1:1 und 1:n möglich)

1:n unidirektional, ohne assoz. Tabelle – Var. 1



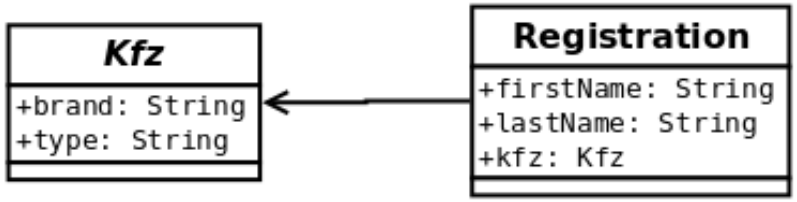


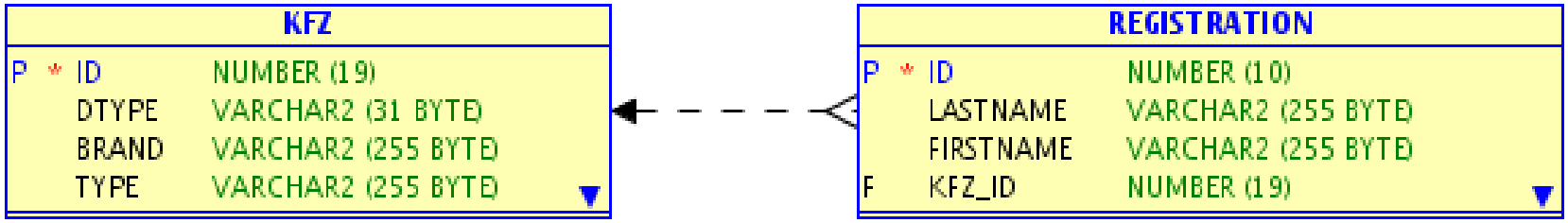

```
@Entity
weitere Annotationen ...
public abstract class Kfz implements Serializable {
    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE, generato
    private Long id;
    private String brand;
    private String type;
    @OneToMany(cascade=CascadeType.ALL)
    @JoinColumn(name="kfz_id")
    private List<Registration> registrations;
```

```
@Entity
weitere Annotationen ...
public class Registration implements Serializable {
    static final long serialVersionUID = 1;
    @Id
    @GeneratedValue(generator = "registrationSeq")
    private int id;
    private String firstName;
    private String lastName;
```

Durch JoinColumn gibt es keine assoziative Tabelle
CascadeType.ALL wandelt die Beziehung (Assoziation) in eine Komposition um (existenzabhängig)

1:n unidirektional, ohne assoz. Tabelle – Var. 2

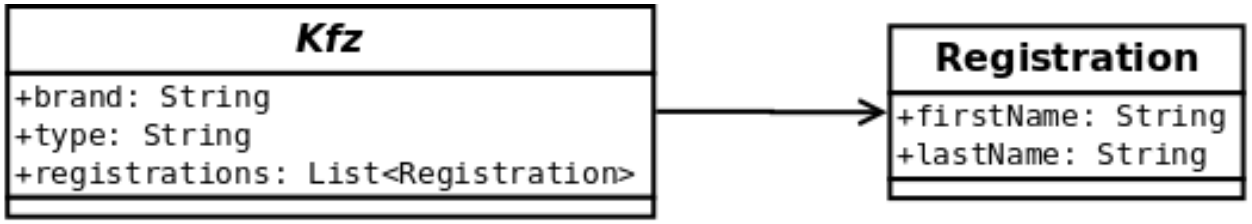


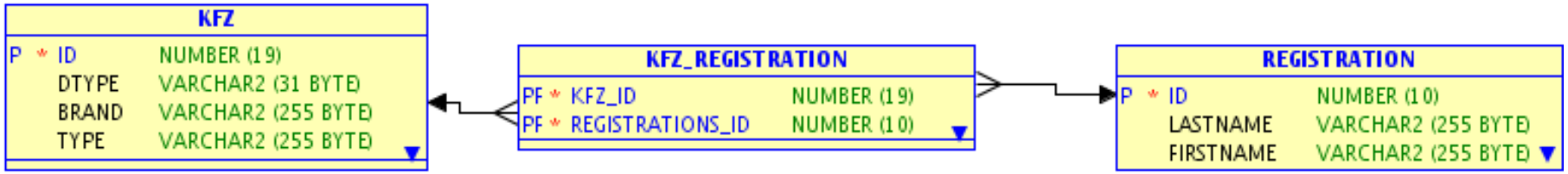



```
@Entity
weitere Annotationen ...
public abstract class Kfz implements Serializable {
    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE, generat
    private Long id;
    private String brand;
    private String type;
```

```
@Entity
weitere Annotationen ...
public class Registration implements Serializable {
    static final long serialVersionUID = 1;
    @Id
    @GeneratedValue(generator = "registrationSeq")
    private int id;
    private String firstName;
    private String lastName;
    @ManyToOne
    private Kfz kfz;
```

1:n unidirektional, mit assoz. Tabelle – Var. 1



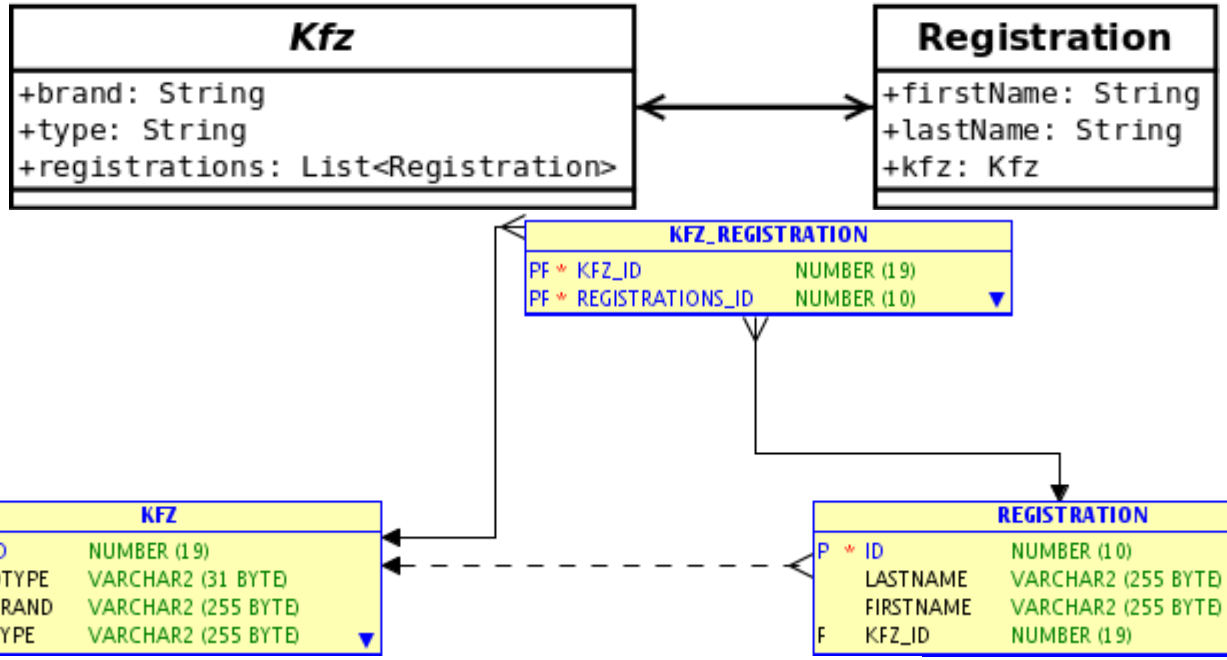



```
@Entity
weitere Annotationen ...
public abstract class Kfz implements Serializable {
    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE, generator=
    private Long id;
    private String brand;
    private String type;
    @OneToMany(cascade=CascadeType.ALL)
    private List<Registration> registrations;
```

```
@Entity
weitere Annotationen ...
public class Registration implements Serializable {
    static final long serialVersionUID = 1;
    @Id
    @GeneratedValue(generator = "registrationSeq")
    private int id;
    private String firstName;
    private String lastName;
```

Anstelle von Lists können auch Sets oder Bags vorteilhaft sein

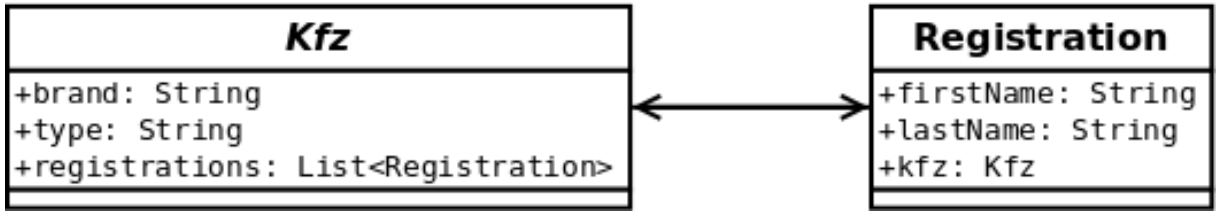
1:n bidirektional, mit assoz. Tabelle

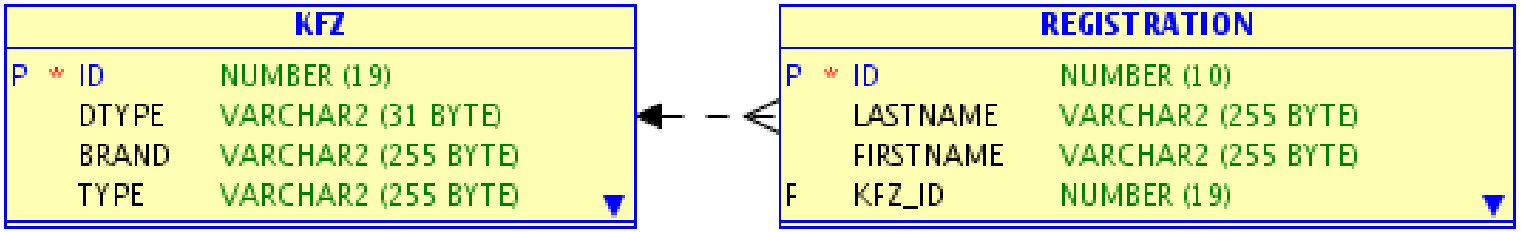



```java
@Entity
weitere Annotationen ...
public abstract class Kfz implements Serializable {
    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="K
    private Long id;
    private String brand;
    private String type;
    @OneToMany(cascade=CascadeType.ALL)
    private List<Registration> registrations;
```

```java
@Entity
weitere Annotationen ...
public class Registration implements Serializable {
    static final long serialVersionUID = 1;
    @Id
    @GeneratedValue(generator = "registrationSeq")
    private int id;
    private String firstName;
    private String lastName;
    @ManyToOne
    private Kfz kfz;
```

1:n bidirektional, ohne assoz. Tabelle






```java
@Entity
weitere Annotationen ...
public abstract class Kfz implements Serializable {
    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE, generato
    private Long id;
    private String brand;
    private String type;
    @OneToMany(mappedBy="kfz",cascade=CascadeType.ALL)
    private List<Registration> registrations;
```

```java
@Entity
weitere Annotationen ...
public class Registration implements Serializable {
    static final long serialVersionUID = 1;
    @Id
    @GeneratedValue(generator = "registrationSeq")
    private int id;
    private String firstName;
    private String lastName;
    @ManyToOne
    private Kfz kfz;
```

Mit @JoinColumn(name="kfz_id", nullable=false) in der Klasse Registration kann verhindert werden, dass es eine Registration ohne Kfz gibt.

Troubleshooting – Immer Logs kontrollieren

```
[EL Fine]: Connection(1943219781)--CREATE TABLE jpademo_kfz_JPADEMO_REGISTRATION (Kfz_ID NUMBER(19) NOT NULL, registrati
[EL Fine]: SELECT 1 FROM DUAL
[EL Warning]: Exception [EclipseLink-4002] (Eclipse Persistence Services - 2.0.0.v20091127-r5931): org.eclipse.persisten
Internal Exception: java.sql.SQLSyntaxErrorException: ORA-00972: identifier is too long

Error Code: 972



943219781)--ALTER TABLE jpademo_kfz_JPADEMO_REGISTRATION ADD CONSTRAINT jpdmkfzJPDMRGSTRATIONrgstrtnsD FOREIGN KEY
M DUAL
 [EclipseLink-4002] (Eclipse Persistence Services - 2.0.0.v20091127-r5931): org.eclipse.persistence.exceptions.Dat
a.sql.SQLSyntaxErrorException: ORA-00972: identifier is too long


mo_kfz_JPADEMO_REGISTRATION ADD CONSTRAINT jpdmkfzJPDMRGSTRATIONrgstrtnsD FOREIGN KEY (registrations_ID) REFERENCE
```

HTL LEONDING

Schön, hier zu lernen.