Hochschule **Rosenheim**
University of Applied Sciences

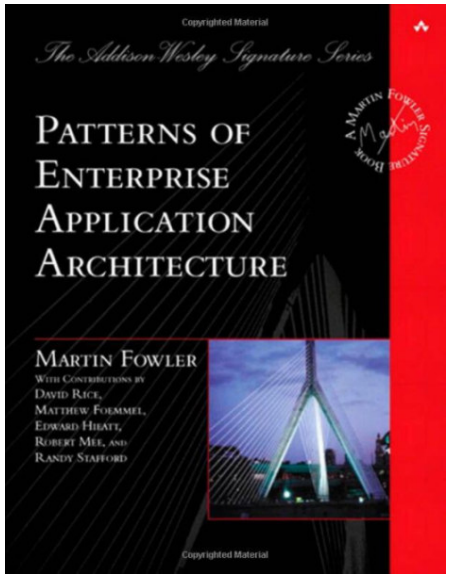
Chapter 9 – Application programming and JPA

Databases lectures

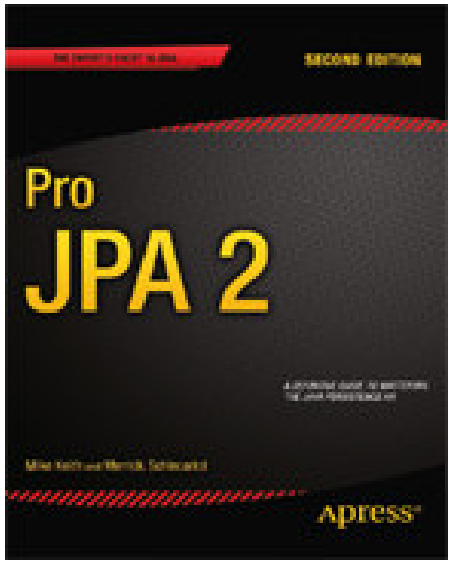
Prof. Dr Kai Höfig

Further literature

- Martin Fowler: Patterns of Enterprise Application Architecture (as an ebook in OPAC)



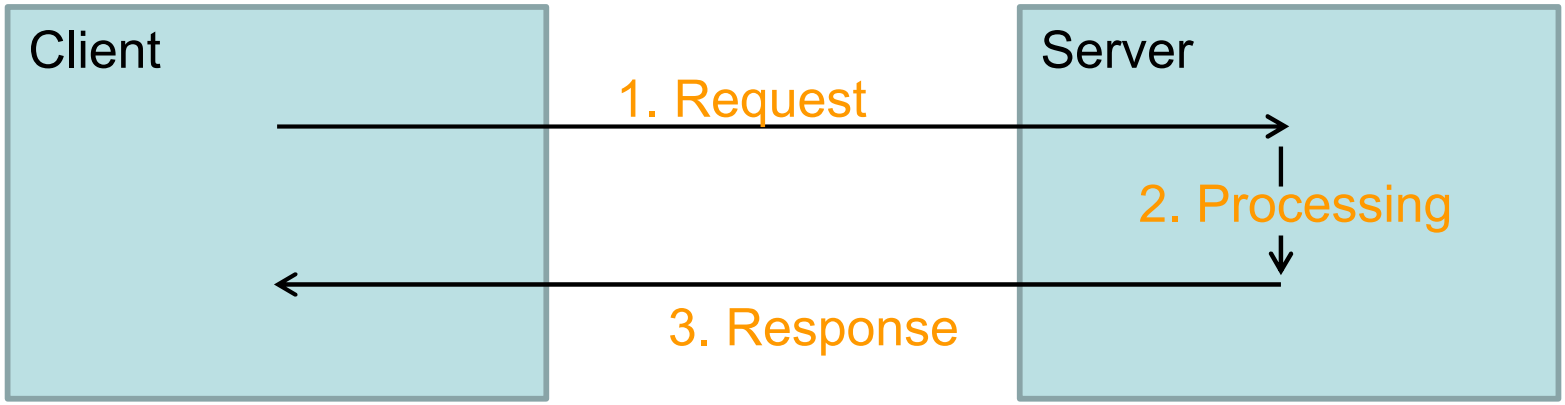
- Mike Keith; Merrick Schincariol: Pro JPA 2, Second Edition (as an ebook in OPAC)



- Michael Inden Persistenzlösungen und REST-Services

The client-server model and how we have worked so far

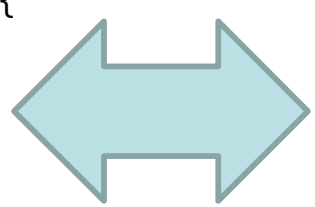
- Principle: client makes use of services from a server



- The MS SQL DBMS was our client so far.
- In this chapter we will look at how applications access a database and learn about this using the example of Java JPA.

Goal today: persistence of objects

```
public class Car {
    private String manufacturer;
    private String name;
    private int price;

    public Car(String manufacturer, String name){
        this.manufacturer=manufacturer;
        this.name=name;
        this.price=0;
    }

    public void setPrice(int price){
        this.price=price;
    }
}
```



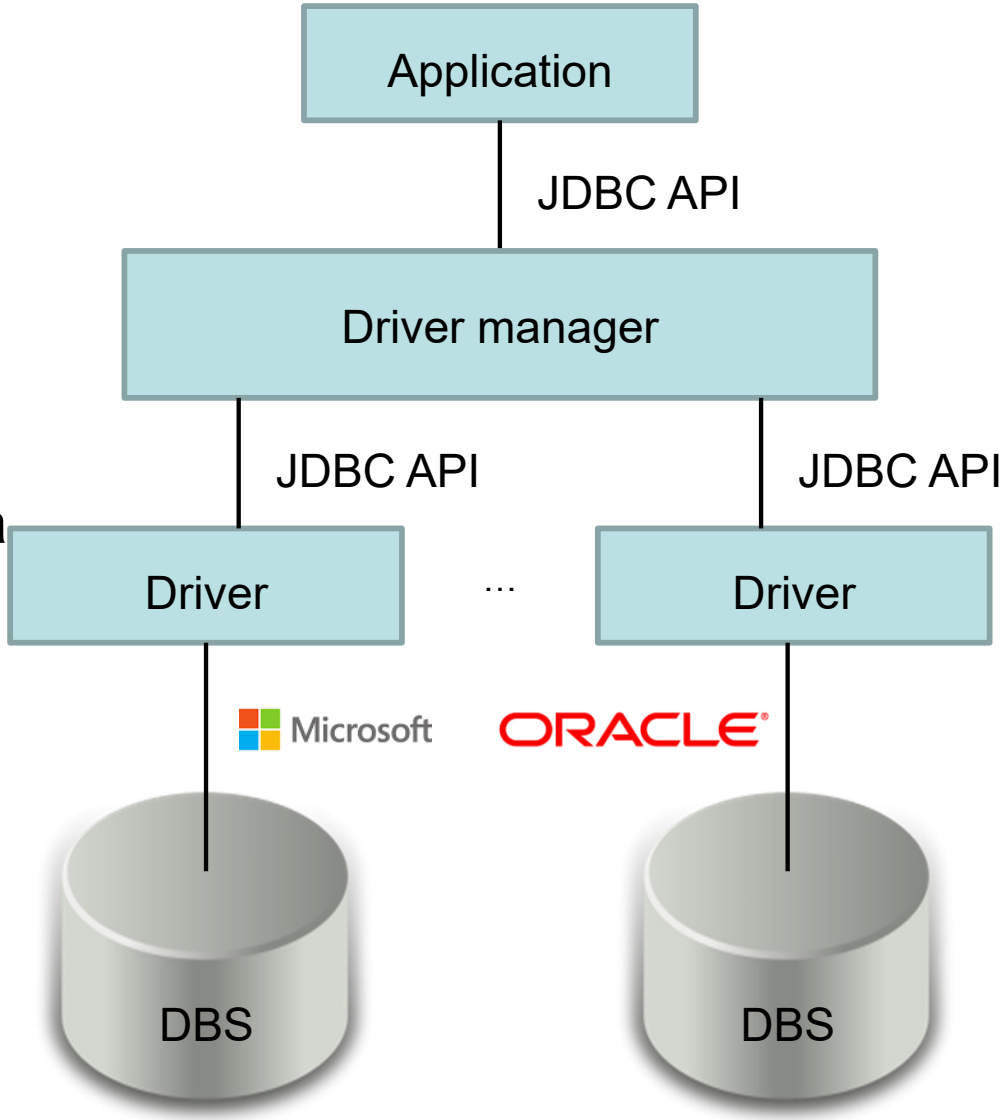
Ergebnisse | Meldungen

	name	hersteller	preis
1	Polo	VW	29000
2	1er	BMW	36500

- We now want to store (**persist**) objects that have been created and later load them again from the database.
- To do so, they must be **serialised** (made storable) and then deserialised (loaded).
- Objects and tuples are different models. That is why we also talk about **impedance mismatch** in persistence.

An overview of JDBC and persistence

- JDBC
  - Database access interface for Java
  - Abstract, database-neutral interface: access to various database systems is possible using system-specific drivers
  - Low-level API: direct use of SQL
- **Persistence** with built-in JDBC tools is a manual task, which is time-consuming and error-prone.
- JDBC is a widely-used standard and makes it relatively easy to query a database. There are better alternatives for persistence.



Java.sql

- Java package `java.sql`
  - `DriverManager`: starting point, loading of drivers

```
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
```

  - `Connection`: database connection

```
Connection conn = DriverManager.getConnection(connectionString,username,password);
```

  - `Statement`: execution of statements (instructions) via a connection

```
conn.createStatement().execute("INSERT INTO Person (name) VALUES ('Maria')");
```

  - `ResultSet`: manages the results of a query, access to individual columns

```
ResultSet result = conn.createStatement().executeQuery("SELECT name FROM Person");
```

Overview of `Java.sql.ResultSet`

- `boolean next()`
  Moves the cursor forward one row from its current position. Can be used to iterate through result set using a while loop.

- `boolean first()`
  Moves the cursor to the first row in this ResultSet object.

- `double getString(int columnIndex)`
  Retrieves the value of the designated column in the current row of this ResultSet object as a string in the Java programming language. (**many more data types available**)

- `void close()`
  Releases this ResultSet object's database and JDBC resources immediately instead of waiting for this to happen when it is automatically closed.

```
ResultSet allPersons = conn.createStatement().executeQuery("SELECT name FROM person");
while(allPersons.next()){
    System.out.println(allPersons.getString("name"));
}
allPersons.close();
```

https://docs.oracle.com/javase/7/docs/api/java/sql/ResultSet.html

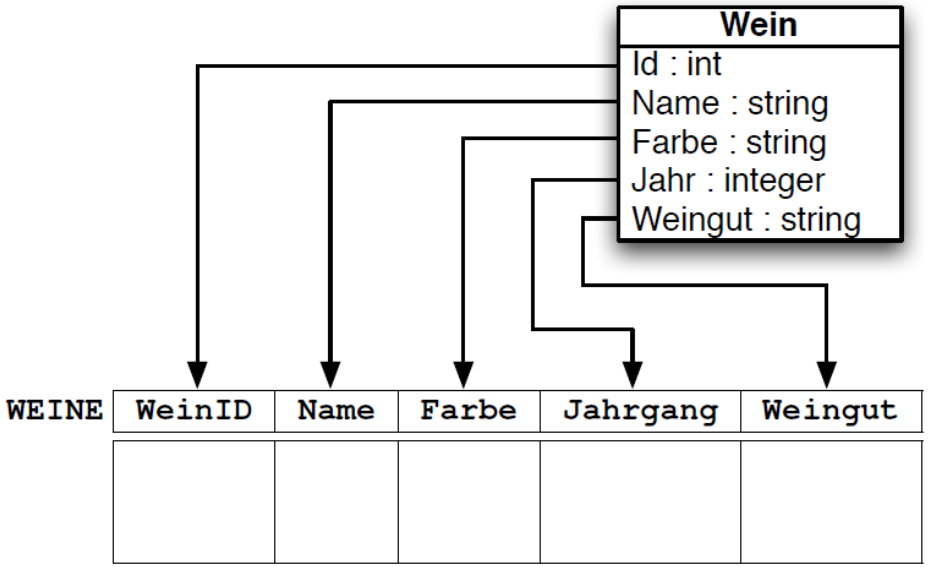
Embedded SQL

- An alternative notation to the classes in `Java.sql` is embedded SQL.
- All statements (instructions) are then translated into regular JDBC transactions.
- Example:

```
String name;
String growing_area = "Toscana";
String region = "Italy";
#sql { SELECT vineyard INTO :name FROM producer WHERE growing_area = :growing_area AND region = :region };
#sql iter = { SELECT name, colour, vintage FROM wines };
while (iter.next ()) {
    System.out.println(iter.name()+":"+iter.colour()+" "+iter.year());
}
```

Java Persistence API (JPA)

- JPA offers better abstraction than just using JDBC on its own.
- Good integration with the Java collections framework.
- High degree of automation for persistence. Serialisation and deserialisation using tags.
- 3 core elements
  1. annotation of the classes to be serialised
  2. (minimal) configuration in the `persistence.xml` file
  3. persistence through use of `EntityManager`



Example of annotated class

- `@Entity` marks a class as intended for persistence. Instances of a class annotated as an entity become tuples in the database.
- `@Table` specifies a particular table for persistence (optional).
- `@Id` specifies the key attribute(s).
- `@GeneratedValue` marks the attribute as a generated key.

```java
@Entity
@Table(name = "Person")
public class Person implements Serializable
{
    @Id
    @GeneratedValue
    //@GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "First name")
    private String firstName;

    @Column(name = "Last name")
    private String lastName;
```

- `@Column` specifies an attribute as a column. (Name optional)
- Full list available at https://www.objectdb.com/api/java/jpa/annotations

Annotations for relationships

- `@ManyToMany`
  A new table is automatically created for the representation of this relationship. Additional attributes can only be saved if the relationship is split manually.

- `@ManyToOne`
  The class in which the annotation is located references an element of another entity.

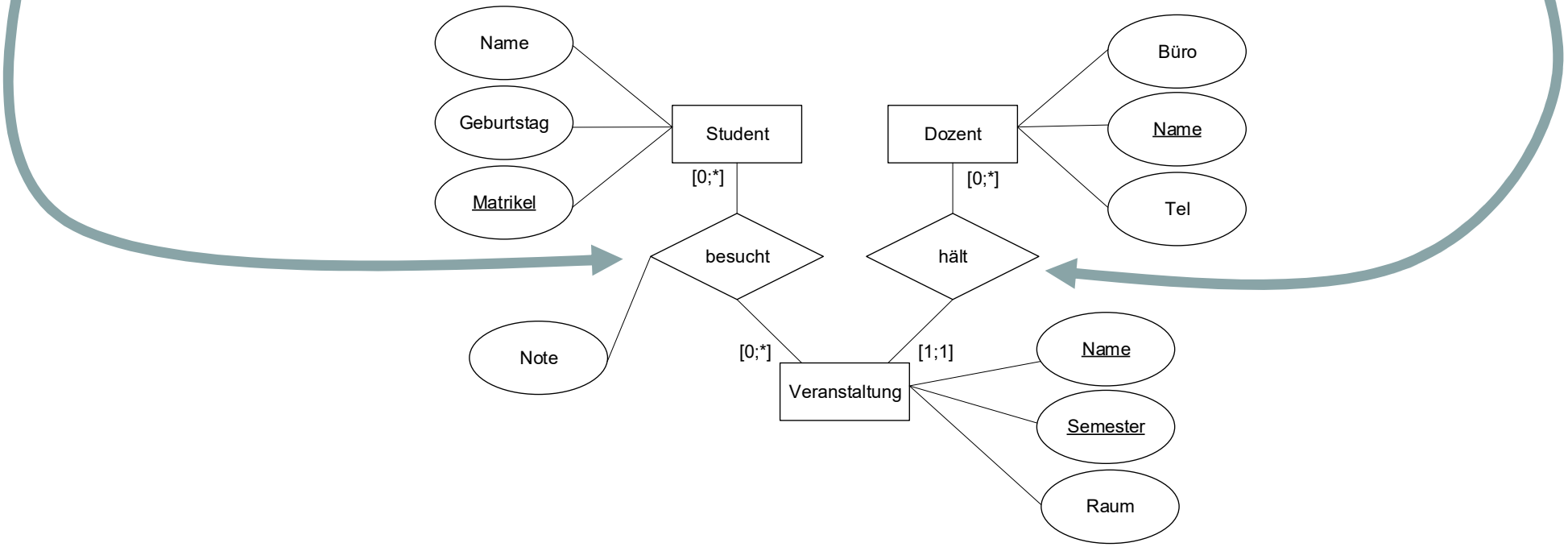
- `@OneToMany`
  The class in which the annotation is located references multiple elements of another entity. Most of the time, this is a list type of the Java collections framework.

- `@OneToOne`
  The class in which the annotation is located references an element of another entity. The attribute becomes a key.

Annotation of relationships - example

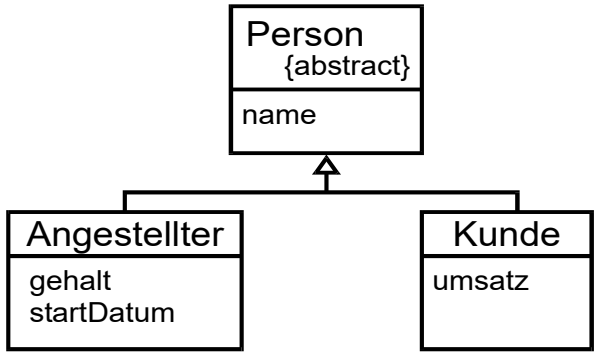
```java
@Entity
@Table(name = "Student")
public class Student  implements Serializable {

    ...

    @OneToMany
    private List<Participation> lectures;
 ...
```

```java
@Entity
@Table(name = "Lecture")
public class Lecture implements Serializable {

    @ManyToOne
    @JoinColumn(name = "Lecturer")
    private Professor prof;
 ...
```



Due to the annotation of the keys with `@Id` of the classes referenced, the specification of `@JoinColumn` is usually unnecessary, except if (as in this example) the name of the attributes is specified manually.

Mapping inheritance



default

null-value style (`single_table`)

Person					
OID	name	gehalt	startDatum	umsatz	objTyp

`table_per_class`

Kunde		
OID	name	umsatz

Angestellter			
OID	name	gehalt	startDatum

ER style (`joined`)

Person		
OID	name	objTyp

Kunde	
OID (FS)	umsatz

Angestellter		
OID (FS)	gehalt	startDatum

```java
@Entity
@Inheritance(strategy= InheritanceType.JOINED)
public class Employee extends Person implements Serializable {
    @Column
    private String company;

    public Employee(String firstName, String lastName, String company) {
        super(firstName, lastName);
        this.company = company;
    }
}
```

Configuration using `persistence.xml`

- A minimal configuration for the JPA framework is specified in the persistence.xml file.
  1. Connection settings for the database (URL, user name, password, driver).
  2. Which classes should be persistent (all or some).
  3. How to handle changes to attributes and classes.

`none`
No schema creation or deletion will take place.

`create`
The provider will create the database artefacts on application deployment. The artefacts will remain unchanged after application redeployment.

`drop-and-create`
Any artefacts in the database will be deleted, and the provider will create the database artefacts on deployment.

`drop`
Any artefacts in the database will be deleted on application deployment.

Access to the database using EntityManager

- EntityManager
  - Is the context that manages entities/objects
  - Provides connection to the database
  - Provides environment for transactions
- Working with the EntityManager

```
EntityManagerFactory emf =
   Persistence.createEntityManagerFactory("WineManagement");
   // cf. persistence.xml, see later

EntityManager em = emf.createEntityManager();

// … Do something exciting

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

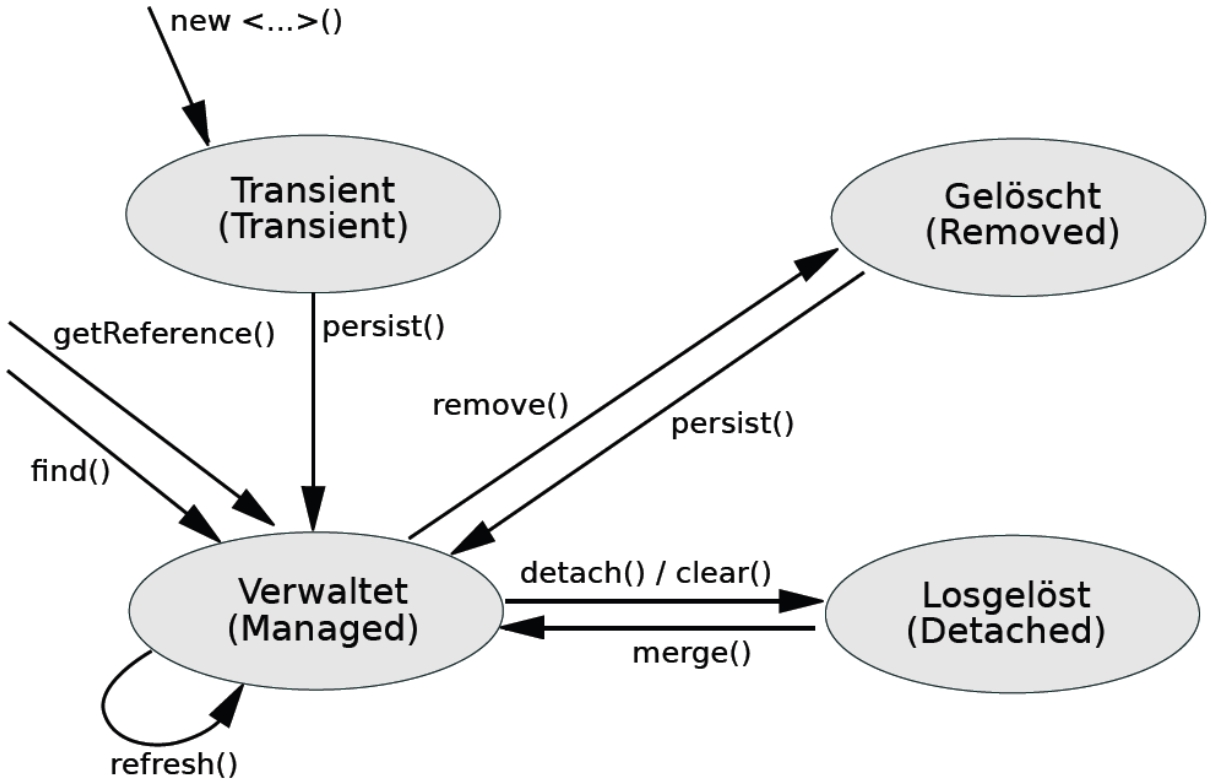
Access to the database using EntityManager

- Create / change / delete data

```
EntityManager em = …;
em.getTransaction().begin();

Producer mueller = em.find( Producer.class, "Müller");

Wine w = new wine (4714, " Dom Perinjong", WineColour.ROSE, 2013, mueller);
em.persist(w);

Wine w2 = em. find ( Wine.class, 4713); // saved earlier
System.out.println("Found:" + w2);

em.remove(w2);

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

- As soon as an entity is managed by EntityManager, all changes are automatically saved
- Persist and remove and/or detach are explicitly required

Life cycle of an entity



- = state model of the entities with respect to EntityManager

Source of diagram: Java Persistencs API 2.0, Hanser

Queries in JPA

JPA offers many options for queries

- "pure, low-level" SQL
- JPA QL - "object-oriented SQL standardised for Java"
  - Example:

```
TypedQuery<Wine> q
  = em.createQuery("select w from Wine w where w.vintage > :aVintage",
Wine.class);
q.setParameter("aVintage", 2000);

List<Wine> newerThan2000 = q.getResultList();

for(Wine fromDB: newerThan2000) {
   System.out.println("Newer than 2000:" + fromDB);
}
```

- Query by Criteria (similar to Query by Example)
  - Programme structure of the query