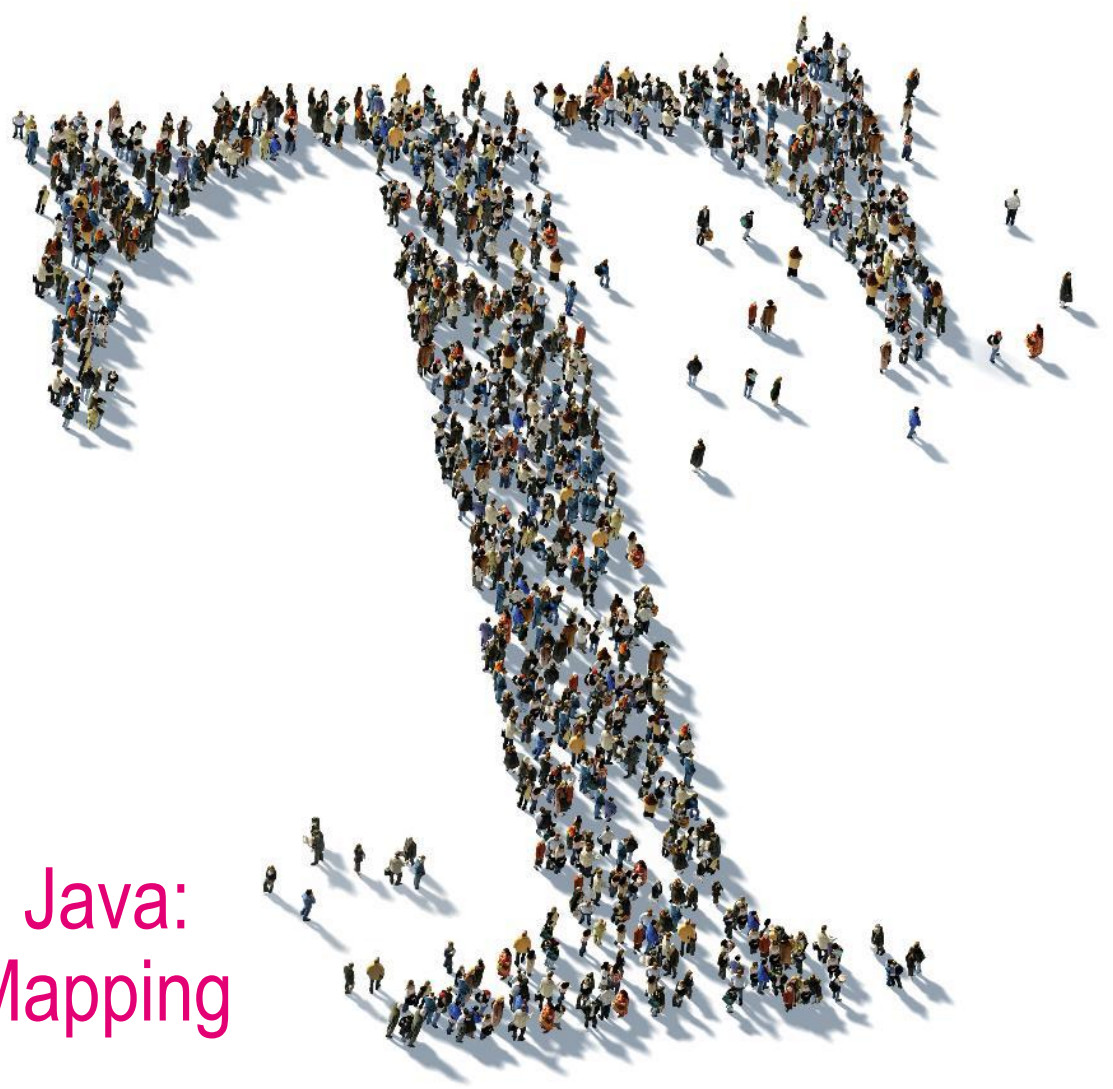


Java Lecture #8

Database Usage in Java: Object-Relational Mapping



Persistence

Под Persistence понимается методы которые описывает действия работы с данными некоторого процесса

В Java существует много способов работы с данными:

JDBC

Serialization

File IO

JCA

Java Persistence Architecture API ([JPA](#)) это Java спецификация для доступа, сохранения и манипулирования данными между Java объектами и реляционной базой данных.

JPA было представлено как часть спецификации EJB 3.0 как замена EJB 2 CMP Entity Beans. Сейчас JPA это промышленный стандарт ORM (Object to Relational Mapping)

JPA состоит из следующих основных разделов:

- **The Java Persistence API**
- **The query language**
- **The Java Persistence Criteria API**
- **Object/relational mapping metadata**

Entities

Энтити это легковесные доменные объекты. Обычно энтити представляют таблицу в реляционной базе данных., каждый инстанс энтити представляет собой запись в таблице..

Энтити должны удовлетворять следующим требованиям.

- Должны быть анотированны `javax.persistence.Entity`.
- Должен быть публичный конструктор по умолчанию. Но при этом класс может иметь и другие конструкторы.
- Класс не должен быть `final`, поля и методы для работы с персистентными данными тоже
- Если энтити будет использоваться в ремоутном бизнес интерфейсе то она должна реализовывать интерфейс `Serializable`.
- Энтити могут расширять как другие энтити, так и обычные классы.

Интерфейс EntityManager

Энтити управляются при помощи EntityManager, который ассоциируется с персистент контекстом – набором managed entity.

EntityManager API CRUD операции + запросы :

- persist (INSERT)
- merge (UPDATE)
- remove (DELETE)
- find (SELECT)
- ...

Типы EntityManager

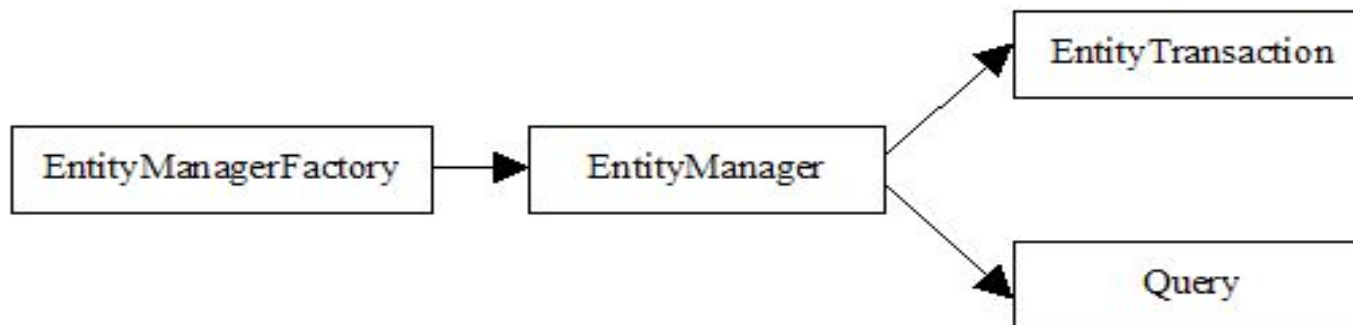
Существуют два типа **EntityManager** :

Container-Managed Entity Managers – контролируется автоматически J2EE контейнером. Использует JTA для работы с транзакциями.

@PersistenceContext

EntityManager em;

Application-Managed Entity Managers – контролируется пользовательским приложением .



Application-Managed Entity Managers

```
EntityManager em = ...
```

```
....
```

```
EntityTransaction trx = em.getTransaction();
```

```
try {
```

```
    trx.begin();
```

```
        // Operations that modify the database should come here.
```

```
    trx.commit();
```

```
} finally
```

```
{
```

```
    if (trx.isActive())
```

```
        trx.rollback();
```

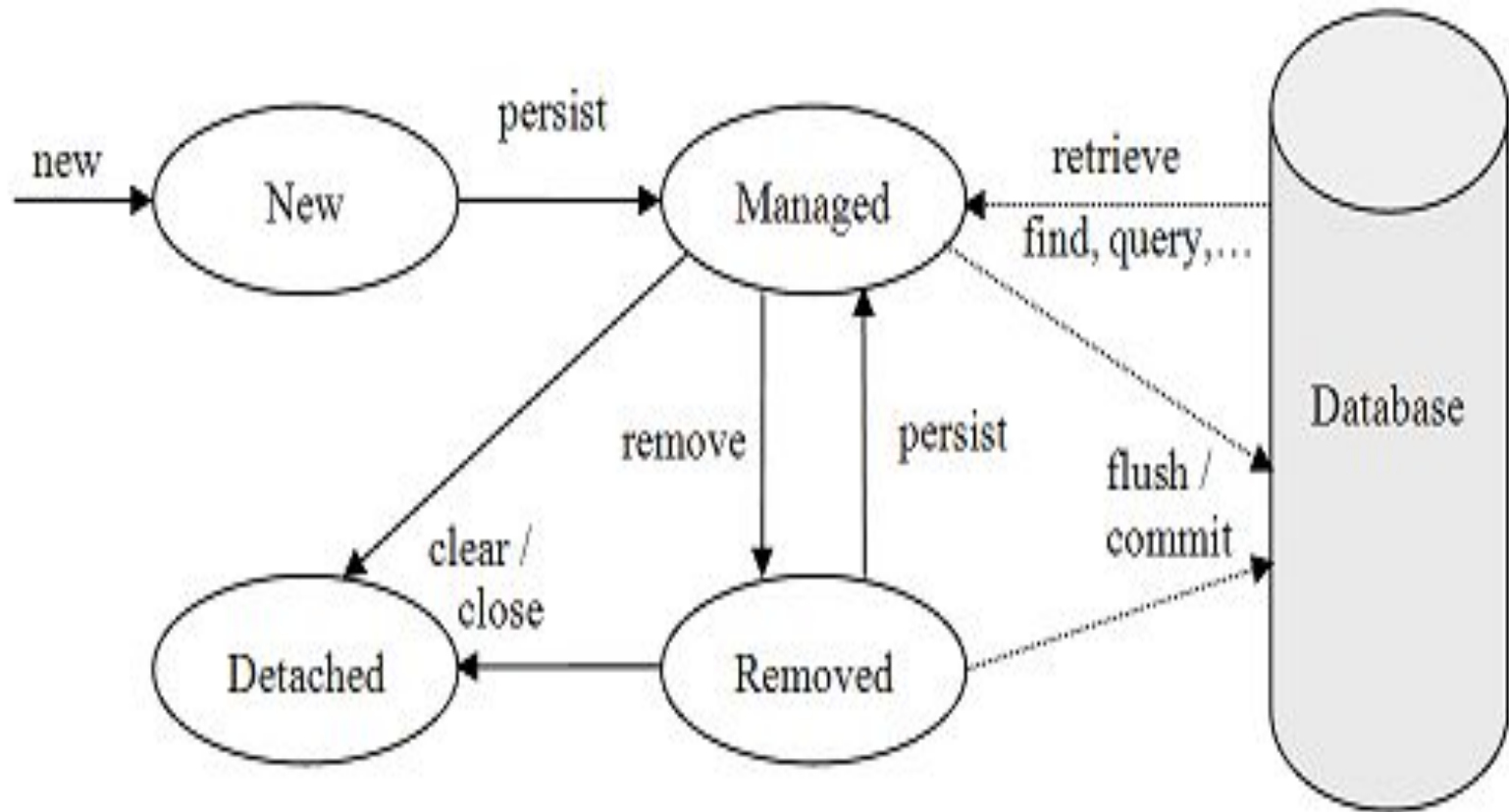
```
}
```

Persistence Units

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
  <persistence-unit name="myapp" transaction-type="RESOURCE_LOCAL">
  <!-- <persistence-unit name="myapp" transaction-type="JTA" --> -->
  <provider>org.hibernate.ejb.HibernatePersistence</provider>
  <!-- <jta-data-source>jdbc/_default</jta-data-source> -->
  <properties>
    <property name="javax.persistence.jdbc.driver" value="org.apache.derby.jdbc.ClientDriver"/>
    <property name="javax.persistence.jdbc.url" value="jdbc:derby://localhost:1527/jpaexample;create=true"/>
    <property name="javax.persistence.jdbc.user" value="app" />
    <property name="javax.persistence.jdbc.password" value="app" />
    <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>
    <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
  </properties>
</persistence-unit>
</persistence>
```

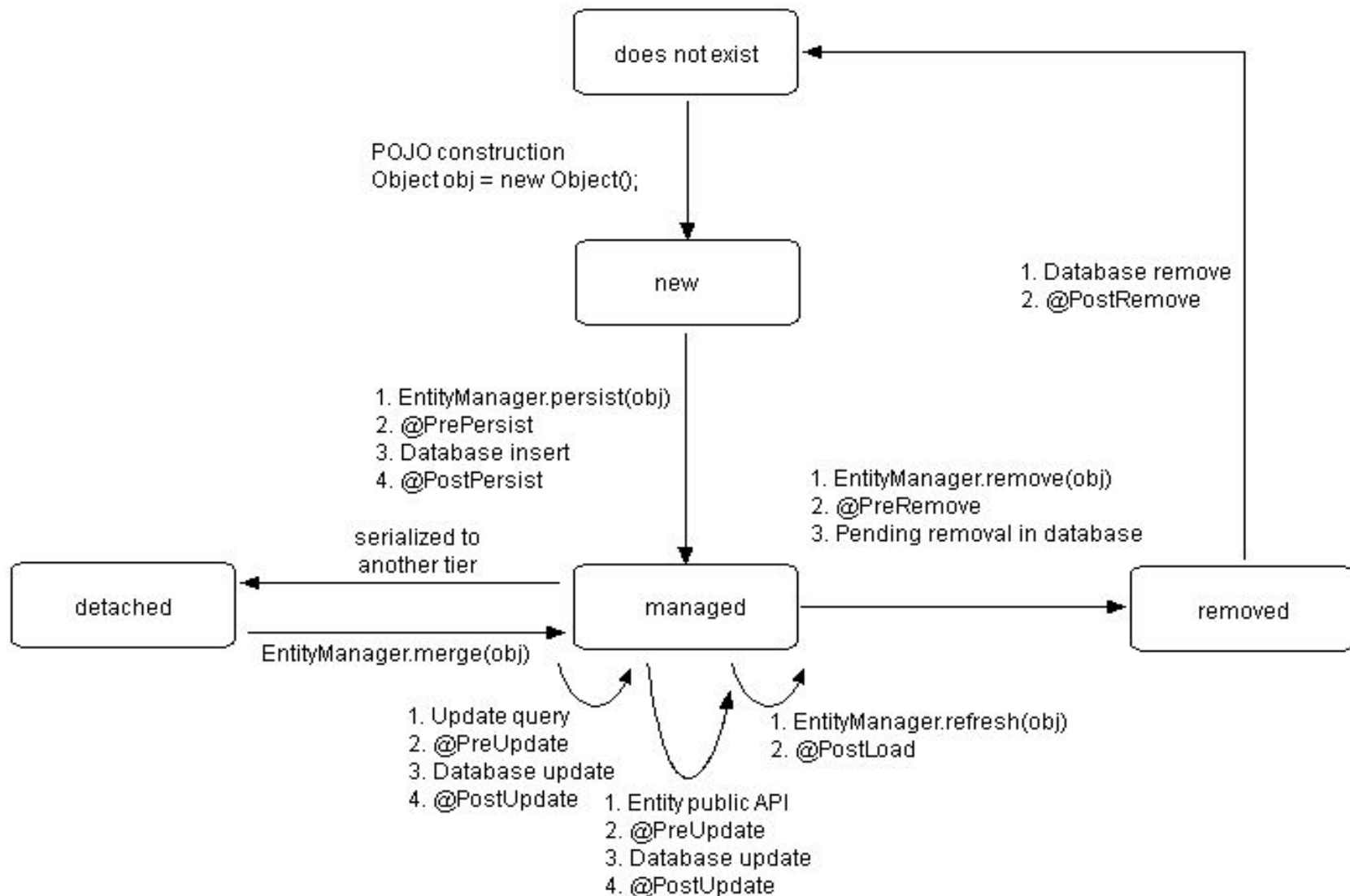
```
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("mya
pp");
EntityManager em =
emf.createEntityManager();
```


EntityManager lifecycle



... T ... Systems ...

EntityManager lifecycle annatation



Persistent Fields and Properties in Entity Classes

Состояние объекта храниться в полях и свойствах объекта, которые могут быть:

- Java примитивы
- `java.lang.String`
- Реализующие `serializable types`, including:
 - Врaперы примитивов
 - `java.util.Date`
 - `java.math.BigDecimal`
 - ...
- Эnumераторы
- Другие энтити или коллекции из энтитей
- `Embeddable` классы

Persistent Properties

Сигнатура доступа к персистентному полю должна быть следующей:
`getProperty()` `void setProperty(Type type)`

Использование коллекций в качестве персистентных полей.

Могут использоваться следующие интерфейсы коллекций:

- `java.util.Collection`
- `java.util.Set`
- `java.util.List`
- `java.util.Map`

Validating Persistent Fields and Properties

Для персистентных полей может использоваться Bean Validation API который предоставляет механизм верификации данных энити.

@AssertFalse boolean isUnsupported; **@AssertTrue** boolean isActive;
@DecimalMax("30.00") BigDecimal discount; **@DecimalMin("5.00")** BigDecimal discount; **@Digits(integer=6, fraction=2)** BigDecimal price; **@Future** Date eventDate;
@Max(10) int quantity; **@Min(5)** int quantity; **@NotNull** String username; **@Null** String unusedString; **@Past** Date birthday; **Pattern**(regexp="\\(\\d{3}\\)\\d{3}-\\d{4}") String phoneNumber; **@Size**(min=2, max=240) String briefMessage;

@Entity @Table @Column

@Entity

@Table(uniqueConstraints = @UniqueConstraint(columnNames = { "NAME", "SURNAME" })), name = "EMPLOYEE")

public class Employee {

 @Id

private long id;

 @Column(name = "NAME", nullable = **false**, **length = 20**)

private String name;

 @Column(name = "SURNAME", nullable = **false**, **length = 20**)

private String surname;

 ...

}

Primary Keys in Entities

Каждая Entity должна иметь уникальный идентификатор (первичный ключ), например:

Простой первичный ключ

@Id

private long id;

Первичный ключ может быть: примитивный тип, wrapper примитивного типа, String, Date, BigDecimal, BigInteger.

Композитный первичный ключ (ProjectId.class).

Embedded первичный ключ ()

должен переопределять hashCode() и equals(Object other), реализовывать Serializable.

Entity Relationships

Существуют следующие отношения между энтити в JPA

One-to-one: при этом каждый экземпляр одного класса энтити имеет ссылку на экземпля инстанса другой энтити. `@OneToOne`

One-to-many: при этом инстанс одной энтити имеет ссылку на коллекцию инстансов другой энтити. `@OneToMany`

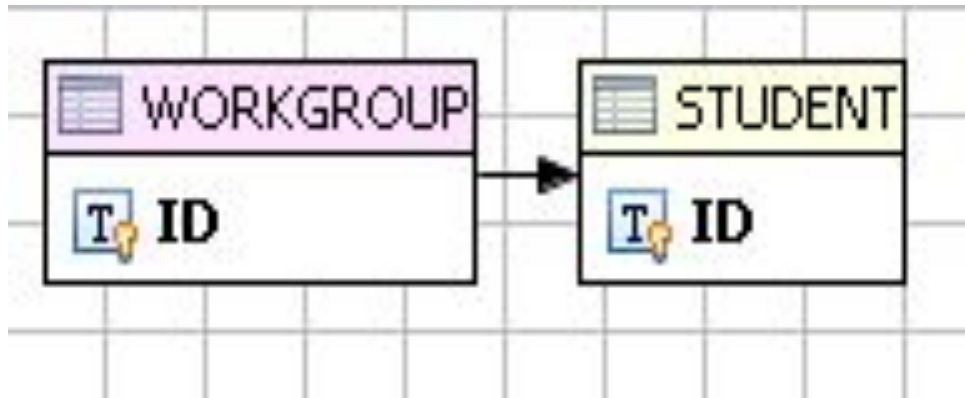
Many-to-one: несколько инстансов одной энтити имеют ссылку на один инстанс другой. `@ManyToOne`

Many-to-many: несколько инстансов одной энтити имеют ссылки на несколько инстансов другой. `@ManyToMany`

Отношения могут быть однонаправленными и двунаправленными:

Двунаправленные связи для `@OneToOne`, `@OneToMany`, `@ManyToMany` должны использовать `mappedBy` для ссылки на поле обратное поле.

One-to-many и Many-to-many:



@ManyToOne

```
private WorkGroup workGroup;
```

@OneToMany(mappedBy = "workGroup")

```
private List<Student> student;
```

Отношения и каскадные операции

| Каскадная операция | описание |
|--------------------|--|
| ALL | Все каскадные операции будут применены к энтити. Равносильно <code>cascade={DETACH, MERGE, PERSIST, REFRESH, REMOVE}</code> |
| DETACH | Если родительская энтити перейдет в состояние <code>detached</code> , то связанные с ней энтити так же перейдут в это состояние. |
| MERGE | Если над родительской энтити будет произведена операция <code>merge()</code> – то то же будет и для связанных с ней энтити |
| PERSIST | аналогично |
| REFRESH | аналогично |
| REMOVE | аналогично |

```
//@OneToMany(mappedBy = "workGroup")  
//@OneToMany(mappedBy = "workGroup", cascade =  
CascadeType.REMOVE)  
@OneToMany(mappedBy = "workGroup", fetch =  
FetchType.EAGER, orphanRemoval = true)  
private List<Student> student;
```

@OneToOne

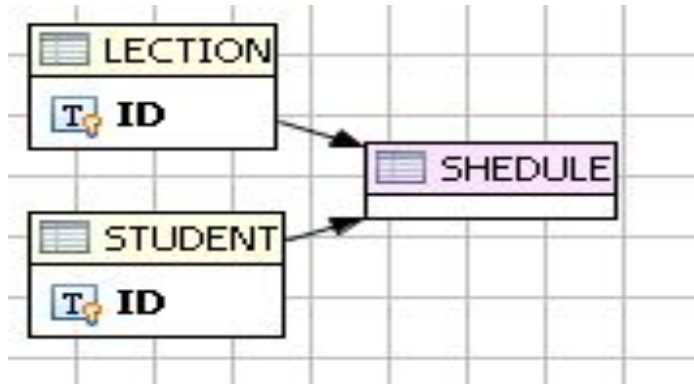
@Entity

```
public class Address {  
    @OneToOne  
    @PrimaryKeyJoinColumn  
    private Student student;  
}
```

@Entity

```
public class Student {  
    @OneToOne(mappedBy = "student")  
    private Address address;  
}
```

@ManyToMany



```
class Lektion {  
    @ManyToMany(mappedBy="lections")  
    private List<Student> students;
```

```
class Student {  
    @ManyToMany(cascade = {CascadeType.ALL})  
    @JoinTable(name="SCHEDULE", joinColumns={@JoinColumn(name="STUDENTID")},  
    inverseJoinColumns={@JoinColumn(name="LECTIONID")})  
    private List<Lektion> lections;
```

@ElementCollection

@Entity

public class Person {

//can be used to define a one-to-many relationship to an Embeddable object, or a Basic value (such
//as a collection of Strings) and can also be used in combination with a Map

 @ElementCollection

 @MapKeyEnumerated

private Map<PhoneType ,Phone> phones;

@Embeddable

public class Phone {

 @Enumerated(EnumType.STRING)

private PhoneType type;

private String areaCode;

 @Column(name = "P_NUMBER")

private String number;

public enum PhoneType { *HOME, WORK, MOBILE ...*

Embeddable Classes

@Entity

```
public class Employee {  
    @Embedded  
    private Skill skill;  
}
```

@Embeddable

```
public class Skill {  
    private String skillName;  
    private String skillLevel;  
}
```

```
CREATE TABLE APP.EMPLOYEE ( ID BIGINT NOT NULL,  
NAME VARCHAR(20) NOT NULL,  
SKILLLEVEL VARCHAR(255),  
SKILLNAME VARCHAR(255),  
SURNAME VARCHAR(20) NOT NULL
```

Embeddable Classes

@Entity

```
public class Employee {  
    @Embedded  
    private Skill skill;  
}
```

@Embeddable

```
public class Skill {  
    private String skillName;  
    private String skillLevel;  
}
```

```
CREATE TABLE APP.EMPLOYEE ( ID BIGINT NOT NULL,  
NAME VARCHAR(20) NOT NULL,  
SKILLLEVEL VARCHAR(255),  
SKILLNAME VARCHAR(255),  
SURNAME VARCHAR(20) NOT NULL
```

Entity Inheritance: Abstract Entities

Абстрактная энтити может быть определена при помощи **абстрактного класса** с аннотацией **@Entity**.

Если абстрактная энтити выступает в качестве параметра запроса или связана отношением с другой энтити – то результат будет содержать все подклассы данной абстрактной энтити.

Существуют следующие **стратегии мапинга энтитей на таблицы** базы данных при наследовании:

- в одной таблице хранятся все подклассы (**InheritanceType. SINGLE_TABLE**)
- каждый подкласс в своей таблице (**InheritanceType. TABLE_PER_CLASS**)
- стратегия “join”, когда каждый подкласс мапится на таблицы (включая абстрактный), при этом в каждой таблице содержится первичный ключ из базового класса и только специфичные для данного класса поля (**InheritanceType. JOINED**)

Entity Inheritance: Abstract Entities

```
@Entity
```

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

```
public abstract class Animal {
```

```
    @Id
```

```
    private String id;
```

```
    private String woolColor;
```

```
    @ManyToOne
```

```
    private Forest forest;
```

```
@Entity
```

```
public class Bear extends Animal {
```

```
    private String bearFeature;
```

```
@Entity
```

```
public class Fox extends Animal {
```

```
    private String foxFeature;
```

Entity Inheritance: InheritanceType.TABLE_PER_CLASS

@Entity

@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)

public abstract class Animal {

 @Id

private String id;

private String woolColor;

 @ManyToOne

private Forest forest;

@Entity

public class Bear extends Animal {

private String bearFeature;

@Entity

public class Fox extends Animal {

private String foxFeature;

Entity Inheritance: InheritanceType.SINGLE_TABLE

```
@Entity
```

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

```
@DiscriminatorColumn(name="PROJ_TYPE")
```

```
public abstract class ItProject {
```

```
    @Id
```

```
    private long id;
```

```
@Entity
```

```
@DiscriminatorValue("L")
```

```
public class LargeProject extends ItProject{
```

```
    private String largeFeature;
```

```
@Entity
```

```
@DiscriminatorValue("S")
```

```
public class SmallProject extends ItProject{
```

```
    private String smallFeature;
```

Entity Inheritance: InheritanceType.JOINED

```
@Entity
```

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

```
public abstract class Vehicle {
```

```
    @Id
```

```
    private long id;
```

```
    private String common;
```

```
@Entity
```

```
public class Car extends Vehicle {
```

```
    private String carFeature;
```

```
@Entity
```

```
public class Plane extends Vehicle {
```

```
    private String carFeature;
```

Mapped Superclasses

Энтити могут быть наследованны от суперкласса который сам при этом не является энтити, но содержит информацию о мапинге и персистенсе.

@MappedSuperclass

```
public class Employee {  
    @Id  
    protected Integer employeeld
```

@Entity

```
public class FullTimeEmployee extends Employee {  
    protected Integer salary
```

@Entity

```
public class PartTimeEmployee extends Employee {  
    protected Float hourlyWage
```

Java Persistence Query Language

Java Persistence Query Language (JPQL) — платформо-независимый объектно-ориентированный язык запросов являющийся частью Java Persistence API спецификации.

```
SELECT a FROM Author a ORDER BY a.firstName, a.lastName
```

```
SELECT DISTINCT a FROM Author a INNER JOIN a.books b WHERE b.publisher.name =  
'XYZ Press'
```

Динамический запрос:

Query и TypedQuery (JPA 2.0)

```
public List<Author> getAuthorsByLastName(String lastName) {  
    String queryString = "SELECT a FROM Author a " + "WHERE :lastName IS NULL OR  
    LOWER(a.lastName) = :lastName";  
    Query query = getEntityManager().createQuery(queryString);  
    query.setParameter("lastName", StringUtils.lowerCase(lastName));  
    return query.getResultList();  
}
```

@NamedQuery and @NamedQueries

Именованные запросы являются статическими.

```
@Entity
```

```
@NamedQuery(name="Country.findAll", query="SELECT c FROM Country c")
```

```
public class Country { ... }
```

```
@Entity
```

```
@NamedQueries({
```

```
    @NamedQuery(name="Country.findAll", query="SELECT c FROM Country c"),
```

```
    @NamedQuery(name="Country.findByName", query="SELECT c FROM Country c  
    WHERE c.name = :name"), })
```

```
public class Country {
```

```
TypedQuery<Country> query = em.createNamedQuery("Country.findAll", Country.class);
```

```
List<Country> results = query.getResultList();
```

JPA Criteria Query

Если JPQL queries определяются в виде строки, аналогично SQL, то **JPA criteria queries** позволяют формировать запрос в виде **Java классов**

Запрос вида:

```
SELECT c FROM Country c
```

Можно представить:

```
CriteriaBuilder cb = em.getCriteriaBuilder();  
CriteriaQuery<Country> q = cb.createQuery(Country.class);  
Root<Country> c = q.from(Country.class);  
q.select(c);
```

Выполнить:

```
TypedQuery<Country> query = em.createQuery(q);  
List<Country> results = query.getResultList();
```


Entity Locking and Concurrency

optimistic locking

@Version

protected int version;

javax.persistence.OptimisticLockException

pessimistic locking

EntityManager em = ...;

Person person = ...;

em.lock(person, LockModeType.PESSIMISTIC_WRITE);