

JPA & Hibernate



zaurtregulov@gmail.com

JPA & Hibernate

Для кого предназначен данный курс?

- Для людей, совсем ничего не знающих о JPA и Hibernate
- Для людей, которые хотят расширить знания о JPA и Hibernate и закрепить свои знания различными примерами

JPA & Hibernate

Какие знания вам необходимы для успешного прохождения курса?

- Базовые знания Java

- Базовые знания SQL

Самое главное требование:

- Ваше **ЖЕЛАНИЕ** изучить JPA и Hibernate

JPA & Hibernate

Java для Начинающих



Java - получи Чёрный Пояс



SQL для Начинающих

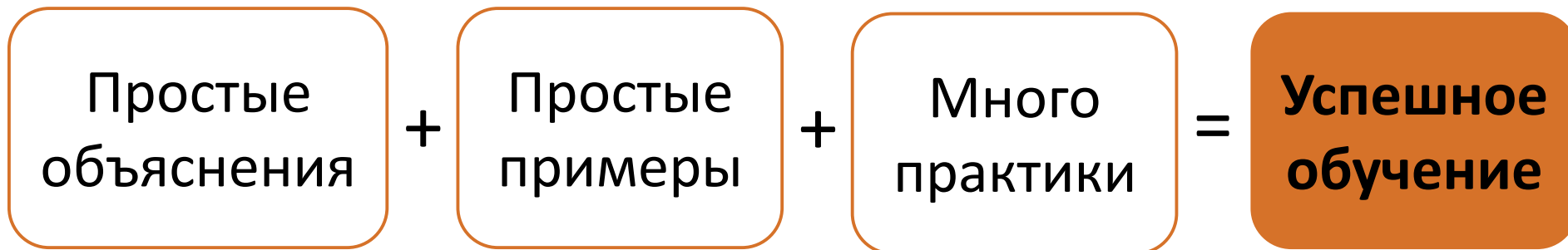


Spring для Начинающих



JPA & Hibernate

Моя методика



Особенности уроков

- Средняя продолжительность = 15 мин
- Уроки сохранили свою эффективность
- Стало ещё больше примеров

JPA & Hibernate

Введение

JPA

JDBC

Relationships

Persistence
Context

Работа с
данными

Criteria API

Hibernate

Inheritance
Mapping

Advanced
mapping

Разное

JPA & Hibernate

Как повысить эффективность
просмотра данного курса?

- Занимайтесь в удобное для вас время
- Занимайтесь комфортное количество времени
- Ведите конспекты
- Пишите код вместе со мной
- Побольше экспериментируйте
- Пересматривайте уроки
- Не делайте больших перерывов

JPA & Hibernate



Спрашивайте и Помогайте

Не забудьте оценить курс



Приятный бонус – презентация со всеми слайдами



Подписывайтесь
на YouTube канал
Программания

JPA & Hibernate

Hibernate

JPA

JDBC

ORM

Установка MySQL

- Root PWD: **jpacourse**
- User: **jpauser**
- PWD: **jpapwd**
- DB: **test_db**
- Connection: **test_connection**

JDBC

Java Database Connectivity

JDBC — это стандарт взаимодействия Java-приложений с различными СУБД

Является частью JDK с 1997 года

JDBC

<pre>public class Student {</pre>		<pre>CREATE TABLE students (</pre>
<pre> private Long id;</pre>	→	<pre> id BIGINT NOT NULL AUTO_INCREMENT,</pre>
<pre> private String name;</pre>	→	<pre> name varchar(25),</pre>
<pre> private String surname;</pre>	→	<pre> surname varchar(25),</pre>
<pre> private Double avgGrade;</pre>	→	<pre> avg_grade DOUBLE,</pre>
<pre> // some code</pre>		<pre> PRIMARY KEY (id)</pre>
<pre>}</pre>		<pre>);</pre>

CRUD

- **CREATE - команда INSERT**
- **READ - команда SELECT**
- **UPDATE - команда UPDATE**
- **DELETE - команда DELETE**

SQL injection

```
String sqlQuery = "UPDATE students set avg_grade = 7.5 WHERE name = '" + enteredName + "'";
```

Enter name

Zaur

```
UPDATE students set avg_grade = 7.5 WHERE name = 'Zaur'
```

Zaur' OR '5' = '5'

```
UPDATE students set avg_grade = 7.5 WHERE name = 'Zaur' OR '5' = '5'
```

JDBC

Connection to DB

```
static final String DB_URL = "jdbc:mysql://localhost:3306/test_db";  
static final String USER = "jpauser";  
static final String PWD = "jppwd";
```

```
Connection connection = DriverManager.getConnection(DB_URL, USER, PWD);
```

```
connection.close();
```

JDBC

SELECT

```
PreparedStatement statement = connection.prepareStatement(
    sql: "SELECT * FROM students WHERE avg_grade>?");

statement.setDouble( parameterIndex: 1, x: 7.0);

ResultSet resultSet = statement.executeQuery();

List<Student> students = new ArrayList<>();
while(resultSet.next()){
    Student student = new Student();
    student.setId(resultSet.getLong( columnLabel: "id"));
    student.setName(resultSet.getString( columnLabel: "name"));
    student.setSurname(resultSet.getString( columnLabel: "surname"));
    student.setAvgGrade(resultSet.getDouble( columnLabel: "avg_grade"));
    students.add(student);
}

statement.close();
resultSet.close();
```


JDBC

INSERT

```
PreparedStatement statement = connection.prepareStatement(  
    sql: "INSERT INTO students (name, surname, avg_grade) VALUES(?,?,?)");
```

```
statement.setString( parameterIndex: 1, student.getName());  
statement.setString( parameterIndex: 2, student.getSurname());  
statement.setDouble( parameterIndex: 3, student.getAvgGrade());
```

```
statement.executeUpdate();
```

```
statement.close();
```

JDBC

INSERT and get ID

```
PreparedStatement statement = connection.prepareStatement(
    sql: "INSERT INTO students (name, surname, avg_grade) " +
        "VALUES(?,?,?)", Statement.RETURN_GENERATED_KEYS);

statement.setString( parameterIndex: 1, student.getName());
statement.setString( parameterIndex: 2, student.getSurname());
statement.setDouble( parameterIndex: 3, student.getAvgGrade());

statement.executeUpdate();

ResultSet generatedKeys = statement.getGeneratedKeys();
    if (generatedKeys.next()) {
        student.setId(generatedKeys.getLong( columnIndex: 1));
    }

statement.close();
generatedKeys.close();
```

JDBC

UPDATE

```
PreparedStatement statement = connection.prepareStatement(  
    sql: "UPDATE students SET avg_grade = 6.5 WHERE name = ?");  
  
statement.setString( parameterIndex: 1, enteredName);  
  
statement.executeUpdate();  
  
statement.close();
```

JDBC

DELETE

```
PreparedStatement statement = connection.prepareStatement(  
    sql: "DELETE FROM students WHERE surname = ?");  
  
statement.setString( parameterIndex: 1, x: "Tregulov");  
  
statement.executeUpdate();  
  
statement.close();
```

JDBC

Минусы

- Очень много кода
- Легко допустить ошибку
- Тяжело тестировать
- Необходимы хорошие знания **SQL**
- Отсутствие синхронизации между объектом и строкой
- Возможна проблема с безопасностью
- Зависимость от СУБД, в которой хранятся данные

JPA

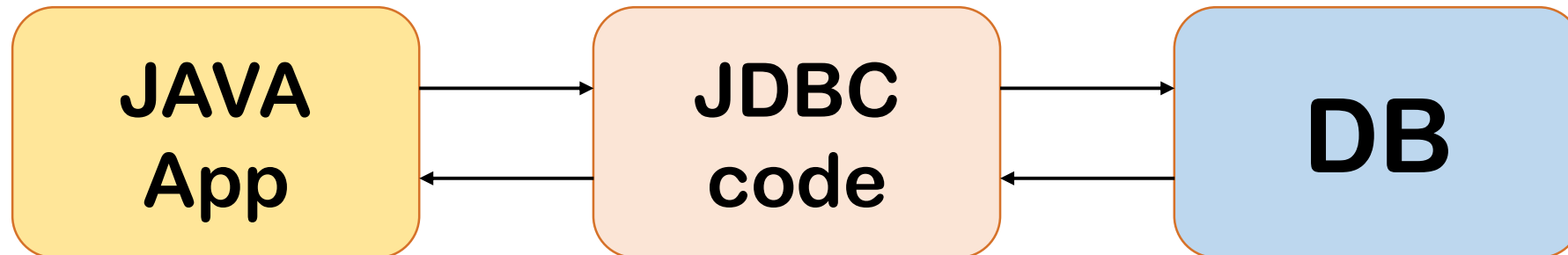
Почему мы добавили в проект
зависимость Hibernate ???

JDBC

```
Student student = new Student(  
    name: "Chane1", surname: "King", avgGrade: 9.1  
);
```



id	name	surname	avg_grade
4	Chane1	King	9.1



JPA & Hibernate

JPA – Java Persistence API

JPA – стандартная спецификация, которая описывает систему для управления Java объектами в таблицах БД-х

Hibernate – самая популярная реализация спецификации JPA

Hibernate – это framework, который используется для сохранения, получения, изменения и удаления Java объектов из БД

JPA описывает правила, а **Hibernate** их реализует

JPA & Hibernate

JPA содержит список того,
что надо делать:

- надо уметь сохранять объект в БД
- надо уметь изменять объект
- надо уметь удалять объект из БД

.....

Hibernate умеет это
делать:

- умею сохранять объект в БД
- умею изменять объект
- умею удалять объект из БД

.....

JPA & Hibernate

```
interface WhatToDo{  
    void persist();  
    void remove();  
    // specification of other methods  
}
```

```
class HowToDo implements WhatToDo{  
    public void persist(){  
        // code#1  
    }  
    public void remove(){  
        // #code2  
    }  
    //realization of other methods  
}
```

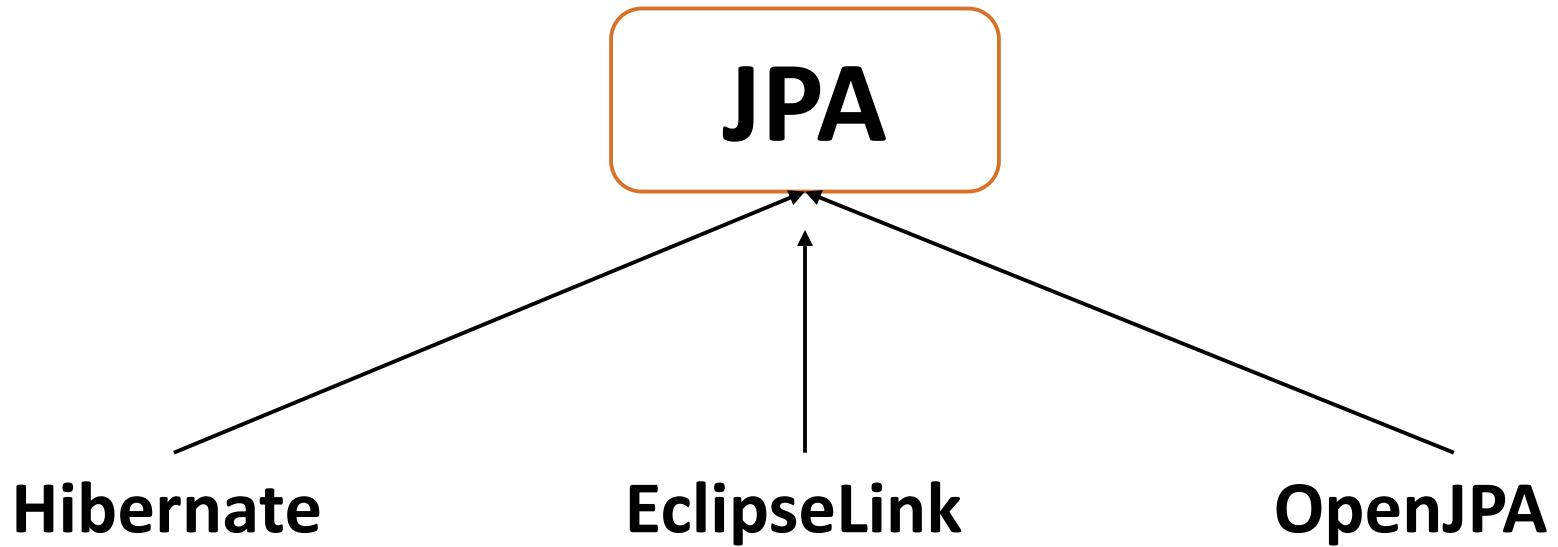
Hibernate

Hibernate

Реализация
спецификации
JPA

Функционал, не
связанный с **JPA**

JPA

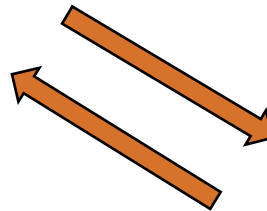


ORM





ORM – Object to Relational Mapping

ORM – это преобразование объекта в строку в таблице и обратное преобразование

```
public class Student {  
    private Long id;  
    private String name;  
    private String surname;  
    private Double avgGrade;  
  
    // other code  
}
```











students

 id ▼	 name ▼	 surname ▼	 avg_grade ▼
--	--	---	---

Mapping

```
public class Student {  
    private Long id;  
    private String name;  
    private String surname;  
    private Double avgGrade;  
  
    // other code  
}
```

students

 id   name   surname   avg_grade 

Mapping

POJO (Plain Old Java Object) – обычно простой по функциональности Java класс, имеющий private поля, getter-ы и setter-ы, конструктор без аргументов, ...

@Entity превращает обычный Java класс в Entity класс

Entity класс – это Java класс (POJO), который отображает информацию определённой таблицы в БД

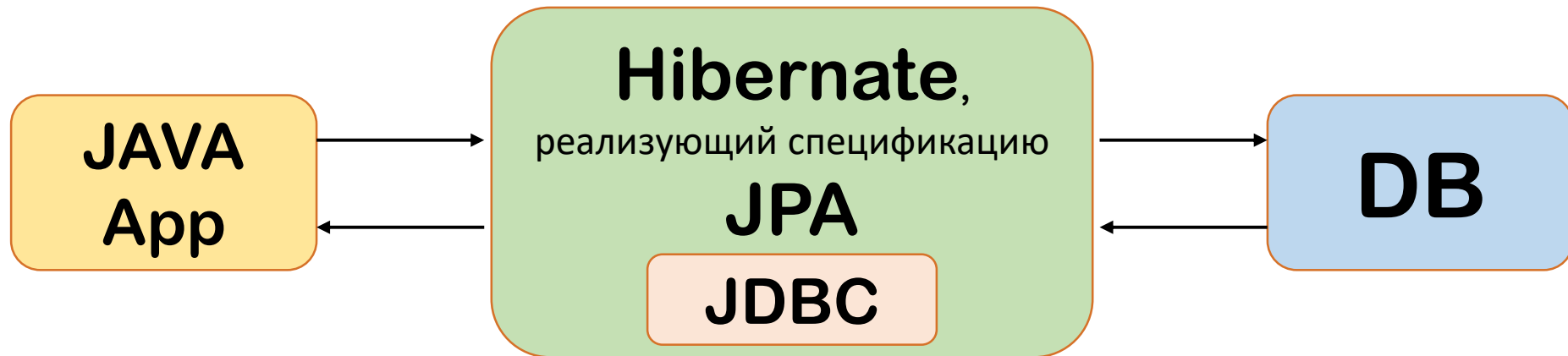
Mapping

@Table показывает, к какой именно таблице мы привязываем класс

@Column показывает, к какому именно столбцу из таблицы мы привязываем поле класса

@Id показывает, какой именно столбец в таблице является primary key

JPA & Hibernate



GenerationType

IDENTITY – стратегия, при которой Primary Key изменяется в соответствии с правилами, прописанными при создании таблицы

SEQUENCE – стратегия, основанная на генерации значений объектом Sequence, созданного в БД

TABLE – стратегия, основанная на использовании значений столбца специальной таблицы, созданной в БД

AUTO – тип по умолчанию. Выбор стратегии будет зависеть от типа БД и версии Hibernate

UUID – стратегия, основанная на использовании UUID, которые генерируются Java приложением.

GenerationType

id
3

UUID – Universally Unique ID

students

id	name	surname	avg_grade
1
2
3
...

university

id	name	founding_date
1
2
...

JPA

persistence.xml

```
<persistence-unit name="jpa-course">
  <properties>
    <property name="jakarta.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
    <property name="jakarta.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/test_db" />
    <property name="jakarta.persistence.jdbc.user" value="jpauser" />
    <property name="jakarta.persistence.jdbc.password" value="jpapwd" />
    <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />-->
    <property name="hibernate.show_sql" value="true" />
    <property name="hibernate.format_sql" value="true" />
    <property name="hibernate.hbm2ddl.auto" value="update"/> // create create-drop
  </properties>
</persistence-unit>
```

JPA

mapping

```
import jakarta.persistence.*;

@Entity
@Table(name = "students")
public class Student {

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

    @Column(name="name", nullable = false, unique = true)
    private String name;

    @Transient
    private LocalDateTime createTime;
```

JPA

INSERT

```
EntityManagerFactory factory = Persistence
    .createEntityManagerFactory(persistenceUnitName: "jpa-course");
EntityManager entityManager = factory.createEntityManager();

EntityTransaction transaction = entityManager.getTransaction();

    transaction.begin();

    entityManager.persist(student);

    transaction.commit();

entityManager.close();
factory.close();
```

JPA

SELECT

```
EntityManagerFactory factory = Persistence
    .createEntityManagerFactory(persistenceUnitName: "jpa-course");
EntityManager entityManager = factory.createEntityManager();

Student st = entityManager.find(Student.class, o: 5);

entityManager.close();
factory.close();
```

JPA

UPDATE

```
EntityManagerFactory factory = Persistence
    .createEntityManagerFactory(persistenceUnitName: "jpa-course");
EntityManager entityManager = factory.createEntityManager();

EntityTransaction transaction = entityManager.getTransaction();

transaction.begin();

Student st = entityManager.find(Student.class, 5);

st.setAvgGrade(9.8);

transaction.commit();

entityManager.close();
factory.close();
```


JPA

DELETE

```
EntityManagerFactory factory = Persistence
    .createEntityManagerFactory(persistenceUnitName: "jpa-course");
EntityManager entityManager = factory.createEntityManager();

EntityTransaction transaction = entityManager.getTransaction();

transaction.begin();

Student st = entityManager.find(Student.class, 5);

entityManager.remove(st);

transaction.commit();

entityManager.close();
factory.close();
```

JPA

Плюсы

- Удобная настройка подключения к БД
- Лёгкий процесс знакомства класса с таблицей
- Немного кода
- Можно обойтись без использования **SQL**
- Присутствует синхронизация между объектом и строкой
- Независимость от СУБД, в которой хранятся данные

Table relationships

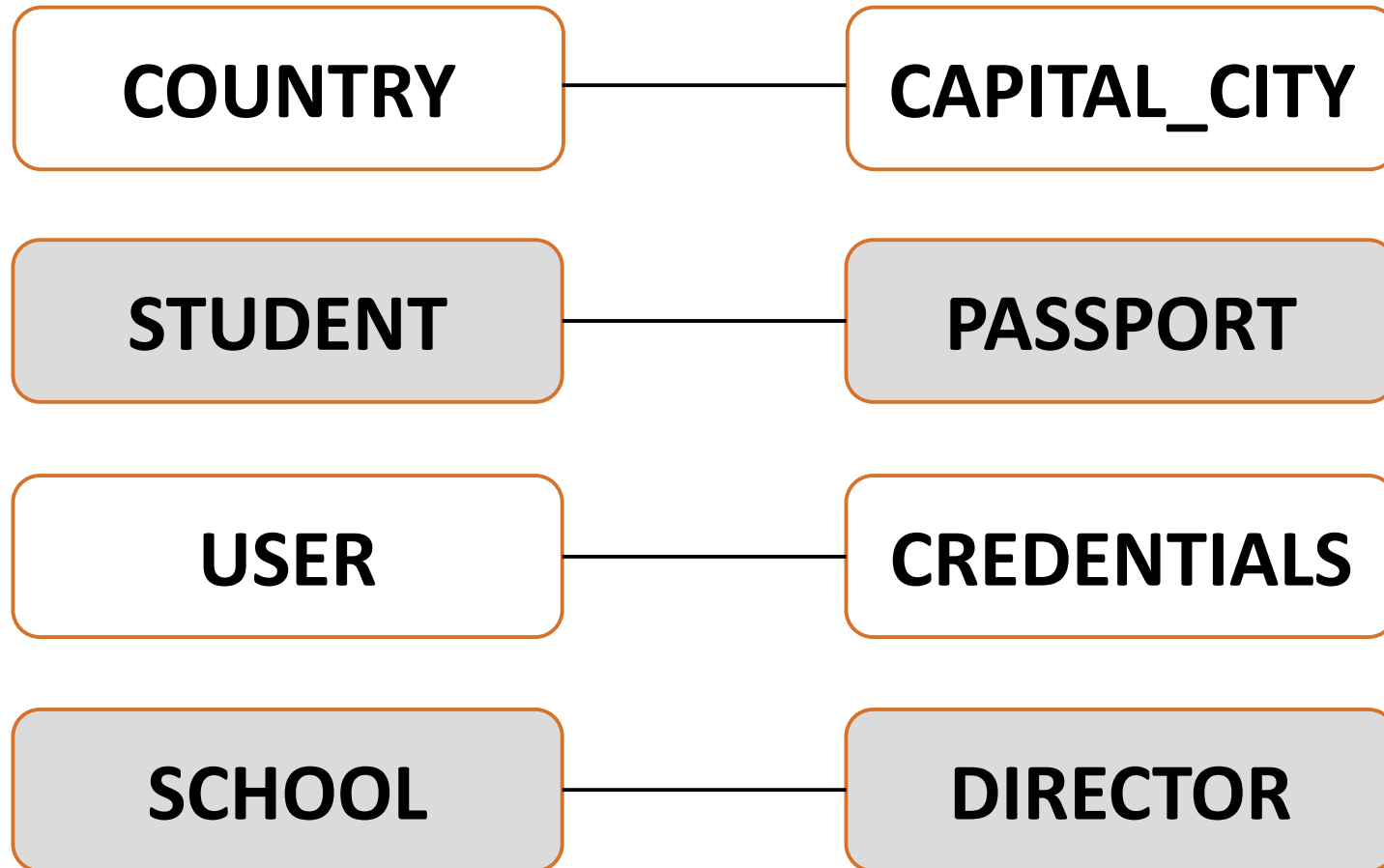
One-to-One

One-to-Many

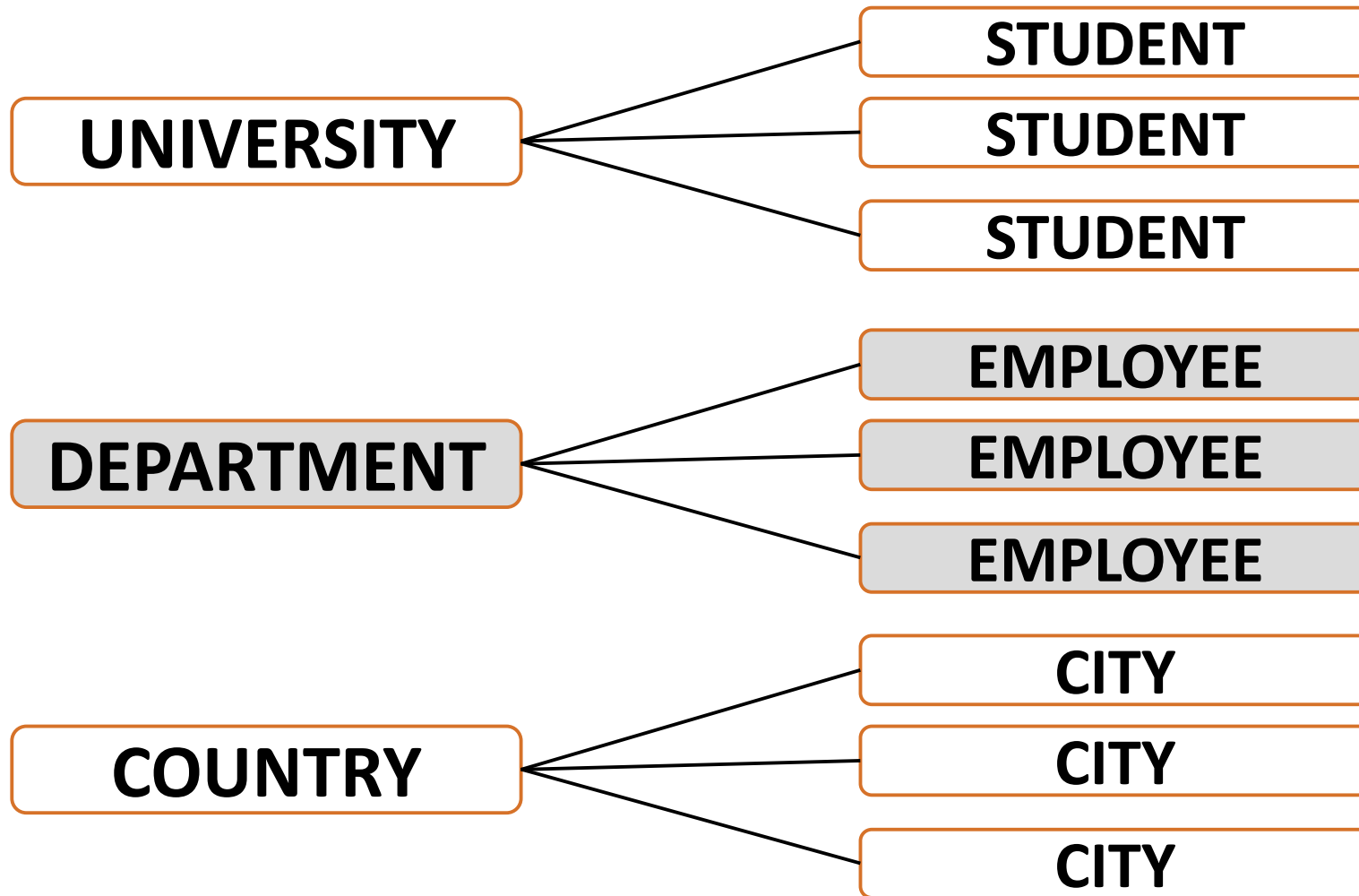
Many-to-One

Many-to-Many

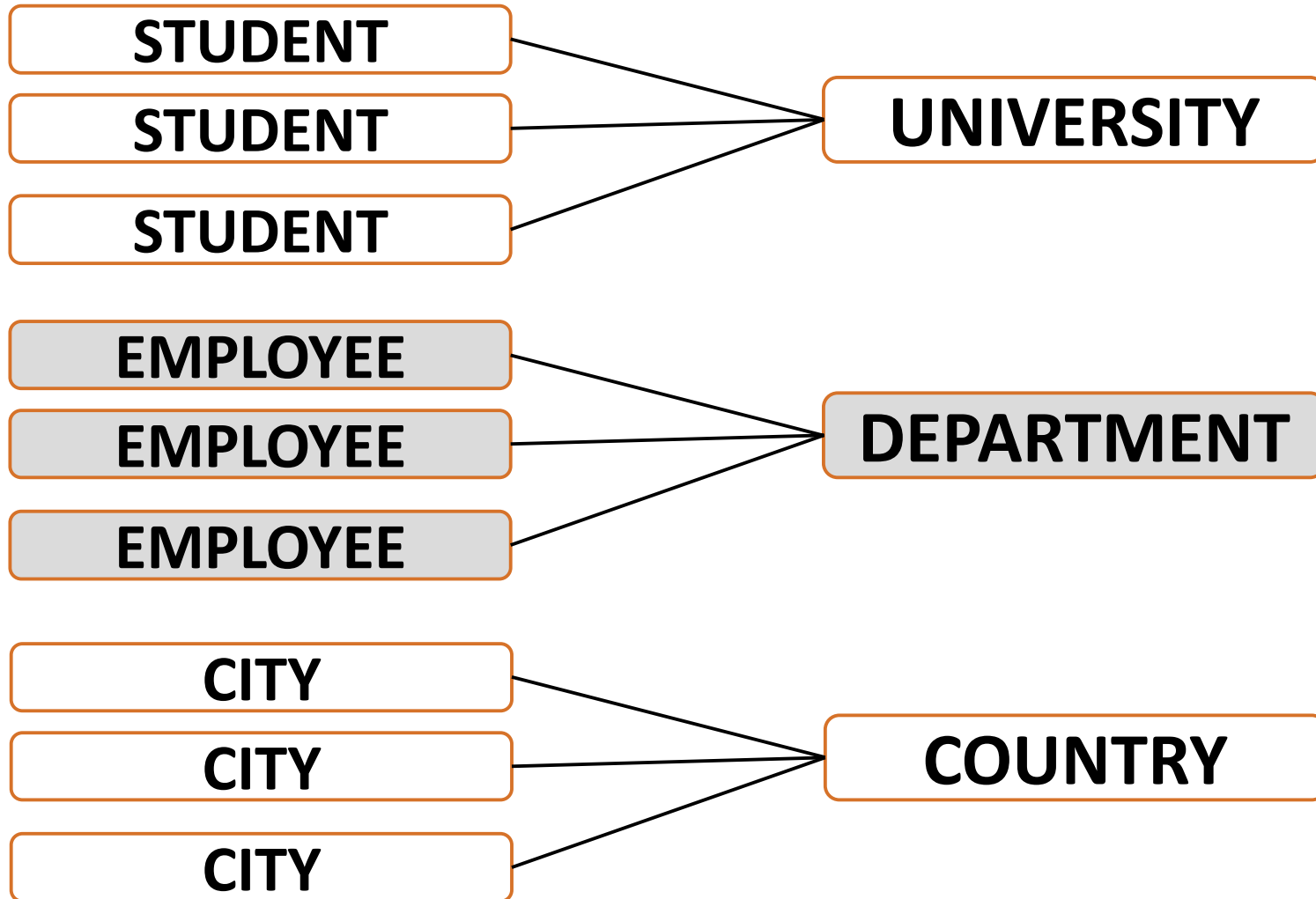
One-to-One relationship



One-to-Many relationship



Many-to-One relationship



Many-to-Many relationship

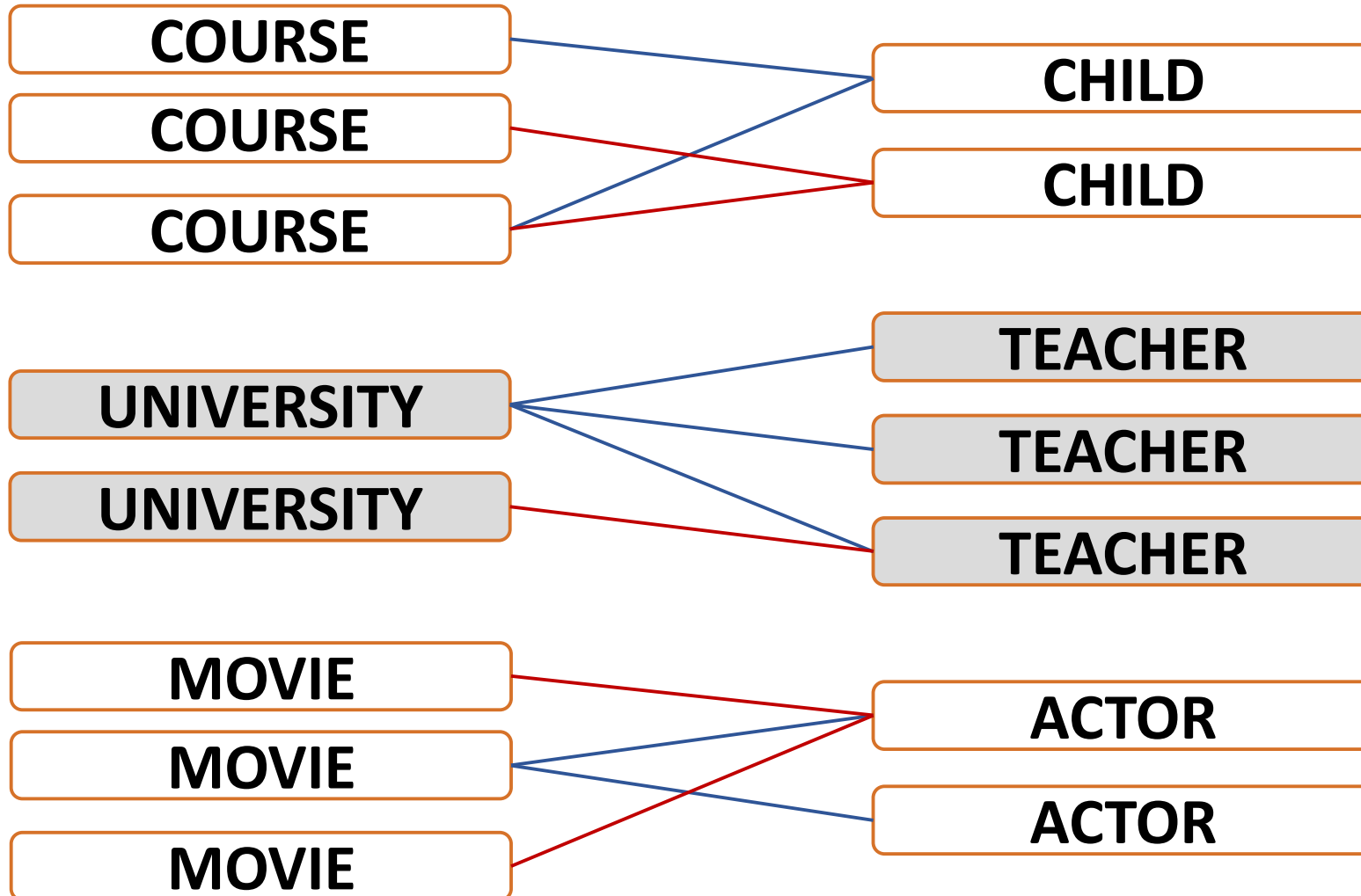


Table relationships

students
id BIGINT
name VARCHAR(25)
surname VARCHAR(25)
avg_grade DOUBLE

passports
id BIGINT
email VARCHAR(50)
height INT
eye_color VARCHAR(25)

id	name	surname	avg_grade
1	Chanel	King	9.1
2	Leo	Farrell	8.4
3	Julia	Dean	8.7

id	email	height	eye_color
10	chanel.king@gmail.com	174	blue
11	leo.farrell@yahoo.com	178	black
12	julia.dean@gmail.com	168	green

Foreign Key

Foreign Key служит для создания связи между двумя таблицами

Как правило, столбец с **Foreign Key** ссылается на столбец с **Primary Key** другой таблицы

Столбец с Foreign Key может содержать только те данные, которые есть в столбце, на который он ссылается

Table relationships

```
CREATE TABLE test_db.students (  
  id BIGINT NOT NULL AUTO_INCREMENT,  
  name varchar(25),  
  surname varchar(25),  
  avg_grade DOUBLE,  
  passport_id BIGINT,  
  PRIMARY KEY (id),  
  FOREIGN KEY (passport_id) references test_db.passports(id)  
);
```

students
id BIGINT
name VARCHAR(25)
surname VARCHAR(25)
avg_grade DOUBLE
passport_id BIGINT

passports
id BIGINT
email VARCHAR(50)
height INT
eye_color VARCHAR(25)

id	name	surname	avg_grade	passport_id
1	Chanel	King	9.1	10
2	Leo	Farrell	8.4	11
3	Julia	Dean	8.7	15

id	email	height	eye_color
10	chanel.king@gmail.com	174	blue
11	leo.farrell@yahoo.com	178	black
12	julia.dean@gmail.com	168	green

One-to-One

COUNTRY

CAPITAL_CITY

STUDENT

PASSPORT

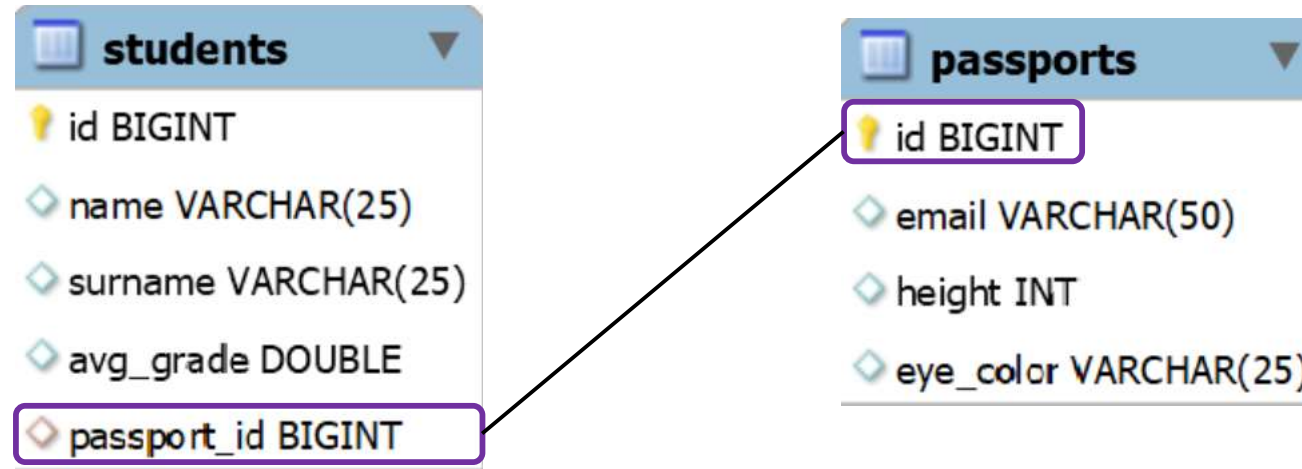
USER

CREDENTIALS

SCHOOL

DIRECTOR

One-to-One



id	name	surname	avg_grade	passport_id
1	Chanel	King	9.1	10
2	Leo	Farrell	8.4	11
3	Julia	Dean	8.7	15

id	email	height	eye_color
10	chanel.king@gmail.com	174	blue
11	leo.farrell@yahoo.com	178	black
12	julia.dean@gmail.com	168	green

Associations

```
class Student {  
    Passport p;  
}  
  
class Passport {  
}
```

Uni-directional
(однонаправленная)
ассоциация – это
отношения, когда о них
знает только одна сторона

```
class Student {  
    Passport p;  
}  
  
class Passport {  
    Student s;  
}
```

Bi-directional
(двунаправленная)
ассоциация – это
отношения, когда о них
знают обе стороны

One-to-One

```
@OneToOne  
@JoinColumn(name = "passport_id")  
private Passport passport;
```

@OneToOne показывает нам тип отношений между объектами

@JoinColumn показывает нам столбец, с помощью которого осуществляется связь с другим объектом

One-to-One

```
@OneToOne(cascade = CascadeType.ALL)
```

Cascade операций – это выполнение операции не только для Entity, на котором операция вызывается, но и на связанных с ним Entity

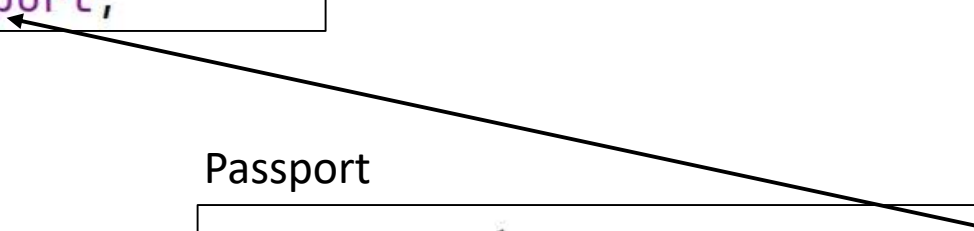
One-to-One Bi

Student

```
@OneToOne  
@JoinColumn(name = "passport_id")  
private Passport passport;
```

Passport

```
@OneToOne(mappedBy = "passport")  
private Student student;
```

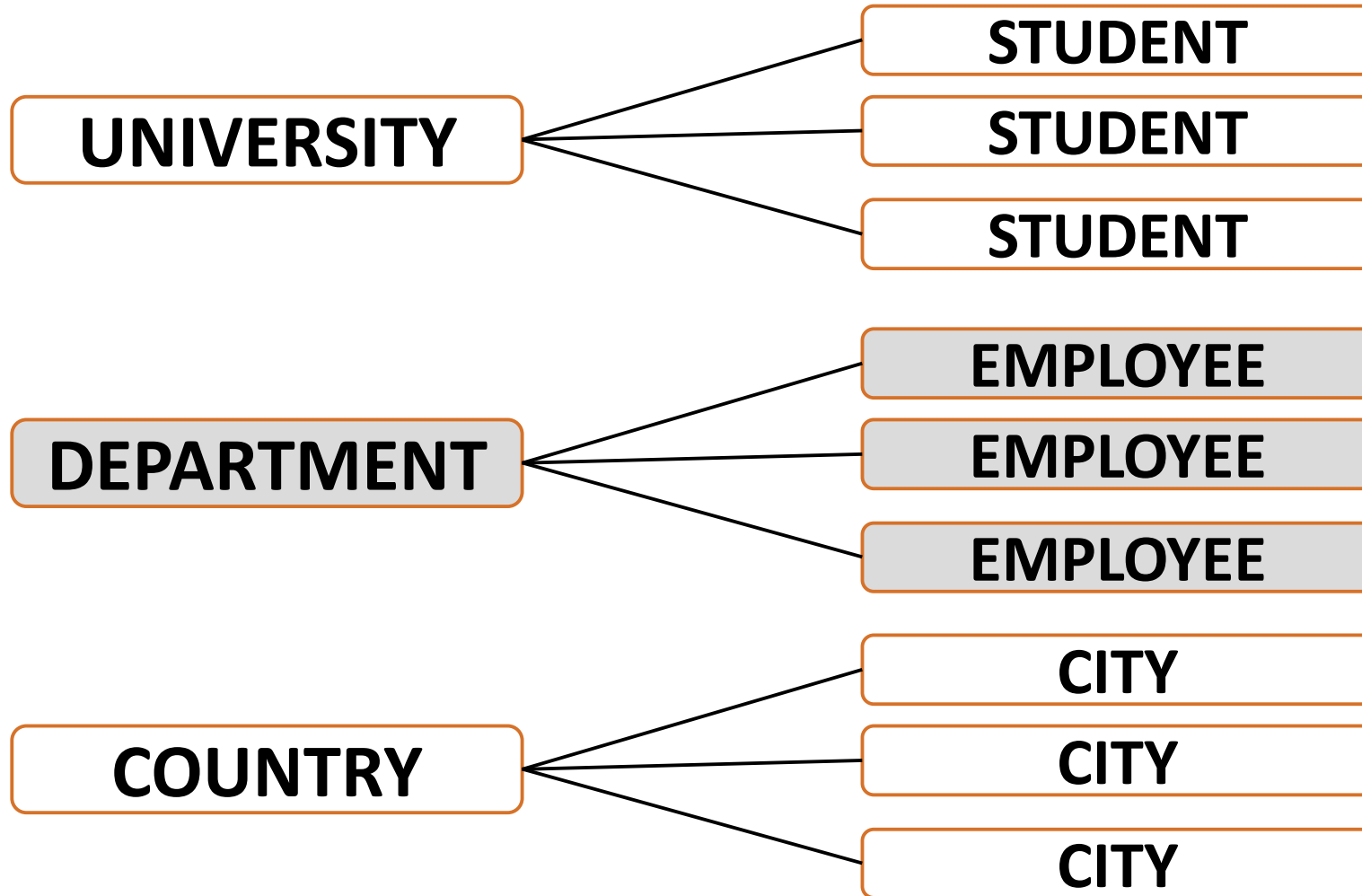


Enumerated

```
public enum EyeColor {  
    BLACK, BLUE, GREEN, BROWN  
}
```

```
@Column(name="eye_color")  
@Enumerated(EnumType.STRING)  
private EyeColor eyeColor;
```

One-to-Many relationship



One-to-Many

id	name	surname	avg_grade	university_id
1	Chanel	King	9.1	3
2	Leo	Farrell	8.4	1
3	Julia	Dean	8.7	4

id	name	founding_date	student_id
1	Harvard	28.10.1636	1, 3
2	MIT	10.04.1861	
3	Oxford	01.09.1200	

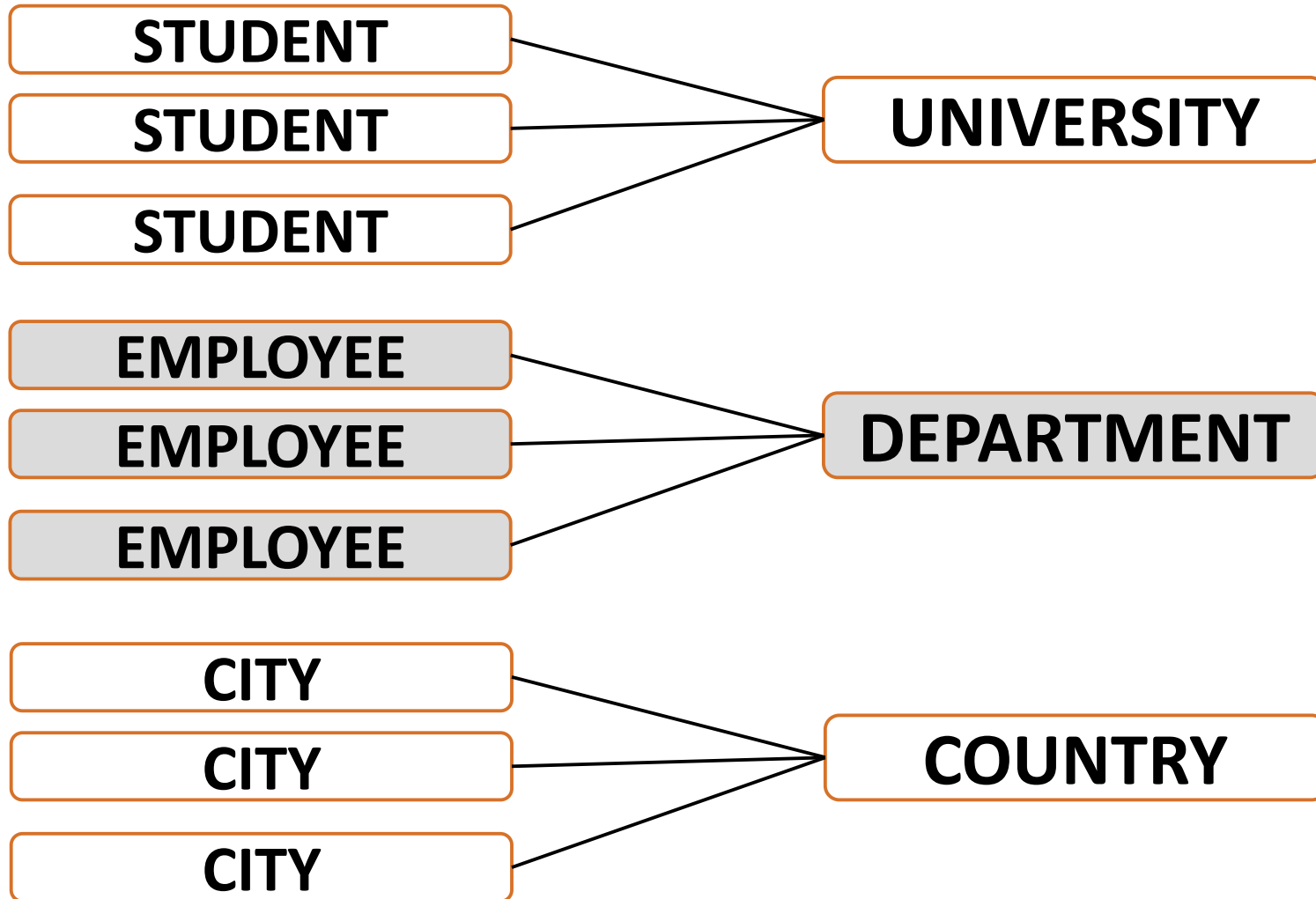
One-to-Many

```
@OneToMany  
@JoinColumn(name = "university_id")  
private List<Student> students;
```

@OneToMany показывает нам тип отношений между объектами

@JoinColumn показывает нам столбец, с помощью которого осуществляется связь с другим объектом

Many-to-One relationship



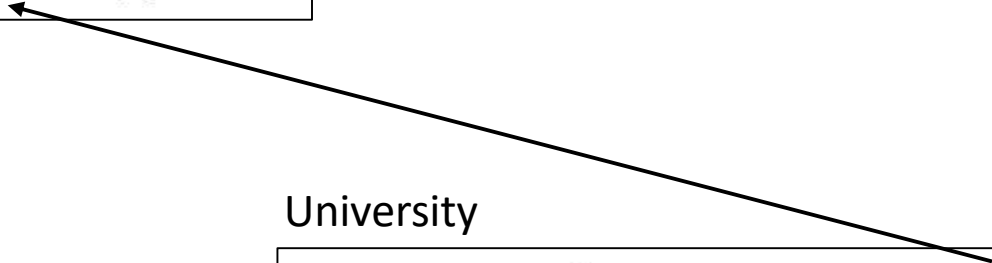
Many-to-One Bi

Student

```
@ManyToOne  
@JoinColumn(name = "university_id")  
private University university;
```

University

```
@OneToMany(mappedBy = "university")  
private List<Student> students;
```



@ManyToOne показывает нам тип отношений между объектами

OrderBy

```
@OneToMany(mappedBy = "university"  
            ,cascade = CascadeType.PERSIST)  
@OrderBy("avgGrade, name DESC")  
private List<Student> students = new ArrayList<>();
```

@OrderBy используется для сортировки
возвращаемого результата

Loading Types

Eager – это нетерпеливая загрузка, при которой связанные сущности загружаются сразу вместе с загрузкой основной сущности

Lazy – это ленивая загрузка, при которой связанные сущности **НЕ** загружаются сразу вместе с загрузкой основной сущности. Связанные сущности загружаются только при первом обращении к ним

Default Fetch Types

Relationship	Loading
One-to-One	Eager
One-to-Many	Lazy
Many-to-One	Eager
Many-to-Many	Lazy

Loading Types

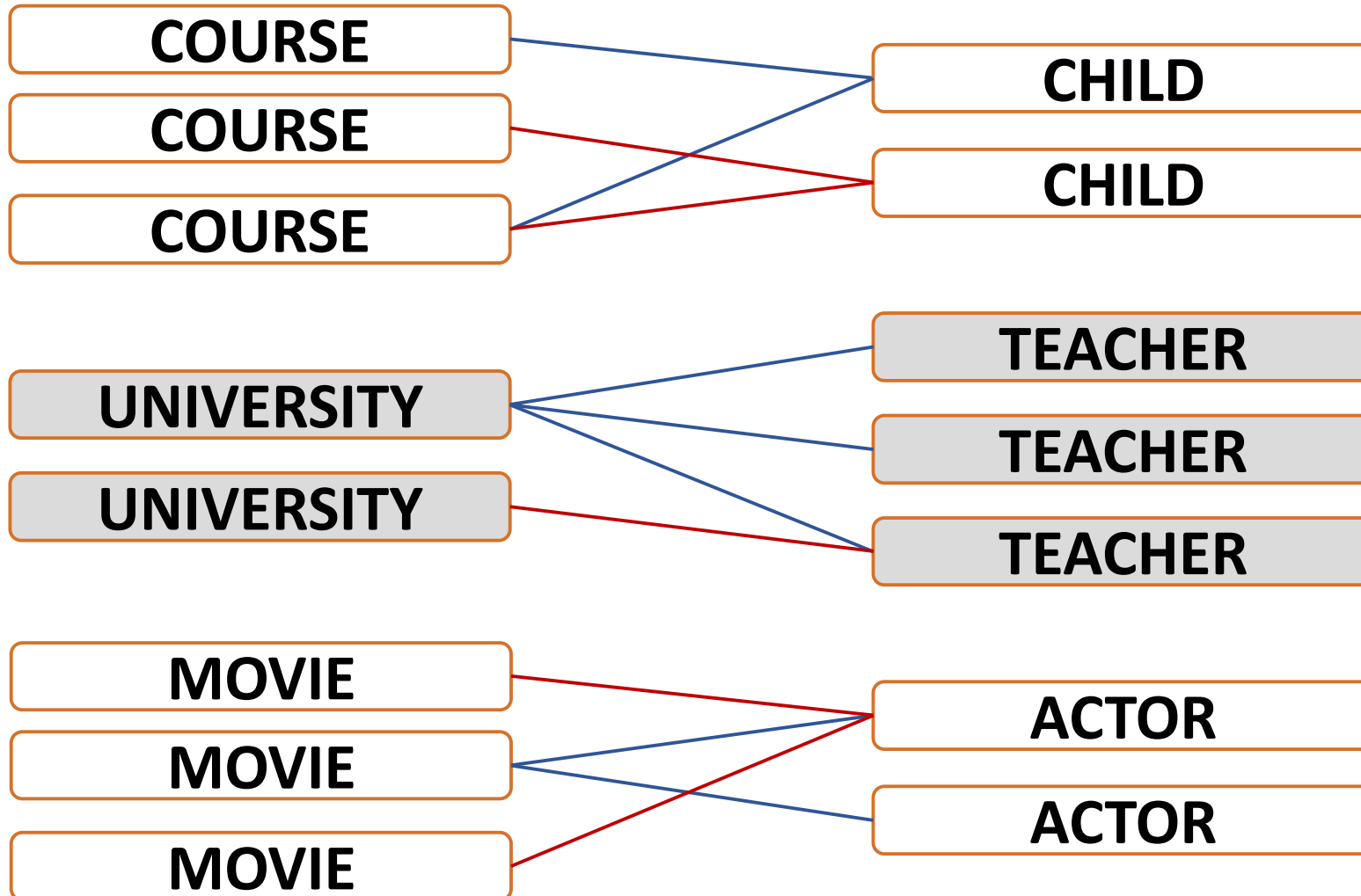
EAGER Loading

```
@OneToMany(mappedBy = "university"  
    , fetch = FetchType.EAGER)  
private List<Student> students = new ArrayList<>();
```

LAZY Loading

```
@ManyToOne(fetch = FetchType.LAZY)  
@JoinColumn(name= "university_id")  
private University university;
```

Many-to-Many



Many-to-Many

teachers

id	name	surname	subject	is_professor	university_id
1	Alessandro	Lozano	CS	TRUE	1, 3
2	Rio	Berger	Biology	FALSE	
3	Landry	Shelton	Math	TRUE	

universities

id	name	founding_date	teacher_id
1	Harvard	28.10.1636	2, 3
2	MIT	10.04.1861	
3	Oxford	01.09.1200	

teacher_uni

teacher_id	university_id
1	1
1	3
2	1
3	1

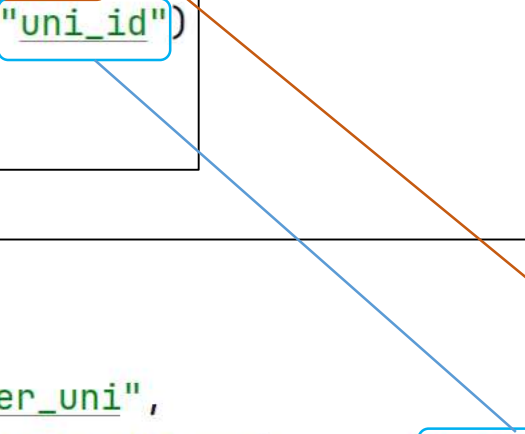
Many-to-Many

Join Table – это таблица, которая отображает связь между строками двух других таблиц

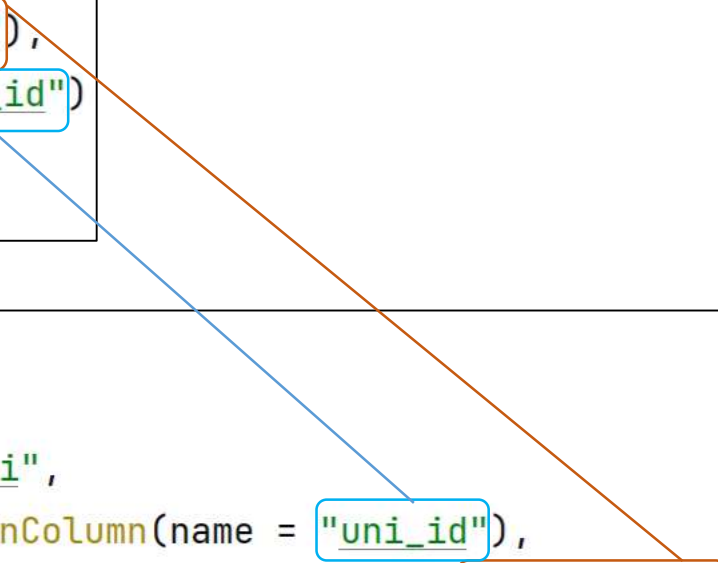
Столбцы **Join Table** – это Foreign Key, которые ссылаются на Primary Key связываемых таблиц

Many-to-Many

```
@ManyToMany
@JoinTable(
    name = "teacher_uni",
    joinColumns = @JoinColumn(name = "teacher_id"),
    inverseJoinColumns = @JoinColumn(name = "uni_id")
)
List<University> universities;
```



```
@ManyToMany
@JoinTable(
    name = "teacher_uni",
    joinColumns = @JoinColumn(name = "uni_id"),
    inverseJoinColumns = @JoinColumn(name = "teacher_id")
)
private List<Teacher> teachers;
```



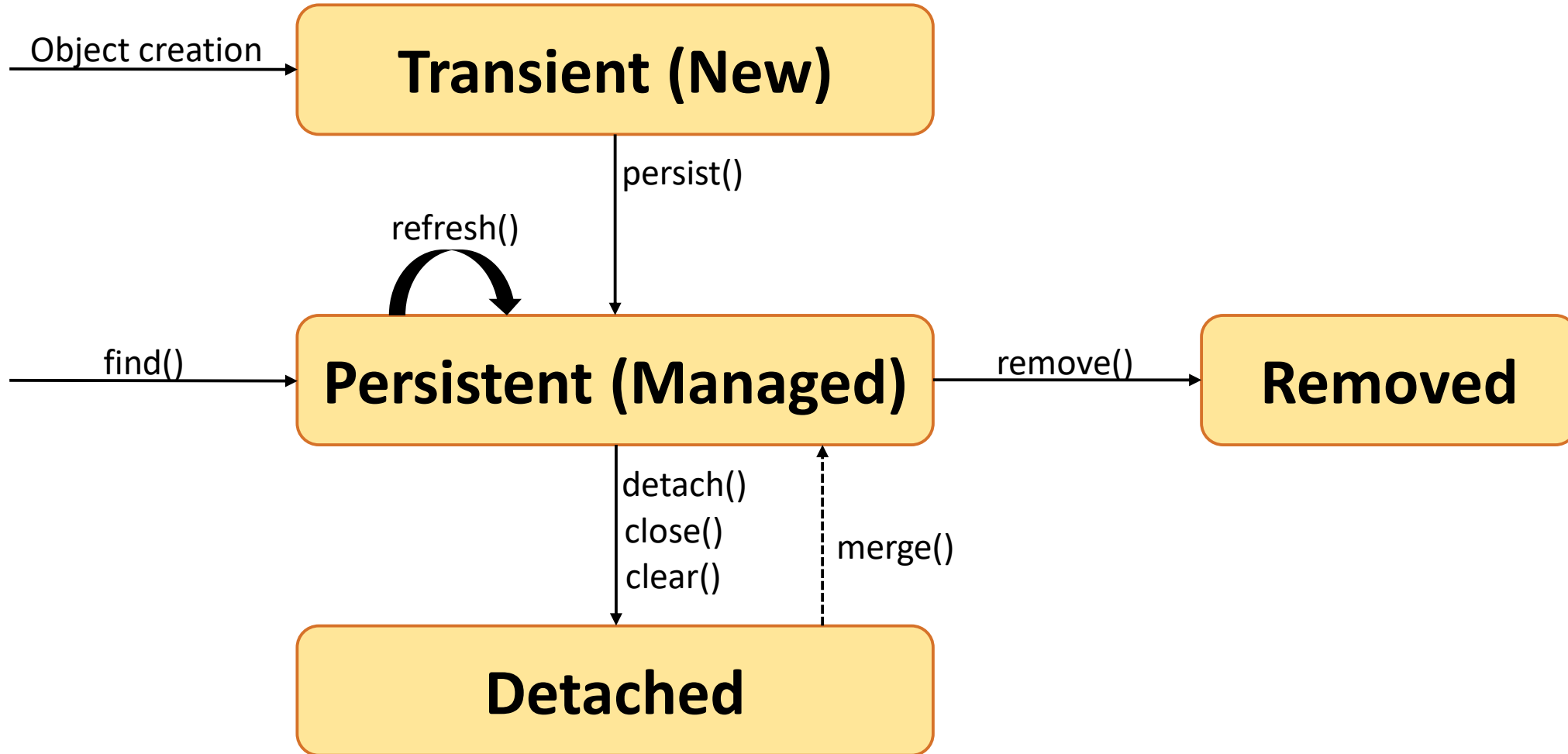
Many-to-Many

@ManyToMany показывает нам тип отношений между объектами

В **@JoinTable** мы прописываем

- название таблицы Join Table
- названия столбцов таблицы Join Table, которые ссылаются на Primary Key столбцы таблиц, находящихся в отношении Many-to-Many

Entity States



Persistence Context

Persistence Context – это эффективный управленец нашими entities, которые в нём содержатся.

Persistence Context – это сервис/место, где помнят все модификации entities, а также изменения их статусов.

Entity Manager – это интерфейс/представитель, посредством которого мы взаимодействуем с **Persistence Context**

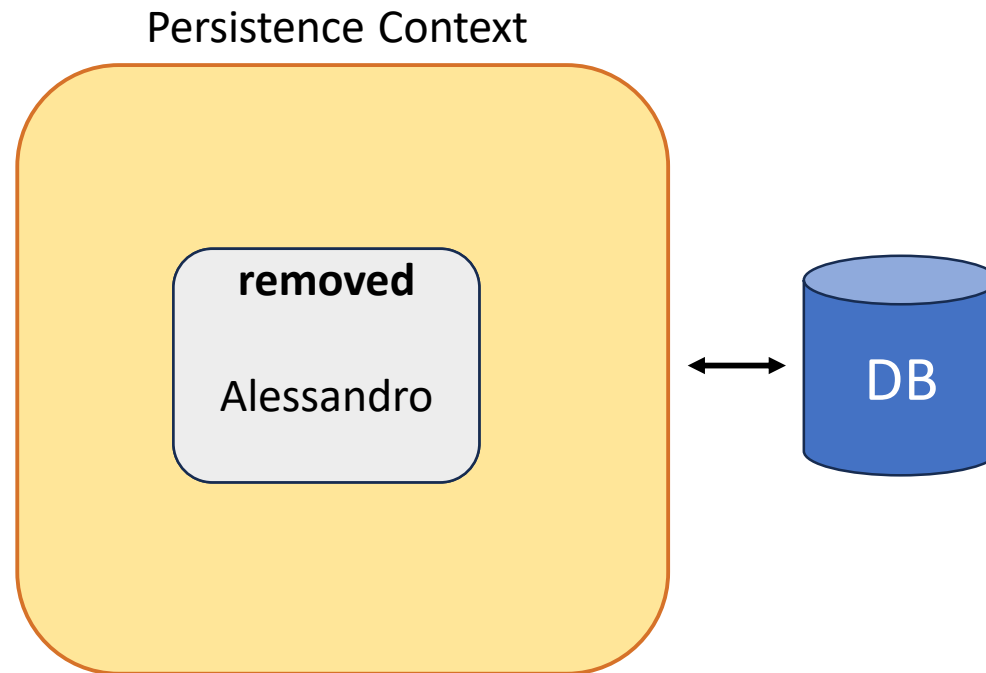
Persistence Context

```
Teacher teacher = new Teacher("Alessandro", "Lozano", "CS", true);  
EntityManager entityManager=factory.createEntityManager();  
entityManager.persist(teacher);  
transaction.commit();  
entityManager.close();
```

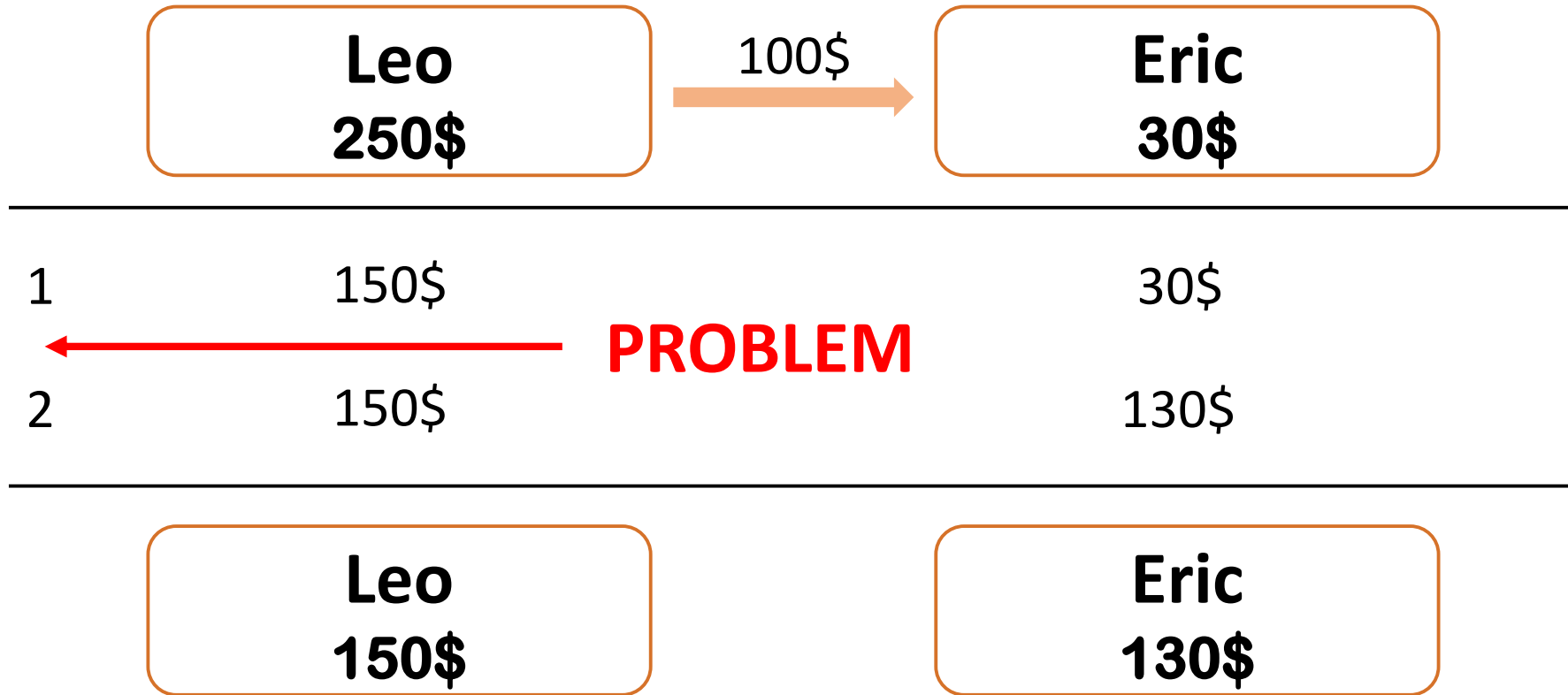


Persistence Context

```
EntityManager entityManager=factory.createEntityManager();  
Teacher teacher = entityManager.find(Teacher.class, 1);  
entityManager.remove(teacher);  
transaction.commit();  
entityManager.close();
```



Transaction



Transaction

Транзакция – это группа операций по работе с DB, которые объединены в цельный юнит

Всё или ничего!

Persistence Context

Функции:

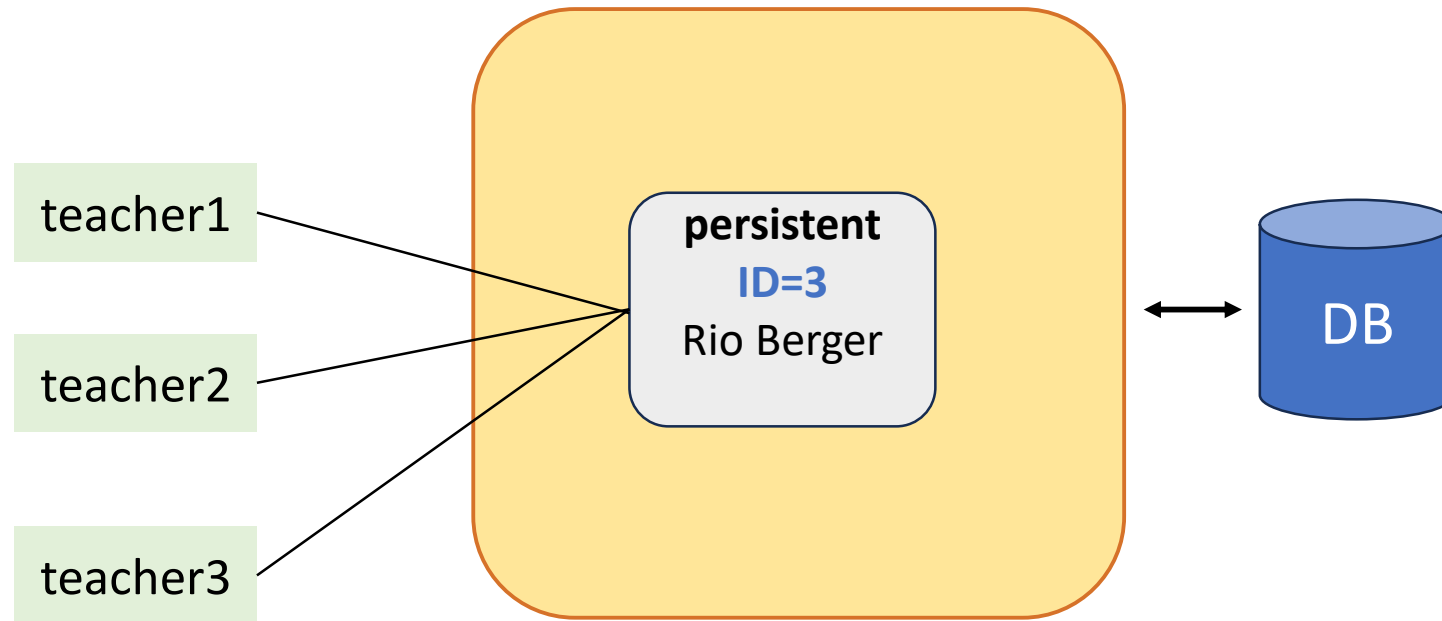
- First Level Cache
- Repeatable Read
- Automatic Dirty Checking

First Level Cache

```
EntityManager entityManager = factory.createEntityManager();  
Teacher teacher1 = entityManager.find(Teacher.class, 3);  
Teacher teacher2 = entityManager.find(Teacher.class, 3);  
Teacher teacher3 = entityManager.find(Teacher.class, 3);
```

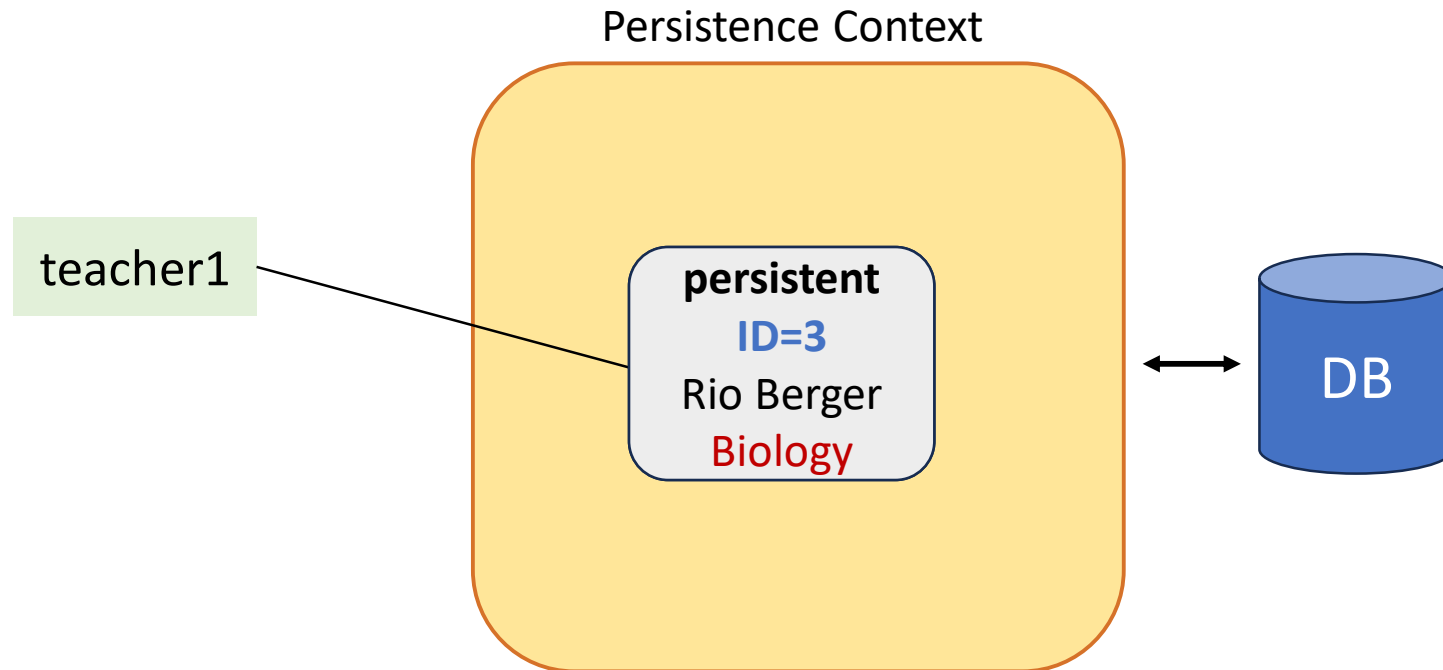
id	name	surname
3	Rio	Harper

Persistence Context



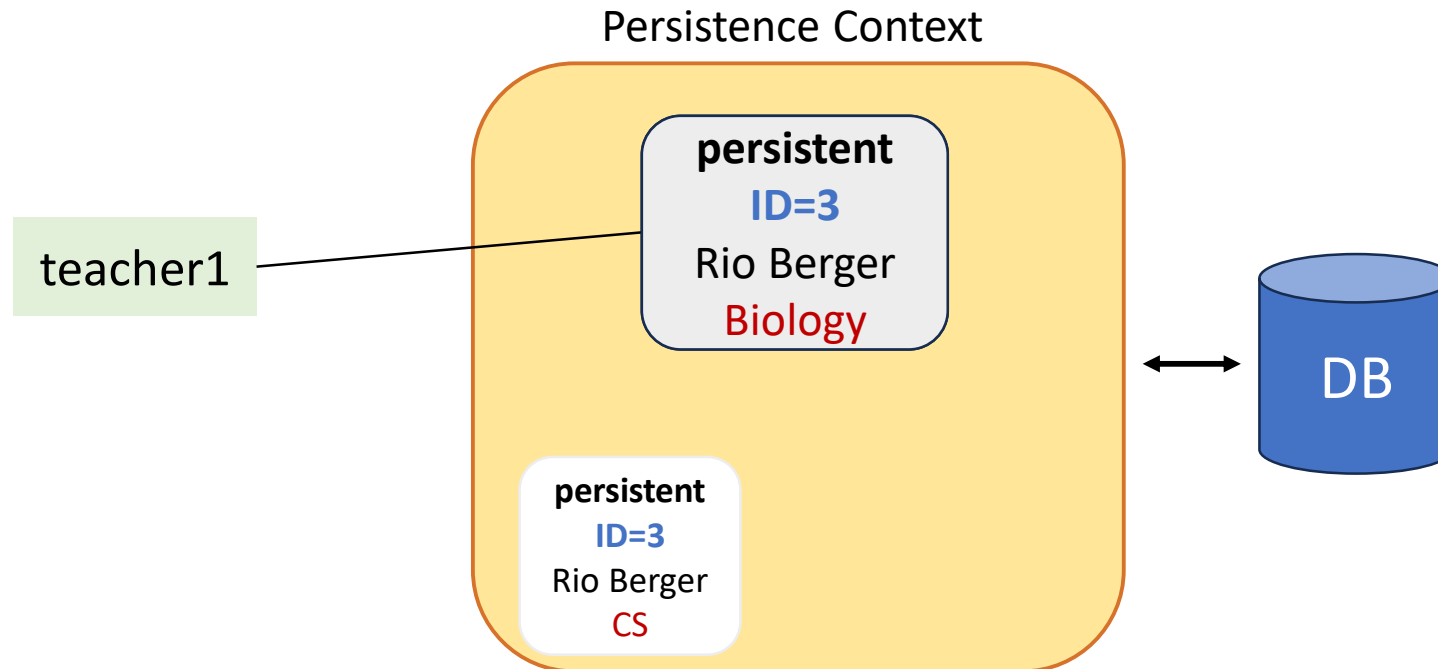
Automatic Dirty Checking

```
EntityManager entityManager = factory.createEntityManager();  
Teacher teacher1 = entityManager.find(Teacher.class, 3);  
teacher1.setSubject("Biology");  
transaction.commit();
```



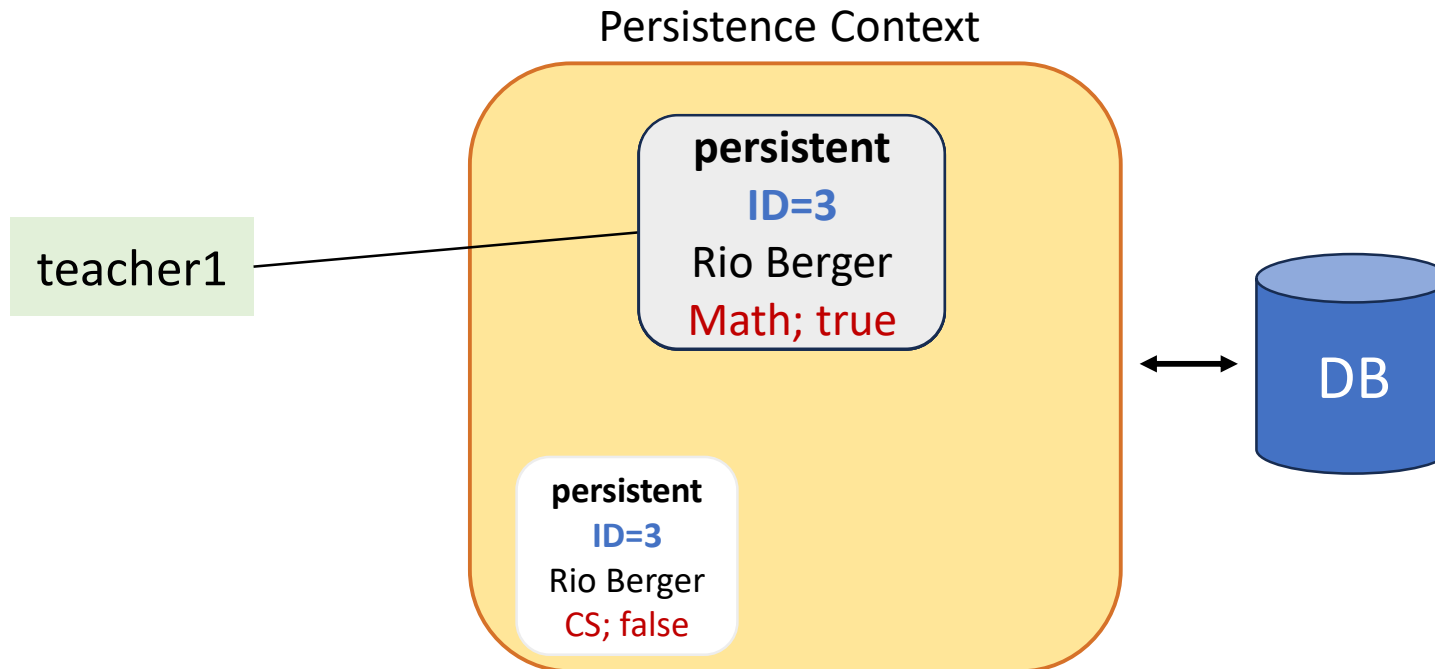
Automatic Dirty Checking

```
EntityManager entityManager = factory.createEntityManager();  
Teacher teacher1 = entityManager.find(Teacher.class, 3);  
teacher1.setSubject("Biology");  
transaction.commit();
```



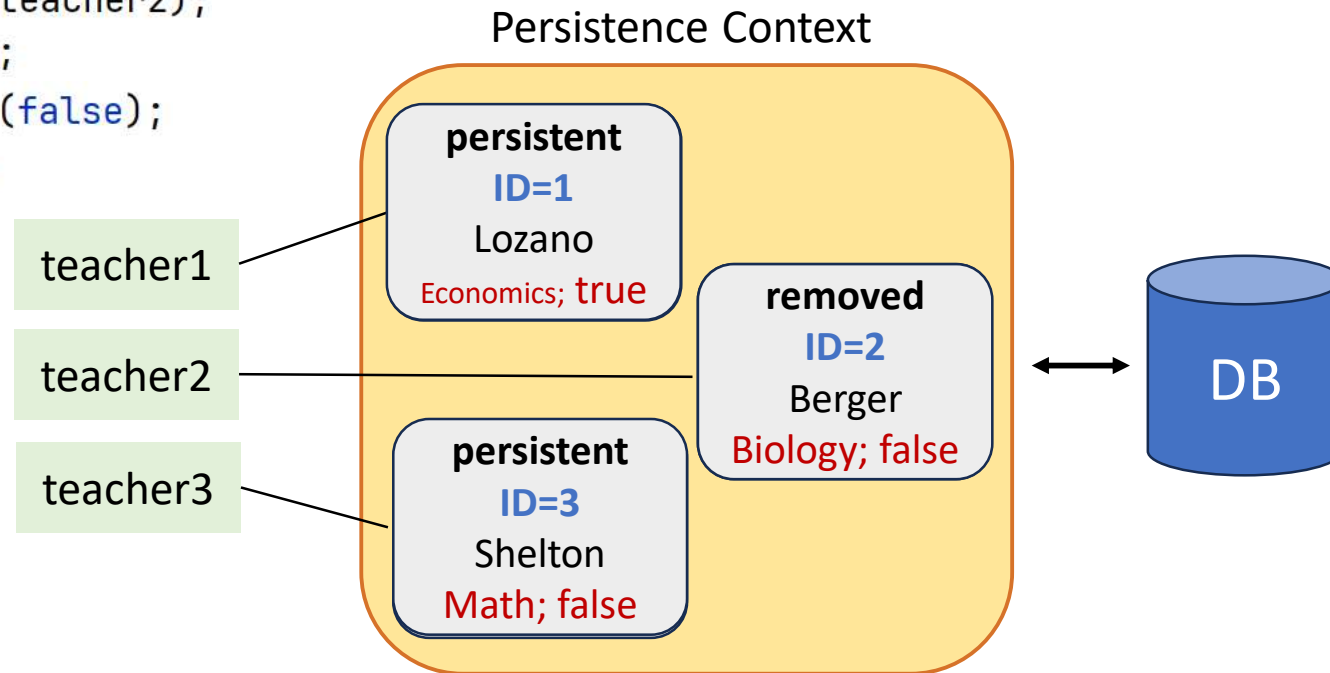
Automatic Dirty Checking

```
EntityManager entityManager = factory.createEntityManager();  
Teacher teacher1 = entityManager.find(Teacher.class, 3);  
teacher1.setSubject("Biology");    teacher1.setSubject("Math");  
teacher1.setProfessor(true);        transaction.commit();
```



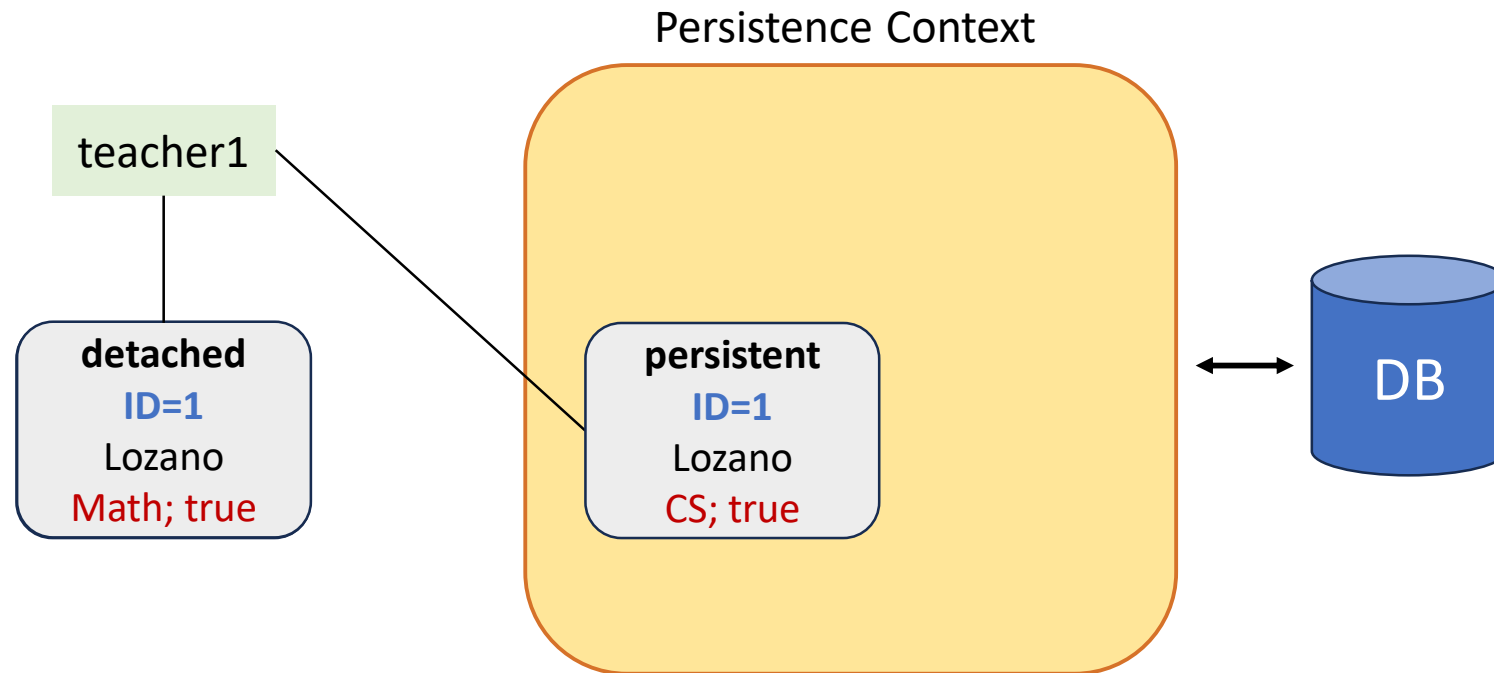
Method flush

```
Teacher teacher1 = entityManager.find(Teacher.class,1);  
Teacher teacher2 = entityManager.find(Teacher.class,1);  
Teacher teacher3 = entityManager.find(Teacher.class,1);  
teacher1.setSubject("Economics");  
entityManager.remove(teacher2);  
entityManager.flush();  
teacher3.setProfessor(false);  
transaction.commit();
```

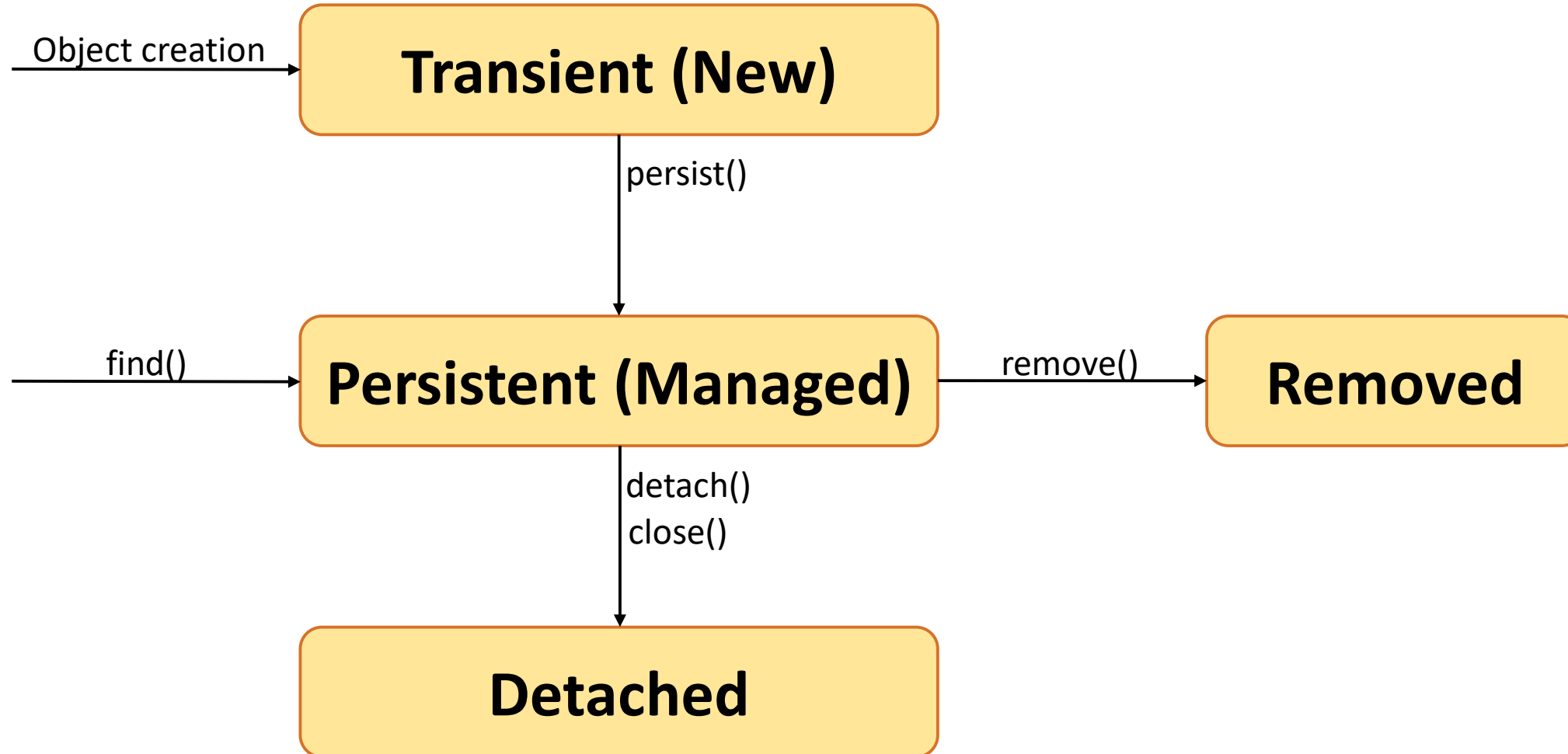


Method detach

```
Teacher teacher1 = entityManager.find(Teacher.class, 1);  
entityManager.detach(teacher1);  
teacher1.setSubject("Math");  
transaction.commit();
```

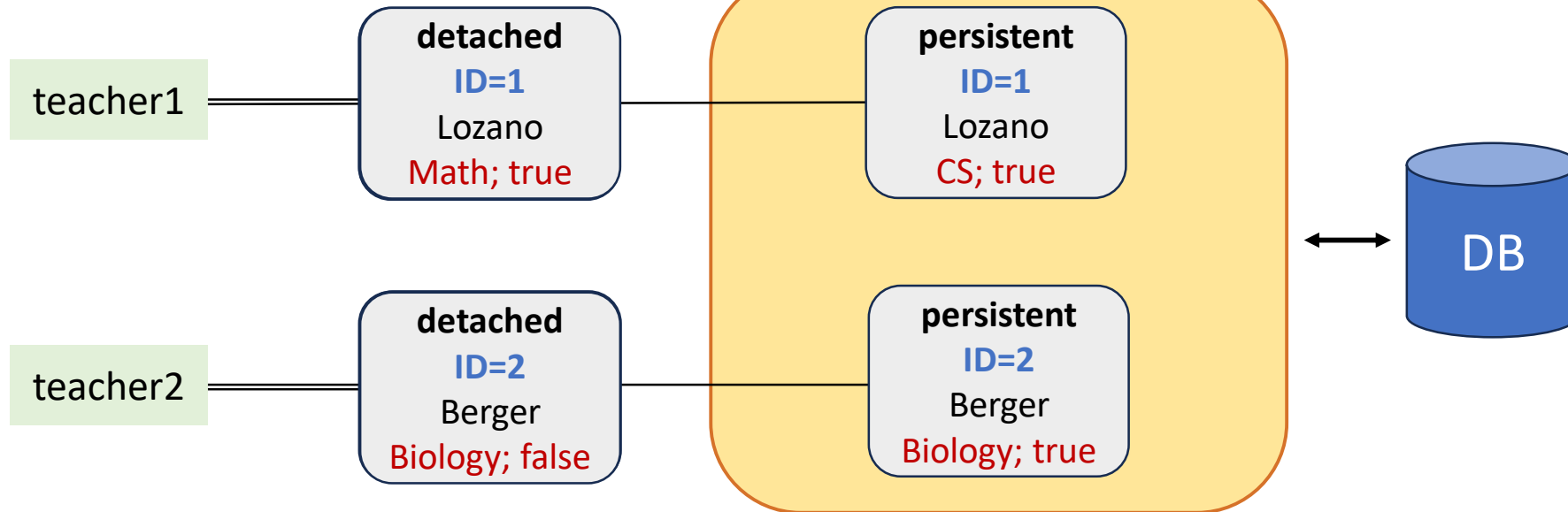


Entity States

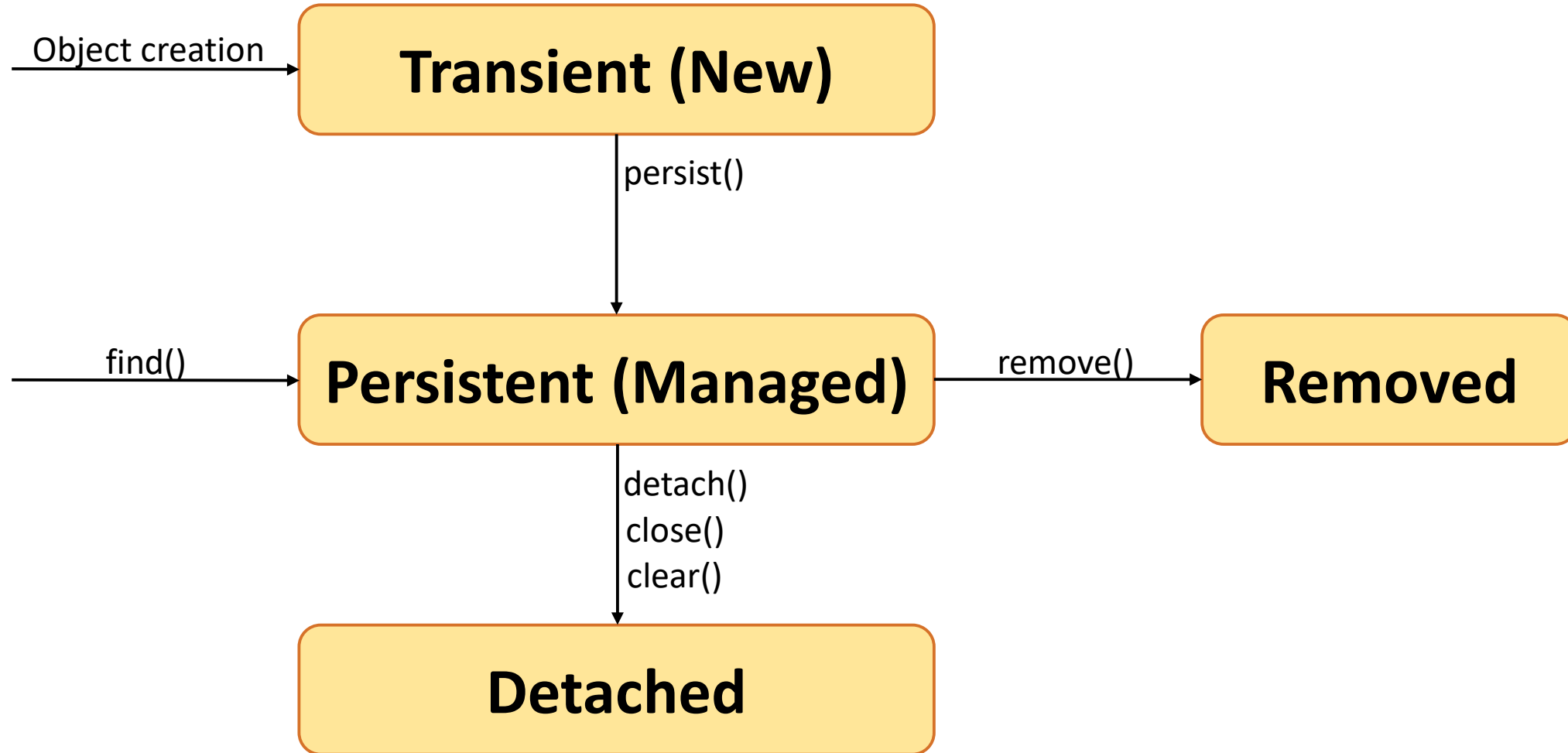


Method clear

```
Teacher teacher1 = entityManager.find(Teacher.class, 1);  
Teacher teacher2 = entityManager.find(Teacher.class, 2);  
entityManager.clear();  
teacher1.setSubject("Math");  
teacher2.setProfessor(false);  
transaction.commit();
```



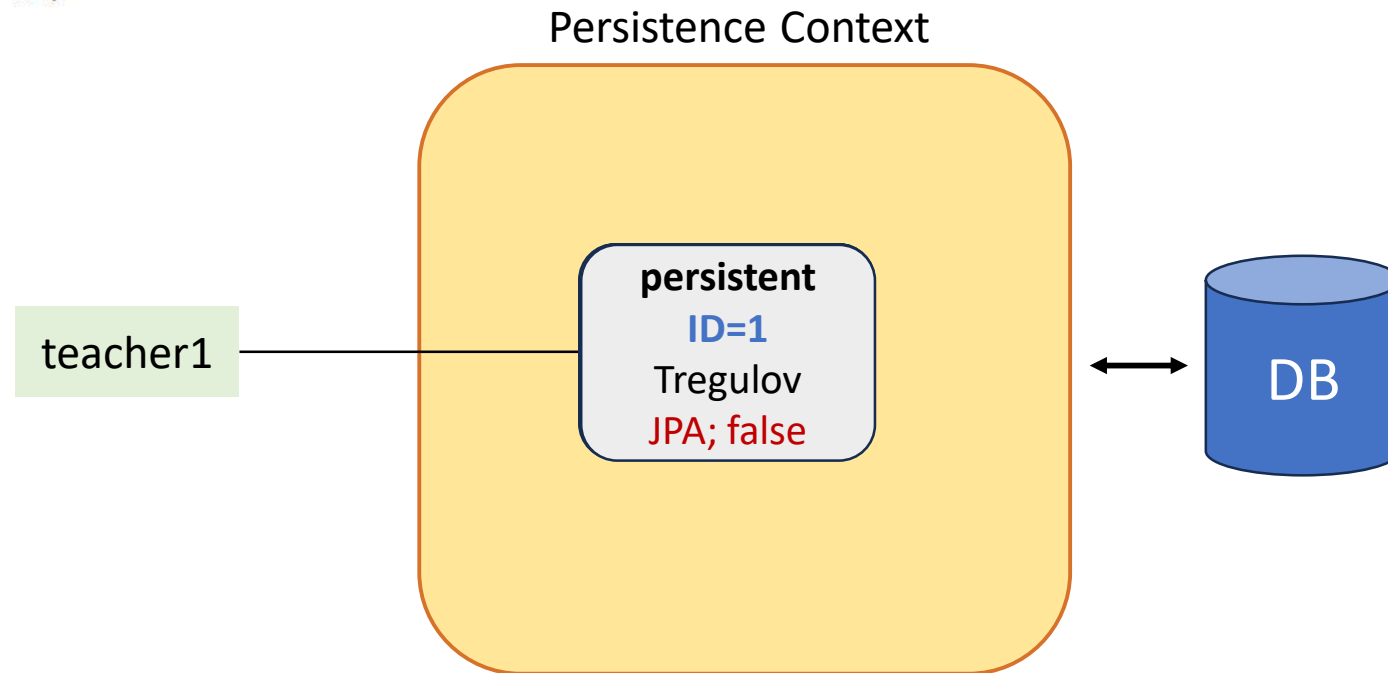
Entity States



Method refresh

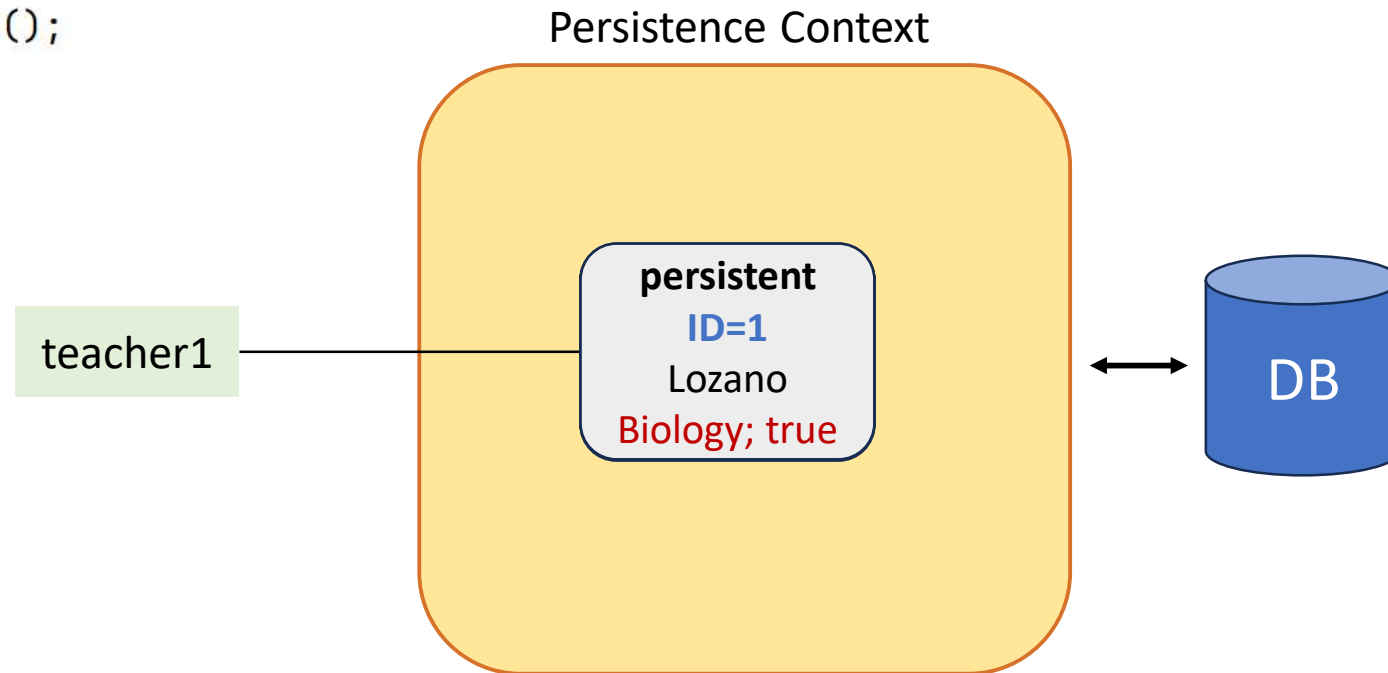
```
Teacher teacher1 = entityManager.find(Teacher.class, 1);  
teacher1.setSubject("Biology");  
teacher1.setProfessor(false);  
entityManager.refresh(teacher1);  
transaction.commit();
```

id	name	surname	subject	is_professor
1	Zaur	Tregulov	JPA	false

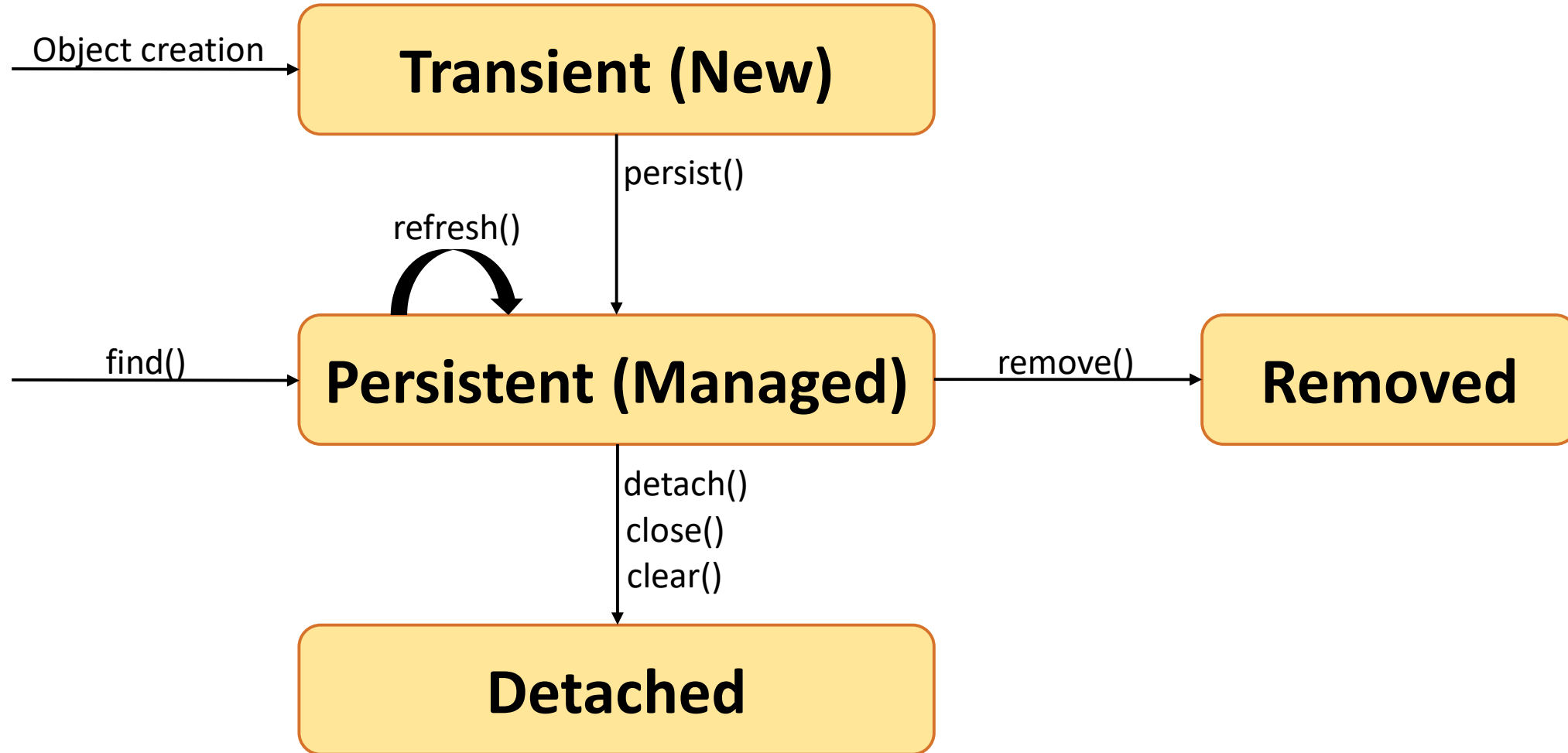


Method refresh

```
Teacher teacher1 = entityManager.find(Teacher.class, 1);  
teacher1.setSubject("Biology");  
entityManager.flush();  
teacher1.setProfessor(false);  
entityManager.refresh(teacher1);  
transaction.commit();
```

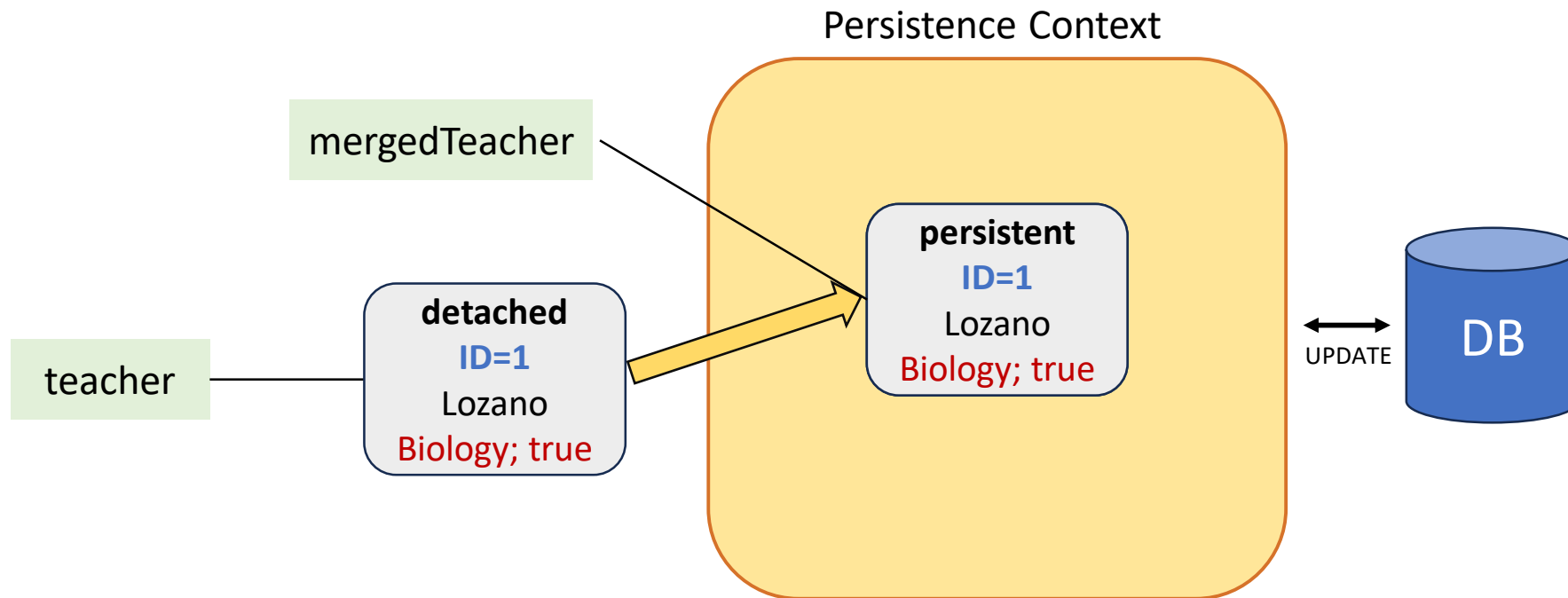


Entity States



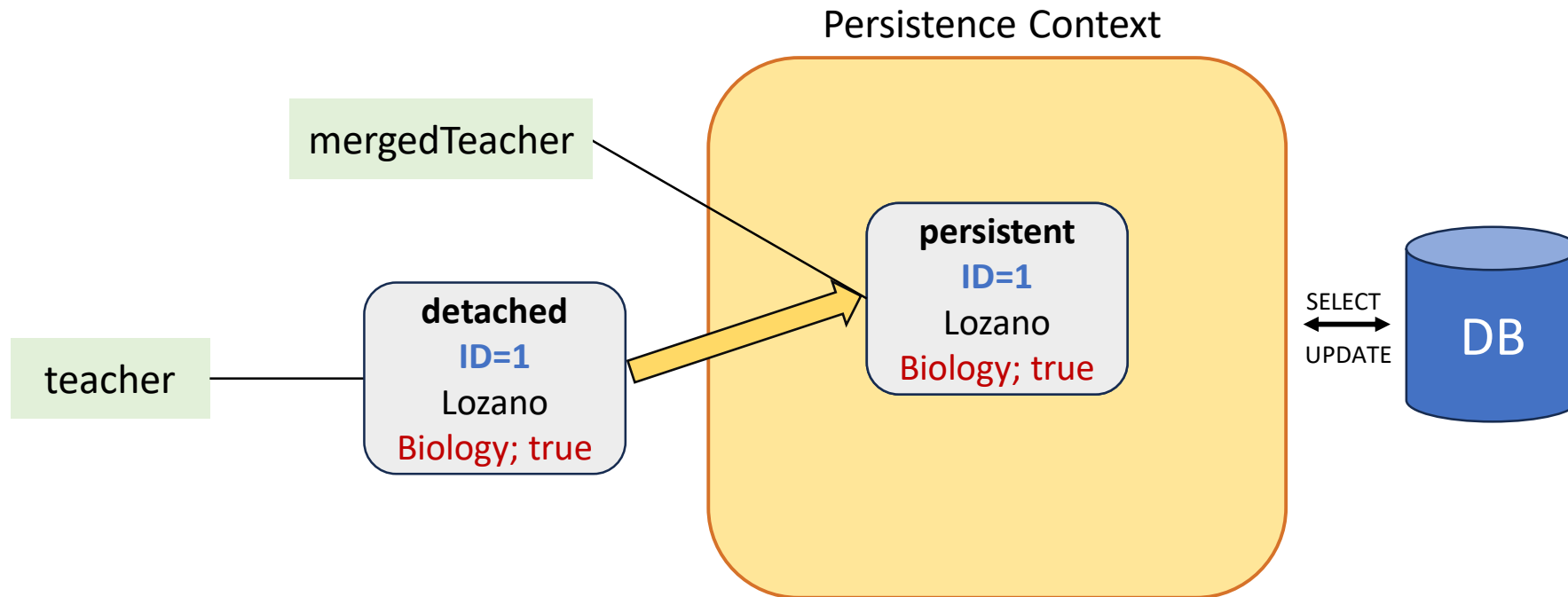
Method merge

```
Teacher mergedTeacher = entityManager.merge(teacher);  
.....  
transaction.commit();
```



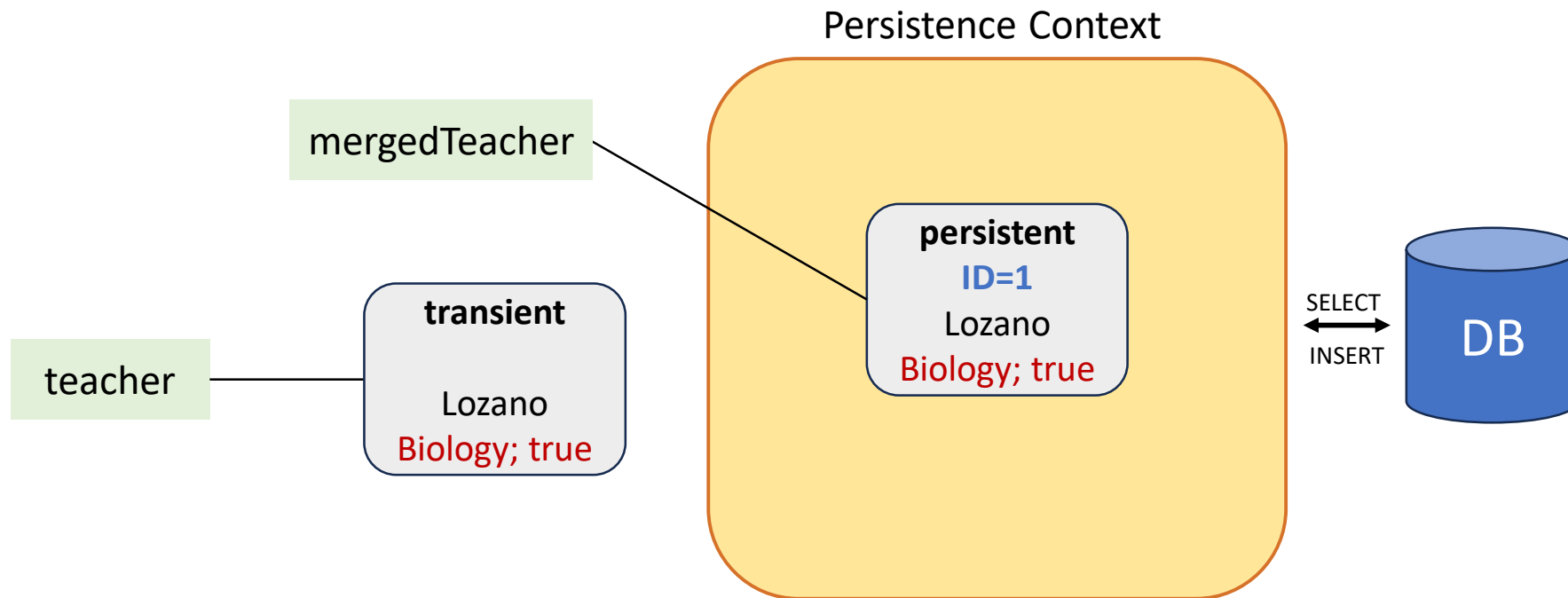
Method merge

```
Teacher mergedTeacher = entityManager.merge(teacher);  
.....  
transaction.commit();
```



Method merge

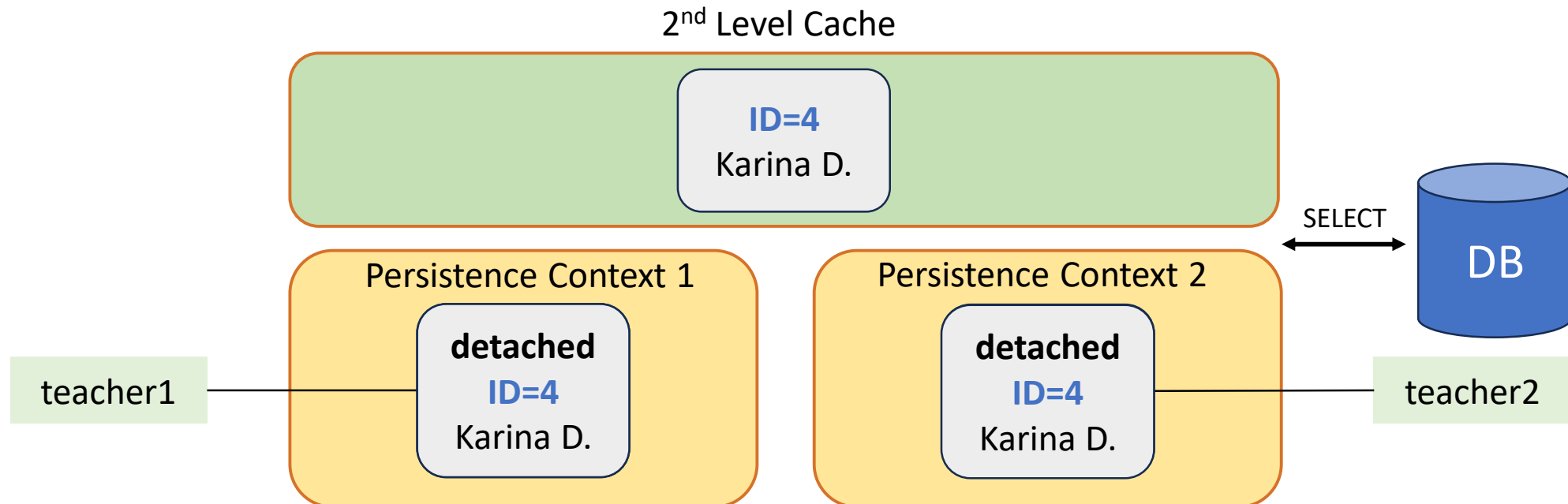
```
Teacher teacher = new Teacher("Alessandro", "Lozano", "Biology", true);  
Teacher mergedTeacher = entityManager.merge(teacher);  
.....  
transaction.commit();
```



Second Level Cache

```
EntityManagerFactory factory = Persistence
    .createEntityManagerFactory("jpa-course");
EntityManager entityManager1=factory.createEntityManager();
Teacher teacher1 = entityManager1.find(Teacher.class, 4);
entityManager1.close();

EntityManager entityManager2=factory.createEntityManager();
Teacher teacher2 = entityManager2.find(Teacher.class, 4);
entityManager2.close();
```



Entity Lifecycle Callback methods

@PrePersist

@PostPersist

@PreUpdate

@PostUpdate

@PreRemove

@PostRemove

@PostLoad

JPQL

JPQL – Java Persistence Query Language

- При работе с **SQL** мы используем имя таблицы и названия её столбцов.
- При работе с **JPQL** мы используем имя Entity и названия его полей.

JPQL

```
Query query = entityManager.createQuery("select s from Student s");  
List<Student> students1 = query.getResultList();  
  
List<Student> students2 = entityManager.createQuery("select s from Student s").getResultList();
```

```
List<Student> students = entityManager.createQuery("select s from Student s" +  
    " where s.name = 'Leo' ").getResultList();
```

```
List<Student> students = entityManager.createQuery("select s from Student s" +  
    " where avgGrade > 8.5").getResultList();
```

```
List<Student> students = entityManager.createQuery("select s from Student s" +  
    " where avgGrade between 7 and 8").getResultList();
```

JPQL

```
List<Student> students = entityManager.createQuery("select s from Student s" +  
    " where s.name LIKE '%a%' ").getResultList();
```

```
List<Student> students = entityManager.createQuery("select s from Student s" +  
    " where lower(s.name) LIKE '%a%' ").getResultList();
```

```
List<Student> students = entityManager.createQuery("select s from Student s" +  
    " where s.avgGrade IS NULL").getResultList();
```

```
List<Student> students = entityManager.createQuery("select s from Student s" +  
    " where s.name LIKE '%l%' AND s.avgGrade>8.5").getResultList();
```

JPQL

```
List<String> names = entityManager.createQuery("select s.name from Student s").getResultList();
```

```
List<Object[]> studentsInfo = entityManager.createQuery("select s.name, s.avgGrade from Student s")  
    .getResultList();
```

```
Query query = entityManager.createQuery("select max(s.avgGrade) from Student s");  
double maxGrade = (Double)query.getSingleResult();
```

```
Query query = entityManager.createQuery("select avg(s.avgGrade) from Student s");  
double avgGrade = (Double)query.getSingleResult();
```

JPQL

```
Query query = entityManager.createQuery("select s from Student s" +  
    " where s.name LIKE ?1 AND s.avgGrade > ?2 ");  
query.setParameter(1, "%l%");  
query.setParameter(2, 8.5);  
List<Student> students = query.getResultList();
```

```
Query query = entityManager.createQuery("select s from Student s" +  
    " where s.name LIKE :letter AND s.avgGrade > :grade ");  
query.setParameter("letter", "%l%");  
query.setParameter("grade", 8.5);  
List<Student> students = query.getResultList();
```

JPQL

```
Query query = entityManager.createQuery("update Student s set avgGrade = 7.0 " +  
    "where length(surname)>6 ");  
query.executeUpdate();
```

```
Query query = entityManager.createQuery("delete Student s where s.avgGrade<7.5 OR s.avgGrade IS NULL");  
query.executeUpdate();
```


JPQL

```
Query query = entityManager.createQuery("select u from University u where u.students is empty");  
List<University> universities = query.getResultList();
```

```
Query query = entityManager.createQuery("select u from University u where size(u.students) = 1");  
List<University> universities = query.getResultList();
```

```
Query query = entityManager.createQuery("SELECT u FROM University u ORDER BY SIZE(u.students) DESC");  
List<University> universities = query.getResultList();
```

```
Query query = entityManager.createQuery("select u, s from University u, Student s");  
List<Object[]> results = query.getResultList();  
for(Object[] result:results){  
    System.out.println(result[0] + " ---> " + result[1]);  
}
```

```
Query query = entityManager.createQuery("select u, s from University u JOIN u.students s");  
List<Object[]> results = query.getResultList();
```

JPQL

```
@NamedQuery(name = "University.allUniversitiesLessOrEqualTo2"  
    , query = "select u from University u where size(u.students) <= 2")
```

```
Query query = entityManager.createNamedQuery("University.allUniversitiesLessOrEqualTo2");
```

```
@NamedQueries(  
    {  
        @NamedQuery(name = "University.allUniversitiesLessOrEqualTo2"  
            , query = "select u from University u where size(u.students) <= 2"),  
        @NamedQuery(name = "University.studentsWithAvgGradeBetween"  
            , query = "select s from Student s where avgGrade between :from and :to")  
    }  
)
```

Native Query

```
Query query = entityManager.createNativeQuery("select * from students", Student.class);
```

```
Query query = entityManager.createNativeQuery("select * from students where avg_grade > ?1 and " +  
    "length(name) = ?2", Student.class);  
query.setParameter(1, 8);  
query.setParameter(2, 5);
```


JPQL

- Метод flush()

- First Level Cache

SQL

SQL – Structured Query Language



SQL для начинающих:
с нуля до сертификата Oracle

zaurtregulov@gmail.com

Criteria API

Плюсы:

- **Type safety**

- **Динамическое формирование запросов**

Criteria API

```
// JPQL: select s from Student s;

//1 Creation of Criteria Builder
CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();

//2 Creation of Criteria Query
CriteriaQuery<Student> criteriaQuery = criteriaBuilder.createQuery(Student.class);

//3 Root creation
Root<Student> root = criteriaQuery.from(Student.class); //from Student s

//4 Adding root to Criteria Query
criteriaQuery.select(root); // select s from Student s

//5 Query creation
TypedQuery<Student> query = entityManager.createQuery(criteriaQuery);

List<Student> students = query.getResultList();
```

Criteria API

```
// JPQL: select s.name from Student s;

//1 Creation of Criteria Builder
CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();

//2 Creation of Criteria Query
CriteriaQuery<String> criteriaQuery = criteriaBuilder.createQuery(String.class);

//3 Root creation
Root<Student> root = criteriaQuery.from(Student.class); //from Student s

//4 Adding root to Criteria Query
criteriaQuery.select(root.get("name")); // select s.name from Student s

//5 Query creation
TypedQuery<String> query = entityManager.createQuery(criteriaQuery);

List<String> names = query.getResultList();
```

Criteria API

```
// JPQL: select s from Student s where avgGrade>=7.5;

//1 Creation of Criteria Builder
CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();

//2 Creation of Criteria Query
CriteriaQuery<Student> criteriaQuery = criteriaBuilder.createQuery(Student.class);

//3 Root creation
Root<Student> root = criteriaQuery.from(Student.class); //from Student s

//3.1 Condition creation
Predicate predicate = criteriaBuilder.greaterThanOrEqualTo(root.get("avgGrade"), 7.5);

//3.2 Adding condition to Criteria Query
criteriaQuery.where(predicate);

//4 Adding root to Criteria Query
criteriaQuery.select(root); // select s from Student s where s.avgGrade>=7.5

//5 Query creation
TypedQuery<Student> query = entityManager.createQuery(criteriaQuery);
List<Student> names = query.getResultList();
```

Criteria API

```
// JPQL: select s.name, s.avgGrade from Student s;

//1 Creation of Criteria Builder
CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();

//2 Creation of Criteria Query
CriteriaQuery<Object[]> criteriaQuery = criteriaBuilder.createQuery(Object[].class);

//3 Root creation
Root<Student> root = criteriaQuery.from(Student.class); //from Student s

//4 Adding root to Criteria Query
criteriaQuery.multiselect(root.get("name"), root.get("avgGrade")); // select s.name, s.avgGrade from Student s

//5 Query creation
TypedQuery<Object[]> query = entityManager.createQuery(criteriaQuery);

List<Object[]> studentInfo = query.getResultList();
```

Criteria API

```
// JPQL: select u, s from University u JOIN u.students s;

//1 Creation of Criteria Builder
CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();

//2 Creation of Criteria Query
CriteriaQuery<Object[]> criteriaQuery = criteriaBuilder.createQuery(Object[].class);

//3 Root creation
Root<University> root = criteriaQuery.from(University.class); //from University u

//3.1 JOIN
Join<University, Student> join = root.join("students");

//4 Adding root to Criteria Query
criteriaQuery.multiselect(root, join); //select u, s from University u JOIN
                                         // u.students s;

//5 Query creation
TypedQuery<Object[]> query = entityManager.createQuery(criteriaQuery);

List<Object[]> students = query.getResultList();
```


JPA & Hibernate

JPA – Java Persistence API

JPA – стандартная спецификация, которая описывает систему для управления Java объектами в таблицах БД-х

Hibernate – самая популярная реализация спецификации JPA

Hibernate – это framework, который используется для сохранения, получения, изменения и удаления Java объектов из БД

JPA описывает правила, а **Hibernate** их реализует

JPA & Hibernate

JPA содержит список того, что надо делать:

- надо уметь сохранять объект в БД
- надо уметь изменять объект
- надо уметь удалять объект из БД

.....

Hibernate умеет это делать:

- умею сохранять объект в БД
- умею изменять объект
- умею удалять объект из БД

.....

JPA & Hibernate

```
interface WhatToDo{  
    void persist();  
    void remove();  
    // specification of other methods  
}
```

```
class HowToDo implements WhatToDo{  
    public void persist(){  
        //          code#1  
    }  
    public void remove(){  
        //          #code2  
    }  
    //realization of other methods  
}
```

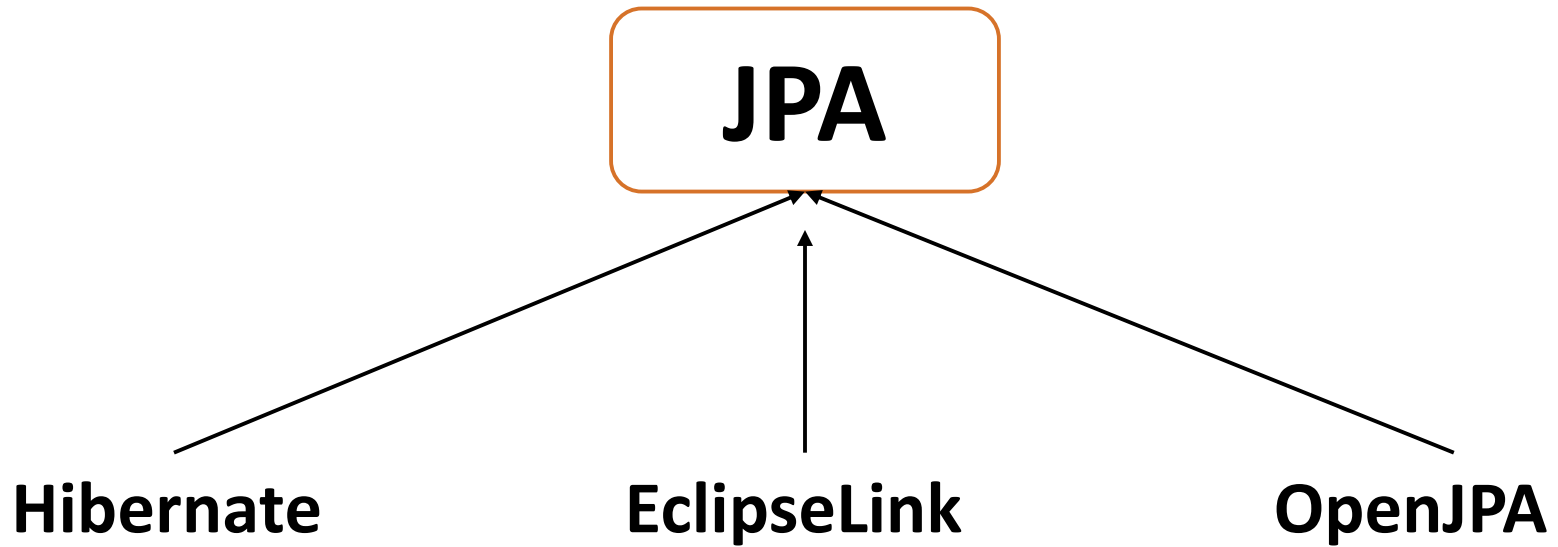
Hibernate

Hibernate

Реализация
спецификации
JPA

Функционал, не
связанный с **JPA**

JPA

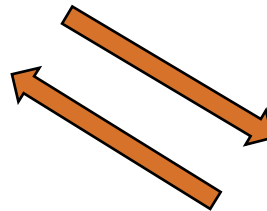


ORM





ORM – Object to Relational Mapping

ORM – это преобразование объекта в строку в таблице и обратное преобразование

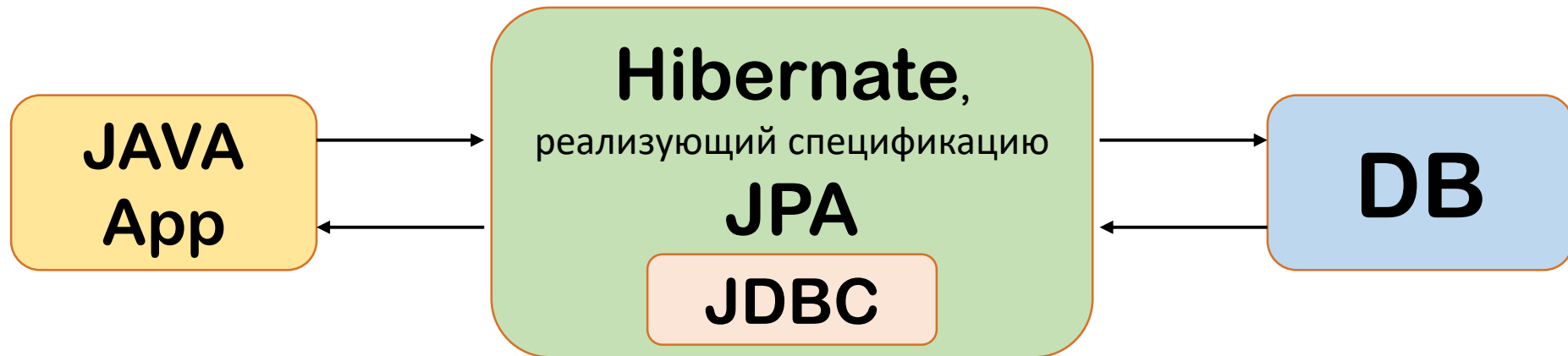
```
public class Student {  
    private Long id;  
    private String name;  
    private String surname;  
    private Double avgGrade;  
  
    // other code  
}
```



students

 id ▼	 name ▼	 surname ▼	 avg_grade ▼
--	--	---	---

JPA & Hibernate



JPA & Hibernate

Плюсы

- Удобная настройка подключения к БД
- Лёгкий процесс знакомства класса с таблицей
- Немного кода
- Можно обойтись без использования **SQL**
- Присутствует синхронизация между объектом и строкой
- Независимость от СУБД, в которой хранятся данные

CRUD

- **CREATE - команда INSERT**
- **READ - команда SELECT**
- **UPDATE - команда UPDATE**
- **DELETE - команда DELETE**

HQL & JPQL

HQL

JPQL



JPA & Hibernate

JPA – стандартная спецификация, которая описывает систему для управления Java объектами в таблицах БД-х

Hibernate – самая популярная реализация спецификации JPA

JPA описывает правила, а **Hibernate** их реализует

HQL

```
// ALL Students  
// select * from students;  
Query<Student> query = session.createQuery("from Student");  
List<Student> students = query.getResultList();
```

```
//UPDATE  
session.createQuery("update Student s set s.avgGrade = 10.0 where length(s.name)=5 ").executeUpdate();  
  
//DELETE  
session.createQuery("delete Student s where s.avgGrade<9").executeUpdate();
```

Hibernate INSERT

```
SessionFactory factory = new Configuration()
    .configure("hibernate.cfg.xml")
    .addAnnotatedClass(Student.class)
    .buildSessionFactory();
Session session = factory.getCurrentSession();
Transaction transaction = session.getTransaction();

try{
    Student student = new Student("Leo", "Farrell", 8.4);
    transaction.begin();

    session.persist(student);

    transaction.commit();
}
```

Hibernate SELECT

```
SessionFactory factory = new Configuration()
    .configure("hibernate.cfg.xml")
    .addAnnotatedClass(Student.class)
    .buildSessionFactory();
Session session = factory.getCurrentSession();
Transaction transaction = session.getTransaction();

try{
    transaction.begin();

    Student student = session.get(Student.class, 1);

    transaction.commit();
}
```

Hibernate UPDATE

```
SessionFactory factory = new Configuration()
    .configure("hibernate.cfg.xml")
    .addAnnotatedClass(Student.class)
    .buildSessionFactory();
Session session = factory.getCurrentSession();
Transaction transaction = session.getTransaction();

try{
    transaction.begin();

    Student student = session.get(Student.class, 2);

    student.setAvgGrade(9.0);

    transaction.commit();
}
```

Hibernate DELETE

```
SessionFactory factory = new Configuration()
    .configure("hibernate.cfg.xml")
    .addAnnotatedClass(Student.class)
    .buildSessionFactory();
Session session = factory.getCurrentSession();
Transaction transaction = session.getTransaction();

try{
    transaction.begin();

    Student student = session.get(Student.class, 2);

    session.remove(student);

    transaction.commit();
}
```


One-to-One relationship

STUDENT

PASSPORT

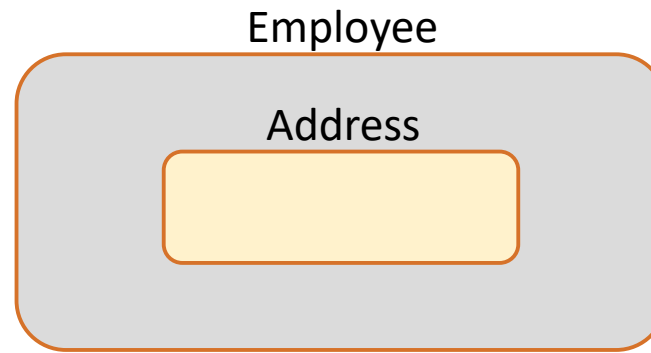
```
@Entity
@Table(name = "students")
public class Student {
    // some code
}
```

```
@Entity
@Table(name = "passports")
public class Passport {
    //some code
}
```

id	name	surname	avg_grade	passport_id
1	Chanel	King	9.1	10
2	Leo	Farrell	8.4	11

id	email	height	eye_color
10	chanel.king@gmail.com	174	blue
11	leo.farrell@yahoo.com	178	black

Composite type mapping



```
@Entity
@Table(name = "employees")
public class Employee {
    Address address;
    //some code
}
```

```
@Embeddable
public class Address {
    //some code
}
```

id	name	salary	experience	country	city	street	house
1	Michael	4000	15	USA	Chicago	Dempster	40
2	Rudolf	3500	10	Germany	Berlin	Karl-Marx-Alee	25

Collection mapping

```
@Entity
public class Employee {
    @ElementCollection
    List<String> friends = new ArrayList<>();
    //other code
}
```

id	name	salary	experience	friends
1	Michael	4000	15	Chanel, Leo, Julia
2	Rudolf	3500	10	Roy, Kynlee, Eric

id	name	salary	experience	friend
1	Michael	4000	15	Chanel
1	Michael	4000	15	Leo
1	Michael	4000	15	Julia
2	Rudolf	3500	10	Roy
3	Rudolf	3500	10	Kynlee
4	Rudolf	3500	10	Eric

employees

id	name	salary	experience
1	Michael	4000	15
2	Rudolf	3500	10

emp_id	friend_name
1	Chanel
1	Leo
1	Julia
2	Roy
2	Kynlee
2	Eric

emp_friends

Collection mapping

```
@Entity
public class Employee {
    @ElementCollection
    List<Friend> friends = new ArrayList<>();
    //other code
```

```
@Embeddable
public class Friend {
    private String name;
    private String surname;
    private int age;
    //other code
```

employees

id	name	salary	experience
1	Michael	4000	15
2	Rudolf	3500	10

emp_friends

emp_id	emp_name	emp_surname	emp_age
1	Chanel	King	22
1	Leo	Farrell	24
1	Julia	Dean	23
2	Roy	Harper	22
2	Kynlee	Boyer	24
2	Eric	Scott	23

Collection mapping

```
@Entity
public class Employee {
    @ElementCollection
    List<String> friends = new ArrayList<>();
    //other code
}
```

employees

id	name	salary	experience
1	Michael	4000	15
2	Rudolf	3500	10

emp_id	friend_name
1	Chanel
1	Leo
1	Julia
2	Roy
2	Kynlee
2	Eric

emp_friends

Collection mapping

```
@ElementCollection
@CollectionTable(name = "emp_friends", joinColumns = @JoinColumn(name = "emp_id"))
@Column(name = "friend_name")
List<String> friends = new ArrayList<>();
```

Collection mapping

```
@ElementCollection
@CollectionTable(name = "emp_friends", joinColumns = @JoinColumn(name = "emp_id"))
@AttributeOverrides({
    @AttributeOverride(name = "name", column = @Column(name = "emp_name")),
    @AttributeOverride(name = "surname", column = @Column(name = "emp_surname")),
    @AttributeOverride(name = "age", column = @Column(name = "emp_age"))
})
List<Friend> friends = new ArrayList<>();
```

Inheritance mapping

Стратегии:

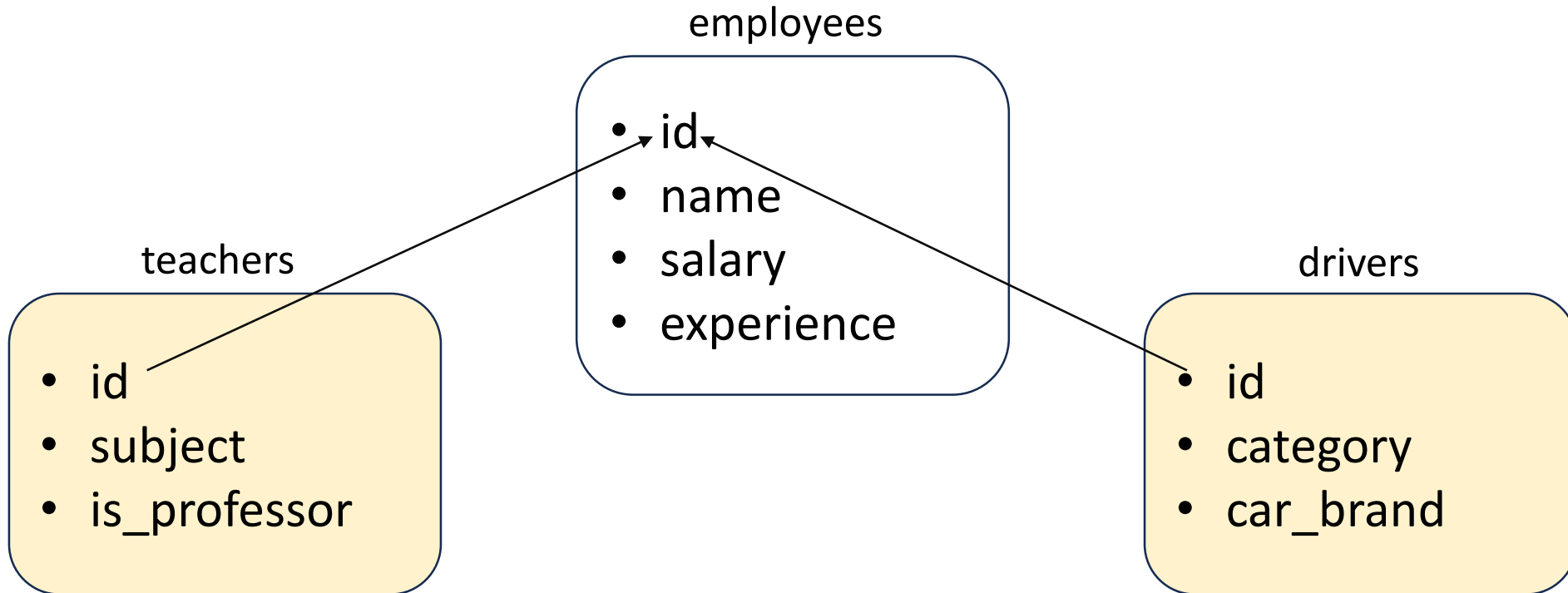
- **Single Table**
- **Joined**
- **Table per Class**

SINGLE_TABLE strategy

employees

- id
- name
- salary
- experience
- subject
- is_professor
- category
- car_brand

JOINED strategy



TABLE_PER_CLASS strategy

teachers

- id
- name
- salary
- experience
- subject
- is_professor

drivers

- id
- name
- salary
- experience
- category
- car_brand

@MappedSuperclass

teachers

- id
- name
- salary
- experience
- subject
- is_professor

drivers

- id
- name
- salary
- experience
- category
- car_brand

Inheritance mapping

Стратегии:

- **Single Table**
- **Joined**
- **Table per Class**

SINGLE_TABLE strategy

employees

- id
- name
- salary
- experience
- subject
- is_professor
- category
- car_brand

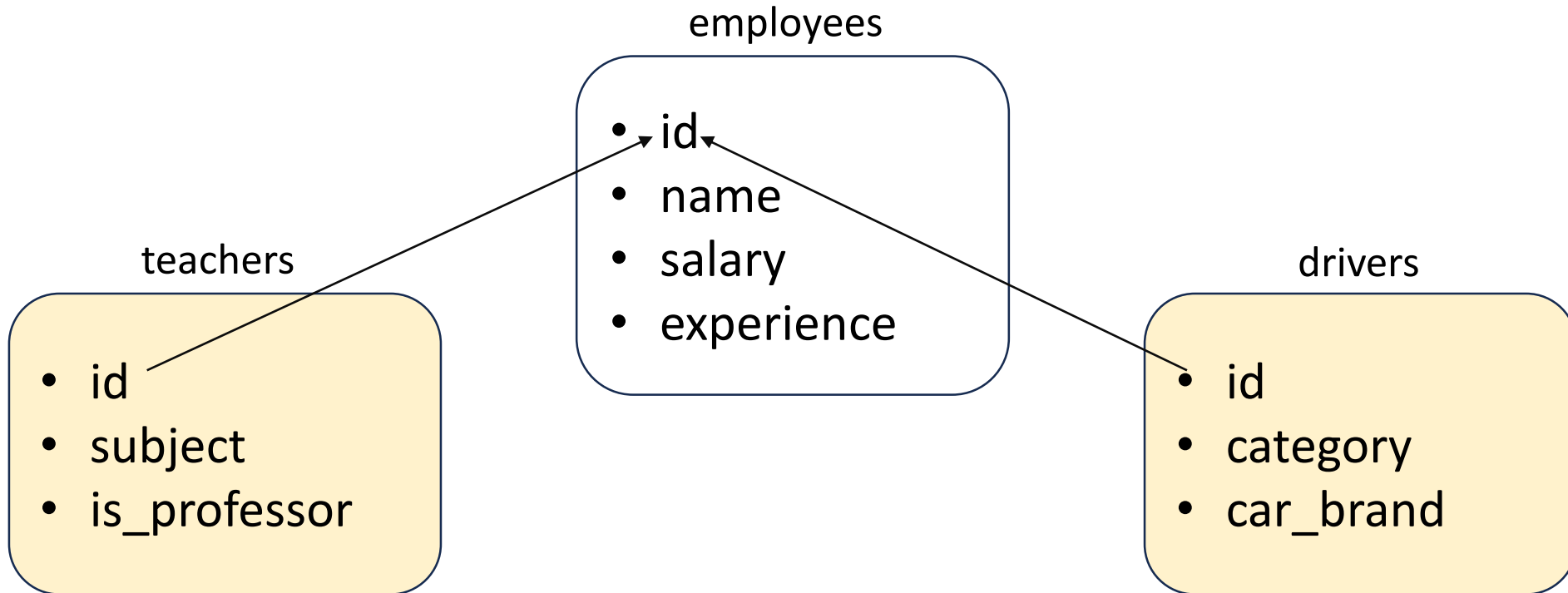
SINGLE_TABLE strategy

```
@Entity
@Table(name = "employees")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public abstract class Employee {
```

```
@Entity
public class Teacher extends Employee{
```

```
@Entity
public class Driver extends Employee{
```

JOINED strategy



JOINED strategy

```
@Entity
@Table(name = "employees")
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Employee {
```

```
@Entity
@Table(name = "teachers")
public class Teacher extends Employee{
```

```
@Entity
@Table(name = "drivers")
public class Driver extends Employee{
```

TABLE_PER_CLASS strategy

teachers

- id
- name
- salary
- experience
- subject
- is_professor

drivers

- id
- name
- salary
- experience
- category
- car_brand

TABLE_PER_CLASS strategy

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Employee {

    @Column(name = "id")
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE)
    private Long id;
```

```
@Entity
@Table(name = "teachers")
public class Teacher extends Employee{
```

```
@Entity
@Table(name = "drivers")
public class Driver extends Employee{
```

@MappedSuperclass

teachers

- id
- name
- salary
- experience
- subject
- is_professor

drivers

- id
- name
- salary
- experience
- category
- car_brand

@MappedSuperclass

```
@MappedSuperclass  
public abstract class Employee {
```

```
@Entity  
@Table(name = "teachers")  
public class Teacher extends Employee{
```

```
@Entity  
@Table(name = "drivers")  
public class Driver extends Employee{
```

Compound Primary Key

id	author	name	publication_year	rating
1	Dostoevsky	Crime and Punishment	1866	4.8
2	Dostoevsky	The Brothers Karamazov	1880	4.6
3	Tolstoy	War and Peace	1867	4.7

author	name	publication_year	rating
Dostoevsky	Crime and Punishment	1866	4.8
Dostoevsky	The Brothers Karamazov	1880	4.6
Tolstoy	War and Peace	1867	4.7

JPA & Hibernate

JDBC

JPA

Relationships

**Persistence
Context**

Работа с данными

Criteria API

Hibernate

**Advanced
mapping**

**Inheritance
mapping**

Разное

JPA & Hibernate

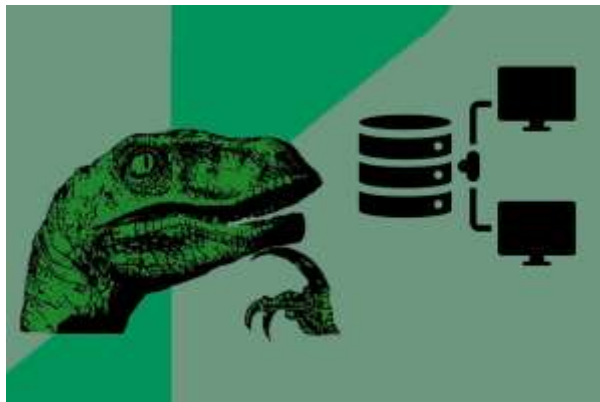
Java для Начинающих



Java - получи Чёрный Пояс



SQL для Начинающих



Spring для Начинающих



JPA & Hibernate



zaurtregulov@gmail.com