Jakarta Persistence

1.1. ORM

ИТМО ВТ

Объектно-реляционное отображение

ORM — Object/Relational Mapping — преобразование данных из объектной формы в реляционную и наоборот.
Подходы к выполнению ORM:

1) **Top-Down** (Сверху-Вниз) – модель классов приложения определяет реляционную.
2) **Bottom-up** (Снизу-Вверх) – модель классов строится на основании реляционной схемы.
3) **Meet-in-the-Middle** – параллельная разработка доменной модели (модели классов) и реляционной с учетом особенностей друг друга.

ИТМО ВТ

Объектно-реляционное несоответствие

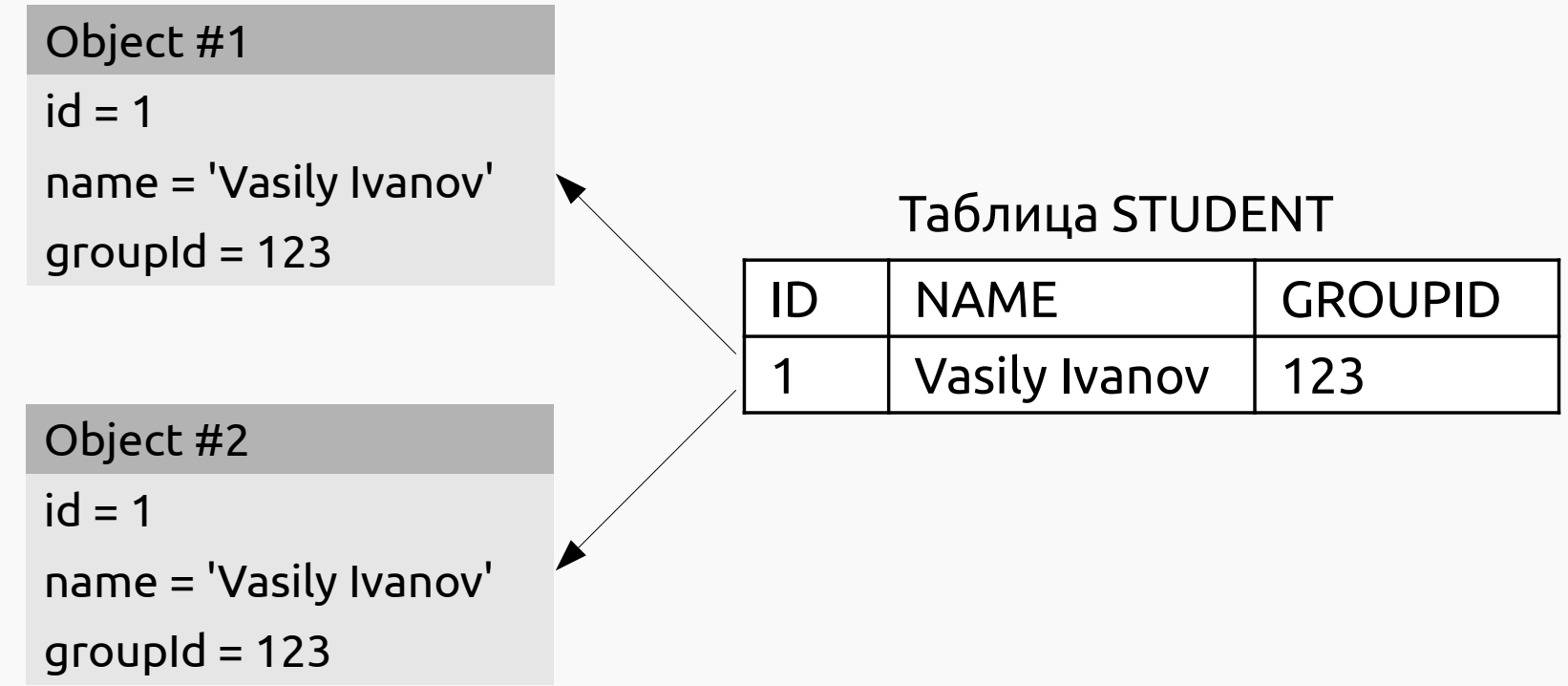
Проблемы при отображении:

Object-relational impedance mistmatch — из-за несоответствия объектно-ориентированной и реляционной модели друг другу:

1) Идентичность данных
2) Наследование
3) Связи между данными

Идентичность объектов и записей (1)

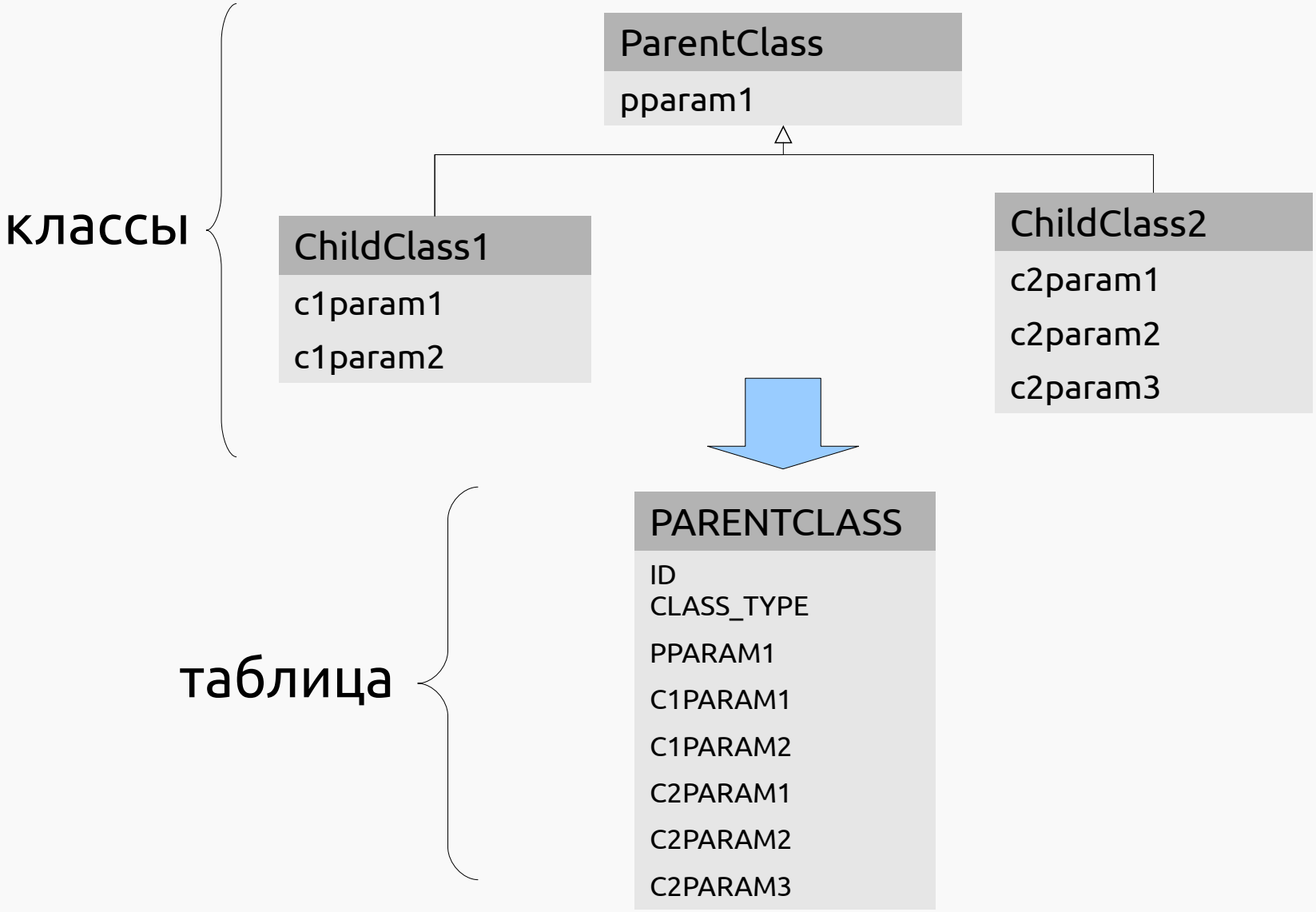
Проблема: несколько неидентичных Java-объектов соответствуют одной записи в базе данных



объекты (класс Student)

ИТМО ВТ

Наследование: Single Table Inheritance pattern



ИТМО ВТ

Single Table Inheritance pattern

При использовании данной стратегии все классы иерархии отображаются на одну таблицу базы данных.

Достоинства:

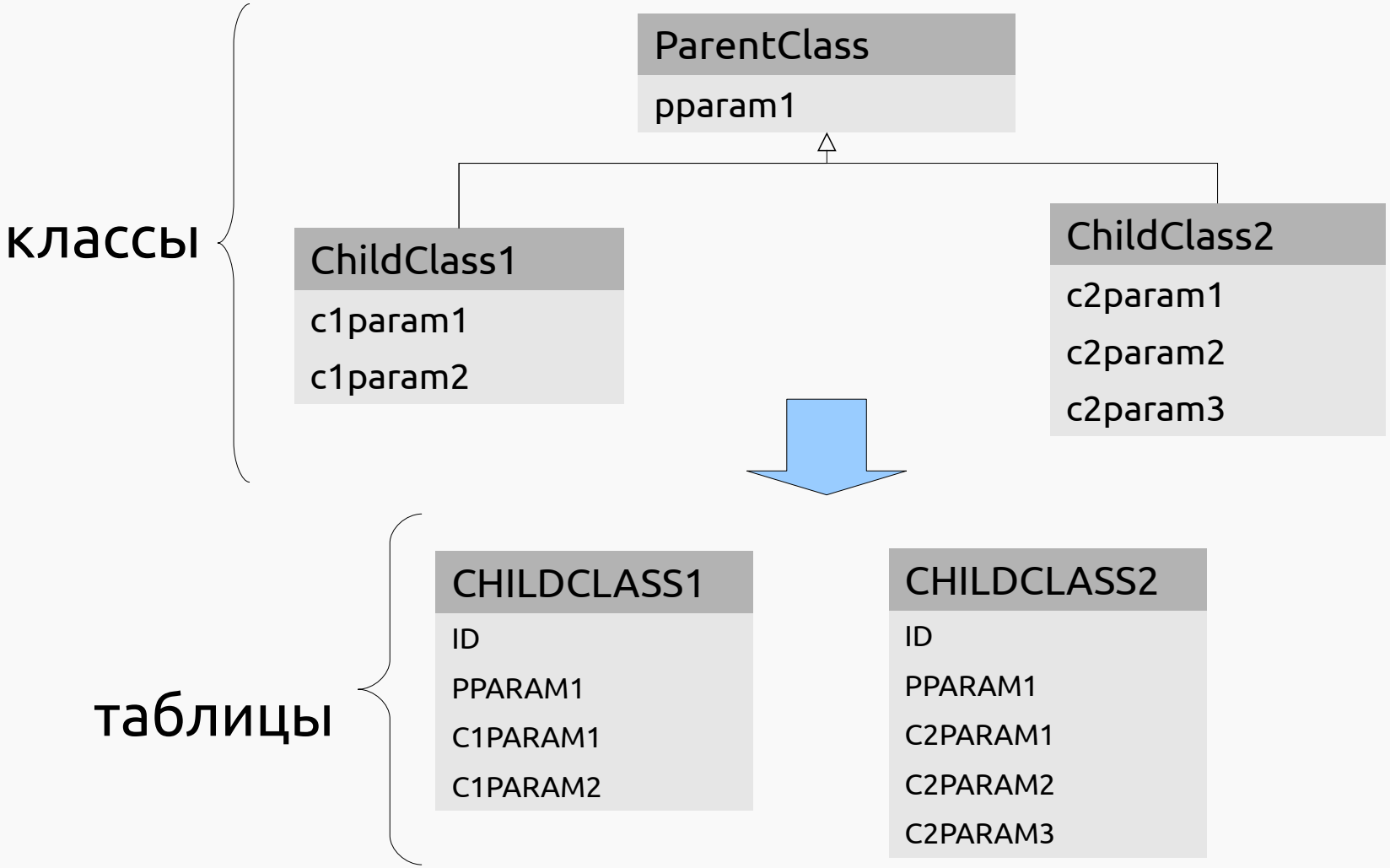
1) Наиболее простое решение;
2) Наиболее производительное решение;

Недостатки:

1) На поля подклассов нельзя накладывать null-ограничения;
2) Полученная таблица не нормализована;

ИТМО ВТ

Наследование: Concrete Table Inheritance pattern



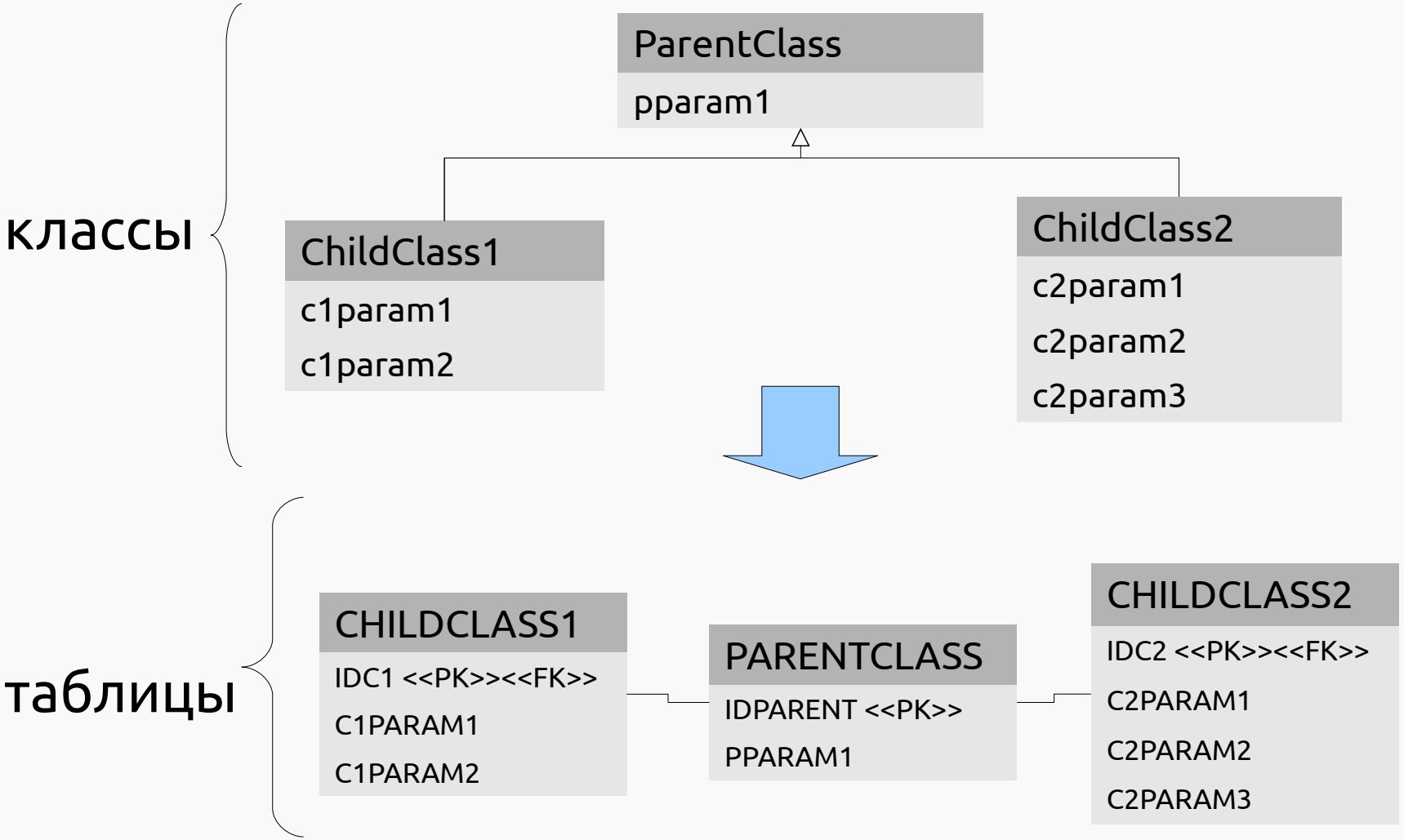
Concrete Table Inheritance pattern

Достоинства:
Можно накладывать ограничения (not null) на поля подклассов;

Недостатки:
1) Полученные таблицы не нормализованы;
2) Плохая поддержка полиморфных запросов;
3) Низкая производительность;

ИТМО ВТ

Наследование: Class Table Inheritance pattern



ИТМО ВТ

Concrete Table Inheritance pattern

Каждый подкласс отображается на отдельную таблицу, которая содержит колонки, соответствующие только полям этого подкласса.

Достоинства:

1) Таблицы нормализованы;
3) Существует возможность задавать ограничения на поля подклассов;

Недостатки:

1) Запросы выполняются медленнее, чем при использовании одной таблицы;

Как реализовать ORM на Java?

- JDBC;
- ORM-фреймворки (Hibernate, EclipseLink, …);
- Java Persistense API/Jakarta Persistence.

ИТМО ВТ

Hibernate ORM

ORM-фреймворк от Red Hat, разрабатывается с 2001 г.

Ключевые особенности:

- Таблицы БД описываются в XML-файле, либо с помощью аннотаций.
- 2 способа написания запросов — HQL и Criteria API.
- Есть возможность написания native SQL запросов.
- Есть возможность интеграции с Apache Lucene для полнотекстового поиска по БД (Hibernate Search).

ИТМО ВТ

EclipseLink ORM

ORM-фреймворк от Eclipse Foundation.

Ключевые особенности:

- Основан на кодовой базе Oracle TopLink.
- Является эталонной реализацией (reference implementation) для JPA.

ИТМО ВТ

Java Persistence API (JPA)

Java-стандарт (JSR 220, JSR 317), который определяет:

- как Java-объекты хранятся в базе;
- API для работы с хранимыми Java-объектами;
- язык запросов (JPQL);
- возможности использования в различных окружениях.

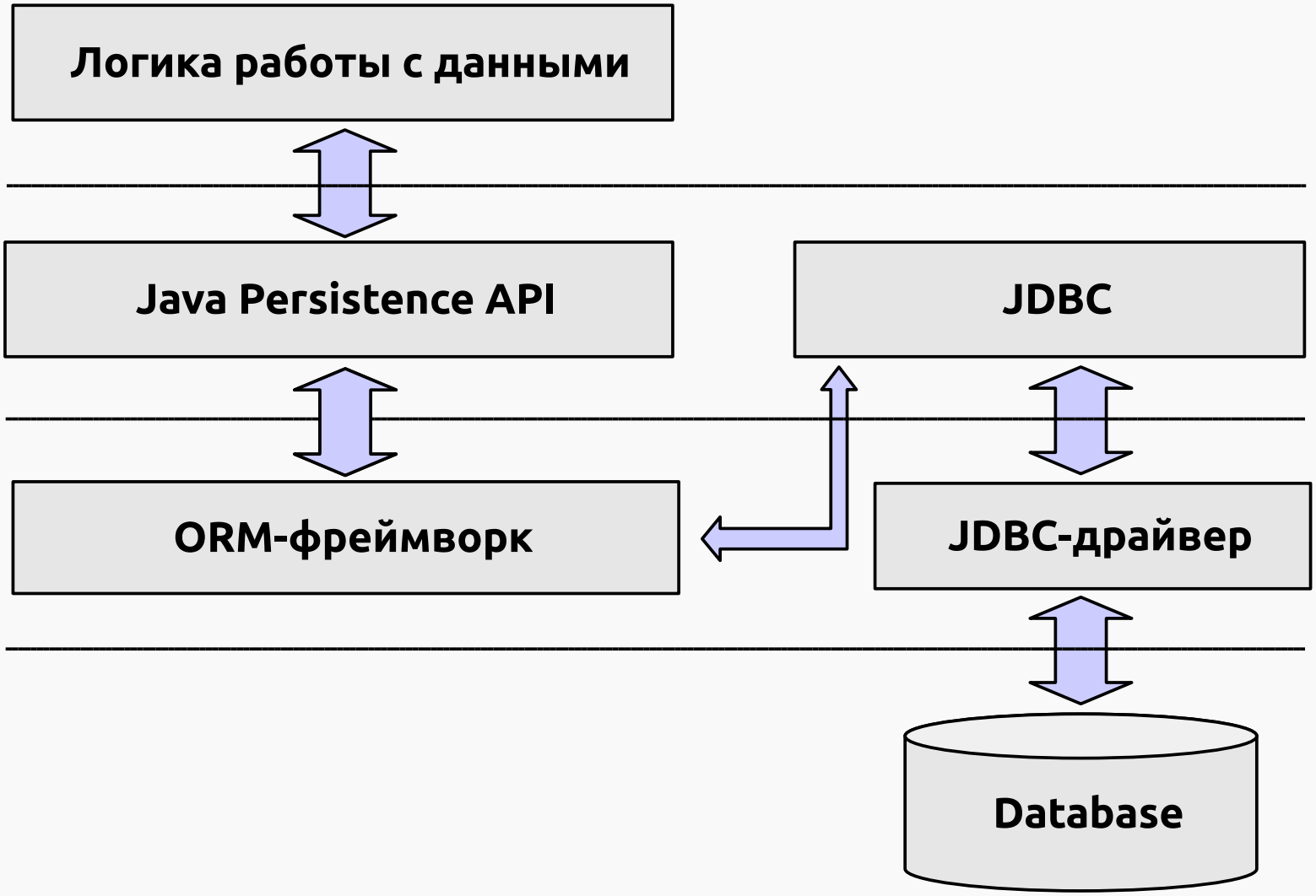
ИТМО ВТ

Что даёт использование JPA?

- Достижение лучшей переносимости.
- Упрощение кода.
- Сокращение времени разработки.
- Независимость от ORM-фреймворков.

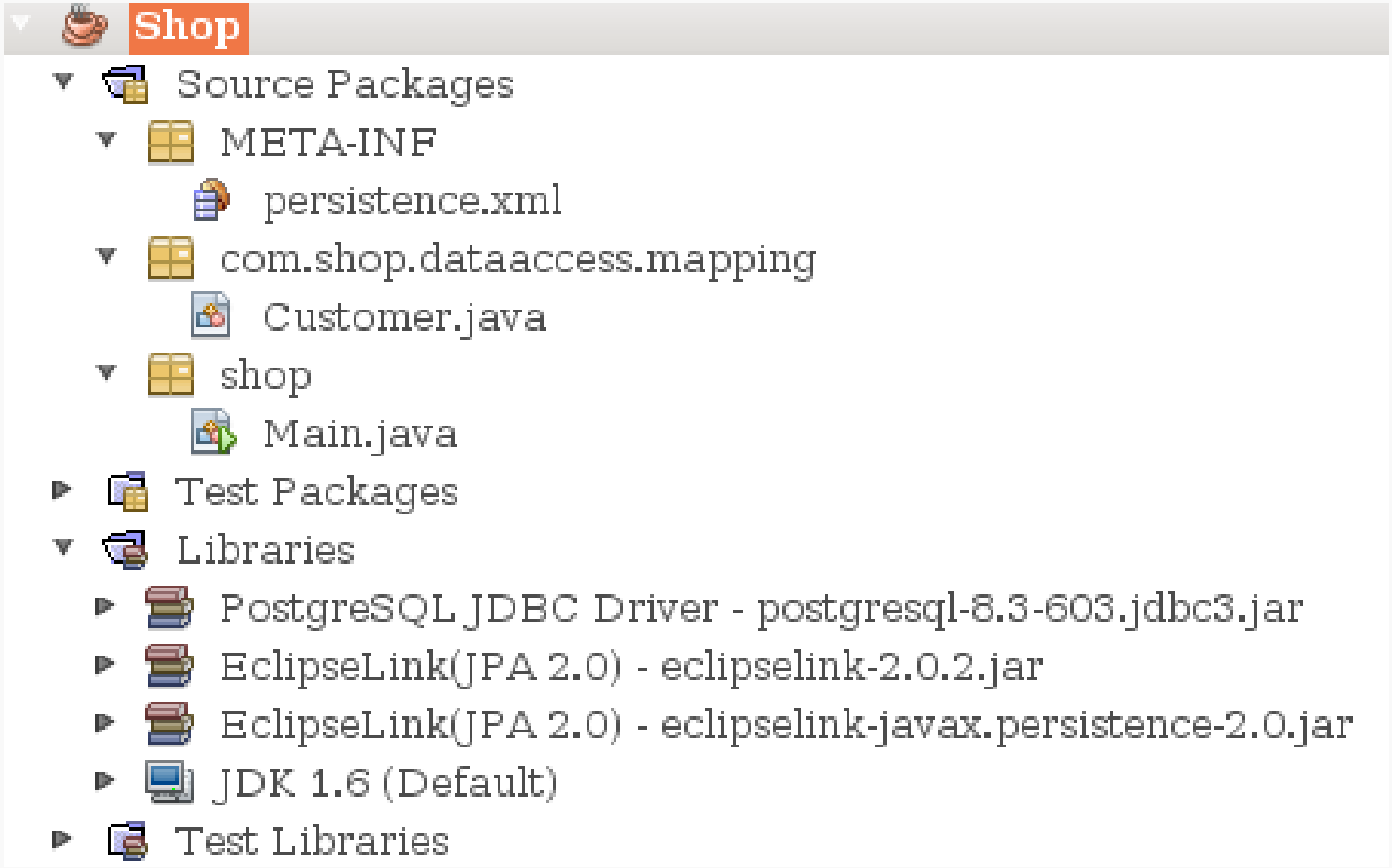
ИТМО ВТ

Взаимодействие приложения с БД через JPA



ИТМО ВТ

Использование JPA



ИТМО ВТ

Entity

Entity - простой Java класс (POJO), удовлетворяющий следующим требованиям:

- Не должен быть внутренним (inner).
- Не должен быть final.
- Не должен иметь final методов.
- Должен иметь public/protected конструктор без аргументов.
- Атрибуты класса не должны быть public.

ИТМО ВТ

Entity

Класс должен быть обозначен как Entity (2 способа):

1) @Entity annotation (jakarta.persistence.Entity):

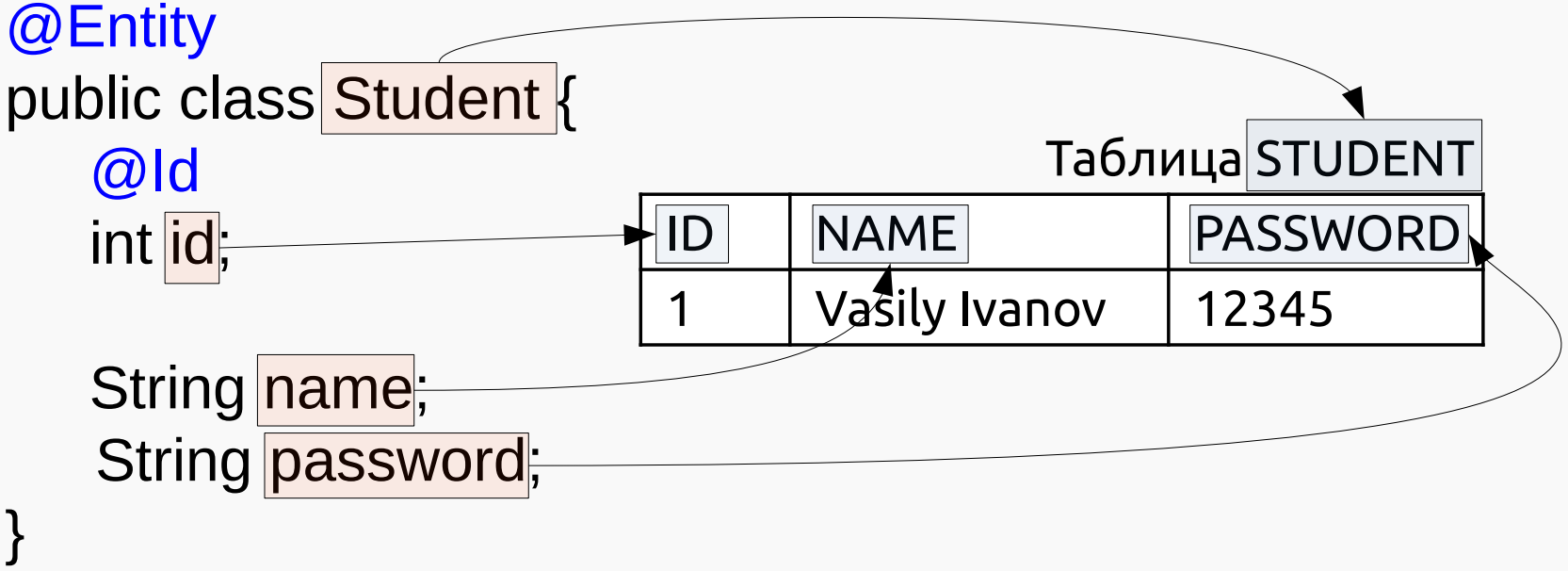
```
@Entity
public class Employee { ... }
```

2) Entity-параметр в XML mapping файле:

```
<entity class="com.project.jpa.entity.Employee"/>
```

Entity через аннотации

В классе должен быть объявлен идентификатор:



ИТМО ВТ

Jakarta persistence аннотации

- Jakarta Persistence аннотации определены в пакете jakarta.persistence.
- Две стратегии типа доступа: аннотации могут относиться к полям или соответствующим свойствам.

ИТМО ВТ

Смешанный тип доступа

```
@Entity @Access(AccessType.FIELD)
public class Student {
    @Id int id;
    String name;
    @Transient String password;

    @Access(AccessType.PROPERTY)
    protected String getPassword() {
        return password;
    }
    protected void setPassword(String password) {
        this.password = password;
    }
}
```

ИТМО ВТ

Идентичность

- Идентификатор (id) сущности, первичный ключ в БД.
- Уникально определяет сущность в памяти и БД.

1. Простой id
```
@Id int id;
```
2. Составной id
```
@Id String name;
@Id String login;
```
3. Embedded id
```
@EmbeddedId StudentPK id;
```

Должен быть Embeddable

ИТМО ВТ

@GeneratedValue

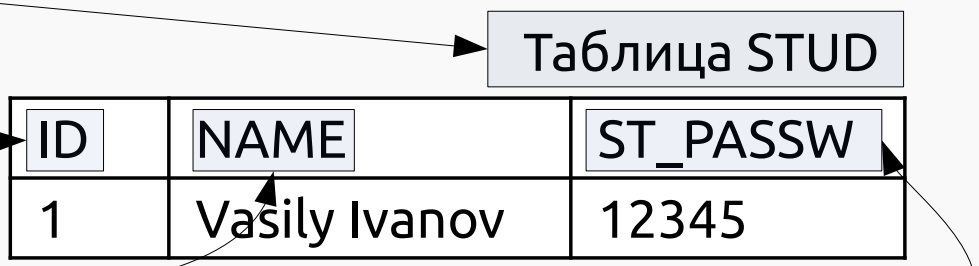
- Поддерживаются автогенерируемые значения первичных ключей.
- Стратегии, определенные GenerationType enum:
  - GenerationType.AUTO
  - GenerationType.IDENTITY
  - GenerationType.SEQUENCE
  - GenerationType.TABLE

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Integer id;
```

@Table, @Column

- @Table - отображение класса на таблицу.
- @Column - отображение полей класса на столбцы таблицы.

```
@Entity
@Table(name = "STUD")
public class Student {
    @Id
    int id;

    String name;
    @Column(name = "ST_PASSW")
    String password;
}
```



ИТМО ВТ

@Enumerated

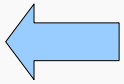
- Используется для определения стратегии хранения значений Java-перечислений (enum) в БД.
- Способы отображения:
  - EnumType.ORDINAL (по умолчанию)
  - EnumType.STRING

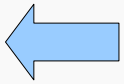
ИТМО ВТ

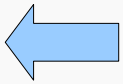
@Transient

- По умолчанию – все поля хранимые (persistent).
- Нехранимые поля – с модификатором transient (или аннотированы @Transient).

```
@Entity
public class EnumTest {
    @Id
    private int id;

    private transient String test1;
    @Transient
    private String test2;
}
```

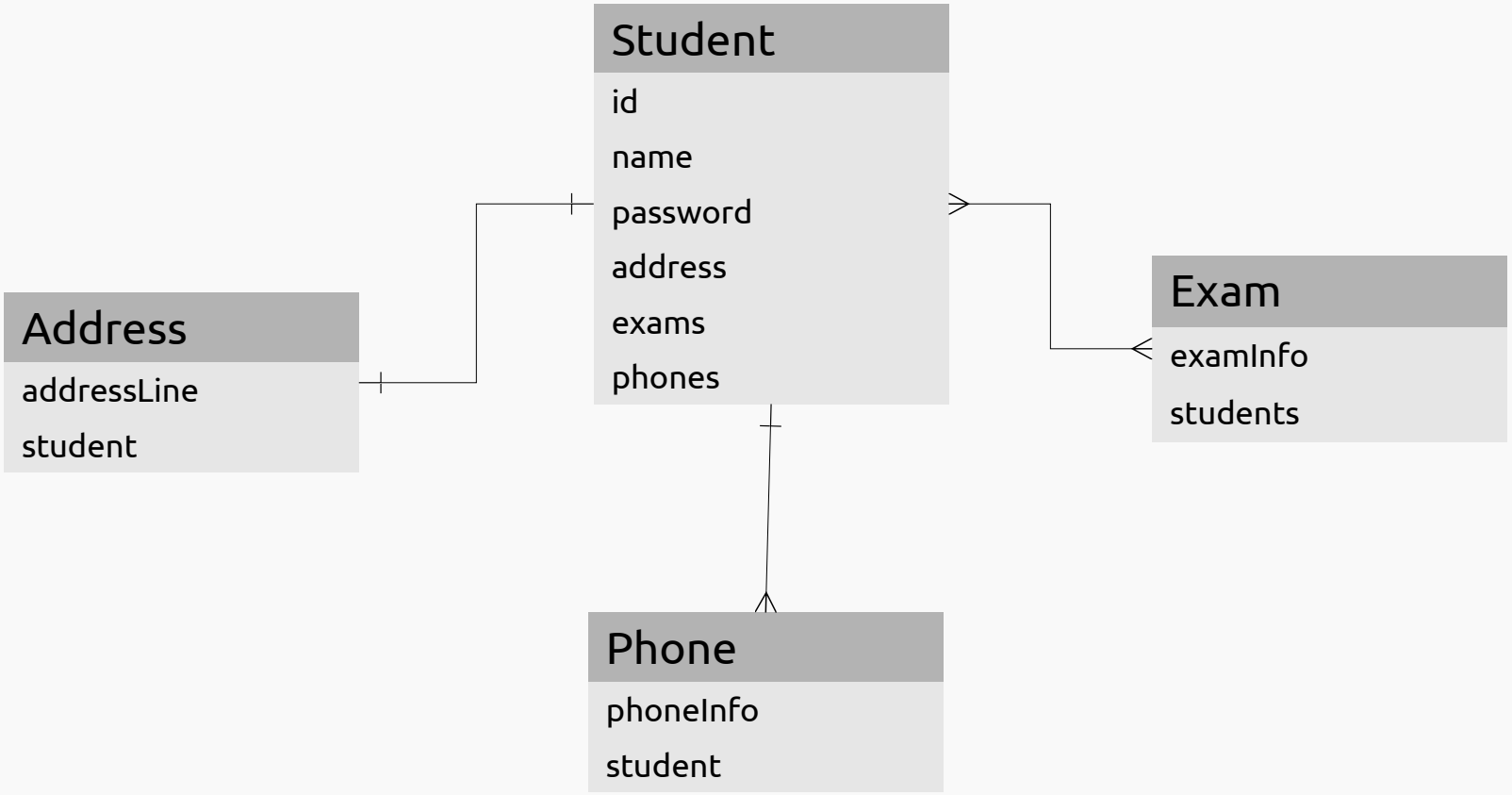
 хранимое

 нехранимое

 нехранимое

ИТМО ВТ

ER-модель



@OneToOne

```
@Entity
public class Student {
    @Id
    int id;
     /* other impl */
    @OneToOne(mappedBy="student")
     Address address;
}
```

Таблица STUDENT

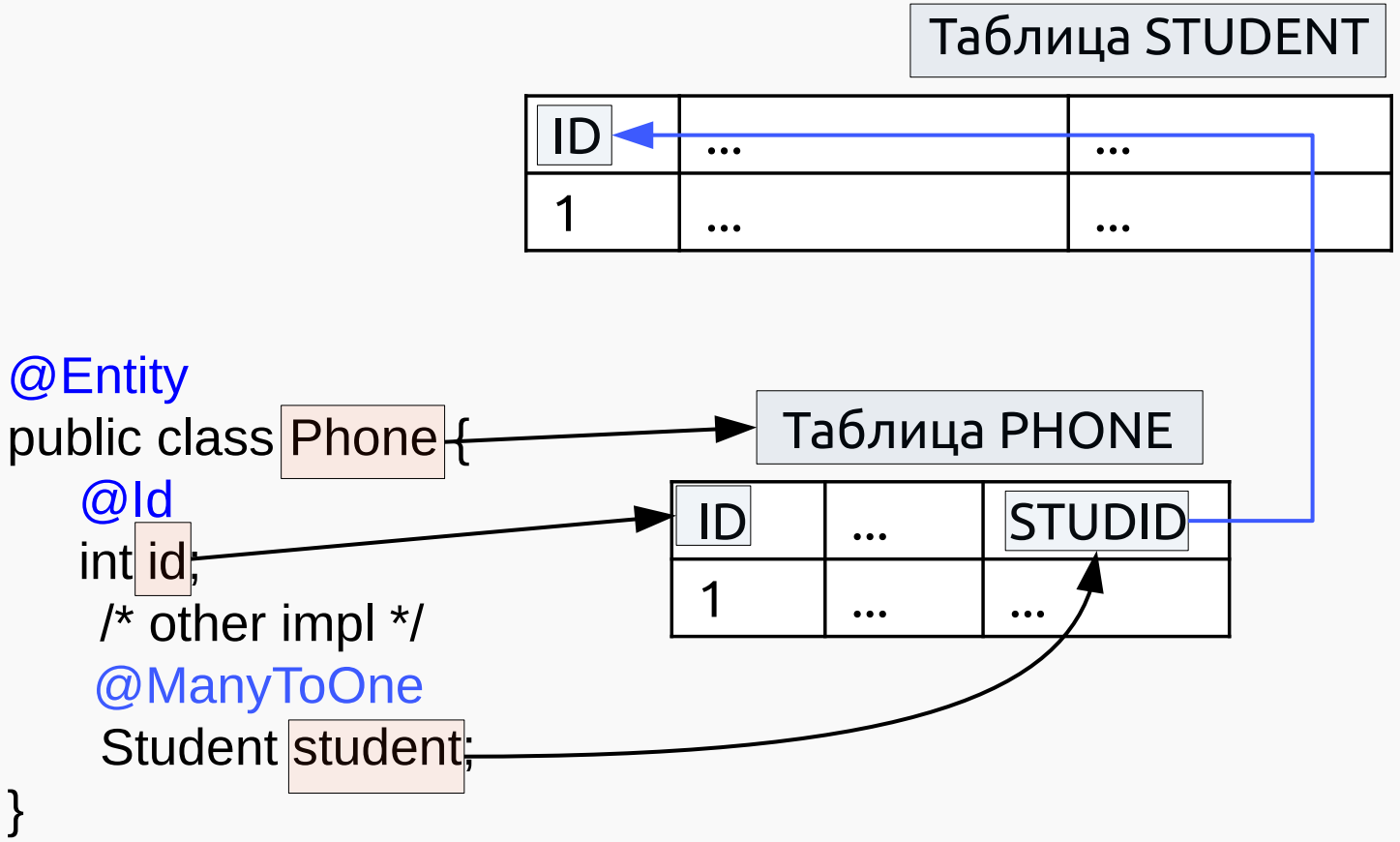
ID	...	...
1	...	...

```
@Entity
public class Address {
    @Id
    int id;
     /* other impl */
    @OneToOne
    @JoinColumn(name="STUDID")
     Student student;
}
```

Таблица ADDRESS

ID	...	STUDID
1	...	...

@ManyToOne



“many” сторона – главная (владеющая) сторона связи;

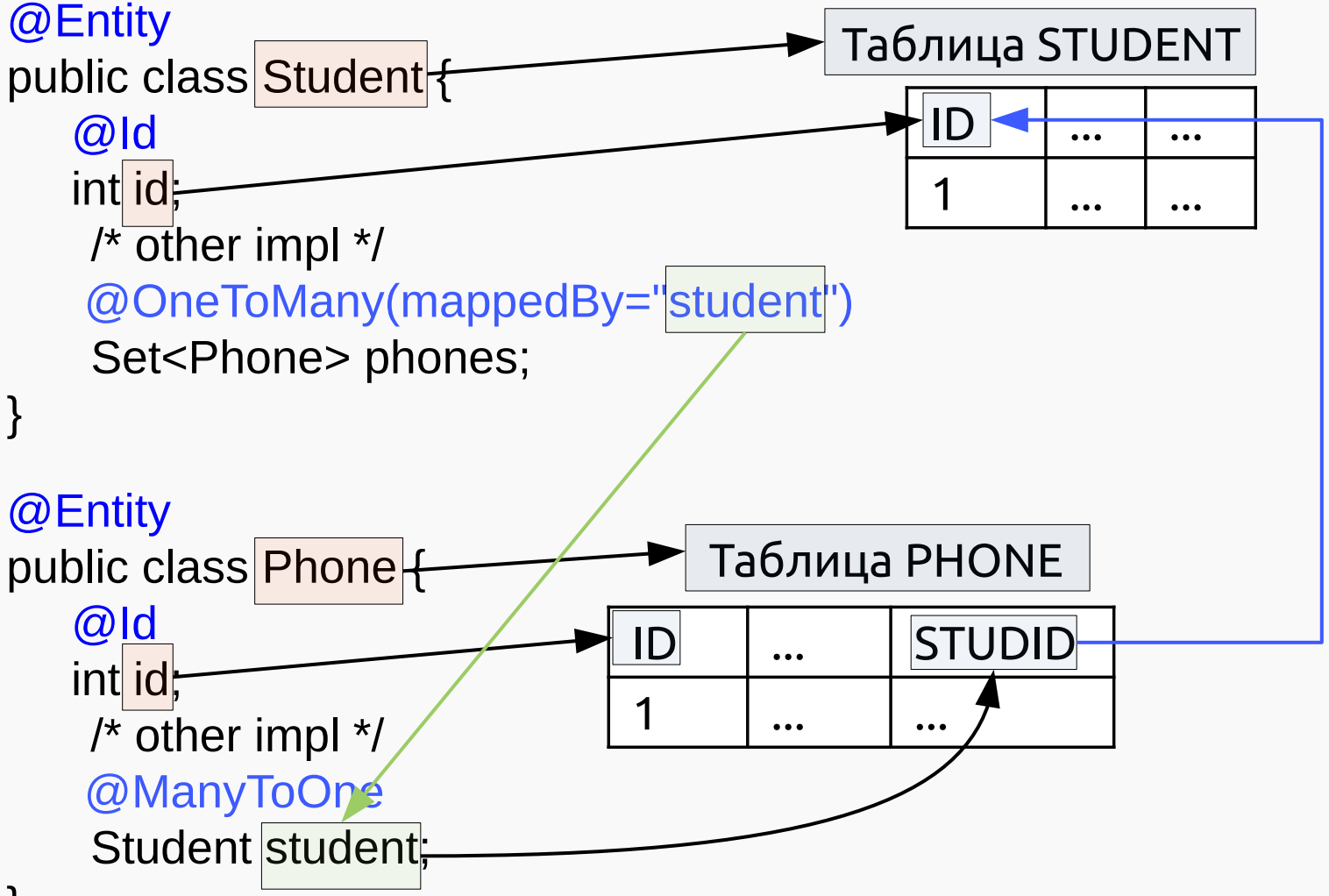
ИТМО ВТ

@OneToMany

- @OneToMany определяет “one” сторону отношения one-to-many.
- Элемент аннотации mappedBy определяет ссылку, используемую стороной “many”.
- @JoinColumn позволяет настроить отображение внешних ключей.
- “many” сторона представляется одной из реализаций java.util.Collection.
- @OrderBy определяет порядок сортировки, необходимый при получении коллекции.

@OneToMany

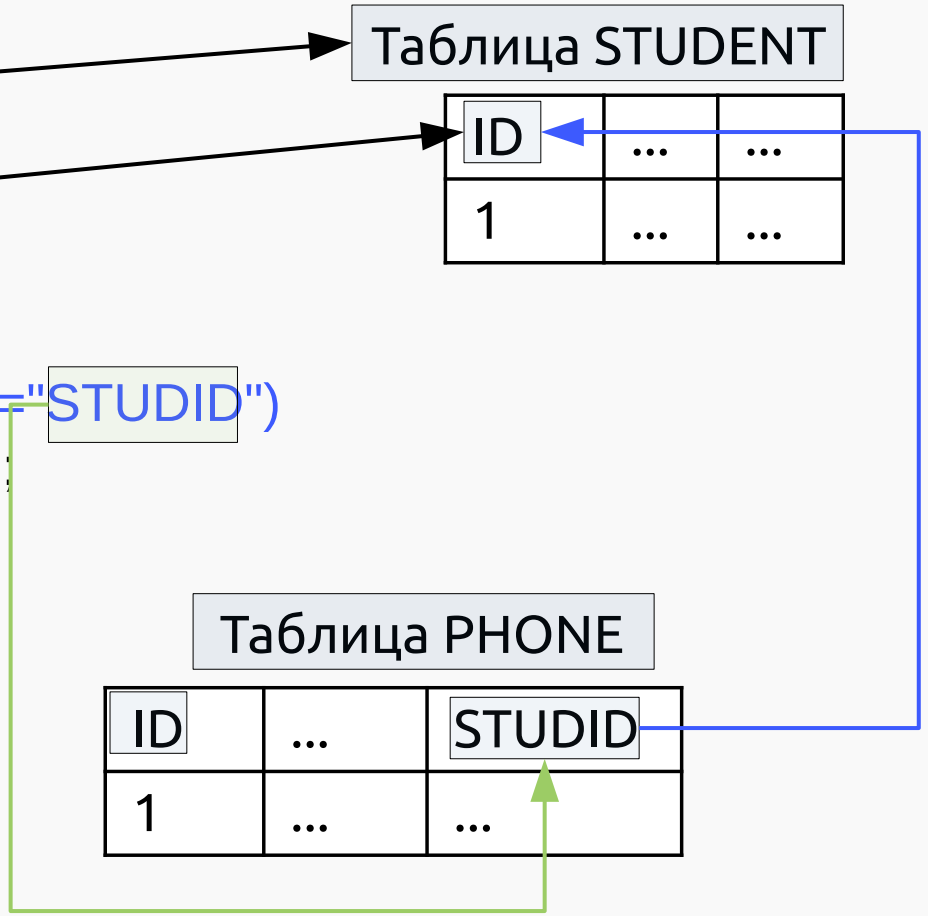
```
@Entity
public class Student {
    @Id
    int id;
     /* other impl */
     @OneToMany(mappedBy="student")
     Set<Phone> phones;
}

@Entity
public class Phone {
    @Id
    int id;
     /* other impl */
     @ManyToOne
     Student student;
}
```



@OneToMany (с JPA 2.0)

Теперь можно задать однонаправленный OneToMany с использованием @JoinColumn

```
@Entity
public class Student {
    @Id
    int id;
     /* other impl */
     @OneToMany
     @JoinColumn(name="STUDID")
     Set<Phone> phones;
}
```



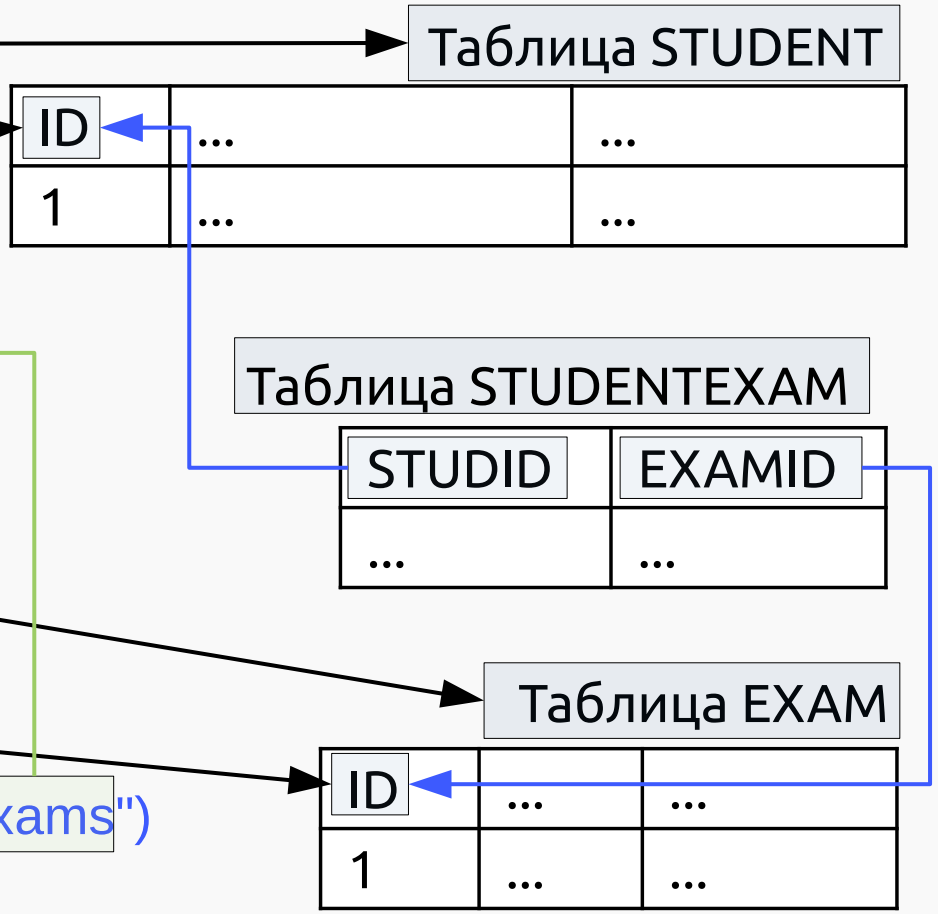
ИТМО ВТ

@ManyToMany

- Аннотация @ManyToMany определяется с каждой стороны отношения.
- Каждая Entity, участвующая в отношении, содержит коллекцию противоположной сущности.
- @JoinTable ставится на владеющей (главной) стороне отношения.
- Владеющая сторона отношения many-to-many условна, определяется произвольно.
- @JoinColumn используется для определения владеющей и обратной колонок таблицы соединений.

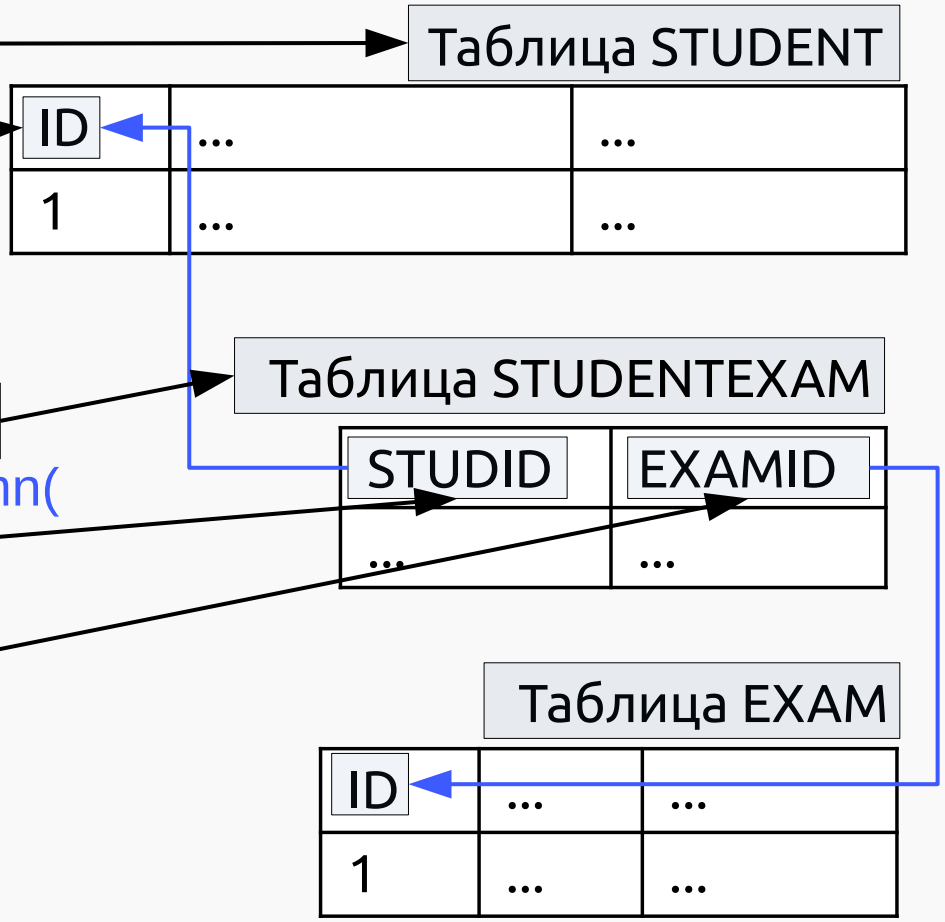
@ManyToMany

```
@Entity
public class Student {
    @Id
    int id;
     /* other impl */
    @ManyToMany
    Collection<Exam> exams;
}
@Entity
public class Exam {
    @Id
    int id;
     /* other impl */
    @ManyToMany(mappedBy="exams")
    Collection<Student> students;
}
```



@ManyToMany

```
@Entity
public class Student {
    @Id
    int id;
     /* other impl */
    @ManyToMany
    @JoinTable(
            name="STUDENTEXAM",
            joinColumns=@JoinColumn(
                name="STUDID"),
            InverseJoinColumns(
                name="EXAMID"))
    Collection<Exam> exams;
}
```



ИТМО ВТ

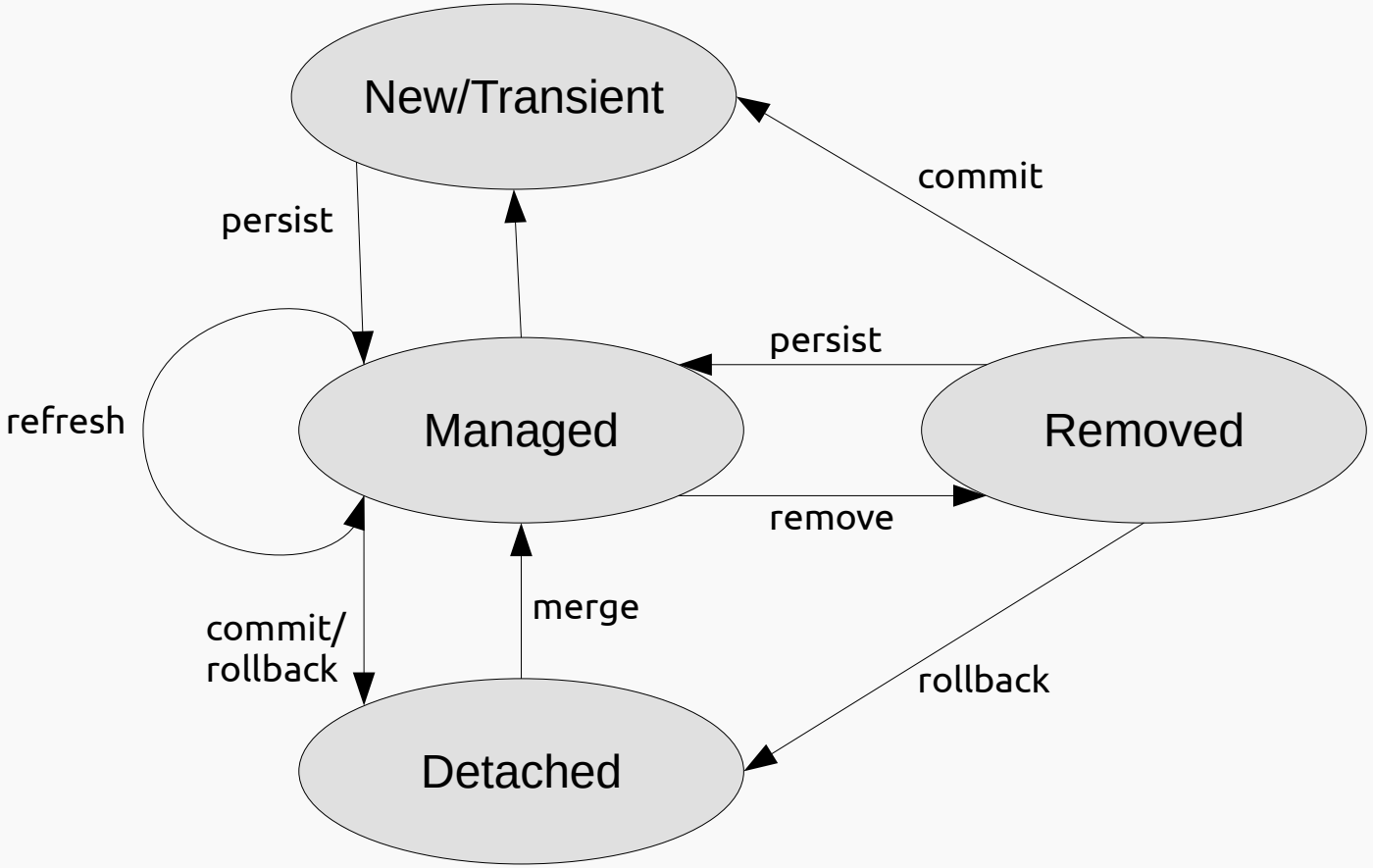
Fetch

Объекты-аттрибуты могут быть загружены или получены (fetch) как EAGER или LAZY:

- LAZY – заставляет JPA откладывать загрузку аттрибутов (сущностей), пока к полю явно не обращаются.
- EAGER – означает то, что поле или отношение будет загружено вместе с объектом, содержащим это поле или отношение.

ИТМО ВТ

Жизненный цикл Entity



ИТМО ВТ

EntityManager

Базовый интерфейс для работы с хранимыми данными:

- Обеспечивает взаимодействие с Persistence Context;
- Можно получить через EntityManagerFactory.
- Обеспечивает базовые операции для работы с данными (CRUD).

ИТМО ВТ

Persistence Context

- Persistence Context – абстрактное представление множества управляемых объектов-сущностей;
- PC контроллируется и управляется EntityManager;
- Содержимое PC изменяется в результате операций, производимых EntityManager API;
- PC может принадлежать нескольким EntityManager;

ИТМО ВТ

Persistence Unit

Persistence Unit содержит:

- Информацию о Persistence Context;
- Настройки источника данных;

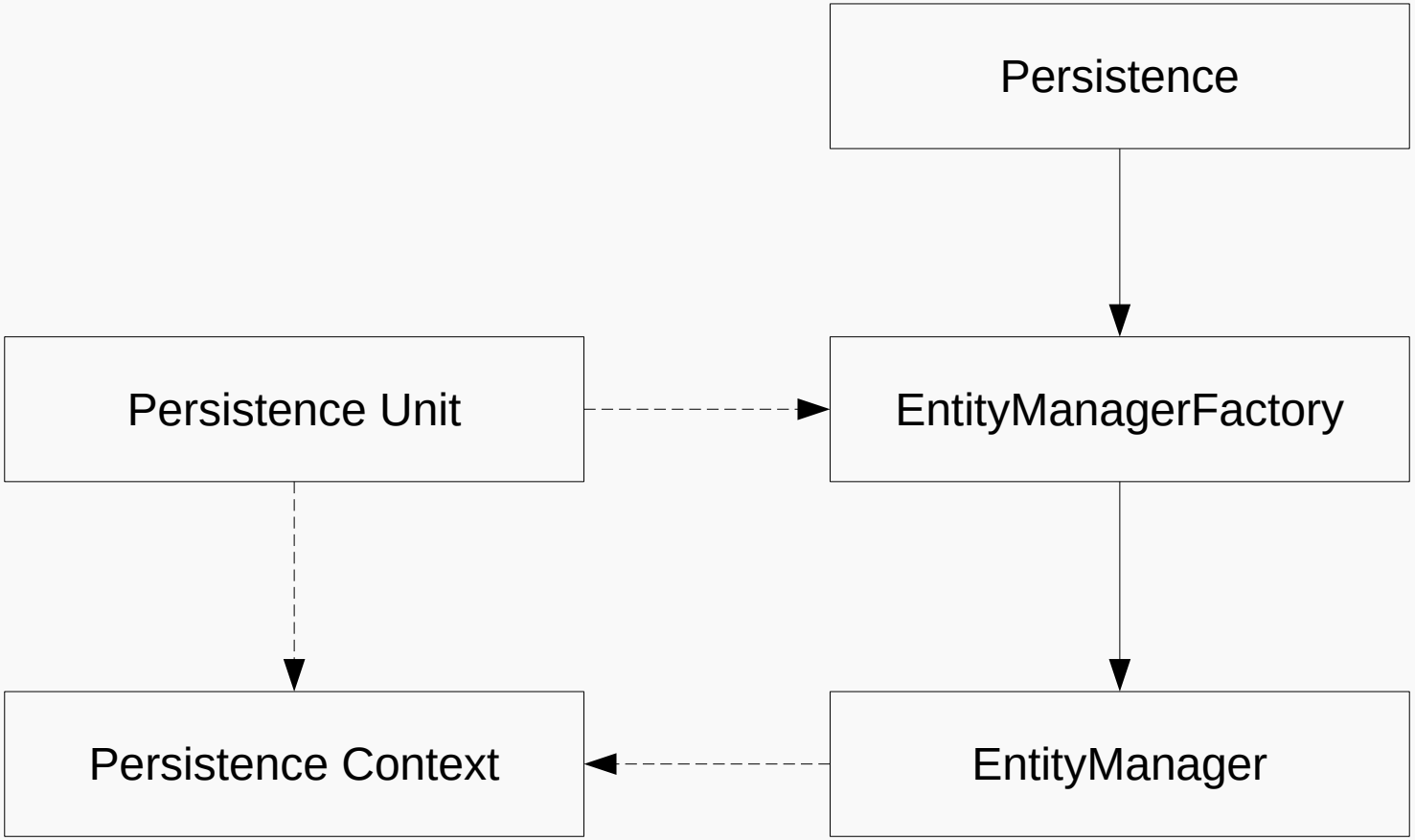
Persistence Unit связан с одной EntityManagerFactory и всеми EntityManagers, созданными ей;

persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
  <persistence-unit name="jpa2SamplePU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>jpa2sample.entity.Employee</class>
    <class>jpa2sample.entity.TelephoneEntity</class>
    <properties>
      <property name="javax.persistence.jdbc.url"
                value="jdbc:postgresql://localhost:5432/jpaBase"/>
      <property name="javax.persistence.jdbc.password"
                value="postgres"/>
      <property name="javax.persistence.jdbc.driver"
                value="org.postgresql.Driver"/>
      <property name="javax.persistence.jdbc.user"
                value="postgres"/>
      <property name="eclipselink.ddl-generation" value="create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```

ИТМО ВТ

JPA


Persistence
Persistence Unit
EntityManagerFactory
Persistence Context
EntityManager

ИТМО ВТ

EntityManager

EntityManager:

- Container-managed EntityManager.
- Application-managed EntityManager:
  - приложение само получает и закрывает PC, когда необходимо.
  - Application-managed API может быть использовано как в Java EE, так и в Java SE приложениях.

ИТМО ВТ

Базовые классы и интерфейсы

jakarta.persistence.Persistence

- Корневой класс для получения Entity Manager (Java SE environment)
- Определяет Persistence Provider для данного Persistence Unit
- Позволяет получить EntityManagerFactory

jakarta.persistence.EntityManagerFactory

- Отвечает за создание EntityManagers для заданного Persistence Unit или конфигурации

ИТМО ВТ

persist()

- Добавляет новый экземпляр Entity в БД.
- Сохраняет состояние Entity и относящихся к нему ссылок.
- Делает экземпляр Entity управляемым PC.

```
public Student createStudent(int id, String name) {
    Student stud = new Student(id, name);
    entityManager.persist(stud);
    return stud;
}
```

ИТМО ВТ

find(), remove()

- find() - получает управляемый экземпляр Entity (по идентификатору) – возвращает null, если заданный объект не найден.

- remove() - удаляет управляемый Entity. Опционально производит каскадное удаление отмеченных объектов.

```
public Student removeStudent(int studId) {
    Student stud = entityManager.find(Student.class, studId);
    entityManager.remove(stud);
}
```

ИТМО ВТ

flush()

Синхронизирует записи базы данных с управляемыми Entity. Также происходит создание новых записей и удаление существующих (соответственно с изменениями в PC).

```
public Student updateStudentName(int studId, String name) {
    Student stud = entityManager.find(Student.class, studId);
    stud.setName(name);
    entityManager.flush();
    return stud;
}
```

ИТМО ВТ

refresh()

Синхронизирует управляемую Entity в PC с содержимым БД (Entity возвращается к состоянию, в котором находится соответствующая ему запись в БД);

```
public Student refreshStudent(int studId) {
    Student stud = entityManager.find(Student.class, studId);
    entityManager.refresh(stud);
    return stud;
}
```

EntityManager (app-managed)

```
public class EMTest {
    public static void main(String[] args) {
        EntityManagerFactory emFactory =
            Persistence
                .createEntityManagerFactory("studentPU");
        EntityManager entityManager =
                emFactory.createEntityManager();
        entityManager.getTransaction().begin();
        //операции над сущностями
        entityManager.getTransactioin().commit();
        entityManager.close(); emFactory.close();
    }
}
```

EntityManager (container-managed)

```
@Named
public class StudentService {

   @PersistenceContext
   private EntityManager em;

   public Student findStudentOrder(Integer studId) {
       try {
           Student stud = em.find(Student.class, studId);
           return stud;
        }
    }

}
```

ИТМО ВТ

Entity Lifecycle Callbacks

- @PrePersist
- @PostPersist
- @PreRemove
- @PostRemove
- @PreUpdate
- @PostUpdate
- @PostLoad

Вызовы “Post” и PreUpdate могут произойти во время операции с данными или во время успешного завершения транзакции (commit).

ИТМО ВТ

JPA запросы

- Поддерживаются статические и динамические запросы;
- Запросы могут быть написаны на SQL или JPQL
- Поддерживается передача именованных и позиционных параметров
- Поддерживается eager доступ при использовании fetch;

ИТМО ВТ

JPA запросы

Для создания запроса:

- createQuery()
- createNamedQuery()
- createNativeQuery()

Для получения результата:

- getSingleResult()
- getResultList()

ИТМО ВТ

Динамические запросы

- Обеспечивают наибольшую гибкость при задании и исполнении
- Для создания используется метод EM createQuery(), запрос передается в качестве аргумента;

```
public List findAll(String entityName) {
    return entityManager.createQuery (
        "select e from " + entityName + " e")
        .getResultList();
}
```

ИТМО ВТ

Именованные запросы

- Запрос должен быть статически определен в аннотации или XML;
- Для создания используется метод EntityManager createNamedQuery();

```
@NamedQuery (name = "Address.findByAddrId",
                    query = "select  a from Address a
                                where a.student.id = :studId)
public List findAddressByStudentId(Integer studentId) {
    return entityManager.createNamedQuery (
        "Student.findByAddrId")
        .setParameter("studId", studentId)
        .getResultList();
}
```

ИТМО ВТ

Native queries

- Можно использовать SQL в динамических или именованных запросах. Persistence Provider отобразит результат на сущности;
- Во время выполнения используются методы createNativeQuery()/createNamedQuery();

```
Query query = em.createNativeQuery(
        "SELECT DISTINCT s.id, s.name, s.password "
        + "FROM student s "
        + "join address a on s.id = a.stud_id "
        + "WHERE (a.id > 1000) ", Student.class)
    query.getResultList();
```

ИТМО ВТ

Criteria API

Характерные черты:

- Объектно-ориентированное API для построения запросов.
- Есть возможность отобразить любой JPQL-запрос в Criteria.
- Поддерживает построение запросов в runtime.

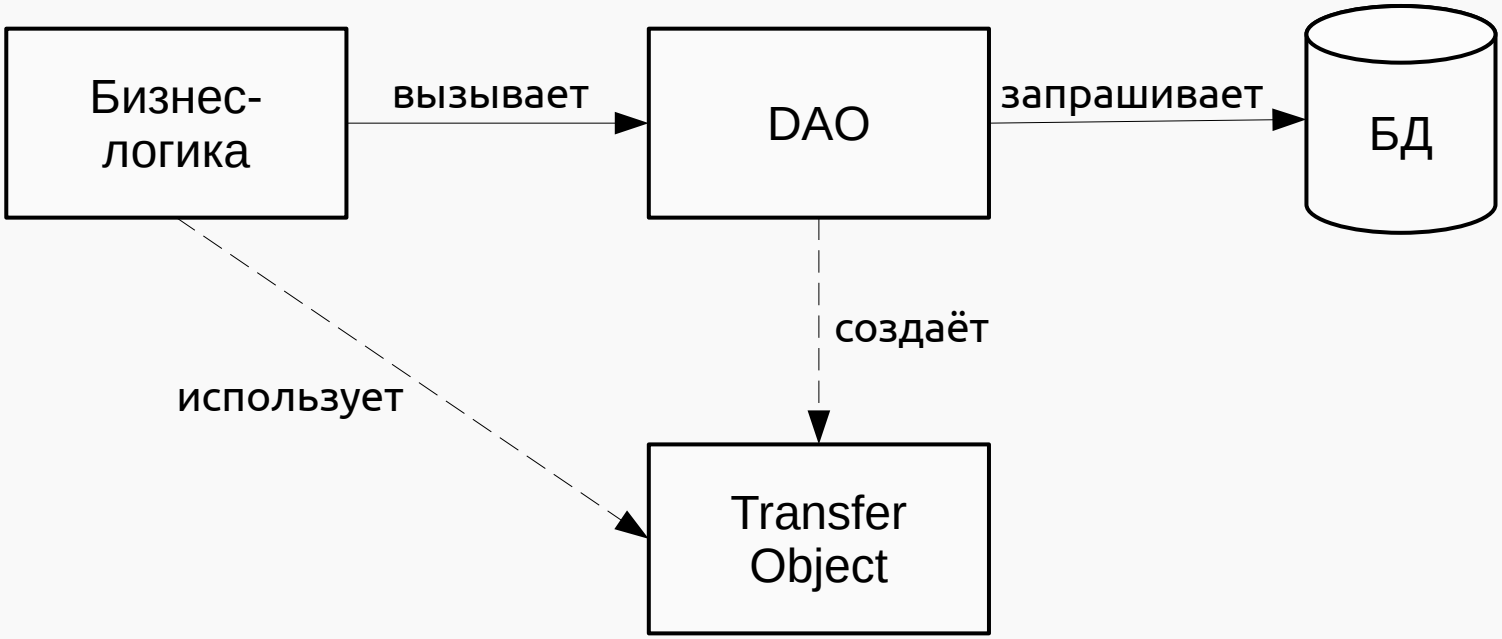
ИТМО ВТ

Динамические запросы

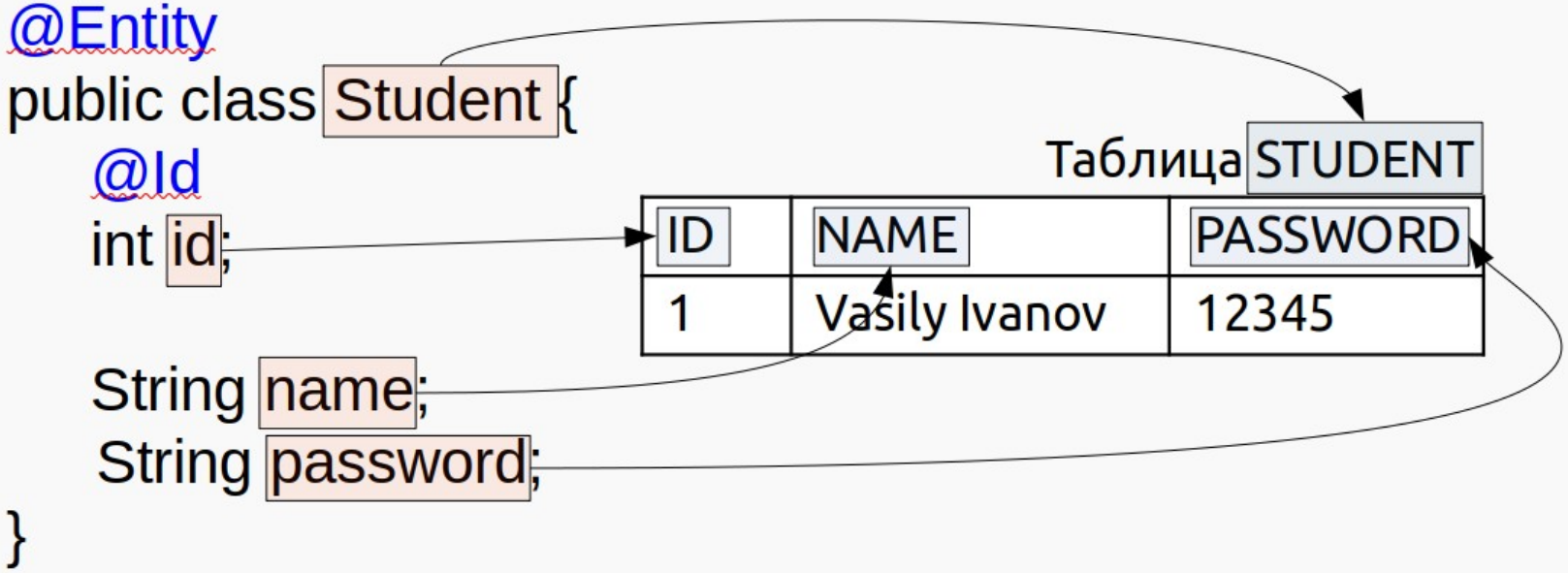
```
EntityManager em = ... ;
CriteriaBuilder queryBuilder = em.getCriteriaBuilder();
CriteriaQuery qdef = queryBuilder.createQuery();

Root student = qdef.from(Student.class);
qdef.select(student)
   .where(queryBuilder.equal(student.get("address"), addr));
    /* addr – сравниваемый объект Address (передан через параметр метода), в котором этот код*/
```

ИТМО ВТ

DAO


Бизнес-
логика
вызывает
DAO
запрашивает
БД
создаёт
использует
Transfer
Object

Student Entity



ИТМО ВТ

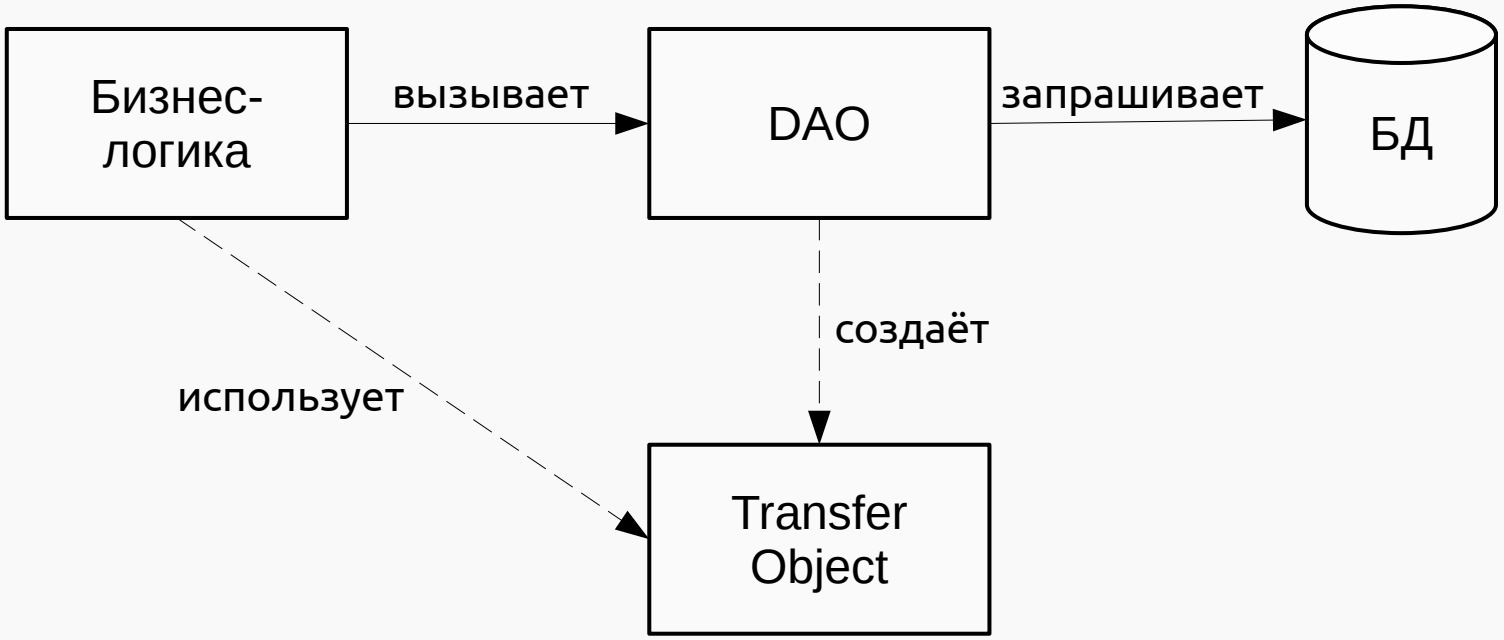
Student DAO

```
@Named
public class StudentDAO {

   @PersistenceContext
   private EntityManager em;

   public Student findStudent(Integer studId) {
      try {
         Student s1 = em.find(Student.class, studId);
         return s1;
       }
   }
   // другие методы
}
```

1.2. Jakarta Data

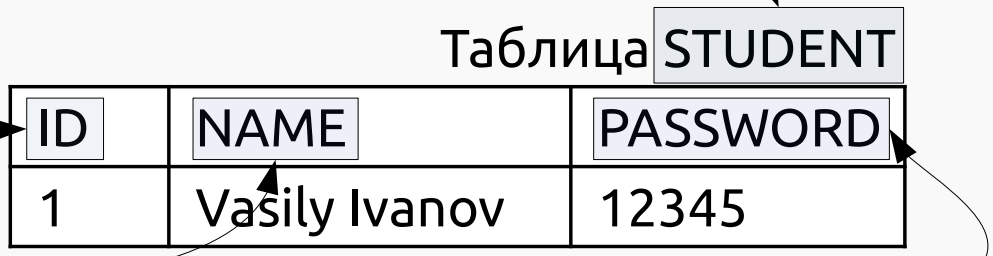
ИТМО ВТ

Взаимодействие с уровнем хранения



Student Entity

```
@Entity
public class Student {
    @Id
    int id;

    String name;
    String password;
}
```



ИТМО ВТ

Repository

```
@Named
@SessionScoped
public class StudentBean {
    @Inject
    Students students;

    ...
}

@Repository
public interface Students { }
```

Репозиторий, чтобы можно совершать операции над данными

BasicRepository

```
@Named
@SessionScoped
public class StudentBean {
    @Inject
    Students students;

    public void studAction(Integer id) {
        customers.findById(id);
        /* other actions */ }
}

@Repository
public interface Students
```

```
delete(T entity)
deleteAll(List<? extends T> entities)
deleteById(K id)
findAll()
findAll(PageRequest pageRequest, Order<T> sortBy)
findById(K id)
save(S entity)
saveAll(List<S> entities)
```

extends **BasicRepository**<Student, Integer> { }

Запросы

```
@Repository
public interface Students
                        extends BasicRepository<Student, Integer> {

    boolean existByName(String name);

    Customer findByNameAndPassword(String name, String password);
}
```