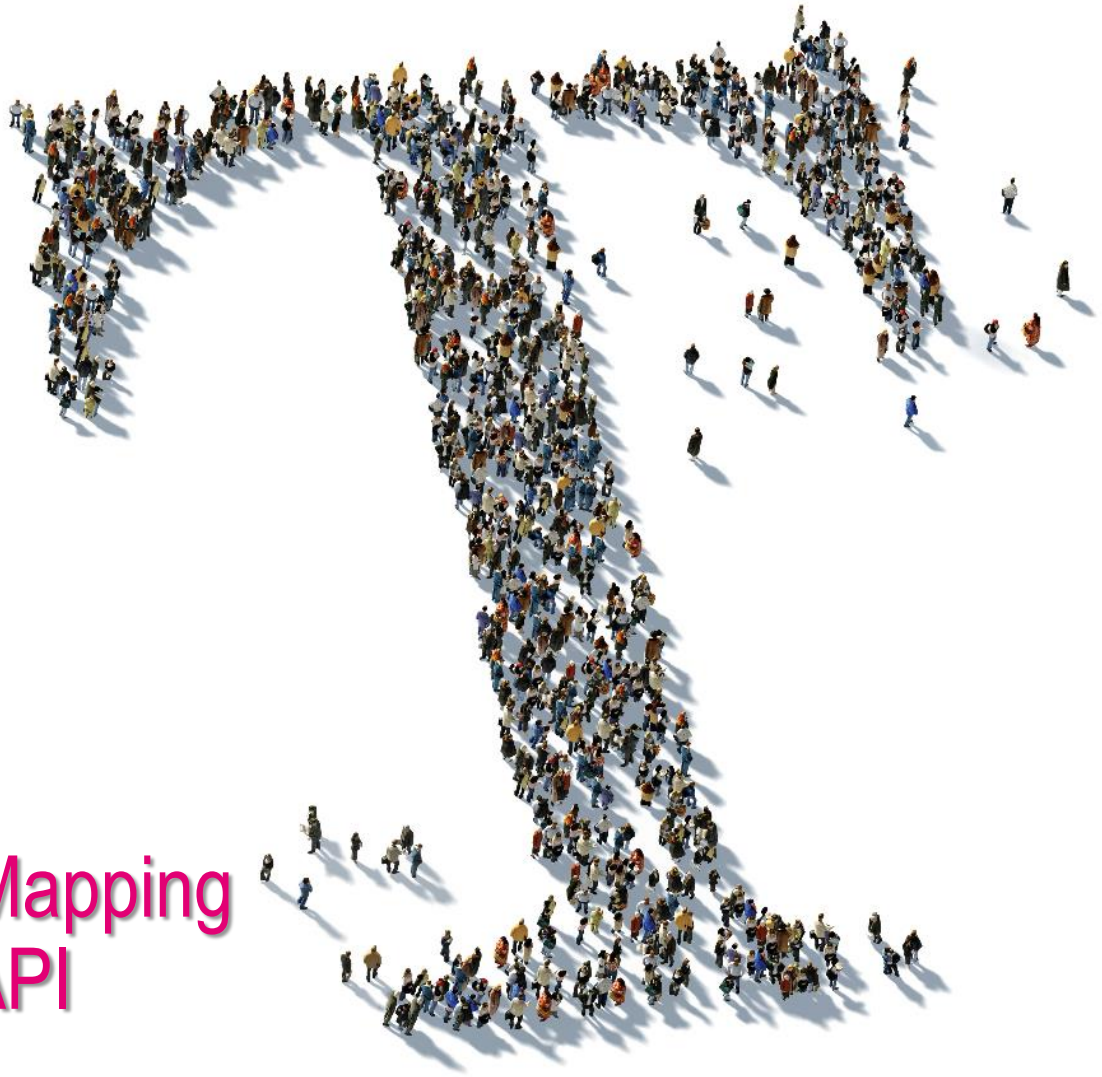


Object-Relational Mapping
Java Persistence API

Voronezh, 2018

T · · Systems

Содержание

- Что такое **O**bject-**R**elational **M**apping и зачем он нужен?
- Как использовать ORM в Java?
- **J**ava **P**ersistence **A**PI

JDBC Overview

1. **Load driver** or obtain data source
2. **Establish connection** using a JDBC URL
3. **Create statement**
4. **Execute statement**
5. Optionally, **process results** in result set
6. **Close database** resources
7. Optionally, **commit** or **rollback transaction**

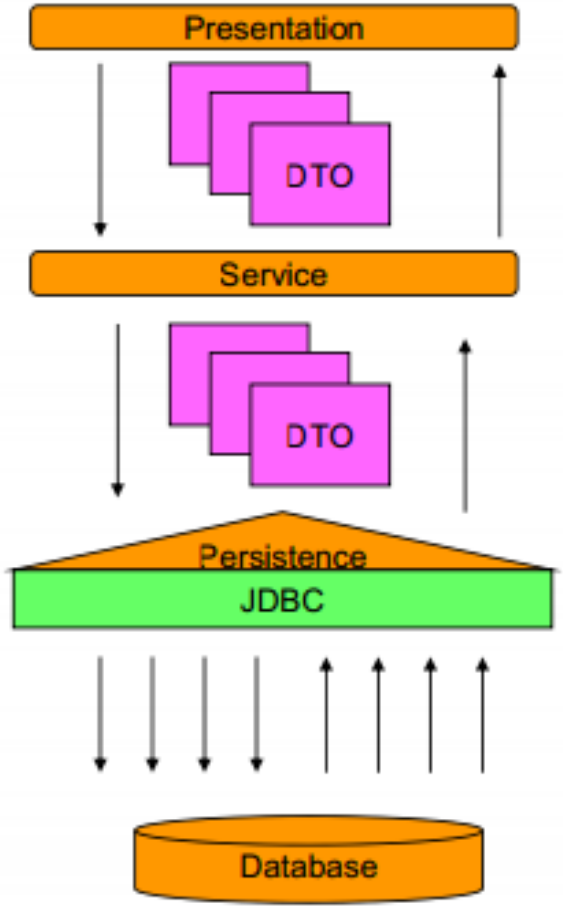
Boiler code

```
public static void main(String[] args) {
    try {
        Connection connection = DriverManager
                .getConnection("jdbc:derby:Console;create=true");
        Statement statement = null;
        statement = connection.createStatement();
        statement.execute("SELECT first_name, last_name FROM persons");
        ResultSet resultSet = null;

        resultSet = statement.getResultSet();
        while (resultSet.next()) {
            String fName = resultSet.getString("first_name");
            System.out.println(resultSet.wasNull() ? "(null)" : fName);
        }
    } catch (SQLException e) {
        // Handle exception thrown while trying to get a connection
    }
}
```

T · · Systems

N-Tier Architecture



T · · Systems

Проблема

- Согласно ООП, объекты программы = объекты реального мира.

```
class Person{
   private String name;
   private List<String> phoneList;
   private List<String> addressList;
}
```

- Такие объекты должны быть преобразованы в форму, в которой они могут быть сохранены в БД

ID	Имя	Телефон	Адрес
1	John	+7 9998887777	St.Petersburg, ...
2	Moritz	+49 9998887777	Berlin, ...

T · · Systems

Зачем нужен ORM? (1)

- Java objects ←→ database tables
- JDBC – потенциальный источник **ошибок** и слишком **сложен**
- Возможности современного ORM:
  - **Прозрачное** хранение объектов (JavaBeans)
  - **Транзитивное** хранение
  - **Проверка** изменения данных (dirty checking)
  - **Inheritance** mapping
  - **Lazy** fetching
  - Outer **join** fetching
  - Runtime **SQL generation**

Зачем нужен ORM? (2)

- **Естественная модель объектов**

- Более **компактный код**, поэтому:
  - Быстрее пишется
  - Проще читается и поддерживается

- Код может быть **протестирован вне контейнеров**

- Классы могут быть **повторно использованы** в non-persistent context

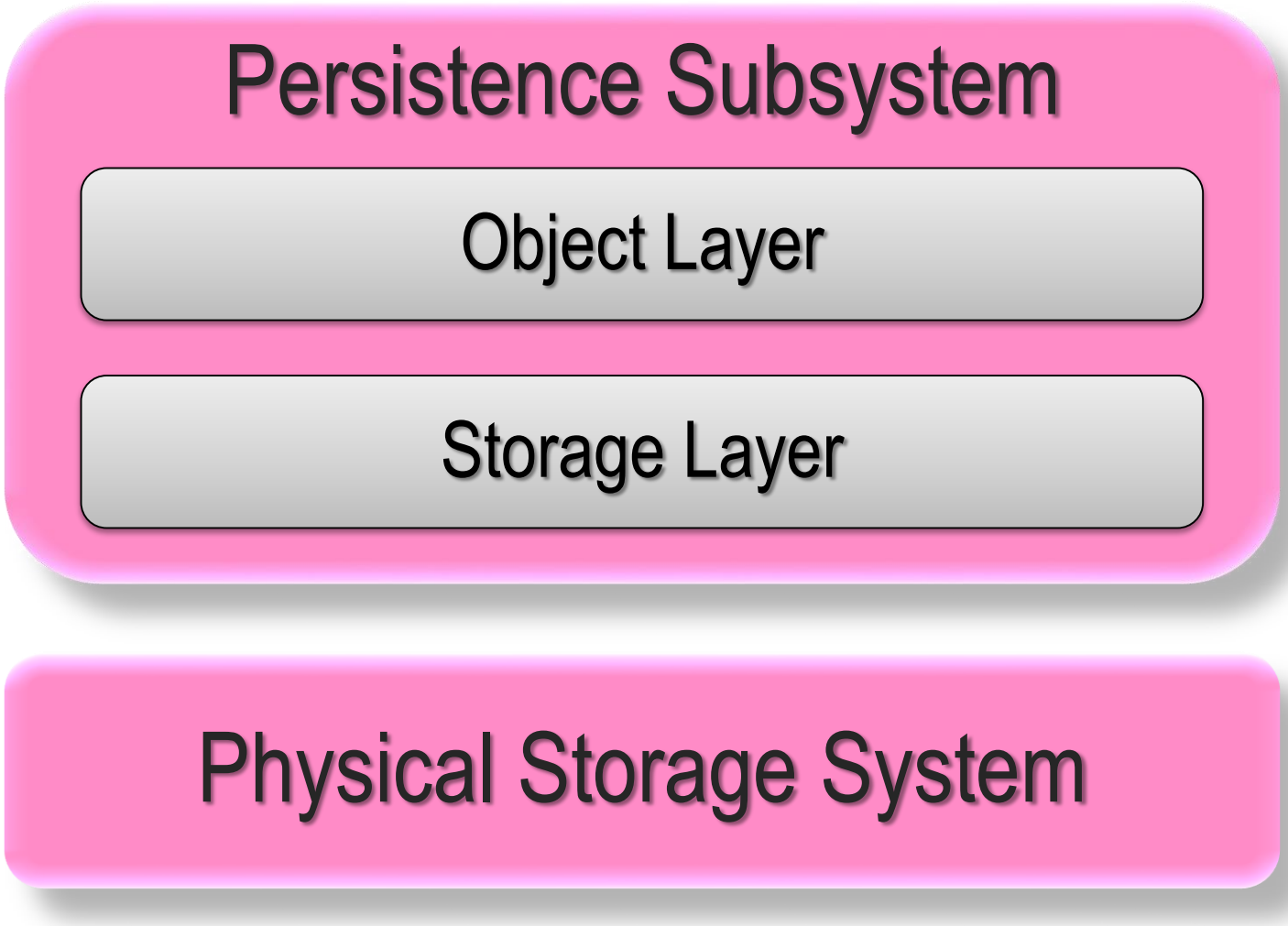
- **Оптимальные запросы** к БД (smart fetching strategies)

- Возможности **кеширования**

- Бóльшая **гибкость при изменении структуры данных/объектов**

T Systems

ORM Structure



T · · Systems

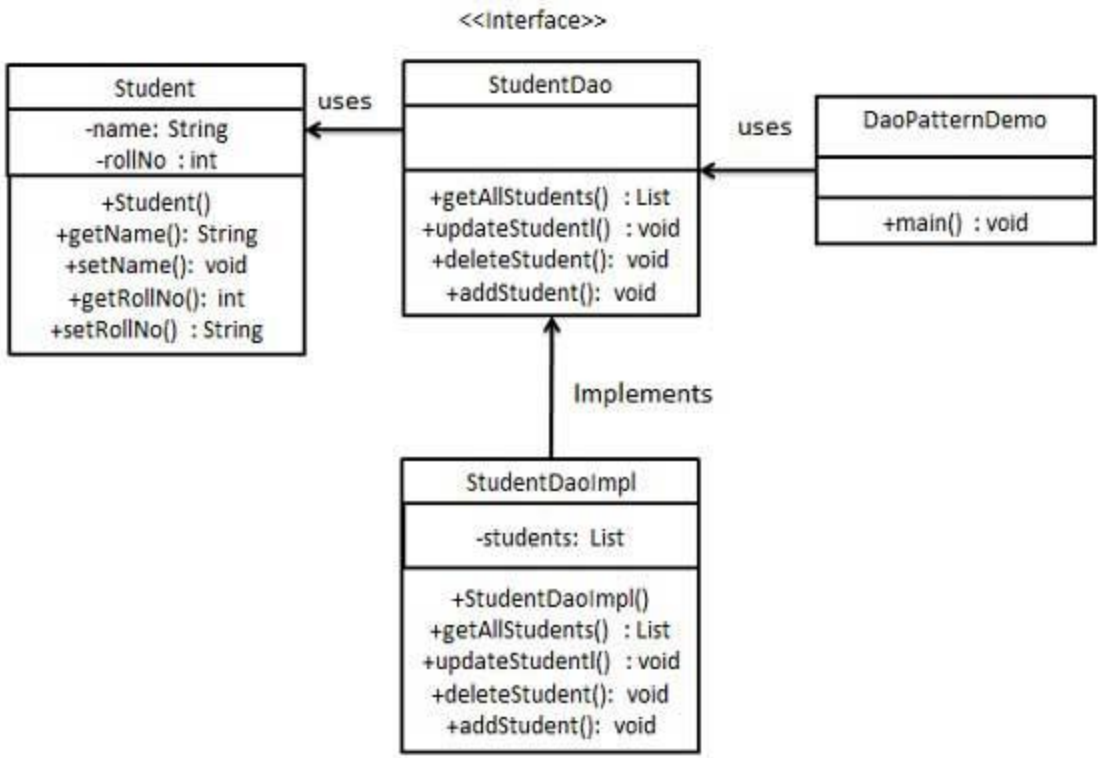
ORM в Java

- Использование **дополнительных библиотек** и **шаблонов** проектирования.

- Например, для Java:
  - EclipseLink JPA
  - Enterprise JavaBeans Entity Beans
  - Java Data Objects
  - Castor
  - TopLink
  - Spring DAO
  - **Hibernate**
  - …

Data Access Object Pattern

- **D**ata **A**ccess **O**bject **Pattern** is used to separate low level data accessing API or operations from high level business services.



Data Access Object

- **DAO** объект, предоставляющий **абстрактный интерфейс** к какому-либо типу базы данных или механизму хранения.

```
interface MyDao {
   List<String> getAllNames();
}
```

- Can be used in a large percentage of applications - **anywhere data storage is required**.
- **Hide all details of data storage** from the rest of the application.
- Act as an **intermediary between the application and the database**. They move data back and forth between objects and database records.

Что такое JPA? (1)

- **J**ava **P**ersistence **A**PI (javax.persistence)
- **Entity**
- A **specification** for generic ORM
  - Not an implementation
  - Vendor-neutral
- Based on **Annotations**
- Developed out of **EJB 3 Specification** (since 2006):
  - Replacement for train wreck EJB 2 persistence.
  - Standalone specification (does not require JEE container).
- **JPQL**:
  - **SELECT** human **FROM** Human human **WHERE** human.name = 'John';

Что такое JPA? (2)

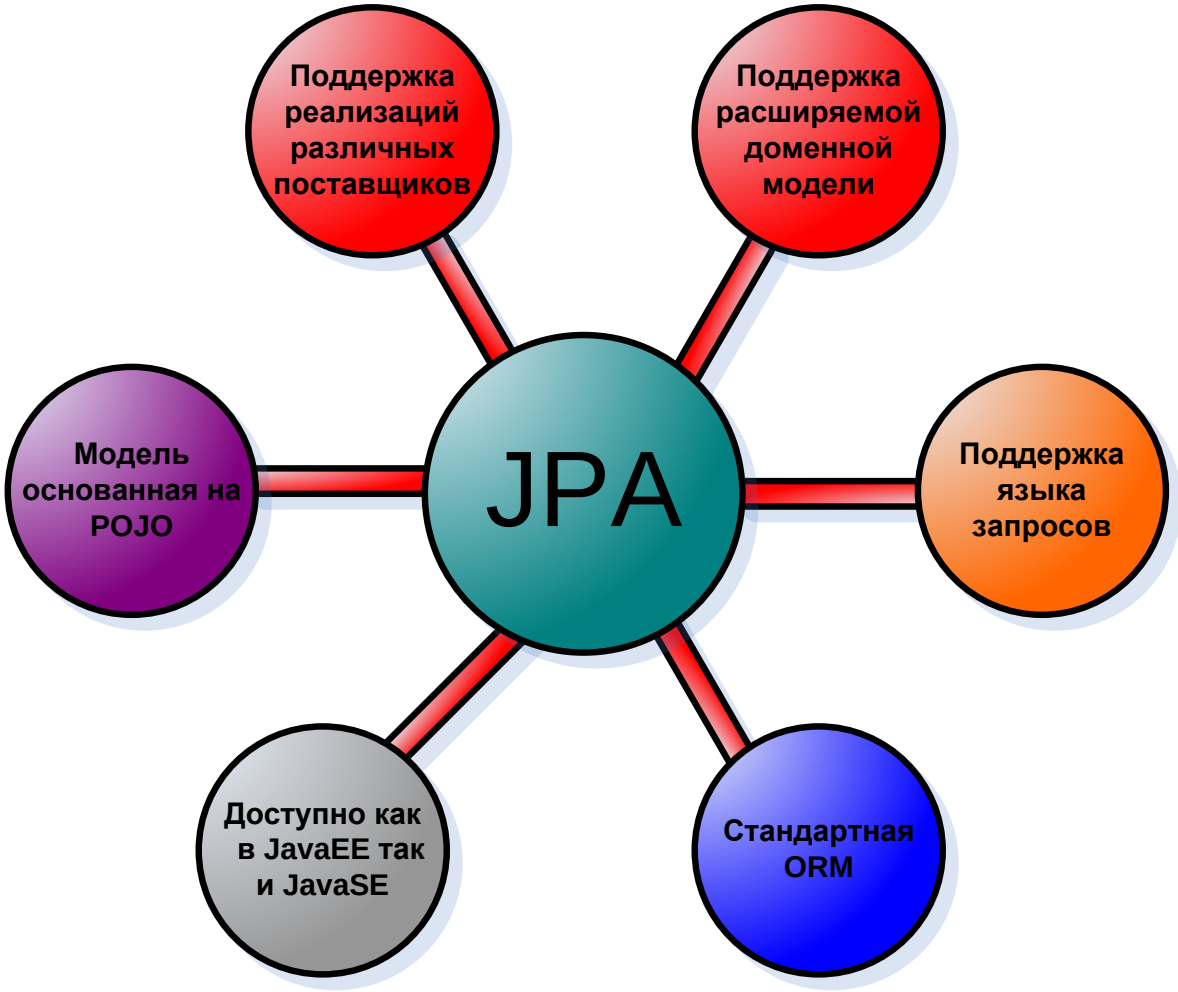
- **Java Persistence Architecture API** (JPA) это Java спецификация для доступа, сохранения и манипулирования данными между Java объектами и реляционной базой данных.

- JPA было представлено как часть **спецификации EJB 3.0**.

- 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**.

Особенности JPA



T · · Systems

Entities (1)

- **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 **аннотированы** 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 удаленном бизнес интерфейсе, то она должна **реализовывать интерфейс Serializable**.
  - Энтити **могут расширять** как другие энтити, так и обычные **классы**.

Entities (2)

- **Состояние объекта храниться в полях и свойствах обьекта**, которые могут быть:
  - Примитивные типы и их обертки Java.
  - Строки.
  - Любые сериализуемые типы Java (реализующие Serializable интерфейс).
  - Enums.
  - Entity классы.
  - Embeddable классы – это класс который не используется сам по себе, только как часть одного или нескольких Entity классов.
  - Коллекции.

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

EntityManager (1)

- **EntityManager** – то интерфейс, который описывает API для всех основных операций над Entity, получение данных и других сущностей JPA.

- По сути главный API для работы с JPA.

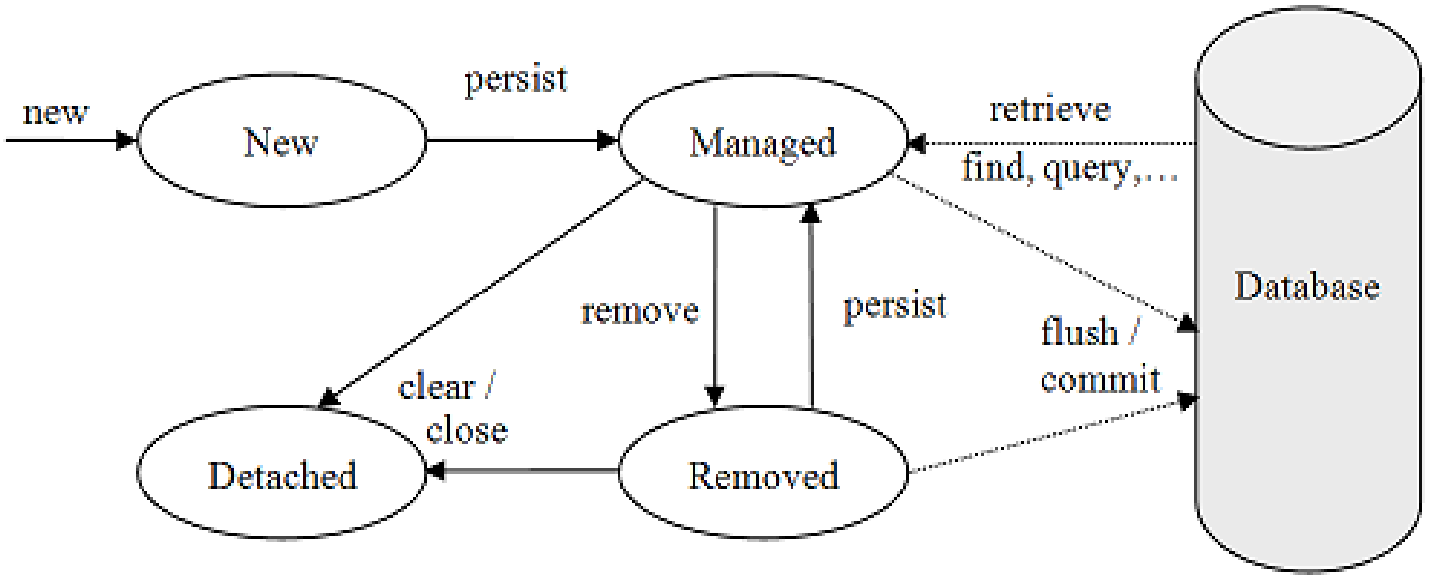
- Основные операции:
  - Для операций над Entity:
    - **persist** (добавление Entity под управление JPA),
    - **merge** (обновление),
    - **remove** (удаления),
    - **refresh** (обновление данных),
    - **detach** (удаление из управление JPA),
    - **lock** (блокирование Entity от изменений в других thread).

EntityManager (2)

- Основные операции:
  - Получение данных:
    - **find** (поиск и получение Entity),
    - **createQuery**, **createNamedQuery**, **createNativeQuery**, contains, createNamedStoredProcedureQuery, createStoredProcedureQuery.
  - Получение других сущностей JPA:
    - **getTransaction**, **getEntityManagerFactory**, **getCriteriaBuilder**, getMetamodel, getDelegate.
  - Работа с EntityGraph:
    - createEntityGraph, getEntityGraph.
  - Общие операции над EntityManager или всеми Entities:
    - close, isOpen, getProperties, setProperty, clear.

T · · Systems

Entity Lifecycle

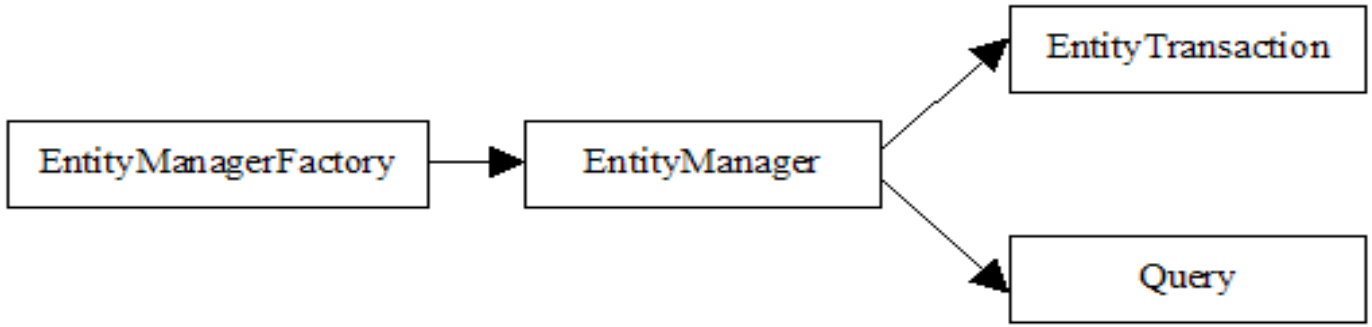


Типы EntityManager

- Существуют два типа EntityManager :
  - **Container-Managed Entity Managers** – контролируется автоматически J2EE контейнером и создается при помощи Dependency Injection (DI). Использует JTA для работы с транзакциями.

```
@PersistenceContext
EntityManager em;
```

  - **Application-Managed Entity Managers** – контролируется пользовательским приложением и создается при помощи фабрики.



Application-Managed Entity Managers

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("yourApplication");
EntityManager em = emf.createEntityManager();
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 (1)

- The **persistence.xml** file is responsible for all **JPA environment configuration**; it can hold **database**, **application** and **JPA** implementation specific **configuration**.

- The persistence.xml file must be **located in** a **META-INF** folder in the same path as the Java classes.

Persistence Units (2)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
    xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">

    <persistence-unit name="jpaHibernateMaven" transaction-type="RESOURCE_LOCAL">

        <provider>org.hibernate.ejb.HibernatePersistence</provider>

        <class>ru.tsystems.jpa.Course</class>
        <class>ru.tsystems.jpa.Student</class>
        <class>ru.tsystems.jpa.Subject</class>
        <class>ru.tsystems.jpa.Teacher</class>
        <class>ru.tsystems.jpa.TeacherCourse</class>
        <class>ru.tsystems.jpa.TeacherReview</class>
        <class>ru.tsystems.jpa.User</class>

        <properties>
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/tuni" />
            <property name="javax.persistence.jdbc.user" value="tuni" />
            <property name="javax.persistence.jdbc.password" value="Qazxsw23" />
            <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />
        </properties>

    </persistence-unit>
</persistence>
```

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

JPA Annotations

- @Entity
- @Table
- @Column
- @Id
- @GeneratedValue
- @Transient
- @JoinColumn
- @JoinTable
- @OneToOne
- @OneToMany
- @ManyToOne
- @ManyToMany
- . . .

JPA Annotation Rules

- Any persistent class must be **annotated with @Entity** (or inherit from a class that does)
- A **public constructor without arguments**
- All persistent classes must have a **field annotated by @Id** signifying the primary key
- All instance **fields** in a persistent class are assumed to be **mapped to columns** with the same name as the field:
  - @Transient will remove a field from mapping
- **Relationships** are **NOT automatically mapped**:
  - Relationships are modeled as aggregate members
  - Require an @OneToOne, @OneToMany, @ManyToOne, or @ManyToMany

T · · Systems

JPA Defaults

- These annotations will create the **relationship between** the **database table and** the **entity**:
  - Class annotation **@Table** defines specific **table mappings**
    - Name
  - Field annotation **@Column** defines specific **column mappings**
    - Name
    - Nullability
    - Size
    - Uniqueness

Primary Keys (1)

- Каждая Entity должна иметь уникальный идентификатор (**первичный ключ**), например:
  - **Простой первичный ключ**: примитивный тип, обертки, String, Date, BigDecimal, BigInteger.

```
@Id private long id;
```

  - **Композитный первичный ключ**

```
@Entity
@IdClass(EmployeeId.class)
public class Employee {
        @Id
        private int id;
        @Id
        private int hrNumber;
        . . .
}
```

```
public class EmployeeId implements Serializable {
        private int id;
        private int hrNumber;
        public EmployeeId() { }
        @Override
        public int hashCode() { . . . }
        @Override
        public boolean equals(Object obj) { . . . }
}
```

Primary Keys (2)

- **Embedded первичный ключ** должен иметь конструктор по умолчанию, переопределять hashCode() и equals(Object other), реализовывать интерфейс Serializable.

```
@Entity
public class Employee {
        @EmbeddedId
        private EmployeeId id;
        . . .
}
```

```
@Embeddable
public class EmployeeId implements Serializable {
        private int id;
        private int hrNumber;
        public EmployeeId() { }
        @Override
        public int hashCode() { . . . }
        @Override
        public boolean equals(Object obj) { . . . }
}
```

Identifier Generation (1)

- Applications can choose one of four different id generation strategies by specifying a strategy in the strategy element:

  - **AUTO** - If an application does not care what kind of generation is used by the provider but wants generation to occur. This is the default value.

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

  - **IDENTITY** – The database controls the ID generation, JPA does not act on the id at all. Thus in order to retrieve the id from the database an entity needs to be persisted first, and after the transaction commits, a query is executed so as to retrieve the generated id for the specific entity.

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

Identifier Generation (2)

- **TABLE** – The most flexible and portable way to generate identifiers is to use a database table. Not only will it port to different databases but it also allows for storing multiple different identifier sequences for different entities within the same table.

```
@TableGenerator(name="Emp_Gen", table="ID_GEN",
        pkColumnName="GEN_NAME", valueColumnName="GEN_VAL")
@Id
@GeneratedValue(generator="Emp_Gen")
private long id;
```

- **SEQUENCE** – Many databases support an internal mechanism for id generation called sequences. A database sequence can be used to generate identifiers when the underlying database supports them.

```
@SequenceGenerator(name="Emp_Gen", sequenceName="Emp_Seq")
@Id
@GeneratedValue(generator="Emp_Gen")
private long id;
```

T · · Systems

@Entity @Table @Column @Id

```
@Entity
@Table(uniqueConstraints = @UniqueConstraint (columnNames = { "NAME", "SUR_NAME" }),
      name = "EMPLOYEE")
public class Employee {
      @Id
      private long id;

      @Column(name = "NAME", nullable = false, length = 20)
      private String name;

      @Column(name = "SUR_NAME", nullable = false, length = 20)
      private String surname;
      . . .
}
```

Validating Persistent Fields and Properties

- Для персистентных полей может использоваться **Bean Validation API** который предоставляет механизм верификации данных энтити:
  - **@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;

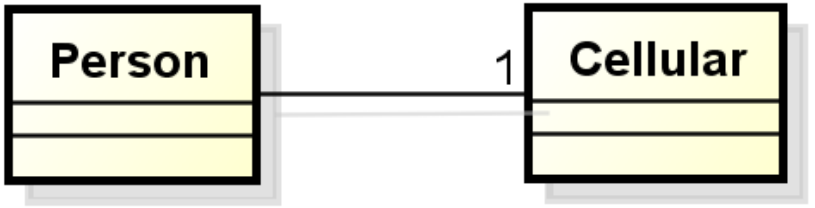
Entity Relationships (1)

- Существуют следующие отношения между энтити в JPA:
  - **@OneToOne**: при этом каждый экземпляр одного класса энтити имеет ссылку на экземпляр инстанса другой энтити.
  - **@OneToMany**: при этом инстанс одной энтити имеет ссылку на коллекцию инстансов другой энтити.
  - **@ManyToOne**: несколько инстансов одной энтити имеют ссылку на один инстанс другой.
  - **@ManyToMany**: несколько инстансов одной энтити имеют ссылки на несколько инстансов другой.

Entity Relationships (2)

- Отношения могут быть **однонаправленными** (unidirectional) и **двунаправленными** (bidirectional):
  - В однонаправленных связях одна из сторон должна быть "владельцем отношений", т.е. иметь внешний (foreign) ключ со ссылкой на другую сторону. Для этих целей используется аннотация **@JoinColumn**:

```
@OneToOne
@JoinColumn(name="cellular_id")
private Cellular cellular;
```



  - Двунаправленные связи для @OneToOne, @OneToMany, @ManyToMany должны использовать **mappedBy** для ссылки на обратное поле:

```
@OneToOne(mappedBy="cellular")
private Person person;
```

Auto Relationship? NO!

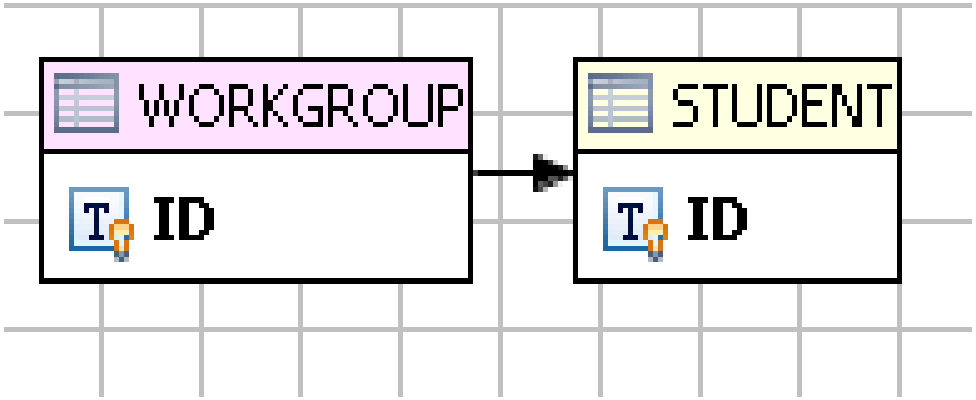
- For a **Bidirectional** relationship to work correctly it is **necessary** to do like below:

```
person.setCellular(cellular);
cellular.setPerson(person);
```

- JPA uses the Java concept of class reference, a **class must maintain a reference to another one if there will be a join between them**.

- **JPA will NOT create a relationship automatically**; to have the relationship in both sides it is needed to do like above.

One-to-Many и Many-to-One




```
@ManyToOne
private WorkGroup workGroup;

@OneToMany(mappedBy = "workGroup")
private List<Student> student;
```

Отношения и каскадные операции

- Каскадные операции определены для аннотаций @OneToOne, @OneToMany и @ManyToMany.

- Каскадные операции javax.persistence.**CascadeType**:
  - DETACH
  - MERGE
  - PERSIST
  - REFRESH
  - REMOVE
  - ALL

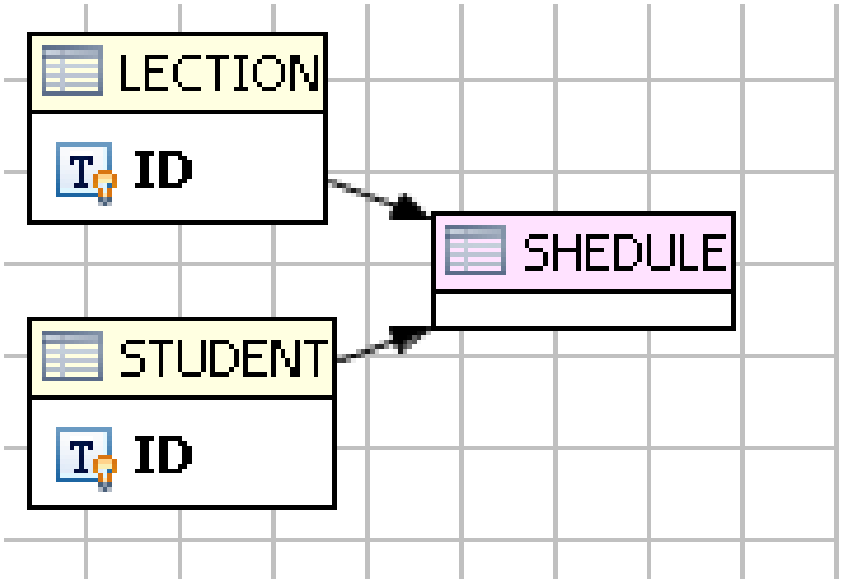
```
// @OneToMany(mappedBy = "workGroup")
// @OneToMany(mappedBy = "workGroup", cascade = CascadeType.REMOVE)
@OneToMany(mappedBy = "workGroup", orphanRemoval = true)
private List<Student> student;
```

@OneToOne

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

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

@ManyToMany




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

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

T · · Systems

Lazy Loading

- **Performance** optimization
- Related items are **not retrieved until** they are **first accessed**:
  - Field level access.
- Limited to work only within the EntityManager or Session that loaded the parent object:
  - **Causes a big problem in web applications**.

Fetch

- Fetch modes:
  - **FetchType.LAZY**
    - The exception LazyInitializationException may happens when the LAZY attribute is accessed without any opened connection.
  - **FetchType.EAGER**
  - **Disable** lazy loading

- Mode can be configured on each relationship.

- Consider performance and use when configuring fetch.

- **Default** values:
  - Every simple attribute will have the FetchType.EAGER option enabled by default.
  - Relationships @OneToOne and @ManyToOne will be EAGERly fetched.
  - Relationships @OneToMany and @ManyToMany will be LAZYly fetched.

T · · Systems

@ElementCollection

- Sometimes it is needed to **map a list of values to an entity but those values are not entities** themselves.

```
@Entity
Public class Person {
    @ElementCollection
    @MapKeyEnumerated
    private Map<PhoneType, Phone> phones;
. . .
}

public enum PhoneType {
    HOME,
    WORK,
    MOBILE
}
```

```
@Embeddable
public class Phone {
    @Enumerated(EnumType.STRING)
    private PhoneType type;
    . . .
}
```

@Embeddable Classes

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

```
@Embeddable
public class Skill {
    private String skillName;
    private String skillLevel;
    . . .
}
```

```
CREATE TABLE EMPLOYEE (
    ID BIGINT NOT NULL,
    NAME VARCHAR(20) NOT NULL,
    SKILLNAME VARCHAR(255),
    SKILLLEVEL 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 связанна 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 мапиться на таблицы (включая абстрактный), при этом в каждой таблице содержится первичный ключ из базового класса и только специфичные для данного класса поля (InheritanceType.**JOINED**)

@MappedSuperclass

- Энтити могут быть **наследованы от суперкласса**, который **сам** при этом **не является энтити**, но содержит информацию о мапинге и персистансе.

```
@MappedSuperclass
public abstract class DogFather {
    private String name;
    . . .
}

@Entity
public class Dog extends DogFather {
    @Id
    private int id;
    private String color;
    . . .
}
```

Entity Inheritance: InheritanceType.SINGLE_TABLE

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "DOG_CLASS_NAME")
public abstract class Dog {
    @Id
    private int id;
    private String name;
    . . .
}
```

id [PK] integer	dog_class_name character varying(31)	name character varyi	littlebark character v	hugepoowe integer
1	SMALL DOG	Red	hau	
2	SMALL DOG	Green	hiu	
3	SMALL DOG	Black	hie	
4	HUGE DOG	Yellow		3
5	HUGE DOG	Brown		3
6	HUGE DOG	Snow		3

```
@Entity
@DiscriminatorValue("SMALL_DOG")
public class SmallDog extends Dog {
    private String littleBark;
    . . .
}
```

```
@Entity
@DiscriminatorValue("HUGE_DOG")
public class HugeDog extends Dog {
    private int hugePooWeight;
    . . .
}
```

T · · Systems

Entity Inheritance: InheritanceType.JOINED

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@DiscriminatorColumn(name = "DOG_CLASS_NAME")
public abstract class Dog {
    @Id
    private int id;
    private String name;
    . . .
}
```

id [PK] integer	hugepooweight integer
4	6
5	3
6	4

id [PK] integer	littlebark character varying(255)
1	hau
2	hiu
3	hie

id [PK] integer	dog_class_name character varying(31)	name character varying(255)
1	SMALL_DOG	Red
2	SMALL_DOG	Green
3	SMALL_DOG	Black
4	HUGE_DOG	Yellow
5	HUGE_DOG	Brown
6	HUGE_DOG	Snow

```
@Entity
@DiscriminatorValue("SMALL_DOG")
public class SmallDog extends Dog {
    private String littleBark;
    . . .
}
```

```
@Entity
@DiscriminatorValue("HUGE_DOG")
public class HugeDog extends Dog {
    private int hugePooWeight;
    . . .
}
```

T · · Systems

Entity Inheritance: InheritanceType.TABLE_PER_CLASS

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Dog {
    @Id
    private int id;
    private String name;
    . . .
}
```

id [PK] integer	hugepooweight integer	name character varying(255)
4	6	Yellow
5	3	Brown
6	4	Snow

id [PK] integer	littlebark character varying(255)	name character varying(255)
1	hau	Red
2	hiu	Green
3	hie	Black

```
@Entity
public class SmallDog extends Dog {
    private String littleBark;
    . . .
}
```

```
@Entity
public class HugeDog extends Dog {
    private int hugePooWeight;
    . . .
}
```

JPQL Features

- **JPQL** is **J**ava **P**ersistence **Q**uery **L**anguage defined in JPA specification. It is used to create queries against entities to store in a relational database.

- JPQL is developed based on SQL syntax. But it won't affect the database directly:
  - **JPQL syntax is very similar to** the syntax of **SQL**.
  - SQL works directly against relational database tables, records and fields, whereas **JPQL works with Java classes and instances**.

- **String functions**
  - upper, lower, length, substring, concat, trim
- **Aggregation functions**
  - count, min, max, sum, avg
  - Can require "group by" clause
  - Also supports "having" on "group by"
- **Subselects**

How To JPQL?

- Must be created from the EntityManager:
  - EntityManger.**createQuery**(String jpql);

- **Query** objects **can be configured**:
  - Maximum number of results.
  - Parameters set.

- Query objects can then **be used to list out the results** of the query:
  - list() returns a java.util.List.

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpaHibernateMaven");
EntityManager em = emf.createEntityManager();
Query query = em.createQuery("SELECT UPPER(u.name) FROM User u");
List<String> list = query.getResultList();
```

T · · Systems

JPQL Query Parameters

- Work just like JDBC Parameters:
  - **?** will act as a placeholder for an indexed parameter
  - **:name** will designate a parameter with the name of "name"

- **Examples**
  - SELECT i FROM Item i WHERE name like **?**
  - SELECT i FROM Item i WHERE name like **:name**

```
public List findUsersByName(String name) {
      return em.createQuery(
          "SELECT u FROM User u WHERE u.name LIKE :userName")
          .setParameter("userName", name)
          .setMaxResults(10)
          .getResultList();
}
```

JPQL Example

- "FROM Item i WHERE i.name = 'foobar'"
- "FROM Item i WHERE i.bidder.name = 'Mike'" // Implicit join
- "FROM Item i WHERE i.bids is not empty"
- "FROM Item i WHERE size(i.bids) > 3" // Implicit join
- "FROM Item i order by i.name asc, i.entryDate asc"
- "FROM Item i join i.bids b WHERE b.amount > 100" // will return rows with an array
- "SELECT distinct(i) FROM Item i join i.bids b WHERE b.amount > 100" // returns Items
- "SELECT i.name, i.description FROM Item i WHERE entryDate > ?"

Criteria API

- The **Criteria API** is a predefined API used to define queries for entities. It is the alternative way of defining a JPQL query.
- These queries are **type-safe**, and **portable** and **easy to modify** by changing the syntax.
- The **major advantage** of the criteria API is that **errors can be detected** earlier **during compile time**.
- String based **JPQL** queries **and** JPA **criteria** based queries **are same in performance and efficiency**.
- Criteria API is **useful when** creating **complex queries**

Criteria Example

- The following simple Criteria query returns all instances of the User entity in the data source:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("yourApp");
EntityManager em = emf.createEntityManager();
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<User> cq = cb.createQuery(User.class);
Root<User> user = cq.from(User.class);
cq.select(user);
TypedQuery<User> q = em.createQuery(cq);
List<User> allUsers = q.getResultList();
```

- The equivalent JPQL query is:
  - **SELECT** u **FROM** User u

Further Reading

- Mike Keith, Merrick Schincariol. Pro JPA 2
- Christian Bauer, Gavin King. Java Persistence with Hibernate
- Hebert Coelho, Byron Kiourtzoglou. Java Persistence API Mini Book
- https://javaee.github.io/tutorial/toc.html
- https://en.wikibooks.org/wiki/Java_Persistence
- http://www.tutorialspoint.com/jpa/
- http://habrahabr.ru/post/265061/