

ENTERPRISE JAVABEANS 3.0

JAVA PERSISTENCE API – ORM PART

Что мы имеем?

- Реляционные Системы Управления Базами Данных (RDBMS)
- Java Database Connectivity (JDBC)

Все ещё верите, что JDBC это идеальное решение?

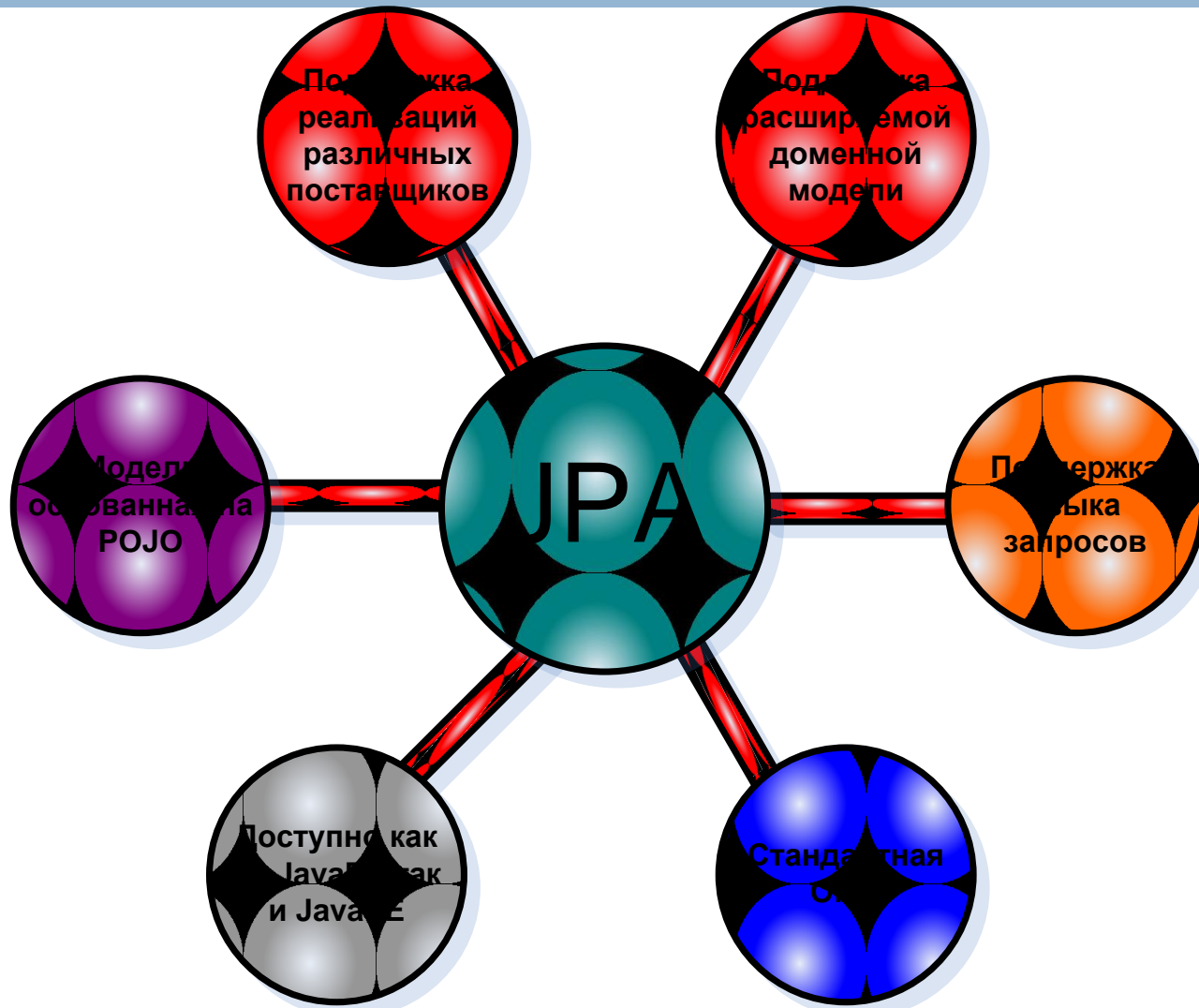
```
public static void main(String[] args) {
    try {
        Connection connection = DriverManager
            .getConnection("jdbc:derby:Console;create=true");
        Statement statement = null;
        statement = connection.createStatement();
        statement.execute("SELECT first_name, last_name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

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

Реляционный мэпинг объектов (Object Relational Mapping)

- Устраняет потребность в JDBC
 - ▣ CRUD операции и запросы
- Управление идентичностью объектов
- Стратегии наследования
 - ▣ Иерархия классов для представления одной или нескольких таблиц
- Ассоциации и Композиция
 - ▣ «Ленивое» управление
 - ▣ Стратегии подгрузки данных (fetching)

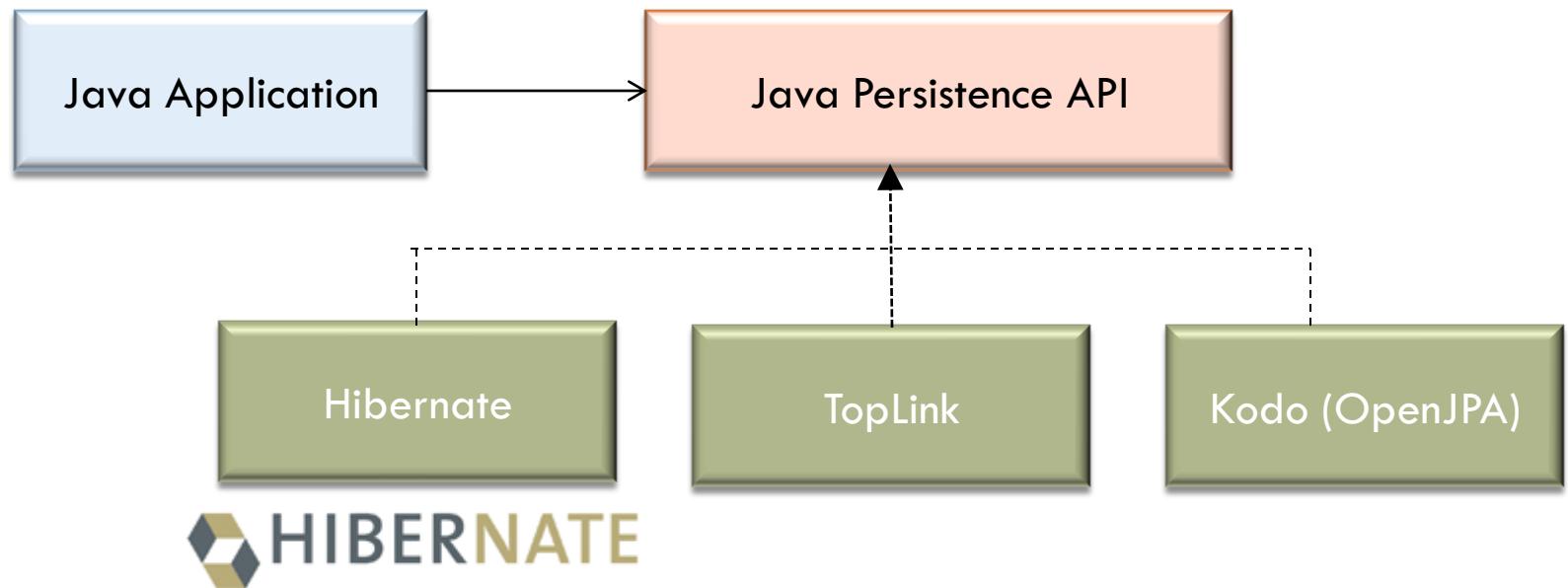
Java Persistence API



Введение в JPA

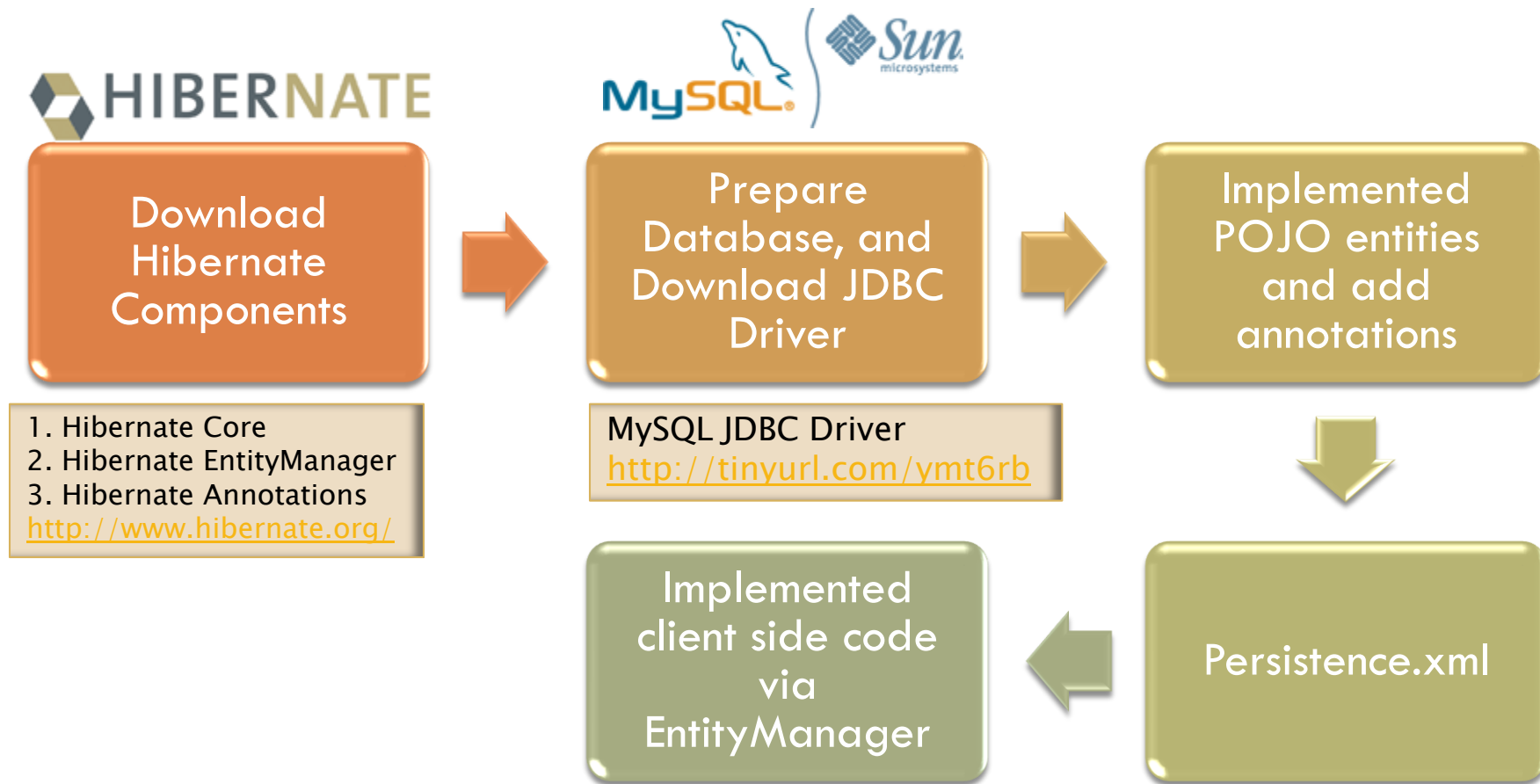
- Независимое от поставщиков решение ORM
- Легко конфигурируется
 - ▣ Конфигурация задается прямо в коде с помощью Java аннотаций
 - ▣ Конфигурация прекрасно настраивается с помощью XML который перекрывает аннотации
- Работает вне JavaEE контейнеров

Архитектура JPA

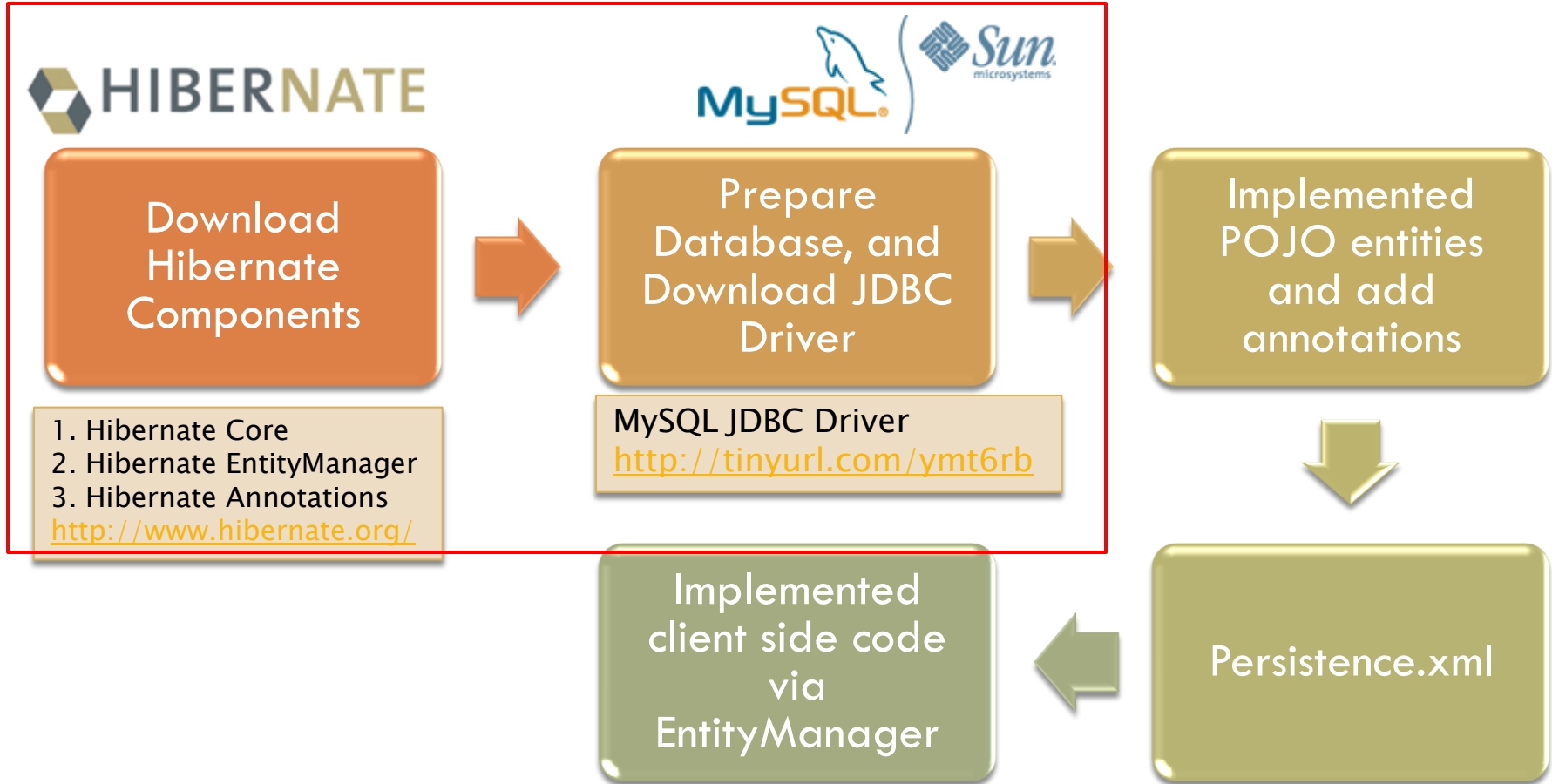


Каждый может выбрать для себя технологию перситетности

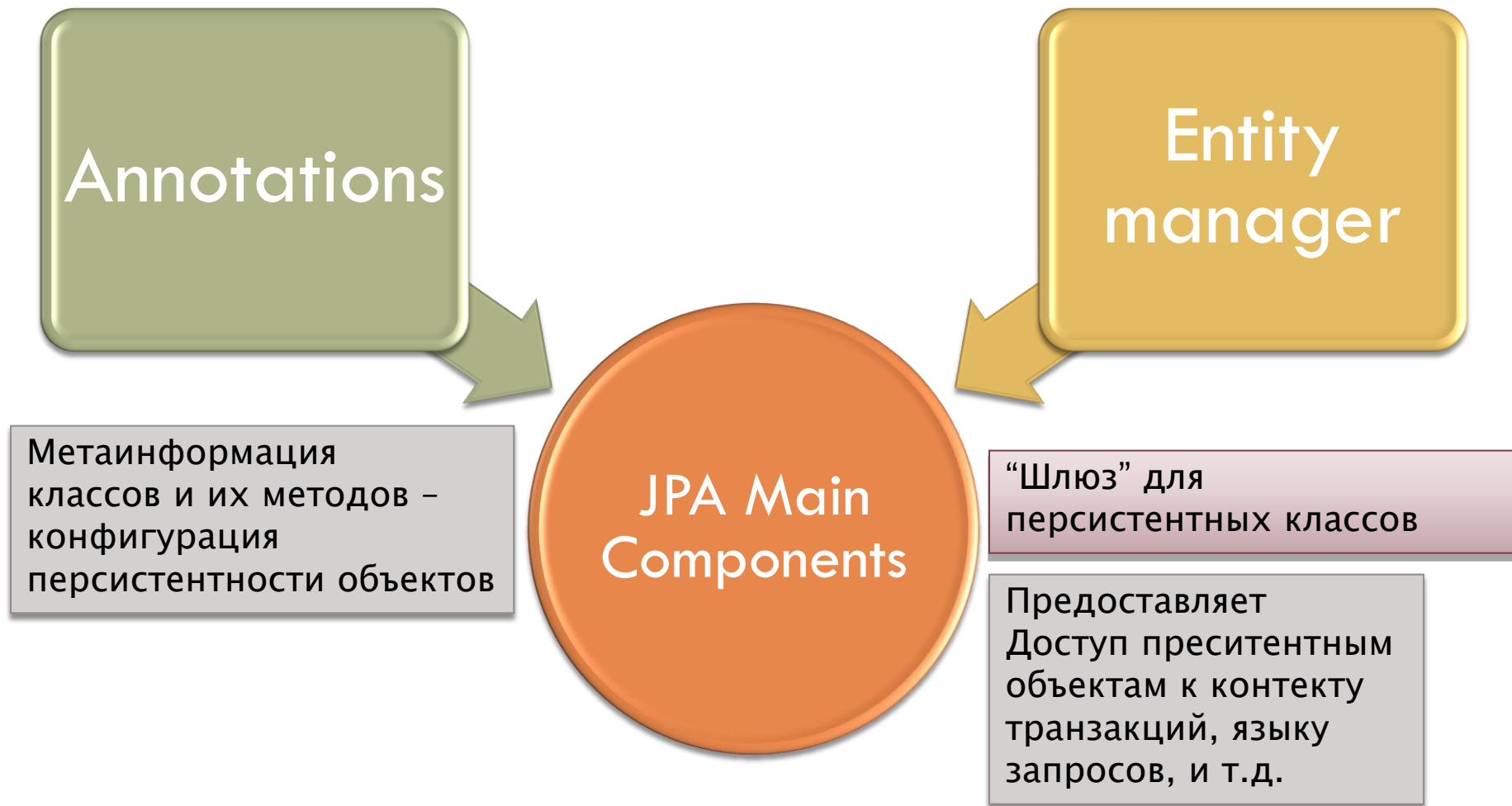
Пять шагов для использования JPA



Шаги №1 и №2



Основные компоненты JPA



Hibernate Annotations/EntityManager

- Аннотации Hibernate включают
 - ▣ Стандартные аннотации JPA и EJB 3.0
 - ▣ Специфичные для Hibernate расширение аннотаций для оптимизации и специального мэппинга
- Hibernate EntityManager включает
 - ▣ Стандартную реализацию управления Java Persistence (JP)
 - ▣ Стандартный язык запросов Java Persistence Query Language (JPQL)
 - ▣ Стандартную реализацию правил жизненного цикла JP объектов
 - ▣ Стандартную реализацию конфигурирования JP

Шаг 3 – имплементация



Download
Hibernate
Components

1. Hibernate Core
2. Hibernate EntityManager
3. Hibernate Annotations
<http://www.hibernate.org/>

Prepare
Database, and
Download JDBC
Driver

MySQL JDBC Driver
<http://tinyurl.com/ymt6rb>

Implemented
POJO entities
and add
annotations

Implemented
client side code
via
EntityManager

Persistence.xml



JPA – возвращаемся к POJOc

- ❑ Не final класс или методы
- ❑ Конструктор по умолчанию (без параметров)
- ❑ Коллекции определяются через интерфейсы
- ❑ Вы не управляете ассоциациями
- ❑ Должно быть объявлено поле идентификатор

Простые типы данных

- Примитивные типы и их обертки
- `java.lang.String`
- `java.math.BigInteger` & `BigDecimal`
- Массивы `Byte` & `Character`
- Java & JDBC временные типы
- Enumeration
- Типы реализующие интерфейс `Serializable`

ORM МЭППИНГ

employee

emp_id INT(10) NOT NULL (PK)
salary DECIMAL(8) NULL
dept_id INT(10) NULL

```
45 @Id
46 @Column(name = "emp_id", unique = true, nullable = false,
47         insertable = true, updatable = true)
48 public Integer getEmpId() {
49     return this.empId;
50 }

67 @Column(name = "salary", unique = false, nullable = true,
68         insertable = true, updatable = true, precision = 8, scale = 0)
69 public Long getSalary() {
70     return this.salary;
71 }

56 @ManyToOne(cascade = {}, fetch = FetchType.LAZY)
57 @JoinColumn(name = "dept_id", unique = false, nullable = true,
58         insertable = true, updatable = true)
59 public Department getDepartment() {
60     return this.department;
61 }
```

@Entity

- Аннотация контекста класса
- Говорит о том что класс является сущностью
- Пример:

```
19 @Entity
20 @Table(name = "department", catalog = "test", uniqueConstraints = {})
21 public class Department implements Serializable {
```

- Сущность должна придерживаться соглашений Java Bean по отношению к свойствам объекта которые будут сохраняться
 - Каждое свойство должно иметь get и set методы

- getDeptId()
- setDeptId(Integer)
- getDeptDesc()
- setDeptDesc(String)

Самый простой пример Entity

```
@Entity
public class Person {

    @Id
    private Long ssn;

    private String firstName;
    private String lastName;

    protected Person() {}

    public Person(Long ssn, String firstName, String lastName) {
        ...
    }
}
```

@Id

- Каждая сущность должна иметь идентификатор (первичный ключ)
- Задается просто аннотированием свойства класса с помощью @Id
- Пример:

```
@Id
@Column(name = "dept_id", unique = true, nullable = false,
        insertable = true, updatable = true)
public Integer getDeptId() {
    return this.deptId;
}
```

Первичный ключ может быть сложным

@Id

- Значение Id может быть автогенерируемым
@Id(generate=GeneratorType.AUTO)
- Дополнительные стратегии генерации первичных ключей:
 - ▣ GeneratorType.SEQUENCE
 - ▣ GeneratorType.TABLE
 - ▣ GeneratorType.IDENTITY
- AUTO – лучше всего подходит для переносимости между различными поставщиками баз данных

Авто-генерация ID

```
@Entity
public class User {

    private Long id;
    private String name;

    @Id
    @GeneratedValue
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

}
```

TABLE-генерация Id

```
@TableGenerator (name="CUST_GENERATOR",  
    table="GENERATOR_TABLE",  
    pkColumnName="PRIMARY_KEY_COLUMN",  
    valueColumnName="VALUE_COLUMN",  
    pkColumnValue="CUST_ID",  
    allocationSize=10)
```

```
@Entity
```

```
public class User {  
  
    private Long id;  
    private String name;
```

```
@Id
```

```
@GeneratedValue
```

```
(strategy=GenerationType.TABLE, generator="CUST_GENERATOR")
```

```
public Long getId() {  
    return id;  
}
```

SQL

```
create table GENERATOR_TABLE  
(  
    PRIMARY_KEY_COLUMN  
    VARCHAR not null,  
    VALUE_COLUMN  
    long not null  
);
```

SEQUENCE-генерация

```
@SequenceGenerator(name="CUSTOMER_SEQUENCE",
                    sequenceName="CUST_SEQ")

@Entity
public class User {

    private Long id;
    private String name;

    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE,
                    generator="CUSTOMER_SEQUENCE")
    public Long getId() {
        return id;
    }
}
```

@Column

- @Column устанавливается на get метод свойства класса (или на свойство класса)
- Задаёт следующую конфигурацию свойства сущности
 - ▣ Updatable/Insertable (boolean)
 - ▣ Nullable (updatable)
 - ▣ Length (int)

□ Пример:

```
58 @Column(name = "dept_desc", unique = false, nullable = true,  
59         insertable = true, updatable = true, length = 100)  
60 public String getDeptDesc() {  
61     return this.deptDesc;  
62 }
```

@Temporal

- Спецификация JPA для свойств типа `java.util.Date` и `java.util.Calendar` требует указания типа временного поля в БД.

Делается это с помощью аннотации `@Temporal`:

```
@Temporal(TemporalType.TIMESTAMP)  
private Date date;
```


Стратегии загрузки данных (fetching)

- Ленивая загрузка (Lazy fetching)
 - ▣ предназначена для полей которые редко используются. Данные загружаются при попытке обращения к данным
- Полная загрузка (Eager fetching) - **default**
 - ▣ Данные подгружаются сразу

Для выбора стратегии загрузки данных используется аннотация

```
@Basic(fetch=FetchType.<LAZY|EAGER>)
```

Пример объявления стратегии «ленивой» загрузки

```
@Entity
public class Person extends BaseEntity {

    private String name;

    @Basic(fetch=FetchType.LAZY)
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Большие объекты

- БД позволяют хранить большие объекты (large objects – LOB) – byte-based объекты, могут быть огромного размера (до нескольких гигабайт – зависимости от возможностей БД)
- Пример LOB объектов: картинки, большие блоки текста, различные двоичные файлы
- Пример мэппинга:

```
@Lob
```

```
private byte[] picture;
```

Связи между сущностями

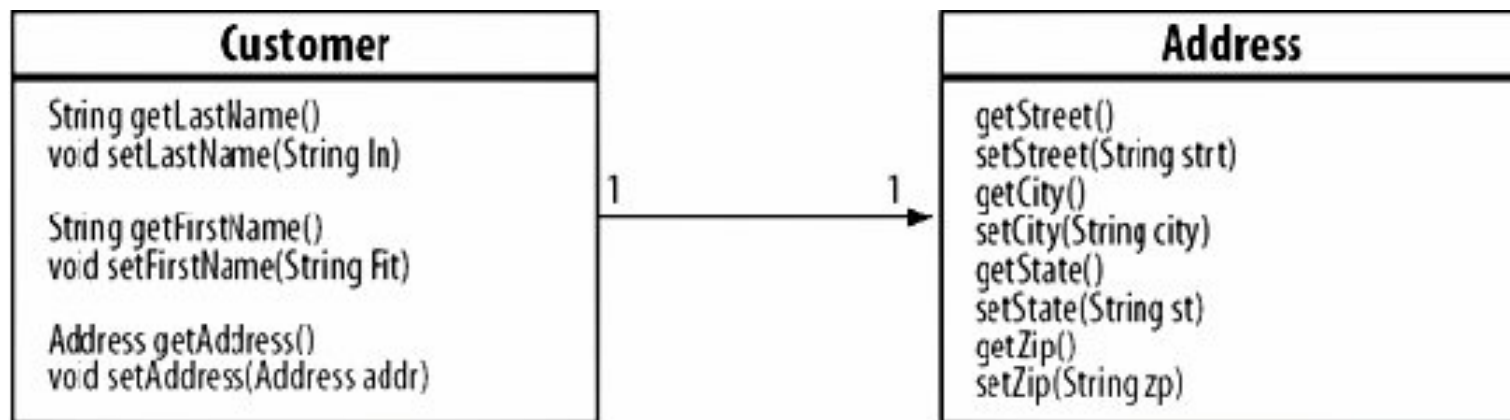
Unidirectional – однонаправленный

Bidirectional - двунаправленный

- One-to-one unidirectional
- One-to-one bidirectional
- One-to-many unidirectional
- One-to-many bidirectional
- Many-to-one unidirectional
- Many-to-many unidirectional
- Many-to-many bidirectional

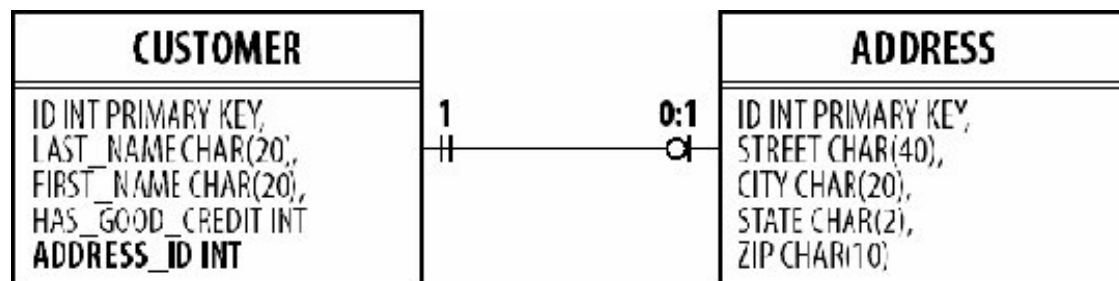
One-to-One Unidirectional

- Например связь «Клиент»-«Адрес»



- Адрес «ничего не знает» о клиенте который с ним связан

Уровень БД:
**FK в таблице
Customer**



One-to-One Unidirectional

Программная модель

```
@Entity
public class Address extends BaseEntity {

    private String street;
    private String building;

    public String getStreet() {
        return street;
    }
    public void setStreet(String street) {
        this.street = street;
    }
    public String getBuilding() {
        return building;
    }
    public void setBuilding(String building) {
        this.building = building;
    }
}
```

```
@Entity
public class Person {

    private Long id;
    private String name;
    private Address address;

    @OneToOne(cascade=CascadeType.ALL)
    @JoinColumn(name="address_id", nullable=false)
    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }
}
```

One-to-One Bidirectional

- Пример связи «Клиент» – «Кредитная карта»
- У клиента есть номер кредитной карты, и в тоже время «Кредитная карта» должна содержать в себе информацию о своем пользователе
- С точки зрения БД те же связи как и в «**One-to-One Unidirectional**»

One-to-One Bidirectional программная модель

```
@Entity
public class Person {

    private Long id;
    private String name;
    private Address address;
    private CreditCard creditCard;

    @OneToOne(cascade=CascadeType.ALL)
    public CreditCard getCreditCard() {
        return creditCard;
    }
}
```

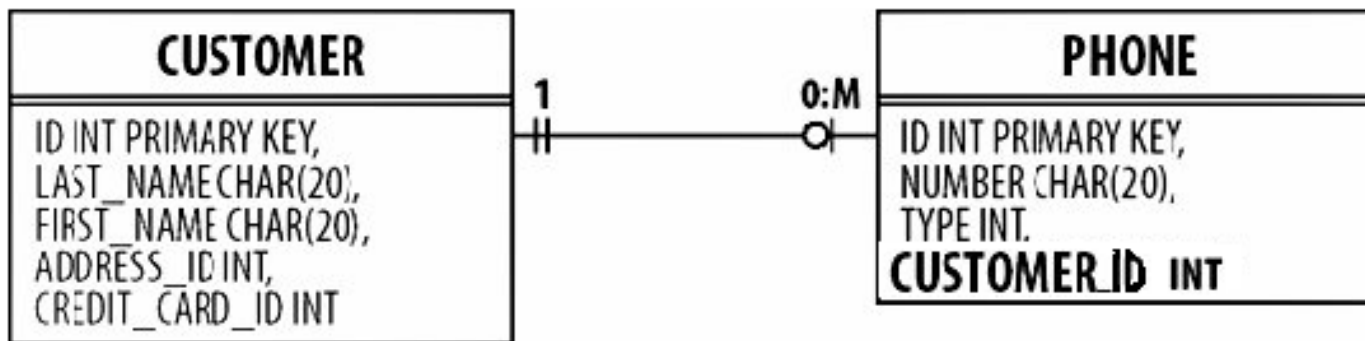
```
@Entity
public class CreditCard extends BaseEntity {

    private Person person;

    @OneToOne(mappedBy="creditCard")
    public Person getPerson() {
        return person;
    }
}
```


One-to-Many Unidirectional

- Пример связи «клиент» - «номер телефона»
- Каждый клиент может иметь несколько телефонных номеров, но не наоборот.
- На уровне БД у таблицы PHONE будет создано поле для хранения «клиентов» и ForeignKey



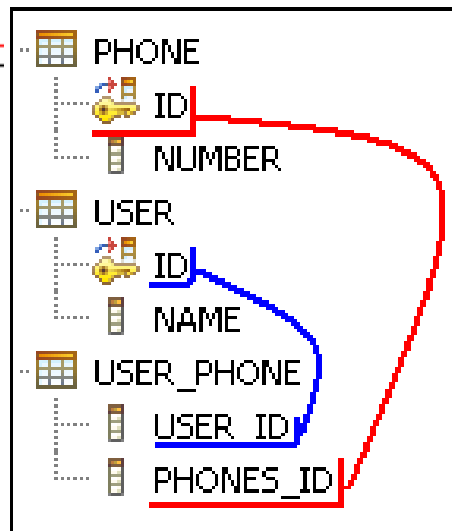
One-to-Many Unidirectional программная модель

```
@JoinTable(name="USER_PHONE",  
          joinColumns={@JoinColumn(name="USER_ID")},  
          inverseJoinColumns={@JoinColumn(name="PHONE_ID")})
```

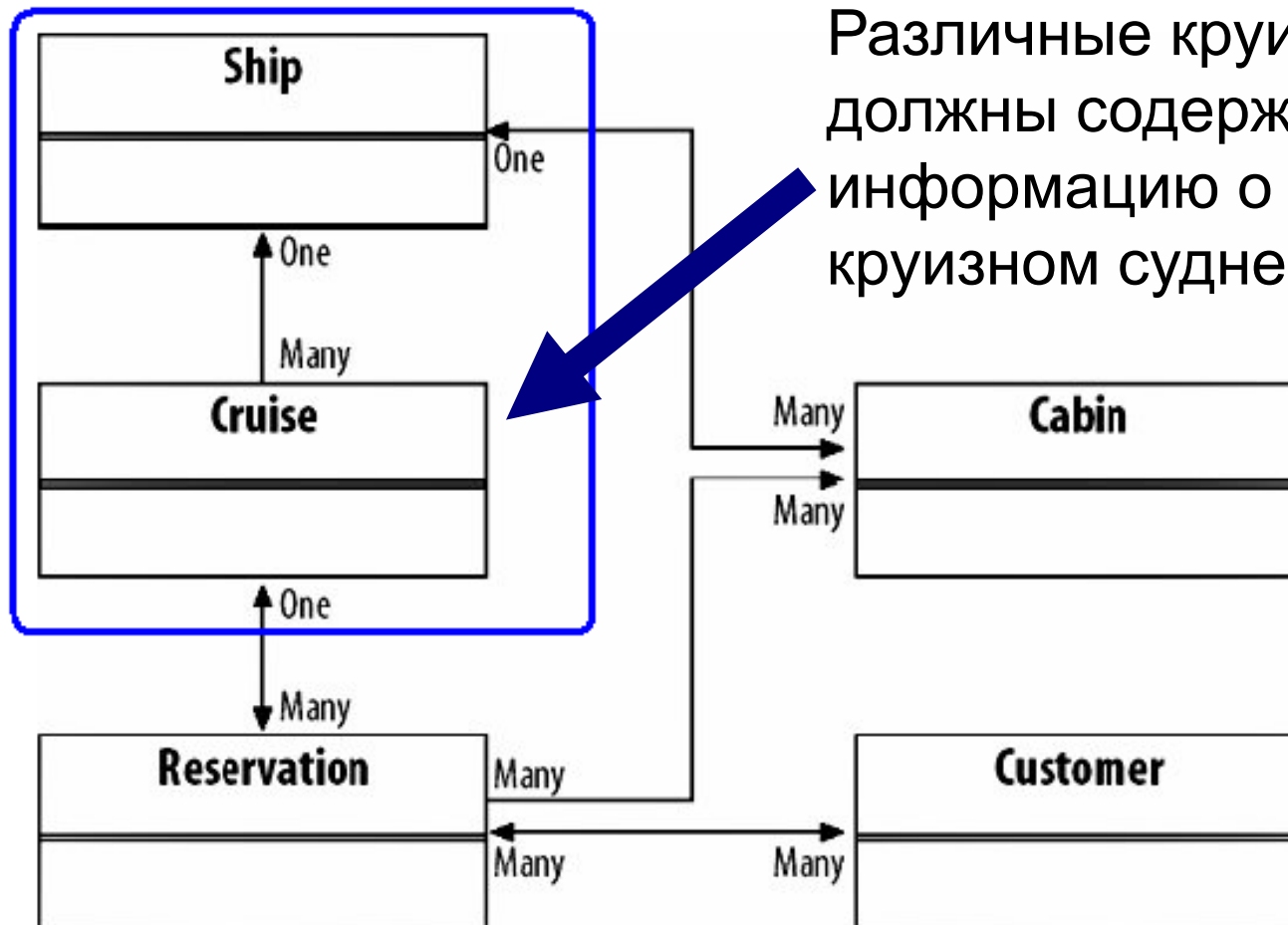
```
@Entity  
public class User extends BaseEntity {  
  
    private String name;  
    private Collection<Phone> phones = new ArrayList<Phone>();  
  
    @OneToMany(cascade={CascadeType.ALL})  
    @JoinColumn(name="USER_ID")  
    public Collection<Phone> getPhones() {  
        return phones;  
    }  
}
```

```
@Entity  
public class Phone extends BaseEntity {  
  
    private String number;  
  
    public String getNumber() {  
        return number;  
    }  
  
    public void setNumber(String number) {  
        this.number = number;  
    }  
}
```

Для связывания
через
промежуточную
таблицу нужно
использовать
`@JoinTable`



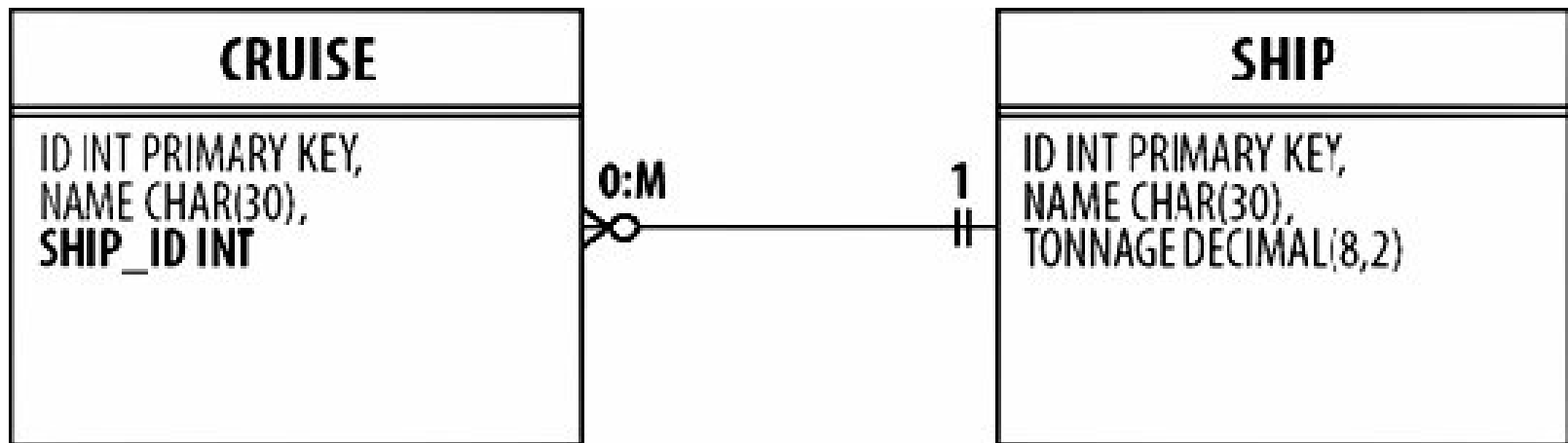
Many-to-One Unidirectional



Различные круизы
должны содержать
информацию о
круизном судне

Many-to-One Unidirectional

- На уровне базы данных связывание через создание колонки в таблице «Cruise» содержащей ID соответствующего корабля и ForeignKey на таблицу «Ship»



Many-to-One Unidirectional программная модель

```
@Entity
public class Cruise extends BaseEntity {
    private String name;
    private Ship ship;

    @ManyToOne
    @JoinColumn(name = "SHIP_ID")
    public Ship getShip() {
        return ship;
    }
}
```

```
@Entity
public class Ship extends BaseEntity {
    private String name;
    private double tonnage;

    public String getName() {
        return name;
    }
}
```

One-to-many/Many-to-one bidirectional – программная модель

```
@Entity
public class Ship extends BaseEntity {
    private String name;
    private double tonnage;
    private Collection<Cruise> cruises = new ArrayList<Cruise>();

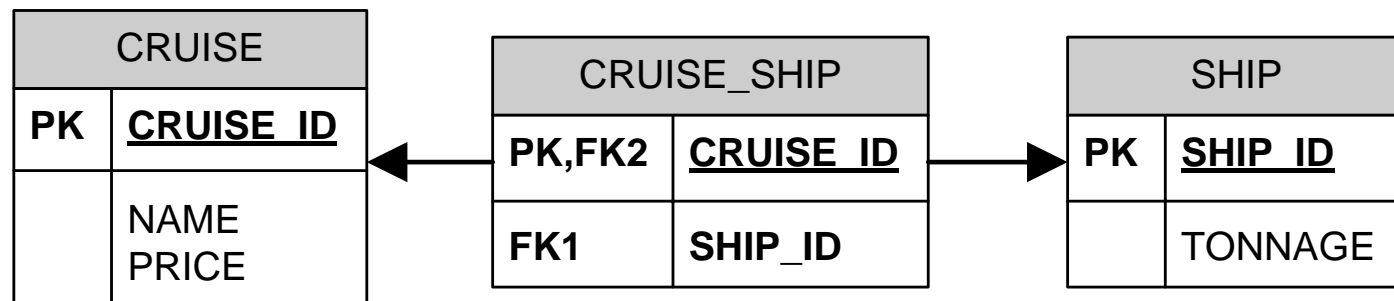
    @OneToMany(mappedBy="ship")
    public Collection<Cruise> getCruises() {
        return cruises;
    }
}
```

```
@Entity
public class Cruise extends BaseEntity {
    private String name;
    private Ship ship;

    @ManyToOne
    @JoinColumn(name = "SHIP_ID")
    public Ship getShip() {
        return ship;
    }
}
```

Many-to-many bidirectional

- Примером можно привести соотношение между круизом и кораблем. Несколько круизов могут происходить на одном корабле и в тоже время несколько кораблей может учувствовать в одном круизе
- Так как с обеих сторон будут коллекции то для мэппинга будет обязательно использоваться промежуточная таблица (@JoinTable)
- Связи на уровне БД:



Many-to-many bidirectional программная модель

```
@Entity
public class Cruise extends BaseEntity {
    private String name;
    private Collection<Ship> ships = new ArrayList<Ship>();

    @ManyToMany
    @JoinTable (name="MY_CRUISE_SHIP",
                joinColumns=@JoinColumn(name="CRUISE_ID"),
                inverseJoinColumns=@JoinColumn(name="SHIP_ID"))
    public Collection<Ship> getShips() {
        return ships;
    }
}
```

```
@Entity
public class Ship extends BaseEntity {
    private String name;
    private double tonnage;
    private Collection<Cruise> cruises = new ArrayList<Cruise>();

    @ManyToMany(mappedBy="ships")
    public Collection<Cruise> getCruises() {
        return cruises;
    }
}
```


Many-to-many bidirectional программная модель

- **Важно что в таком мэппинге у нас Cruise выступает собственника связи! Поэтому все действия добавлению и удалению связей должны идти через Cruise!**

Правильно - Cruise собственник связи

```
Cruise cruise = em.find(Cruise.class, id);  
cruise.getShips().add(ship);
```

Неправильно - добавление будет проигнорировано

```
Ship ship = em.find(Ship.class, id);  
ship.getCruises().add(cruise);
```

Many-to-many Unidirectional программная модель

```
@Entity
public class Cruise extends BaseEntity {
    private String name;
    private Collection<Ship> ships = new ArrayList<Ship>();

    @ManyToMany
    public Collection<Ship> getShips() {
        return ships;
    }
}
```

```
@Entity
public class Ship extends BaseEntity {
    private String name;
    private double tonnage;

    public String getName() {
        return name;
    }
}
```

Каскады

- Существует возможность выполнения автоматических операций с ассоциациями сущностей
 - ▣ Persist – автоматическое создание ассоциированных сущностей
 - ▣ Remove – автоматическое удаление ассоциированных сущностей
 - ▣ Merge – автоматическое обновление/создание ассоциированных сущностей при операции merge
 - ▣ All – автоматически выполнять с ассоциациями любые операции

Пример объявления поддержки каскадных операций

```
@Entity
public class Customer extends Person {

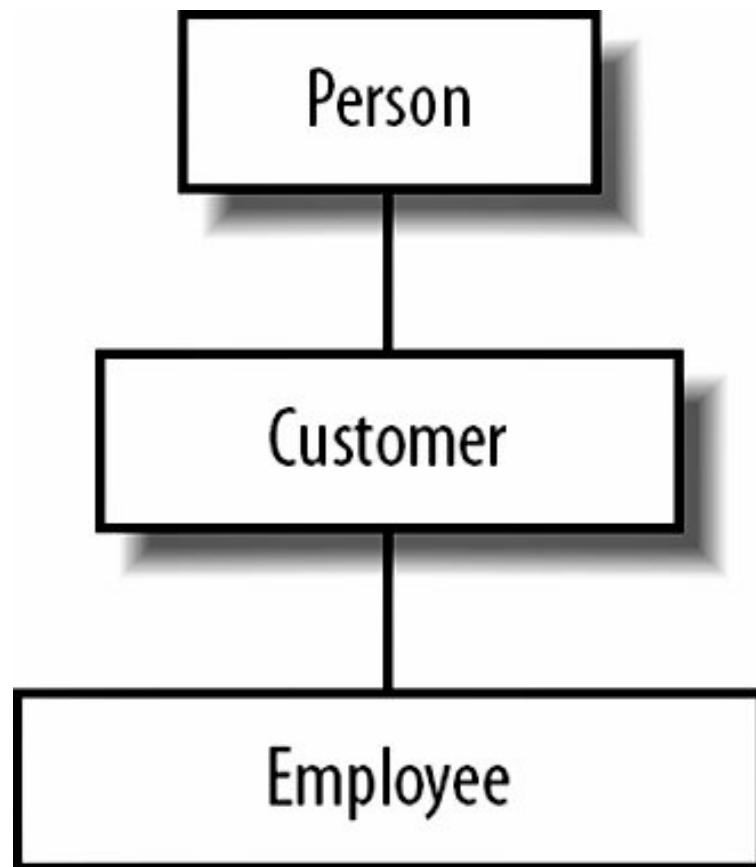
    private String cardId;

    @OneToOne (cascade={CascadeType.PERSIST, CascadeType.REMOVE})
    private Address address;
```

При создании/удалении сущности Customer автоматически будет создаваться/удаляться соответствующая ей сущность Address

Наследование сущностей

- JPA поддерживает мэппинг иерархии классов



Наследование: пример кода

```
@Entity
public class Person {

    @Id
    private Long id;
    private String name;
}
```

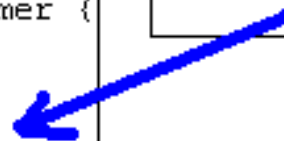
```
@Entity
public class Customer extends Person {

    private String cardId;

    @OneToOne
    private Address address;
}
```

```
@Entity
public class Employee extends Customer {

    private String deptNumber;
    private Long salary;
}
```



Одна таблица на иерархию классов

- Способ наследования по умолчанию
- Для хранения всей структуры классов используется одна таблица
- Для отличия записи одного класса от другого класса в иерархии наследования используется колонка «дискриминатор»

Одна таблица на иерархию классов: пример кода


```
@Entity
@Table(name="PERSON_HIERARCHY")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="DISCRIMINATOR",
                    discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue("PERSON")
public class Person {

    @Id
    private Long id;
    private String name;
}
```

```
@Entity
public class Customer extends Person {

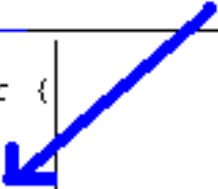
    private String cardId;

    @OneToOne
    private Address address;
}
```



```
@Entity
public class Employee extends Customer {

    private String deptNumber;
    private Long salary;
}
```



Одна таблица на иерархию классов: мэппинг на БД

```
public class Person {  
  
    @Id  
    private Long id;  
    private String name;
```

```
public class Customer extends Person {  
  
    private String cardId;  
  
    @OneToOne  
    private Address address;
```

```
public class Employee extends Customer {  
  
    private String deptNumber;  
    private Long salary;
```

PERSON_HIERARCHY

Column Name	Data Type
DTYPE	VARCHAR
ID	BIGINT
NAME	VARCHAR
CARDID	VARCHAR
DEPTNUMBER	VARCHAR
SALARY	BIGINT
ADDRESS_ID	BIGINT

Одна таблица на иерархию классов: данные в БД

```
Person p = new Person();  
p.setName("Person");
```

```
Customer c = new Customer();  
c.setName("Customer");  
c.setAddress(new Address());  
c.setCardId("123456");
```

```
Employee e = new Employee();  
e.setName("Employee");  
e.setAddress(new Address());  
e.setCardId("0987654");  
e.setDeptNumber("256");  
e.setSalary(50000L);
```

ID	NAME	CARDID	ADDRESS_ID	DEPTNUMBER	SALARY	DISCRIMINATOR
1	Person	<NULL>	<NULL>	<NULL>	<NULL>	Person
2	Customer	123456	1	<NULL>	<NULL>	Customer
3	Employee	0987654	2	256	50000	Employee

Одна таблица на иерархию классов: плюсы и минусы

- Самый простой способ организации иерархии данных
- Самая высокая производительность запросов на выборку
- Очень расточительная по отношению к табличной памяти БД (Много колонок содержащих значение NULL)
- Возможны проблемы с применением некоторых ограничений для свойств сущности (например nullable=false)

Таблица на сущность

- Предусматривает создание отдельной таблицы для каждой сущности иерархии
- Каждая таблица должна содержать колонки представляющие свои свойства и свойства своего суперкласса

Таблица на сущность: пример кода

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

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

```
@Entity
public class Customer extends Person {

    private String cardId;

    @OneToOne (cascade=CascadeType.ALL)
    private Address address;
```

```
@Entity
public class Employee extends Customer {

    private String deptNumber;
    private Long salary;
```

Таблица на сущность: мэппинг на БД

Классы

```
public class Person {  
    @Id  
    @GeneratedValue(strategy=GenerationType.TABLE)  
    private Long id;  
    private String name;
```

```
public class Customer extends Person {  
    private String cardId;  
  
    @OneToOne (cascade=CascadeType.ALL)  
    private Address address;
```

```
public class Employee extends Customer {  
    private String deptNumber;  
    private Long salary;
```

БД

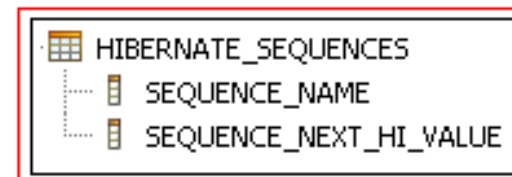
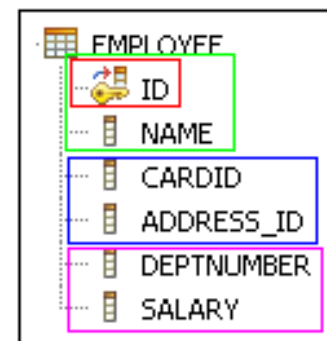


Таблица на сущность: данные в БД

```
Person p = new Person();  
p.setName("Person");
```

```
Customer c = new Customer();  
c.setName("Customer");  
c.setAddress(new Address());  
c.setCardId("123456");
```

```
Employee e = new Employee();  
e.setName("Employee");  
e.setAddress(new Address());  
e.setCardId("0987654");  
e.setDeptNumber("256");  
e.setSalary(50000L);
```

ID	NAME
1	Person

ID	NAME	CARDID	ADDRESS_ID
2	Customer	123456	1

ID	NAME	CARDID	ADDRESS_ID	DEPTNUMBER	SALARY
3	Employee	0987654	2	<NULL>	50000

Таблица на сущность: плюсы и минусы

- Нет проблем с ограничениями свойств
- Простой мэппинг
- Не нормализованная стратегия
- Множественные запросы


Таблица на подкласс

- Предусматривает создание таблицы на каждый подкласс в иерархии классов
- Каждая из таблиц содержит только колонки соответствующего ей подкласса
- Если для сохранения сущности требуется несколько подтаблиц - идентификатор записи в каждой из них будет одинаковый

Таблица на подкласс: пример кода

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Person {

    @Id
    @GeneratedValue
    private Long id;
    private String name;
```



```
@Entity
public class Customer extends Person {

    private String cardId;

    @OneToOne (cascade=CascadeType.ALL)
    private Address address;
```

```
@Entity
public class Employee extends Customer {

    private String deptNumber;
    private Long salary;
```

Таблица на подкласс: мэппинг на БД

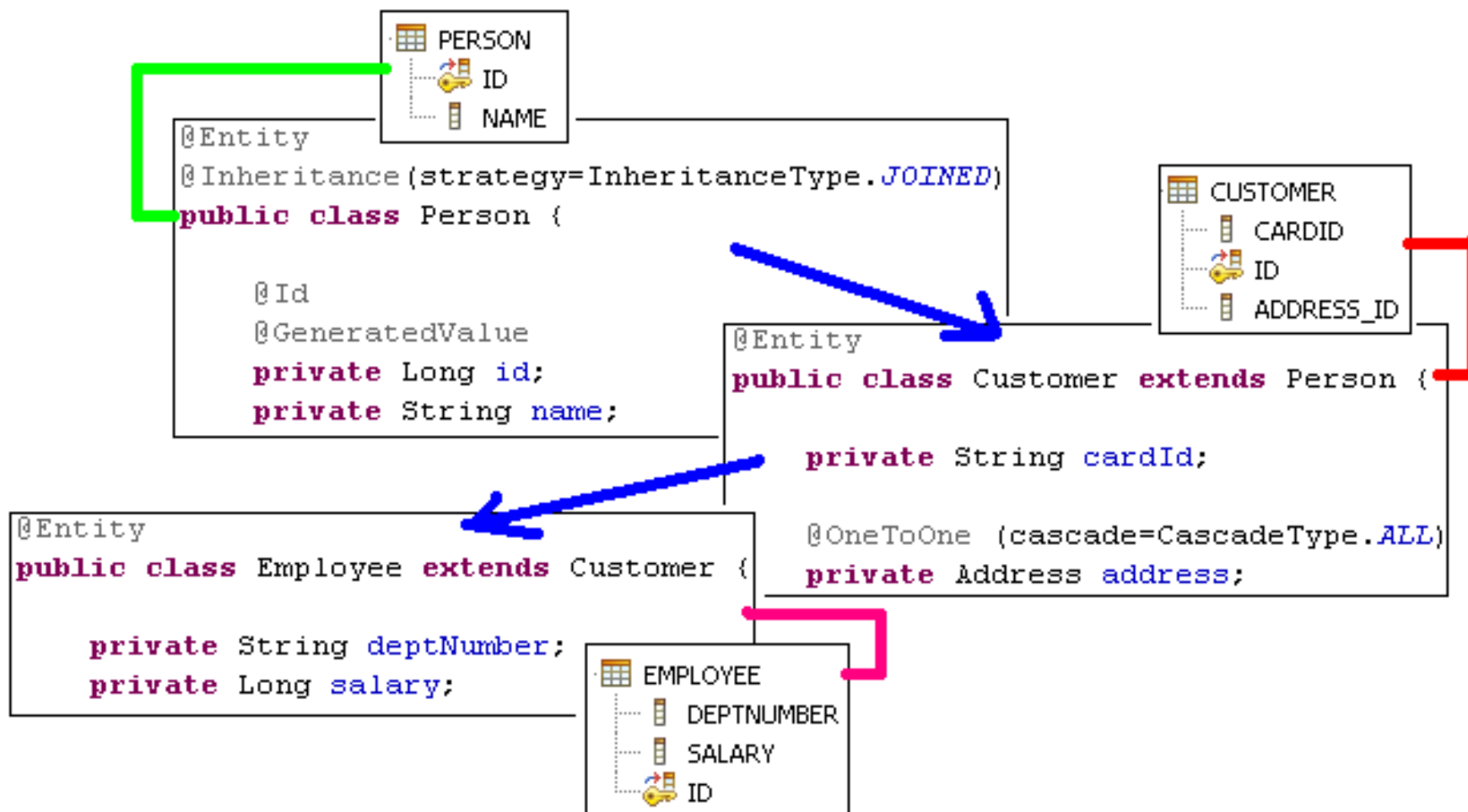


Таблица на подкласс: данные в БД

ID	NAME
①	Person
②	Customer
③	Employee

```
Person p = new Person();  
p.setName("Person");  
  
Customer c = new Customer();  
c.setName("Customer");  
c.setAddress(new Address());  
c.setCardId("123456");  
  
Employee e = new Employee();  
e.setName("Employee");  
e.setAddress(new Address());  
e.setCardId("0987654");  
e.setDeptNumber("256");  
e.setSalary(50000L);
```

CARDID	ID	ADDRESS_ID
123456	②	1
0987654	③	2

DEPTNUMBER	SALARY	ID
256	50000	③

Таблица на подкласс: плюсы и минусы

- Нет проблем установки ограничений
- Экономичное использование ресурсов БД
- Нормализованное решение
- Не так быстро как решение «Одна таблица на иерархию классов»

Абстрактный суперкласс

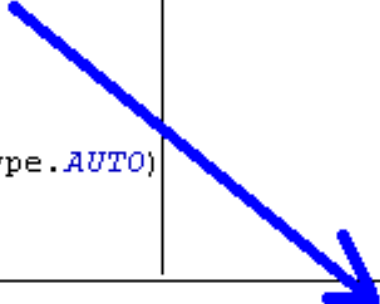
- Требуется создать абстрактный класс для которого не будет соответствующей таблицы, но он планируется использовать его в иерархии сущностей
- Используется аннотация `@MappedSuperclass`

Пример @MappedSuperclass

```
@MappedSuperclass
public abstract class BaseEntity {

    protected Long id;

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    public Long getId() {
        return id;
    }
}
```



```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Person extends BaseEntity {

    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

}
```

Конфигурация хранения



Download
Hibernate
Components

1. Hibernate Core
2. Hibernate EntityManager
3. Hibernate Annotations
<http://www.hibernate.org/>

Prepare
Database, and
Download JDBC
Driver

MySQL JDBC Driver
<http://tinyurl.com/ymt6rb>

Implemented
POJO entities
and add
annotations

Implemented
client side code
via
EntityManager

Persistence.xml



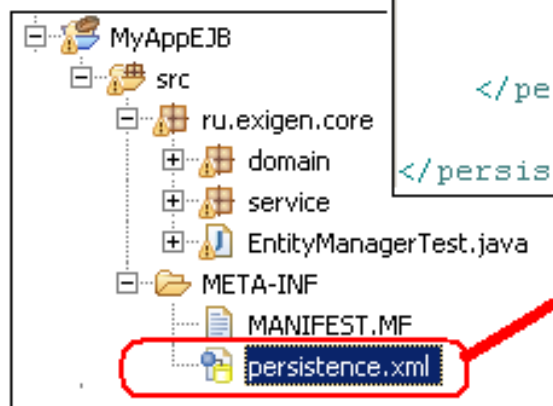
Менеджер сущностей (EntityManager)

- Интерфейс EntityManager служит для управления временем жизни хранимых сущностей
- Пример поддерживаемых операций:
 - ▣ Создание
 - ▣ Поиском
 - ▣ Удалением
 - ▣ и т.д

Контекст хранения (Persistence Context)

- Менеджер сущностей тесно связан с *контекстом хранения (**persistence context**)*.
- Контекст хранения – это множество сущностей, управляемых менеджером сущностей.
- В простейшем случае контекст хранения (и соответствующий менеджер сущностей) предоставляется контейнером (через JNDI).
- В более сложных случаях разработчик может самостоятельно создавать менеджер сущностей.

Конфигурация Persistence Unit (Java EE)



```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">

    <persistence-unit name="MyAppPU" transaction-type="JTA">
        <jta-data-source>java:MyDS</jta-data-source>
        <properties>
            <property name="hibernate.dialect"
                value="org.hibernate.dialect.HSQLDialect" />
            <property name="hibernate.hbm2ddl.auto" value="create-drop" />
        </properties>
    </persistence-unit>

</persistence>
```

Создаем файл Persistence.xml в каталоге META-INF вашего EJB модуля

Конфигурация Persistence Unit (Java SE)

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <persistence xmlns="http://java.sun.com/xml/ns/persistence"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
5     http://java.sun.com/xml/ns/persistence/1_0.xsd" version="1.0">
6
7   <persistence-unit name="JPAPU" transaction-type="RESOURCE_LOCAL">
8     <provider>org.hibernate.ejb.HibernatePersistence</provider>
9     <class>ext.entity.Employee</class>
10    <class>ext.entity.Department</class>
11    <properties>
12      <property name="hibernate.connection.driver_class"
13        value="com.mysql.jdbc.Driver" />
14      <property name="hibernate.connection.url"
15        value="jdbc:mysql://localhost:3306/test" />
16      <property name="hibernate.connection.username" value="root" />
17      <property name="hibernate.connection.password"
18        value="albert" />
19    </properties>
20  </persistence-unit>
21
22 </persistence>
```

EntityManagerFactory Name

Entity classes

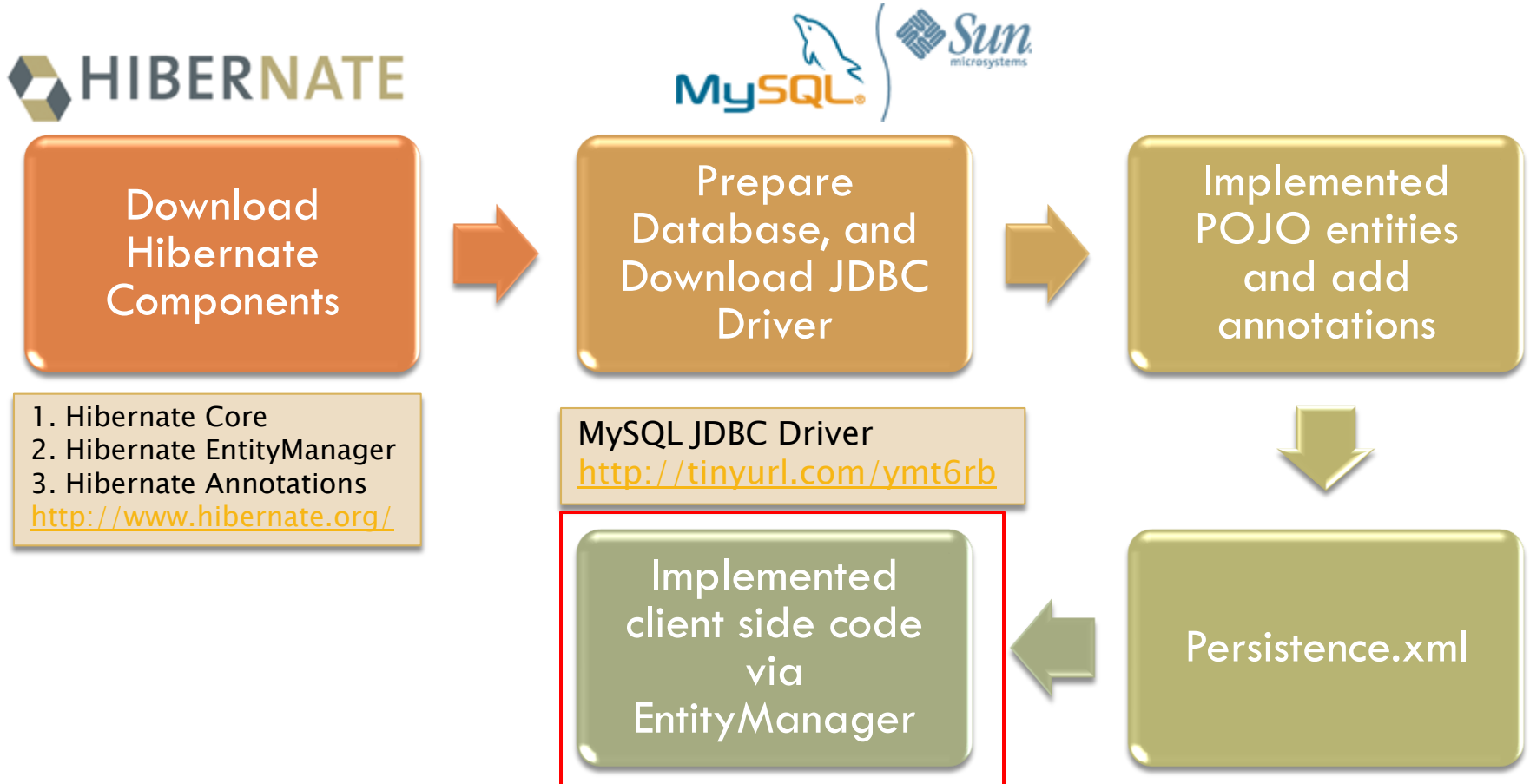
JDBC Driver

JDBC URL

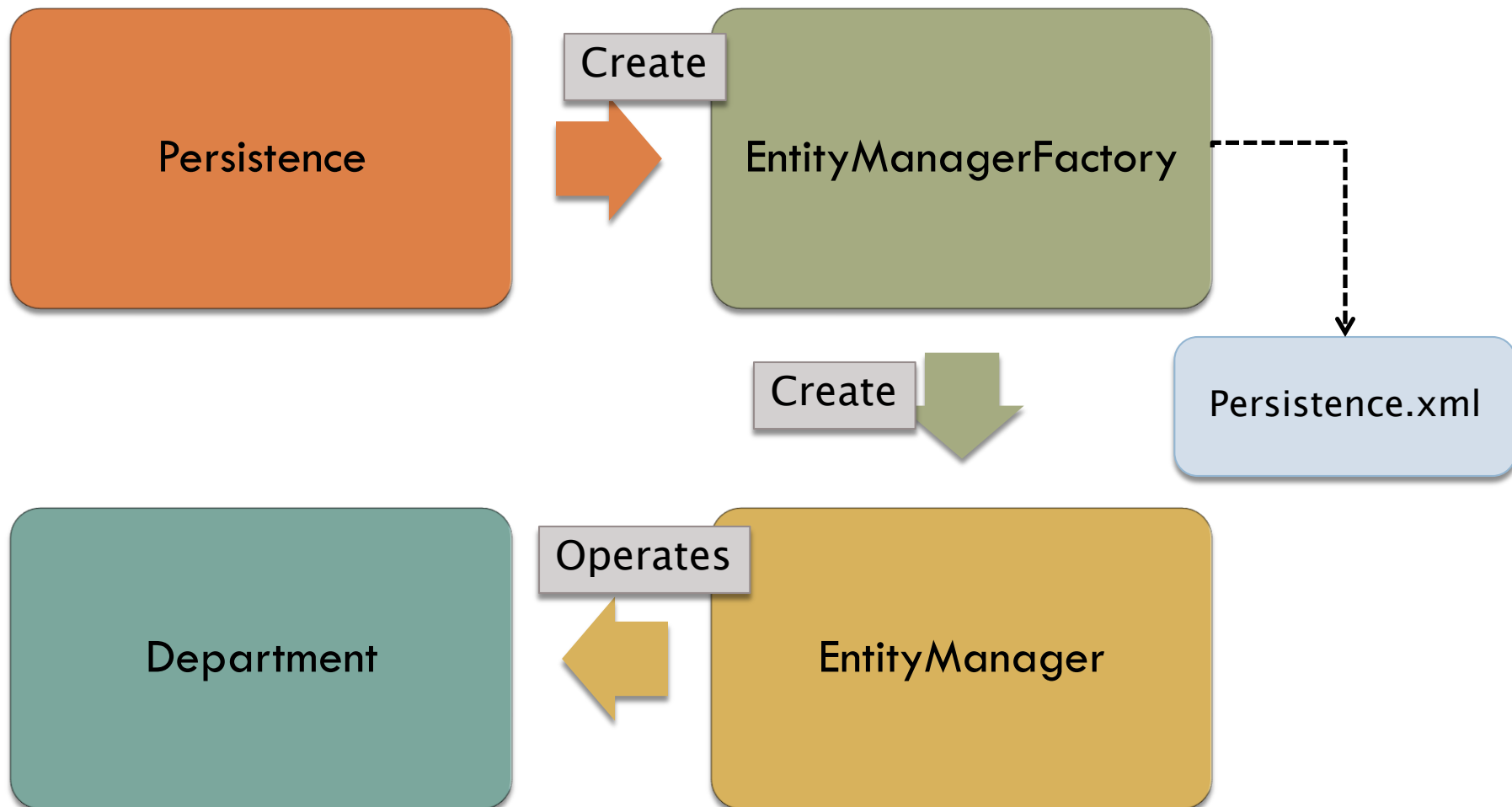
password

User name

Написание кода для управления сущностями через EntityManager



Процесс манипулирования данными JPA



Жизненный цикл сущности

- Сущность может существовать в одном из следующих состояний:
 - **New** – новая сущность, еще не ассоциирована с контекстом хранения
 - **Managed** – сущность ассоциированная с контекстом хранения
 - **Detached** – сущность имеет идентификатор но не ассоциирована с контекстом хранения
 - **Removed** – Сущность ассоциирована с контекстом хранения, но поставленная в очередь для удаления

Основные методы интерфейса EntityManager

- **public void persist(Object entity)** – данный метод позволяет сохранить сущность, представленную объектом сущностного класса в БД
- **public void merge(Object entity)** – данный метод позволяет сохранить или изменить существующую сущность, представленную объектом сущностного класса в БД, с помощью его Detached-версии
- **public void remove(Object entity)** – удалить сущность
- **public <T> T find(Class<T> entityClass, Object primaryKey)** – найти сущность по известному первичному ключу, T – тип сущности (сущностный класс).

Основные методы интерфейса EntityManager

- ❑ **public Query createQuery(String qlString)** – создать запрос БД на языке запросов Java **persistence** Query Language.
- ❑ **public Query createNamedQuery(String name)** – создать заранее predeterminedенный запрос к БД (name – название predeterminedенного запроса).
- ❑ **public Query createNativeQuery(String sqlString)** – создать запрос с использованием языка запросов СУБД (SQL).

Отсутствует метод UPDATE!

Как модифицировать сущность?

- Для операции UPDATE специальный метод не нужен
- Состояние управляемой сущности изменяется внутри транзакции, и все изменения автоматически применяются при завершении транзакции

Пример использования менеджера сущностей предоставляемого контейнером

```
@Stateless
public class PersonService
    implements PersonServiceLocal {
    @PersistenceContext(unitName="MyAppPU")
    EntityManager em;

    public void execute() {
        Person p = new Person();
        p.setName("Person");
        em.persist(p);
        em.flush();
    }
}
```

Пример использования менеджера сущностей (самостоятельное создание)

```
public class EntityManagerTest {

    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("EmployeeService");

        EntityManager em = emf.createEntityManager();
        Collection emps =
            em.createQuery("SELECT e FROM Employee e")
                .getResultList();
        for (Iterator i = emps.iterator(); i.hasNext();) {
            Employee e = (Employee) i.next();
            System.out.println(e.getId() + ", " + e.getName());
        }
        em.close();
        emf.close();
    }
}
```

Пример создания сущности

```
@Stateless
public class PersonService
    implements PersonServiceLocal {

    @PersistenceContext(unitName="MyAppPU")
    EntityManager em;

    public void execute() {

        Customer c = new Customer();
        c.setName("Customer");
        c.setAddress(new Address());
        c.setCardId("123456");
        em.persist(c);
        em.flush();
    }
}
```

```
@Entity
public class Customer extends Person {

    private String cardId;

    @OneToOne (cascade={CascadeType.PERSIST, CascadeType.REMOVE})
    private Address address;
}
```

Пример обновления сущности

```
@Stateless
public class PersonService
    implements PersonServiceLocal {

    @PersistenceContext
    EntityManager em;

    public void execute() {

        Customer cust = em.find(Customer.class, 1L);
        cust.setCardId("444444");

    }
}
```

Пример merge операции

```
@Stateless
public class PersonService
    implements PersonServiceLocal {

    @PersistenceContext(unitName="MyAppPU")
    EntityManager em;

    public void execute(Person detachedPerson) {

        em.merge(detachedPerson);
        em.flush();

    }
}
```

Пример удаления сущности

```
@Stateless
public class PersonService
    implements PersonServiceLocal {

    @PersistenceContext
    EntityManager em;

    public void execute() {

        Customer cust = em.find(Customer.class, 1L);
        em.remove(cust);
        em.flush();

    }
}
```


Пример создания сущности (Java SE)

```
26 public void create() {  
27     // 1. get entity manager  
28     EntityManagerFactory factory = Persistence  
29         .createEntityManagerFactory("JPAPU");  
30     EntityManager entityMgr = factory.createEntityManager();  
31     // 2. prepare entity  
32     Department dept = new Department();  
33     dept.setDeptId(1);  
34     dept.setDeptDesc("test");  
35     // 3. start transaction  
36     entityMgr.getTransaction().begin();  
37     // 4. save entity  
38     entityMgr.persist(dept);  
39     // 5. commit transaction  
40     entityMgr.getTransaction().commit();  
41     // 6. close connection  
42     entityMgr.close();  
43     factory.close();  
44 }
```

Пример поиска сущности (Java SE)

```
46 public void findById(){
47     // 1. get entity manager
48     EntityManagerFactory factory = Persistence
49         .createEntityManagerFactory("JPAPU");
50     EntityManager entityMgr = factory.createEntityManager();
51     // 2. start transaction
52     entityMgr.getTransaction().begin();
53     // 3. find entity by id
54     Department result = entityMgr.find(Department.class, 1);
55     System.out.println(result.getDeptId()+" "+result.getDeptDesc());
56     // 4. commit transaction
57     entityMgr.getTransaction().commit();
58     // 5. close connection
59     entityMgr.close();
60     factory.close();
61 }
```

Пример обновления сущности (Java SE)

```
63 public void update() {  
64     // 1. get entity manager  
65     EntityManagerFactory factory = Persistence  
66         .createEntityManagerFactory("JPAPU");  
67     EntityManager entityMgr = factory.createEntityManager();  
68     // 2. start transaction  
69     entityMgr.getTransaction().begin();  
70     // 3. find entity by id  
71     Department result = entityMgr.find(Department.class, 1);  
72     // 4. give new value  
73     result.setDeptDesc("RD Center");  
74     // 5. commit transaction  
75     entityMgr.getTransaction().commit();  
76     // 6. close connection  
77     entityMgr.close();  
78     factory.close();  
79 }
```

Пример удаления сущности (Java SE)

```
81 public void delete(){
82     // 1. get entity manager
83     EntityManagerFactory factory = Persistence
84         .createEntityManagerFactory("JPAPU");
85     EntityManager entityMgr = factory.createEntityManager();
86     // 2. start transaction
87     entityMgr.getTransaction().begin();
88     // 3. find entity by id
89     Department result = entityMgr.find(Department.class, 1);
90     // 4. delete entity
91     entityMgr.remove(result);
92     // 5. commit transaction
93     entityMgr.getTransaction().commit();
94     // 6. close connection
95     entityMgr.close();
96     factory.close();
97 }
```