# 

[**1. Развёртывание приложения с помощью технологий виртуализации. (Схема, преимущества, недостатки).**](#_ppd6kfuhngma) **2**

[**2. Развёртывание приложения с помощью технологий контейнеризации. (Схема, преимущества, недостатки).**](#_nlckohfbbsrc) **2**

[**3. ОРМ, преимущества и недостатки.**](#_etdyd116nl9n) **2**

[**4. Персистенция, понятие жизненного цикла бизнес-объекта.**](#_7m05hr1yy1xz) **2**

[**5. Бизнес объекты с сохранением статуса, без сохранения статуса, singletone, POJO, транзакционная целостность.**](#_vhm88xbsmml9) **2**

[**6. Микросервисы: преимущества, недостатки, архитектура.**](#_bxi8ah39xt9v) **3**

[**7. Монолитное приложение: преимущества, недостатки, архитектура.**](#_d1gvwk2p60ql) **3**

[**8. Способы конфигурирования приложения с помощью ORM фрэймворка Hibernate.**](#_j3hxloqm2csi) **3**

[**9. Интерфейсы реализации взаимодействия с источниками данных ORM фрэймворка Hibernate. Примеры реализации запросов.**](#_3u6z7anzxyel) **3**

[**10. ОРМ - несоответствие парадигм. Проблема детализации.**](#_27lycj71meie) **3**

[**11. ОРМ - несоответствие парадигм. Проблема подтипов.**](#_73jqfasjl7f) **3**

[**12. ОРМ - несоответствие парадигм. Проблема идентичности.**](#_6zryzptv76j0) **3**

[**13. ОРМ - несоответствие парадигм. Проблемы, связанные с ассоциациями.**](#_ot5rkcdwhu1v) **4**

[**14. ОРМ - несоответствие парадигм. Проблемы навигации по данным.**](#_hoz96acg42le) **4**

[**15. Встраиваемые классы. Переопределение встроенных атрибутов. Отображение вложенных встраиваемых компонентов.**](#_pfh24q5b02p7) **4**

[**16. Отображение наследования. Одна таблица для каждого конкретного класса и неявный полиморфизм.**](#_hqqfzpu3dkh0) **4**

[**17. Отображение наследования. Одна таблица для каждого конкретного класса с объединениями.**](#_tne7l5mzvvm3)

[**4**](#_tne7l5mzvvm3)

[**18. Отображение наследования.. Единая таблица для целой иерархии классов.**](#_nc1ykjpk0v6d)

[**19. Отображение наследования. Одна таблица для каждого подкласса с использованием соединений.**](#_uqyw5ywve1nt) **5**

[**20. Отображение отношений между сущностями. Ассоциация многие-к-одному, один-ко-многим**](#_iha2wfltclue) **5**

[**21. Отображение отношений между сущностями. Ассоциация один-к-одному**](#_50a65js7r2fo) **5**

[**22. Отображение отношений между сущностями. Ассоциация многие-ко-многим**](#_g1rj0nexyh10) **5**

[**23. Отображение коллекции типа Set**](#_c6xuj8s8v3q0) **5**

[**24. Отображение коллекции типа List.**](#_b4suhjruhstw) **5**

[**25. Отображение коллекции тип Map**](#_c0zsk5b8cu37) **5**

[**26. Отображение коллекции компонент**](#_bykay8iavv8b) **5**

[**27. Отображение компонент коллекцией типа Map**](#_omxlvjoqj10g) **6**

# 

# 1. Развёртывание приложения с помощью технологий виртуализации. (Схема, преимущества, недостатки).

## Схема



## Плюсы

1. Виртуализация же дает возможность запускать на одном физическом компьютере несколько серверов
2. Замена физических компьютеров на программно-определяемые ВМ существенно упрощает определение политик для ПО и управление ими
3. Виртуализация позволяет запускать несколько резервных виртуальных машин параллельно и переключаться между ними в случае возникновения неполадок

## Минусы

1. обеспечение единовременной работы нескольких виртуальных машин потребует достаточного количества аппаратных мощностей
2. операционная система виртуальной машины может работать медленнее, чем на аналогичном аппаратном обеспечении

## Доп

Виртуализация использует программное обеспечение для создания уровня абстракции поверх оборудования компьютера, позволяющего разделять аппаратные компоненты одного компьютера на несколько виртуальных машин. Каждая ВМ имеет свою собственную операционную систему (ОС) и работает как независимый компьютер, даже если использует только часть оборудования базового компьютера.

### Еще доп

эта технология лежит в основе экономики [облачных вычислений](https://www.ibm.com/ru-ru/cloud/learn/cloud-computing-gbl). Виртуализация позволяет облачным провайдерам предоставлять клиентам услуги, используя свое физическое оборудование, а пользователи облачных услуг могут приобретать только те вычислительные ресурсы, которые им необходимы в данный момент, и масштабировать их по мере роста потребностей.

# 2. Развёртывание приложения с помощью технологий контейнеризации. (Схема, преимущества, недостатки).

## Схема



## Плюсы

1. Контейнер можно создать быстрее, чем ВМ
2. Контейнер занимает меньше места в хранилище
3. Можно мониторить версионность контейнеров

## Минусы

1. Рост количества контейнеров, работающих с приложением, влияет на сложность управления ими
2. Нередко в контейнеры упаковывается гораздо больше ресурсов, чем реально требуется. Из-за этого образ разрастается, занимая больше места на диске

## Доп

Контейнеризация — это объединение программного кода со всеми его необходимыми компонентами, такими как библиотеки, фреймворки и другие зависимости, чтобы они были изолированы в своем собственном « [контейнере](https://www.redhat.com/en/topics/containers/whats-a-linux-container) ».

Это делается для того, чтобы программное обеспечение или [приложение](https://www.redhat.com/en/topics/cloud-native-apps/what-are-cloud-applications) внутри контейнера можно было перемещать и последовательно запускать в любой среде и на любой инфраструктуре, независимо от этой среды или операционной системы инфраструктуры.

### Еще доп

[Docker](https://www.redhat.com/en/topics/containers/what-is-docker) представил Docker Engine в 2013 году, он установил стандарт использования контейнеров с инструментами, которые были просты в использовании для разработчиков, а также с универсальным подходом к упаковке, который затем ускорил разработку, внедрение [контейнерной технологии](https://www.redhat.com/en/topics/containers) .

# 3. ОРМ, преимущества и недостатки.

**ORM** (англ. Object-Relational Mapping) – это технология программирования, связывающая СУБД с объектно-ориентированным языком программирования, создавая «виртуальную объектную базу данных» Классы будут соответствовать таблицам в базе, а экземпляры этих классов – конкретным строкам таблицы. **Основные преимущества использования ORM**:

* наличие явного описания схемы БД, представленное в терминах какого-либо языка программирования;
* возможность оперировать элементами языка программирования, т.е. классами, объектами, атрибутами, методами
* возможность автоматического создания SQL-запросов;
* не нужно создавать новые SQL-запросы при переносе на другую СУБД.
* ORM избавляет от необходимости работы с SQL и проработки значительного количества программного кода, который зачастую однообразен и подвержен ошибкам.

**недостатки**:

* Эффективность выполнения программы и режим фиксированного мышления приносятся в жертву, что снижает гибкость разработки.
* Когда мы разрабатываем системы, обычно возникают проблемы с производительностью. Код, сгенерированный ORM, как правило, вряд ли напишет очень эффективный алгоритм

# 4. Персистенция, понятие жизненного цикла бизнес-объекта.

Персистенция данных - это средство позволяющее приложению сохранять и извлекать информацию из базы данных. Java Persistence API предоставляет разработчикам Java средство объектно-реляционного сопоставления для управления реляционными данными в приложениях Java

жизненный цикл объекта:

* transient object. Объекты в данном статусе — это заполненные экземпляры классов сущностей. Могут быть сохранены в БД. Не присоединены к сессии. Поле id не должно быть заполнено, иначе объект имеет статус detached ;
* persistent object. Объект в данном статусе — так называемая хранимая сущность, которая присоединена к конкретной сессии. Только в этом статусе объект взаимодействует с базой данных. При работе с объектом данного типа в рамках транзакции все изменения объекта записываются в базу;
* detached object. Объект в данном статусе — это объект, отсоединённый от сессии, может существовать или не существовать в БД.

Согласно JPA объект сущности может иметь один из четырех статусов жизненного цикла:

1. new - объект создан, не имеет primary key, не является частью контекста персистентности (не управляется JPA);
2. managed - объект создан, имеет primary key, является частью контекста персистентности (управляется JPA);
3. detached - объект создан, имеет primary key, не является (или больше не является) частью контекста персистентности (не управляется JPA);
4. removed - объект создан, является частью контекста персистентности (управляется JPA), будет удален при commit-е транзакции.

# 5. Бизнес объекты с сохранением статуса, без сохранения статуса, singletone, POJO, транзакционная целостность.

## Singleton

Singleton гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа. При попытке создания данного объекта он создается только в том случае, если ещё не существует, в противном случае возвращается ссылка на уже существующий экземпляр и нового выделения памяти не происходит.

## Транзакционная целостность.

Под этим термином мы понимаем, что любое обращение к серверу приложений должно либо завершиться успешно, либо все изменения должны быть отменены. Соответственно, при начале обработки серверного вызова создается транзакция и фиксируется (или отменяется) при выходе из вызова

## Статусы

POJO — объект может находиться в одном из 3-х состояний (статусов):

* transient object. Объекты в данном статусе — это заполненные экземпляры классов сущностей. Могут быть сохранены в БД. Не присоединены к сессии. Поле id не должно быть заполнено, иначе объект имеет статус detached ;
* persistent object. Объект в данном статусе — так называемая хранимая сущность, которая присоединена к конкретной сессии. Только в этом статусе объект взаимодействует с базой данных. При работе с объектом данного типа в рамках транзакции все изменения объекта записываются в базу;
* detached object. Объект в данном статусе — это объект, отсоединённый от сессии, может существовать или не существовать в БД.

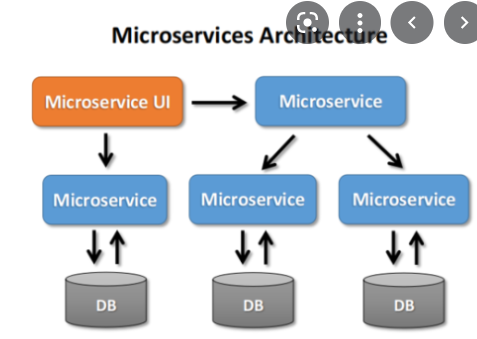
## POJO

Plain Old Java Object — «старый простой Java-объект», простой [Java](https://ru.wikipedia.org/wiki/Java)-объект, не унаследованный от какого-то специфического объекта и не реализующий никаких служебных интерфейсов сверх тех, которые нужны для [бизнес-модели](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%B7%D0%BD%D0%B5%D1%81-%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C).

### Доп про POJO

1. POJO объект должен реализовывать интерфейс Serializable (implements Serializable);
2. POJO объект должен иметь конструктор без параметров (по умолчанию);
3. POJO объект должен иметь методы доступа к свойствам объекта getXXX/setXXX с областью видимостью public, при этом свойства должны иметь область видимости private или protected;
4. POJO объект может иметь методы для манипуляции с атрибутами объекта.

# 6. Микросервисы: преимущества, недостатки, архитектура.

Архитектурный стиль микросервисов — это подход, при котором система строится как набор независимых и слабосвязанных сервисов, которые можно создавать, используя различные языки программирования и технологии хранения данных. Концепция микросервисов позволяет поддерживать слабую связанность сервисов в процессе работы над системой.

* Плюсы микросервисов
  1. Простое масштабирование
  2. Улучшенная отказоустойчивость
  3. Простота понимания кодовой базы
  4. Возможности для экспериментов
  5. Независимое развертывание
* Минусы микросервисов
  1. Повышенная сложность связи
  2. Требуется больше ресурсов
  3. Глобальное тестирование и отладка — это сложно
  4. Не практично для небольших приложений
  5. Относительно сложное развертывание

# 7. Монолитное приложение: преимущества, недостатки, архитектура.

Монолитное приложение (назовем его монолит) представляет собой приложение, доставляемое через единое развертывание

**Достоинства**

1. Большим преимуществом монолита является то, что его легче реализовать. В монолитной архитектуре можно быстро начать реализовывать свою бизнес-логику, вместо того чтобы тратить время на размышления о межпроцессном взаимодействие.
2. Еще одна вещь — это сквозные (End-to-end) тесты. В монолитной архитектуре их легче выполнить.

**Недостатки**

1. Если архитектурные правила были нарушены и со временем компоненты срослись, то это замедляет процесс разработки: каждую будущую функцию будет сложнее развивать.
2. В монолите практически нет изоляции. Проблема или ошибка в модуле может замедлить или разрушить все приложение.
3. Строительство монолита часто протекает с помощью выбора основы. Отключение или обновление нашего первоначального выбора может быть затруднительным, потому что это должно быть сделано сразу и для всех частей нашей системы.

# 8. Способы конфигурирования приложения с помощью ORM фрэймворка Hibernate.

Существует 4 способа конфигурации:

1. аннотации
2. hibernate.cfg.xml
3. hibernate.properties
4. persistence.xml

## Доп

Самый частый способ конфигурации: через аннотации и файл persistence.xml, что касается файлов hibernate.properties и hibernate.cfg.xml, то hibernate.cfg.xml главнее (если в приложение есть оба файла, то принимаются настройки из файла hibernate.cfg.xml). Конфигурация аннотациями, хоть и удобна, но не всегда возможна, к примеру, если для разных баз данных или для разных ситуаций вы хотите иметь разные конфигурацию сущностей, то следует использовать xml файлы конфигураций.

### Еще доп

**Из xml файла hibernate.cfg.xml**

<hibernate-configuration>

<session-factory>

<!-- Настройки подключения к БД -->

<property name="connection.driver\_class">org.postgresql.Driver</property>

<property name="connection.url">jdbc:postgresql://192.168.1.110:5432/shop</property>

<property name="connection.username">postgres</property>

<property name="connection.password">postgres</property>

</session-factory>

</hibernate-configuration>

**Из java кода**

Configuration configuration = new Configuration();

if (configFile != null) {

try {

configuration.configure(configFile);

} catch(Exception ex){

LOG.error("Error loading hibernate.cfg: ", ex);

configuration.configure();

}

} else {

configuration.configure();

}

ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder()

.applySettings(configuration.getProperties()).build();

MetadataSources metadataSources = new MetadataSources(serviceRegistry);

# 9. Интерфейсы реализации взаимодействия с источниками данных ORM фрэймворка Hibernate. Примеры реализации запросов.

## Native

NativeQuery<User> query = session.createNativeQuery("select \* from USER", User.class);

List<User> results = query.list();

## Criteria

CriteriaBuilder cb = session.getCriteriaBuilder();

CriteriaQuery<User> cr = cb.createQuery(User.class);

Root<User> root = cr.from(User.class);

cr.select(root);

Query query = session.createQuery(cr);

List<User> results = query.getResultList();

## HQL

list = session.createQuery("from User", User.class).list();

# 10. ОРМ - несоответствие парадигм. Проблема детализации.

Есть класс User

class User {

String name;

Address address;}

И класс Address

class Address {

String city,

String country}

Теперь имеется выбор между добавлением нескольких столбцов или одного (нового типа данных SQL). Это и является проблемой детализации(problem of granularity)

# 11. ОРМ - несоответствие парадигм. Проблема подтипов.

SQL не предоставляет нативной поддержки наследования и полиморфных запросов.

В базах данных SQL нет способа представления полиморфной ассоциации. Ограничение внешнего ключа ссылается только на одну таблицу; нет простого способа определить внешний ключ, ссылающийся на несколько таблиц.

# 12. ОРМ - несоответствие парадигм. Проблема идентичности.

Java определяет два различных понятия тождественности:

* идентичность экземпляров (грубо говоря, совпадение адресов в памяти; проверяется как a == b);
* равенство экземпляров, определяемое методом equals() (также называется *равенством по значению*).

С другой стороны, идентичность записей в базе данных определяется сравнением значений первичного ключа. Как будет показано в разделе 10.1.2, ни equals(), ни оператор == не всегда эквивалентны сравнению значений первичного ключа.

Для этого создается столбец суррогатного ключа – это столбец первичного ключа, не имеющий значения для пользователя приложения, другими словами, это ключ, скрытый от пользователя приложения. Его единственная цель – идентифицировать данные внутри приложения

# 13. ОРМ - несоответствие парадигм. Проблемы, связанные с ассоциациями.

# В объектно-ориентированных языках ассоциации представлены *объектными ссылками*; но в реляционном мире *столбец внешнего ключа* будет представлять ассоциацию при помощи дублирования значений этого ключа. Ограничение – это правило, гарантирующее целостность ассоциации. Между этими двумя способами существенные различия.

Объектные ссылки по своей природе обладают направленностью; ассоциация идет от одного экземпляра к другому. Они – указатели.

*Навигация* в конкретном направлении не имеет смысла для реляционной модели данных, потому что можно создавать произвольные ассоциации при помощи операций *соединения* и *проекции*.

## Доп

Ассоциации в Java могут иметь вид *многие ко многим*.Чтобы выразить ассоциацию *многие ко многим* в базе данных SQL, придется создать дополнительную таблицу, также называемую *таблицей ссылок* (link table). В большинстве случаев эта таблица отсутствует в предметной модели.

# 14. ОРМ - несоответствие парадигм. Проблемы навигации по данным.

Для доступа к данным в ООП используются последовательные переходы от родительского объекта к свойствам дочерних элементов и инициализации объектов по необходимости. Такой подход считается не эффективным способом извлечения данных из реляционных баз данных. Как правило, количество запросов к БД должно быть сведено к минимуму, необходимые сущности должны по возможности загружаться сразу с использованием JOIN-ов. Но для эффективного использования соединения нужно *заранее* знать, какое подмножество графа объектов понадобится посетить! Но можно также нагрузить базу данных SQL большим декартовым произведением результирующих наборов.

## Доп

Каждая достойная применения система долговременного хранения объектов обеспечивает возможность извлечения данных ассоциированных экземпляров, только когда ассоциация действительно задействуется в Java-коде. Это называется *«отложенной загрузкой»*: данные извлекаются, только когда они действительно необходимы. Такой последовательный стиль доступа к данным крайне неэффективен в контексте баз данных SQL, поскольку требует выполнения одного выражения для каждого узла или коллекции графа объектов, по которому осуществляется обход. Это проблема *n + 1 запроса*.

# 15. Встраиваемые классы. Переопределение встроенных атрибутов. Отображение вложенных встраиваемых компонентов.

При разработке приложения довольно часто приходится сталкиваться с пользовательскими типа, или со структурами данных, которые используются в множестве POJO классах. Один из способов решения данной проблемы, в Java Persistence спецификации, предполагает использование встроенного класса. Таким образом, встроенный класс в POJO сущности отмечается с помощью аннотации @Embedded при его описании. Предположим, нам нужно к сущности User добавить информацию об адресе каждого пользователя:  
*@Entity  
public class User {  
...  
@Embedded*

*private Address homeAddress;*

*...}*

При этом ORM будет искать класс Address с аннотацией @Embeddable. Поэтому встраиваемый класс Address будет выглядеть следующим образом:

*@Embeddable*

*public class Address {*

*private String street;*

*private String zipcode;*

*private String city;*

*...}*

# 16. Отображение наследования. Одна таблица для каждого конкретного класса и неявный полиморфизм.

Все свойства потомка включая свойства родителя отображаются в одной таблице. Основная проблема данного способа заключена в поддержке полиморфизма. Кроме того для получения всех объектов родительского касса, необходимо выполнить два запроса к таблицам потомков. UML диаграмма классов:

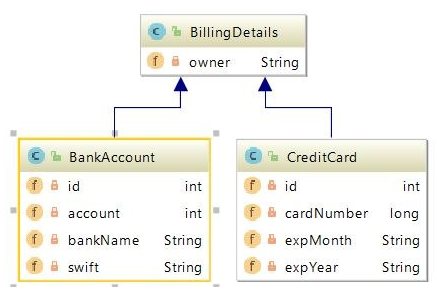
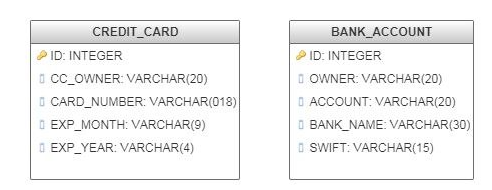
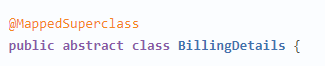


Схема БД выглядит так:



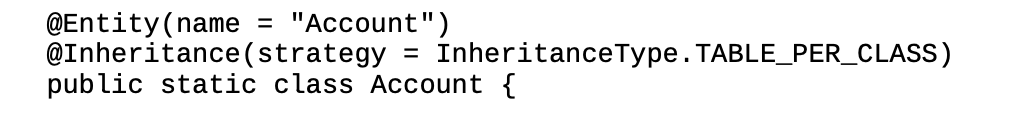
Аннотация для данного способа



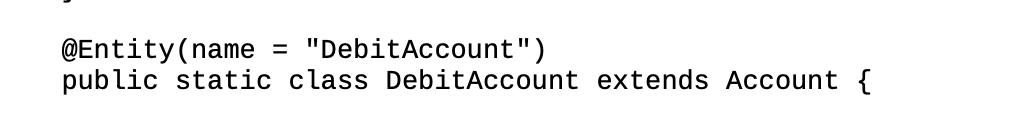
# 17. Отображение наследования. Одна таблица для каждого конкретного класса с объединениями.

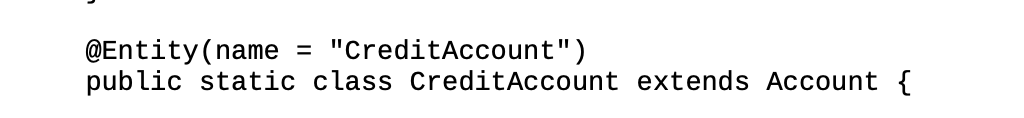
Каждая таблица определяет персистентное состояние каждого класса иерархии. Чтобы получить все экземпляры родительского класса, необходимо сделать объедение всех таблиц.

Класс родитель:



Классы наследники:



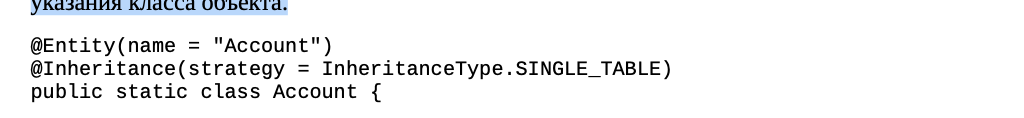


Данная стратегия дает важное преимущество, так как возможна реализация отношения ассоциации от внешней сущности к родительскому классу, за счет использования предиката UNION и представления двух таблиц в одну.

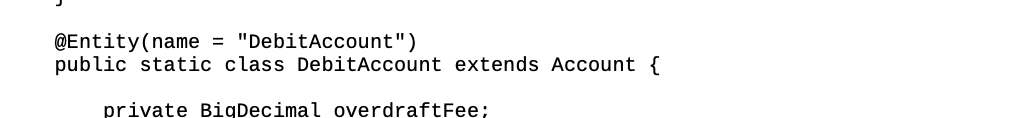
# 18. Отображение наследования. Единая таблица для целой иерархии классов.

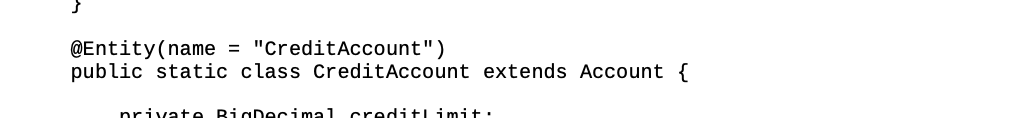
Вся иерархия классов описывается одной таблицей, с реализацией полиморфизма за счёт денормализации таблицы и использованием специального столбца — дискриминатора для указания класса объекта.

Класс родитель:



Классы наследники:



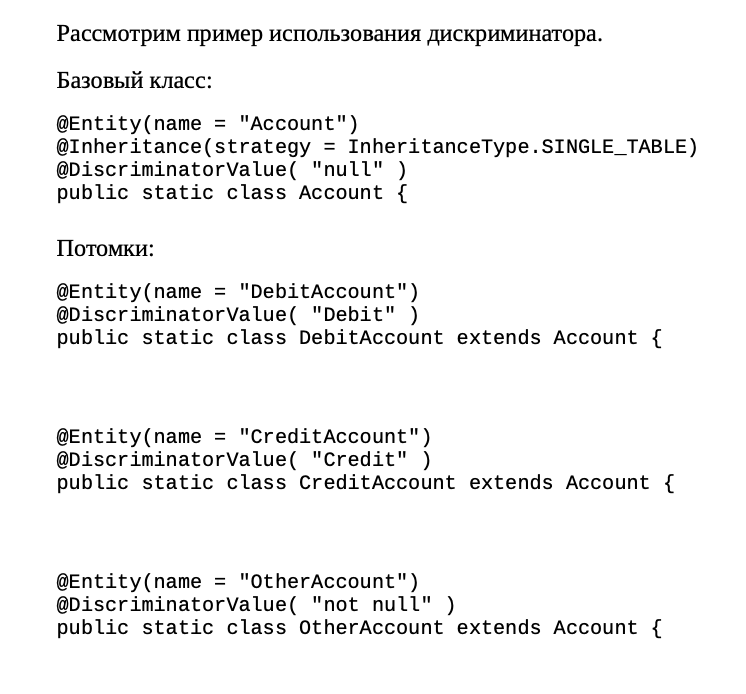


Каждый подкласс иерархии должен иметь уникальное значение дискриминатора, которое используется для идентификации класса по строке таблицы. Если дискриминатор не описан с помощью аннотаций, то по умолчанию в качестве значений дискриминатора используются имена подклассов.

## Доп

Библиотека hibernate-core поддерживает следующие типы данных при определении значения дискриминатора: String, char, int, byte, short, boolean(including yes\_no, true\_false).

### Еще доп



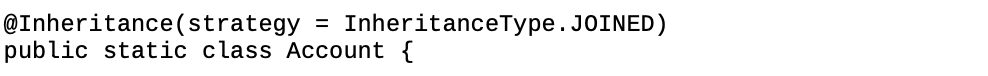
Из примера следует, что класс Account имеет @DiscriminatorValue( "null" ), это означает, что любая строка таблицы не содержащая значения дискриминатора отображает базовый класс. The DebitAccount and CreditAccount используют заданные значения дискриминатора для идентификации класса. The OtherAccount класс используется как обобщенный тип, потому что может отображать любую строку которая явно не связана с текущей иерархией.

# 19. Отображение наследования. Одна таблица для каждого подкласса с использованием соединений.

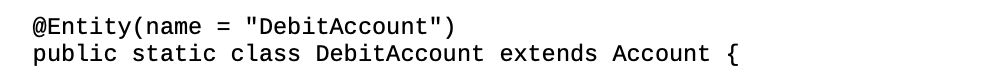
Базовый класс и все подклассы отображаются в собственной таблице. Поля таблицы соответствуют атрибутам сущности иерархии. Первичный ключ таблицы родительского класса является также внешним ключом для связи с таблицами подклассов. Для получения экземпляра подкласса из БД необходимо соединить строку из таблицы родительского класса со строкой из таблицы подкласса (использовать join). Кроме того, с помощью ограничения внешнего ключа, выполняется реализация полиморфизма.

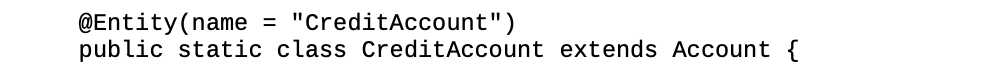
Класс родитель:





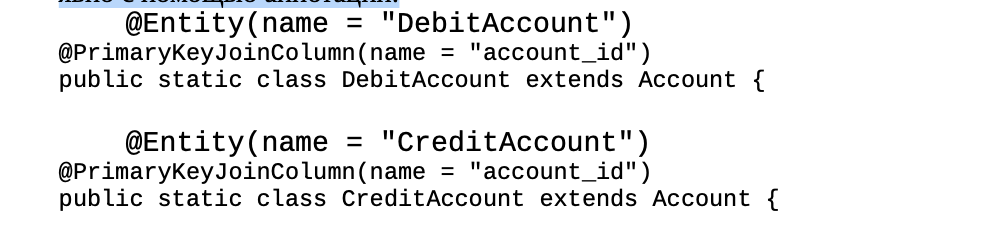
Классы наследники:





## Доп

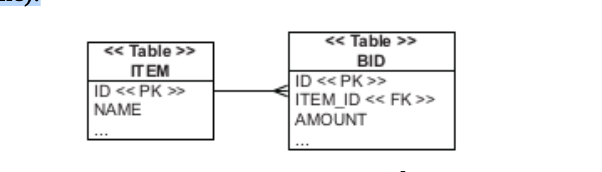
Если не указывать столбец внешнего ключа в таблице подкласса явно с помощью аннотации, то по умолчанию, для имени столбца будет использовано имя подкласса. Однако это можно указать явно с помощью аннотации:



# 20. Отображение отношений между сущностями. Ассоциация многие-к-одному, один-ко-многим

(То что выделено серым необязательно)

Ассоциация двух сущностей в схеме БД, BID и ITEM реализуется отношением многие-к- одному (many-to-one):



**Однонаправленная** ассоциация в сущности Bid определяется:

Тип ассоциации указывается с помощью аннотации @ManyToOne, с помощью атрибутта fetch, указывается тип подгрузки связанной сущности.

**Двунаправленная** ассоциация между сущностями Item и Bid позволит решить следующие проблемы:

* выполнять навигацию по связанному объекту, избегая дополнительных запросов;
* возможность управлять жизненным циклом связанной сущности, настраивая параметр
* каскадного изменения статуса для различных вариантов изменения.

Реализация ассоциации один-ко-многим для сущности Bid остаётся неизменной а для сущности Item добавляются следующие аннотации:



Аннотация @OneToMany является обязательной, и определяет ссылку на поле связанной сущности mappedBy = "item". Параметр mappedBy является обязательным при ассоциации один- ко-многим когда она является двунаправленной, а связанная сущность использует ограничение внешнего ключа для реализации ассоциации многие-к-одному.

## Доп

Главное отличие отображения отношений между сущностями от отображения коллекций заключается в следующем:

* коллекция это набор значений, которые дополняют сущность и являются ее составляющей, жизненный цикл коллекции зависит то жизненного цикла сущности.
* сущности находящиеся в отношении между собой имеют независимый жизненный цикл, т.е. над экземпляром каждой сущности можно выполнять любую CRUD операцию при этом никакого воздействия на связанную сущность не будет.

# 21. Отображение отношений между сущностями. Ассоциация один-к-одному

Один из самых простых и лучших способов для реализации ассоциации один-к-одному, является возможность использования встроенных типов с помощью аннотации @Embeddable. Однако это не совсем удобно, например, когда одну и туже сущность Address могут использовать несколько сущностей: Заказчик, Поставищик, Исполнитель. В этом случае сущность Address должна поддерживать ссылки и идентификацию.

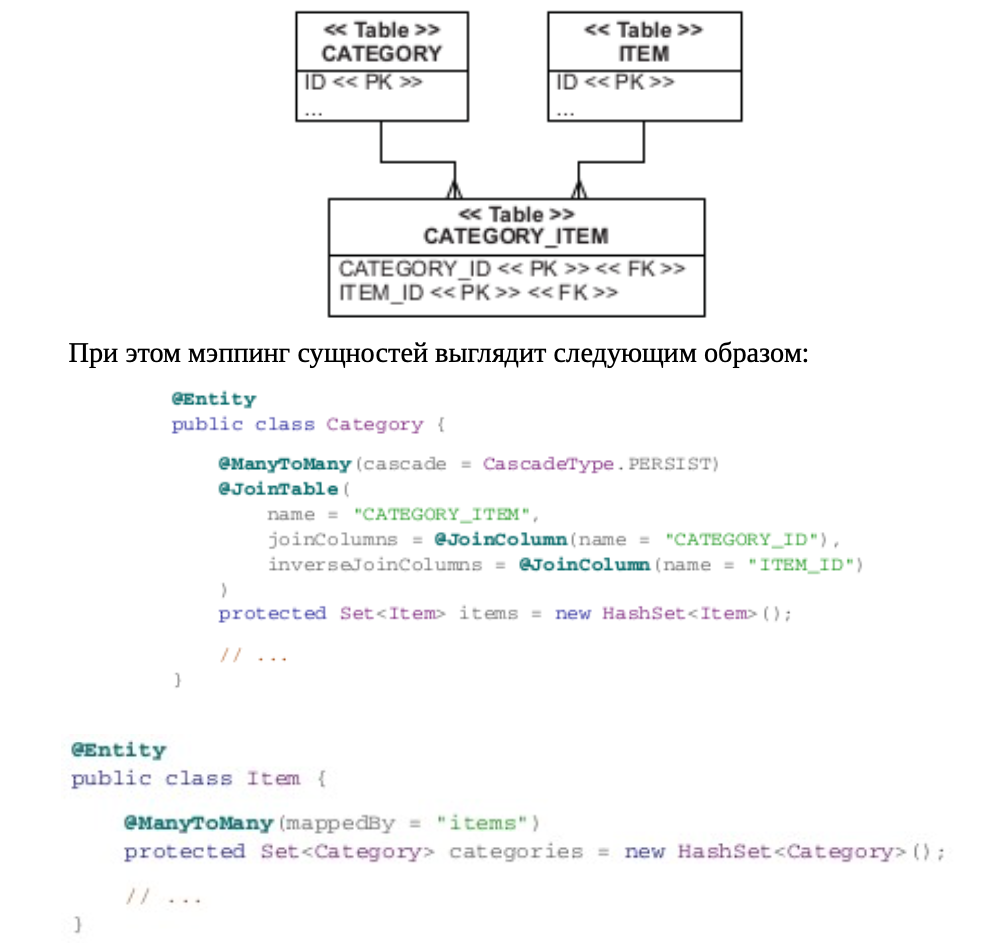
## Доп

Существует несколько способов реализации ассоциации один-к-одному:

1. Разделяемый первичный ключ.
2. Генерируемый внешний первичный ключ
3. Внешний ключ

# 22. Отображение отношений между сущностями. Ассоциация многие-ко-многим

Данный тип ассоциации в реальных системах используется редко. Рассмотрим ее на примере схемы БД:



# 23. Отображение коллекции типа Set

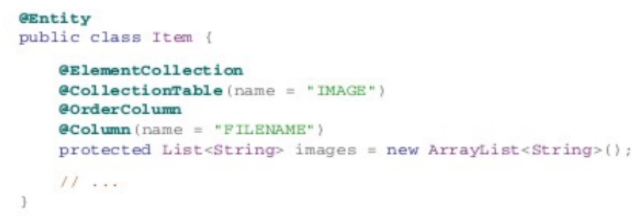
Пометить ее обязательной аннотацией @ElementCollection. Указать ссылку на таблицу-источник коллекции - @CollectionTable и столбец значения которого будут использованы в коллекции - @Column:



Однако, аннотации @CollectionTable и @Column не являются обязательными. В случае их отсутствия он будет пытаться использовать значения по умолчанию из схемы данных.

# 24. Отображение коллекции типа List.

В силу того, что интерфейс List допускает наличие неуникальных значений в коллекции, обязательным является наличие индексного столбца, который указывает на номер строки значения в коллекции. Для это добавляется аннотация @OrderColumn:



Таким образом в таблицу IMAGE добавляется новый столбец IMAGES\_ORDER, который вместе со столбцом IMAGE\_ID образует композитный первичный ключ:

Это позволяет использовать повторяющиеся значения имен файлов в коллекции images. При удалении любого значения из коллекции, автоматически выполняется переиндексация оставшихся значений.

# 25. Отображение коллекции тип Map

Каждая строка коллекции типа Map представляет собой набор ключ-значение.

Дублирование ключей в коллекции типа Map недопустимо. Для реализации коллекции типа имя файла — имя рисунка ( FILENAME — IMAGENAME) необходимо с помощью аннотации @MapKeyColumn указать ключевой столбец, а с помощью @Column - столбец значений:



# 26. Отображение коллекции компонент

Жизненный цикл дочернего объекта определяется родительским. Дочерний объект

описывается с помощью известной аннотации @Embeddable, а нужные аттрибуты с помощью @Column:



Отображение коллекции компонент типа Set имеет следующий вид:



Для получения родительского объекта из дочернего необходимо добавить аннотацию

@org.hibernate.annotations.Parent :



# 27. Отображение компонент коллекцией типа Map

Отображение компонент коллекцией типа Map.

В качестве ключа в родительском классе используется имя файла:

