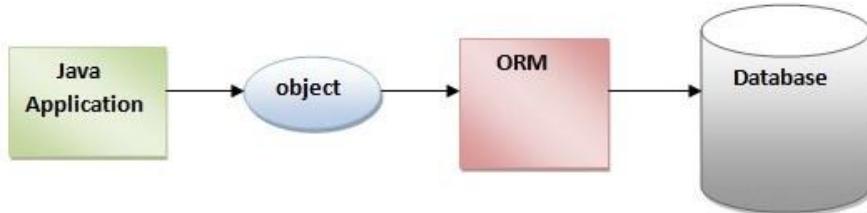


1. Что такое ORM?

Object Relational Mapping – способ представления Java Объекта в базе данных, объектно-реляционное сопоставление, облегчает работу разработчика.

ORM преобразует объект в строку в таблице и обратно, используя **JDBC API** для взаимодействия с базой данных.



Данные в таблицах (реляционные БД) связаны логикой предметной области.

Одна строка/запись = один объект.

Характеристики (атрибуты) объекта описаны в полях класса и размещаются в колонках таблиц.

- Запись – инфо об одном объекте.
- Поле – это атрибут, х-ка объекта.
- Имя поля – название колонки.

РЕЛЯЦИОННЫЕ БАЗЫ ДАННЫХ



Проблема:

Когда мы хотим, чтобы наше приложение получило доступ к информации, хранящейся в БД, мы должны понимать крайне важную деталь. Существует огромная разница между **объектной моделью** и **реляционной**.

Например, после того, как мы создали и java-класс и таблицу в БД, нам необходимо изменить БД, у нас сразу же возникает проблема. К тому же, когда мы записываем или читаем данные в/из БД, у нас есть 5 проблем, которые связаны с разницей между объекто-ориентированной (далее – ОО) моделью и реляционной моделью:

1) Наследование

В реляционной модели нет никакого понятия, похожего на наследование, которое является одним из ключевых принципов ООП.

2) Идентификация

Для БД есть только одна сущность, по которому объект может быть идентифицирован – это Первичный Ключ (Primary Key).

В то время, как в Java у нас есть такие вещи, как (entity1 == entity2) и (object1.equals(object2)).

3) Абстракция

В Java мы используем ссылки на объекты для абстракции, а в реляционной модели – Внешний Ключ (Foreign Key).

4) Доступ

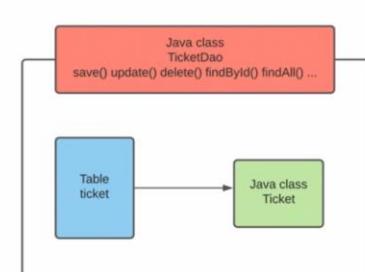
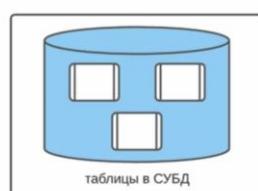
В Java и в реляционной БД абсолютно разные способы получения доступа в объекту.

5) Инкапсуляция

Крайне часто, при разработке приложений, вы будете сталкиваться с тем, что наша ОО модель имеет больше классов, чем таблиц в нашей БД.

Объектно-ориентированная модель (Java)

Реляционная модель (СУБД)



Если нужно создать связь между Объектом и реляционной сущностью, поможет Объектно-Реляционное-Связывание или же – ORM (Object Relational Mapping).

Преимущества ORM в сравнение с JDBC

- Позволяет бизнес методам обращаться не к БД, а к Java-классам
- Ускоряет разработку приложения
- Основан на JDBC
- Отделяет SQL-запросы от ОО модели
- Позволяет не думать о реализации БД
- Сущности основаны на бизнес-задачах, а не на структуре БД
- Управление транзакциями

ORM состоит из:

- API, который реализует базовые операции (СОЗДАНИЕ, ЧТЕНИЕ, ИЗМЕНЕНИЕ, УДАЛЕНИЕ) объектов-моделей.
- Средства настройки метаданных связывания
- Технику взаимодействия с транзакциями, которая позволяет реализовать функции dirty checking, lazy association fetching и т.д.

Самыми распространёнными ORM фреймворками являются: Hibernate, Spring DAO...

ORM регулирует SQL запросы.



CRUD методы: **C**reate/**I**NSE**T**; **R**ead/**S**E**C**T; **U**pdate/**U**P**D**E; **D**elete/**D**E**L**E**T**E.

2. Что такое JPA?

JPA – это стандартная спецификация, описывающая систему управления сохранением Java объектов в базу данных.

JPA (Java Persistence API) описывает правила, а **Hibernate** использует их.

Это спецификация (стандарт, технология), обеспечивающая объектно-реляционное отображение простых JAVA-объектов (Plain Old Java Object - POJO) и предоставляющая универсальный API для сохранения, получения и управления такими объектами.

Сам JPA не умеет ни сохранять, ни управлять объектами, JPA только определяет правила игры: как должен действовать каждый провайдер (Hibernate, EclipseLink, OJB, Torque и т.д.), реализующий стандарт JPA.

Также JPA определяет правила, как должны описываться метаданные отображения и как должны работать провайдеры.

ORM, JPA, Hibernate и как они связаны.

Hibernate — одна из наиболее популярных реализаций ORM-модели.

ORM, объектно-реляционная модель описывает отношения между программными объектами и записями в БД.

Hibernate — самая популярная спецификация JPA (JPA описывает правила, Hibernate их реализует)

*Что появилось раньше JPA или Hibernate?

На основе **Hibernate** (2001) появилась спецификация JPA (2006). Хибер раньше.

3. Что такое Hibernate?

Hibernate – это самая **популярная спецификация JPA** и является одним из самых востребованных ORM фреймворков для Java.

Главная цель – создание объектного слоя между кодом и БД, чтобы работать с таблицами, как с объектами.

Hibernate устраняет многословный спагетти код (повторяющийся), который постоянно преследует разработчика при работе с JDBC. Скрывает от разработчика множество кода, необходимого для управления ресурсами и позволяет сосредоточиться на бизнес логике.

Hibernate поддерживает XML так же, как и JPA аннотации, что позволяет сделать реализацию кода независимой.

Hibernate предоставляет собственный мощный язык запросов (HQL), который похож на SQL. Стоит отметить, что HQL полностью объектно-ориентирован и понимает такие принципы, как наследование, полиморфизм и ассоциации (связи).

Hibernate — широко распространенный open source проект. Благодаря этому доступны тысячи открытых статей, примеров, а также документации по использованию фреймворка.

Hibernate легко интегрируется с другими Java EE фреймворками, например, Spring Framework поддерживает встроенную интеграцию с Hibernate.

Hibernate поддерживает ленивую инициализацию используя **PROXY объекты** и выполняет **запросы к базе данных только по необходимости**.

Hibernate поддерживает разные уровни кэширования, а следовательно, может повысить производительность.

Важно, что Hibernate может использовать чистый SQL, а значит поддерживает возможность оптимизации запросов и работы с любым сторонним вендором БД и его фичами.

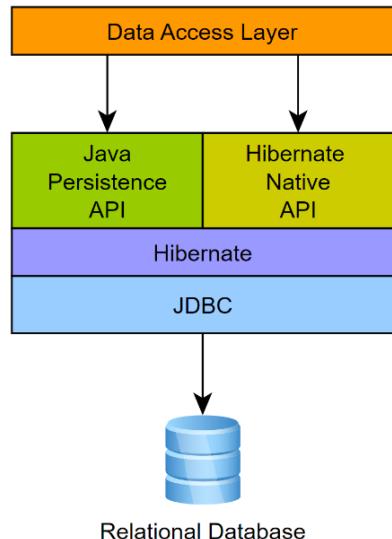
Hibernate **неявно** использует управление транзакциями (большинство запросов нельзя выполнить вне транзакций).

Hibernate использует HibernateException (unchecked), а значит нет необходимости проверять их в коде каждый раз.

Hibernate поддерживает все основные СУБД: MySQL, Oracle, PostgreSQL, Microsoft SQL Server Database, HSQL, DB2.

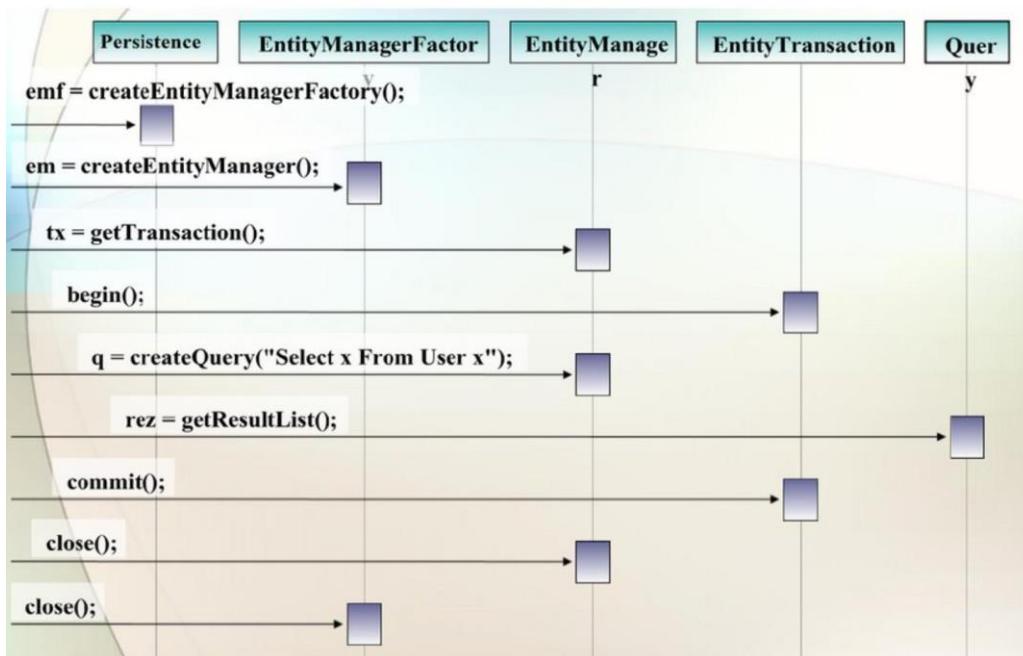
* Какие ключевые интерфейсы использует Hibernate?

Session	Интерфейс Session является основным инструментом, используемым для связи с Hibernate. Он предоставляет API, позволяющий создавать, читать, обновлять и удалять постоянные объекты. Сессия имеет простой жизненный цикл. Открываем, выполняем какие-то операции, а потом закрываем.
SessionFactory	SessionFactory — это интерфейс, который присутствует в org. hibernate, используется для создания объекта сеанса. Он неизменяем и поточно-безопасен по своей природе.
Configuration	Интерфейс используется для настройки и начальной загрузки спящего режима. Экземпляр этого интерфейса используется приложением для указания местоположения документов сопоставления, относящихся к спящему режиму.



Transaction	Определяет единицу работы (транзакцию). Он поддерживает абстракцию от реализации транзакции (JTA, JDBC). Транзакция связана с сеансом и создается путем вызова сеанса.
Query and Criteria	<p>Это объектно-ориентированное представление Hibernate Query. Объект Query можно получить, вызвав интерфейс сеанса метода createQuery(). Интерфейс запроса предоставляет множество методов.</p> <p>Интерфейс Criteria предоставляет методы для применения критериев, таких как получение всех записей таблицы, зарплата которых превышает 50000 и т. д.</p>

Последовательность взаимодействия интерфейсов:



4. PROXY: что это такое, для чего и как применяется в Hibernate.

Proxy используется для замены реальной сущности POJO (Plain Old Java Object).

Класс PROXY генерируется во время выполнения программы и **расширяет исходный класс сущности**, используя extends (наследование).

Поля те же, но они никогда не будут проинициализированы (всегда будут null), т.к. они просто достались по наследству

```

public class CompanyProxy extends Company
    implements HibernateProxy, ProxyConfiguration {
}

f company = {Company$HibernateProxy$PzJ226Gx@5092}
> f $$_hibernate_interceptor = {ByteBuddyInterceptor@5095}
f id = null
name = null
  
```

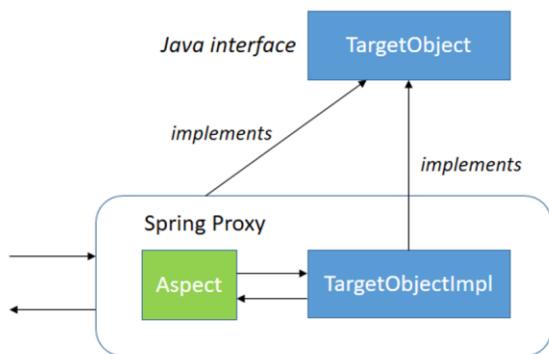
Паттерн Заместитель (Proxy) предоставляет объект-заместитель, который управляет доступом к другому объекту. То есть создается объект-суррогат, который может выступать в роли другого объекта и замещать его (перехватывать все вызовы).

В Spring прокси просто обращает bean, он может добавить логику до и после выполнения методов.

Spring AOP Process

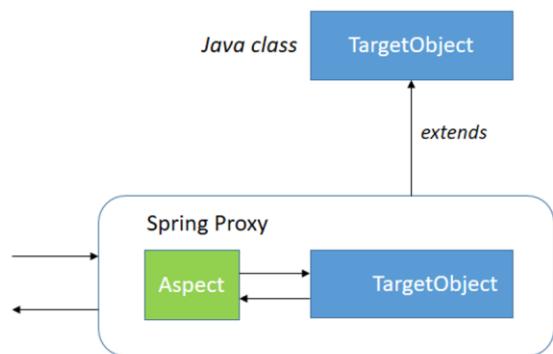
JDK Proxy (interface based)

JDK dynamic proxy - Spring AOP по умолчанию использует JDK dynamic proxy, которые позволяют проксировать любой интерфейс (или набор интерфейсов). Если целевой объект реализует хотя бы один интерфейс, то будет использоваться динамический прокси JDK.



CGLib Proxy (class based)

CGLIB-прокси - используется по умолчанию, если бизнес-объект не реализует ни одного интерфейса.



Hibernate использует объекты Proxy для объектов, чтобы разрешить отложенную LAZY загрузку: `@ManyToOne(optional = false, fetch = FetchType.LAZY)`

Зачем нужна ленивая загрузка? Вы загружаете большую часть своей базы данных, просто загружая один объект Company. Это приведет к проблемам с памятью. Поэтому Hibernate загружает только первый объект и заменяет наборы других объектов прокси. Если вы обращаетесь к прокси-серверу, Hibernate использует текущую сессию для инициализации прокси-сервера и загрузки записей из базы данных.

PROXY создаётся динамически во время компиляции. При доступе к основным свойствам он просто делегирует вызов исходной сущности.

```

public class ProxyTest {

    @Test
    void testDynamic() {
        Company company = new Company();
        Proxy.newProxyInstance(company.getClass().getClassLoader(), company.getClass().getInterfaces(),
            (proxy, method, args) -> method.invoke(company, args));
    }
}

```

Hibernate может использовать разные библиотеки для создания PROXY:

- ByteBuddy (используется в примере выше)
- Javaassist (заменил cglib, но все еще есть в исходниках)
- Cglib (использовался в первых версиях Hibernate)

Каждый List, Set, Map тип в классе сущностей замещен PersistentList, PersistentSet, PersistentMap. Эти классы отвечают за перехват вызова неинициализированной коллекции.

Прокси не выдает никаких операторов SQL. Он просто запускает `InitializeCollectionEvent`, который обрабатывается связанным прослушивателем, который знает, какой запрос инициализации выпустить (зависит от настроенного плана выборки).

Proxy объект получаем через метод `session.load()`, если вызываем геттеры и сеттеры, то выполняется SELECT в базу на получение **реального объекта**.

```

public static Object unproxy(Object proxy) {
    if ( proxy instanceof HibernateProxy ) {
        HibernateProxy hibernateProxy = (HibernateProxy) proxy;
        LazyInitializer initializer = hibernateProxy.getHibernateLazyInitializer();
        return initializer.getImplementation();
    }
    else {
        return proxy;
    }
}

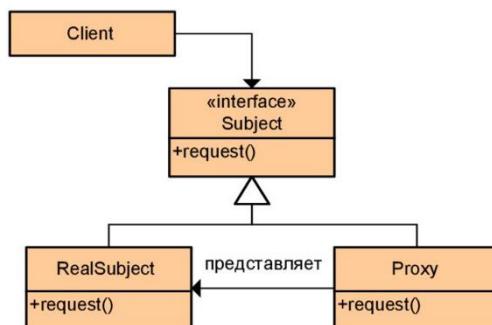
```

Справка: паттерн проектирования PROXY

Прокси Proxy

Тип: Структурный
Что это:

Представляет замену другого объекта для контроля доступа к нему.



Заместитель PROXY — структурный шаблон проектирования.

Представляет объект-суррогат, который контролирует доступ к другому объекту, перехватывая все вызовы.

Когда использовать прокси?

удалённый заместитель (remote proxies)	Когда надо осуществлять взаимодействие по сети, а объект-прокси должен имитировать поведение объекта в другом адресном пространстве. Использование прокси позволяет снизить накладные издержки при передаче данных через сеть.
виртуальный заместитель (virtual proxies)	Когда нужно управлять доступом к ресурсу, создание которого требует больших затрат. Реальный объект создается только тогда, когда он действительно может понадобится , а до этого все запросы к нему обрабатывает прокси-объект.
защищающий заместитель (protection proxies)	Когда необходимо разграничить доступ к вызываемому объекту в зависимости от прав зывающего объекта.
"умные ссылки" (smart reference)	Когда нужно вести подсчет ссылок на объект или обеспечить поток безопасную работу с реальным объектом.

```

public class CompanyProxy extends Company
    implements HibernateProxy, ProxyConfiguration {
}

```

Справка: PersistentBag – это аналог Hibernate Proxy для коллекций (при ленивой инициализации). При отношении @OneToMany по умолчанию fetch type LAZY поэтому в дебаге увидим PersistentBag

f users = {PersistentBag@6416} size = 1

5. Расскажи про Hibernate кэширование (уровни кэша и что под капотом)

Кэширование – один из способов оптимизации работы приложения, ключевой задачей которого является уменьшить количество прямых обращений к БД.

Кэш первого уровня: Session / EntityManager.

Кэшем первого уровня в Hibernate считается СЕССИЯ (либо EntityManager — аналог сессии в JPA). Перед тем, как отправить объект в БД, сессия обязательно хранит, кэширует объект за счёт своих ресурсов.

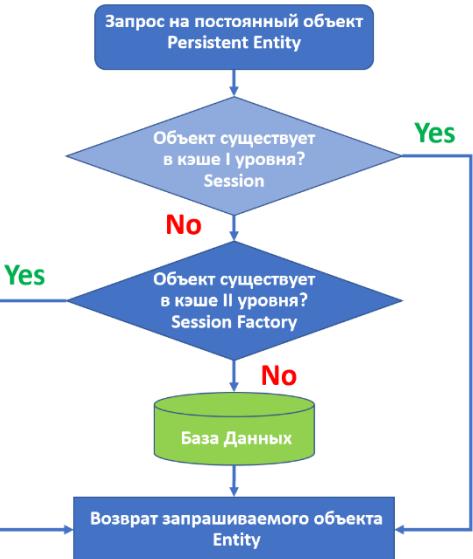
Session — это обёртка вокруг подключения к БД с помощью JDBC, используется один раз.

Включать в настройках кэширование не нужно, так как, это даже не кэш, а одно название — так по умолчанию работает Hibernate под капотом.

Hibernate хранит отслеживаемые сущности в Map, ключами которой являются id сущностей, а значениями — сами объекты-сущности.

Если извлекаем из базы сущность по id с помощью EntityManager.find(), то сущность помещается в Map и хранится там до закрытия сессии. И при повторном find() SQL-команда select в базе данных выполнена не будет. Hibernate возьмет эту сущность из Map — карты отслеживаемых сущностей.

```
@Override
public void putIntoCache(Object key, Object value, SharedSessionContractImplementor session) {
    underlyingCache.put( key, value );
}
```



Кэш второго уровня — общий кэш всех сессий **SessionFactory** (фабрика сессий).

Если сессия привязана к транзакции и закрывается каждый раз по ее окончании, то **SessionFactory создается один раз и далее ПЕРЕИСПОЛЬЗУЕТСЯ** в приложении.

Этот кэш и считается кэшем второго уровня. Это опционально, не обязательно, то есть по умолчанию он не работает, его надо включать. Кэш второго уровня — это прослойка, общая для всех сессий. То есть одна сессия извлекла сущность, а другая может получить к этой сущности потом доступ. Очевидно, что с такой прослойкой есть проблема — данные могут устареть: в базе данные одни, а в кэше второго уровня — другие.

Кэш запросов (Query Cache)

<https://youtu.be/C-wEZjEOhWc?t=1560>

В Hibernate предусмотрен кэш для запросов, и он интегрирован с кэшем второго уровня.

Это требует двух дополнительных физических мест для хранения кэшированных запросов и временных меток для обновления таблицы БД. Этот вид кэширования эффективен только для часто используемых запросов с одинаковыми параметрами.

Важно: хранит результат запроса, причём ключом является сам запрос и те параметры, которые были переданы в запросе. Хитро устроен: **сохраняет не объекты целиком, а их id -шники.**

6. Как работать с кэшем второго уровня?

<https://habr.com/ru/post/135176/>

Если кэш первого уровня привязан к объекту сессии, то **кэш второго уровня привязан к объекту – фабрике сессий** (SessionFactory object) => видимость этого кэша гораздо шире.

Тут будет выполнено 2 запроса в базу, т.к. по умолчанию кэш второго уровня отключен →

Для включения необходимо добавить следующие строки в конфиг файле JPA (persistence.xml):

```
<property name="hibernate.cache.provider_class" value="net.sf.ehcache.hibernate.SingletonEhCacheProvider"/>
//или в более старых версиях
//<property name="hibernate.cache.provider_class" value="org.hibernate.cache.EhCacheProvider"/>
<property name="hibernate.cache.use_second_level_cache" value="true"/>
```

Обратите внимание на первую строчку. Hibernate сам не реализует кеширование как таковое. А лишь предоставляет структуру для его реализации, поэтому подключить можно любую реализацию, которая соответствует спецификации ORM фреймворка. Из популярных реализаций можно выделить: EHCache, OSCache, SwarmCache, JBoss TreeCache.

Также понадобится отдельно настроить и саму реализацию кэша. В случае с провайдером кэширования **EHCache** это нужно сделать в файле **ehcache.xml**. Еще нужно указать самому хибернейту, что именно кэшировать. Это легко можно сделать с помощью аннотаций, например так →

Только после всех этих манипуляций кэш второго уровня будет включен и в примере выше будет выполнен только 1 запрос в базу.

Важно! Hibernate НЕ хранит объекты классов, а хранит информацию в виде массивов строк, чисел и т. д. И идентификатор объекта выступает указателем на эту информацию. Концептуально это нечто вроде Map, в которой id объекта — ключ, а массивы данных — значение. Приблизительно это выглядит так: `1 -> { "Pupkin", 1, null, {1,2,5} }`

Это разумно, учитывая, сколько памяти занимает каждый хранимый объект.

Зависимости класса по умолчанию также не кэшируются. Например, если рассмотреть класс выше — SharedDoc, то при выборке коллекция users будет доставаться из БД, а не из кэша второго уровня. Если нужно кэшировать и зависимости, то класс должен выглядеть так →

```
@Entity
@Table(name = "shared_doc")
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class SharedDoc{
    private Set<User> users;
}
```

```
@Entity
@Table(name = "shared_doc")
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class SharedDoc{
    @Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
    private Set<User> users;
}
```

Чтение из кэша второго уровня происходит только в том случае, если нужный объект не был найден в кэше первого уровня (см. рисунок выше).

Стратегии кэширования

Проблема заключается в том, что кэш второго уровня доступен из нескольких сессий сразу и несколько потоков программы могут одновременно в разных транзакциях работать с одним и тем же объектом.

Следовательно, надо как-то обеспечивать их одинаковым представлением этого объекта.

В Hibernate существует четыре стратегии одновременного доступа к объектам в кэше:

- Read-only (только чтение)
- Read-write (чтение-запись)
- Nonstrict-read-write (не строгое чтение-запись)
- Transactional (транзакционное)

Стратегия	Описание
read-only для Entity, которые никогда не меняются	Используется только для сущностей, которые никогда не изменяются (будет выброшено исключение, если попытаться обновить такую сущность). Очень просто и производительно. Подходит для некоторых статических данных, которые не меняются.
read-write строгая согласованность с использованием мягких блокировок	Эта стратегия гарантирует строгую согласованность, которую она достигает, используя «мягкие» блокировки: когда обновляется кэшированная сущность, на нее накладывается мягкая блокировка, которая снимается после коммита транзакции. Все параллельные транзакции, которые пытаются получить доступ к записям в кэше с наложенной мягкой блокировкой, не смогут их прочитать или записать и отправят запрос в БД. Ehcache использует эту стратегию по умолчанию.
nonstrict read-write (нестрогое) строгая согласованность НЕ гарантируется	Кэш обновляется после совершения транзакции, которая изменила данные в БД и закоммитила их. Таким образом, строгая согласованность не гарантируется, и существует небольшое временное окно между обновлением данных в БД и обновлением тех же данных в кэше, во время которого параллельная транзакция может получить из кэша устаревшие данные.
transactional полноценное разделение транзакций	Полноценное разделение транзакций. Каждая сессия и каждая транзакция видят объекты, как если бы только они с ним работали последовательно одна транзакция за другой. Цена за это — блокировки и потеря производительности.

7. Что такое Persistence Context?

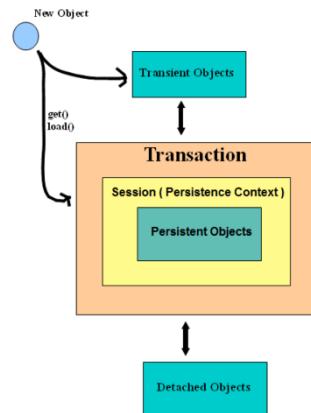
Java Persistence API (JPA) использует интерфейс `javax.persistence.EntityManager` для управления экземплярами Entity и их жизненным циклом (Hibernate делает это с помощью интерфейса `org.hibernate.Session`).

Каждый экземпляр EntityManager связан с Persistence Context, который является своего рода кэшем данных в рамках транзакции — **это и есть кэш первого уровня**.

Persistence контекст бывает двух типов в зависимости от этапа жизненного цикла:

- Persistence контекст области транзакции (Transaction scope persistence context)
- Расширенный persistence контекст (Extended persistence context)

Для EntityManager, управляемого контейнером, можно выбрать тип persistence контекста. Укажите тип контекста в атрибуте `@PersistenceContext`. Типом по умолчанию является Transaction scope. Обратите внимание, что расширенный контекст всегда используется для EntityManager, управляемого приложением (в этом случае нельзя выбирать тип контекста).



8. Что такое EntityManager?

Это **ИНТЕРФЕЙС JPA**, используемый для взаимодействия с персистентным контекстом. EntityManager описывает API для всех основных операций над Entity, а также для получения данных и других сущностей JPA. По сути — это главный API для работы с JPA.

Если проводить аналогию с обычным JDBC, то EntityManagerFactory будет аналогом DataSource, а EntityManager аналогом Connection. Создание EntityManagerFactory довольно дорогая операция, поэтому обычно её создают один раз и на всё приложение.

А чаще всего не создают сами, а **делегируют это фреймворку, такому как Spring**, например Интерфейс Session из Hibernate представлен в JPA как раз интерфейсом EntityManager.

JPA	JDBC по аналогии	Hibernate
EntityManagerFactory	DataSource	SessionFactory
EntityManager	Connection	Session
JPQL		HQL

Примеры:

```
Configuration configuration = new Configuration();
configuration.configure();

try (SessionFactory sessionFactory = configuration.buildSessionFactory();
     Session session = sessionFactory.openSession() {
    System.out.println("OK");
}
}
```

@Autowired
private SessionFactory sessionFactory;

@Override
public void add(User user) {
 sessionFactory.getCurrentSession().save(user);
}

Task 2.2.1.

*Методы интерфейса EntityManager

<https://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html>

<code>persist</code>	<code>void persist(Object entity)</code>	Делает экземпляр управляемым и постоянным.
<code>merge</code>	<code><T> T merge(T entity)</code>	Объединяет состояние данного объекта с текущим контекстом сохраняемости.
<code>remove</code>	<code>void remove(Object entity)</code>	Удаляет экземпляр сущности.
<code>find</code>	<code><T> T find(Class<T> entityClass, Object primaryKey)</code>	Находит по первичному ключу. Поиск объекта указанного класса и первичного ключа. Если экземпляра объекта содержится в контексте постоянства, он возвращается оттуда.
<code>flush</code>	<code>void flush()</code>	Синхронизирует контекст сохраняемости с базовой базой данных.
<code>lock</code>	<code>void lock(Object entity, LockModeType lockMode)</code>	Блокирует экземпляр сущности, содержащийся в контексте постоянства, с указанным типом режима блокировки.
<code>refresh</code>	<code>void refresh(Object entity)</code>	Обновляет состояние экземпляра из базы данных, перезаписав изменения, внесенные в сущность, если таковые имеются.
<code>clear</code>	<code>void clear()</code>	Очищает контекст сохраняемости, в результате чего все управляемые объекты станут отсоединенными. Изменения, внесенные в объекты, которые не были сброшены в базу данных, не будут сохранены.
<code>detach</code>	<code>void detach(Object entity)</code>	Удаляет объект из контекста постоянства, в результате чего управляемый объект станет отсоединенными. Не удаленные изменения, внесенные в объект, если таковые имеются (включая удаление объекта), не будут синхронизированы с базой данных. Объекты, которые ранее ссылались на отсоединенный объект, будут продолжать ссылаться на него.
<code>contains</code>	<code>boolean contains(Object entity)</code>	Проверяет, является ли экземпляр экземпляром управляемого объекта, принадлежащим текущему контексту постоянства.
<code>getLockMode</code>	<code>LockModeType getLockMode(Object entity)</code>	Получает текущий режим блокировки для экземпляра объекта.
<code>createQuery</code>	<code>Query createQuery(String qlString)</code>	Создаёт экземпляр Query для выполнения оператора языка запросов Java Persistence (JPQL)

9. Какие функции он выполняет?

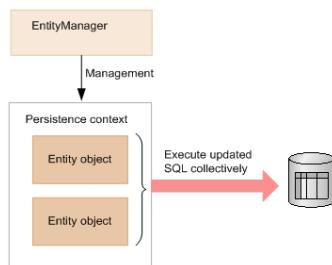
EntityManager вставляет (*insert*) объект управляемой сущности в persistence context (контекст постоянства) и управляет объектами сущности. Когда приложение передает **entity объект в persist method** или обновляет значение поля управляемого entity объекта, **статус entity объекта в persistence контексте изменяется**.

JPA позволяет синхронизировать контекст персистенции и БД при помощи метода `flush`:

```
entityManager.flush();
```

А для синхронизации требуется транспорт и этим транспортом является **транзакция**.

EntityManager синхронизирует статусы объектов сущностей в persistence контексте и таблице базы данных непосредственно перед фиксацией (*commit*-ом) транзакции.



Чтобы применить статусы объектов сущностей в persistence контексте к таблице БД, SQL-операторы обновления выпускаются коллективно и одновременно.

В результате **время блокировки базы данных сокращается**, и, следовательно, можно улучшить параллельное выполнение и эффективно обновлять данные.

Изменения, накопленные в EntityManager, при помощи транзакции были закоммичены (*commit*), т.е. подтверждены и сохранены в БД.

```

@Before
public void init() {
    EntityManagerFactory emf = Persistence.createEntityManagerFactory( "JavaRush" );
    em = emf.createEntityManager();
    em.getTransaction().begin();
}
@After
public void close() {
    if (em.getTransaction().isActive()) {
        em.getTransaction().commit();
    }
    em.getEntityManagerFactory().close();
    em.close();
}

```

Основные функции EntityManager:

Операции над Entity: persist (добавление Entity под управление JPA), merge (изменение), remove (удаление), refresh (обновление данных), detach (удаление из-под управления контекста персистентности), lock (блокирование Entity от изменений в других thread).

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.

Объекты EntityManager НЕ являются поток безопасными. Это означает, что каждый поток должен получить свой экземпляр EntityManager, поработать с ним и закрыть его в конце.

10. Каким условиям должен удовлетворять класс чтобы являться Entity?

Entity – это **легковесный хранимый объект бизнес-логики** (persistent domain object). Основная программная сущность – это entity класс, который так же может использовать дополнительные классы, как вспомогательные или для сохранения состояния entity.

Entity Class – это Java класс, который отображает информацию таблицы в базе данных. Это также **POJO класс**, в котором используются Hibernate аннотации для связи класса с таблицей из базы данных. <https://javarush.com/groups/posts/2259-jpa-znakovstvo-s-tehnologiyey>

Персистентное состояние сущности представлено персистентными **полями** или персистентными **свойствами** (методами). В JPA персистентные поля и свойства принято называть атрибутами класса-сущности.

Персистентное поле – это поле сущности, которое отражается в БД в виде столбца таблицы.

Персистентное свойство – это методы, поведение сущности, которые аннотированы вместо полей для доступа провайдера к ним (полям).

Эти поля или свойства используют аннотации **объектно-реляционного сопоставления (маппинга)** для сопоставления сущностей и отношений между ними с реляционными данными в хранилище данных.

Примеры аннотаций: `@OneToOne`, `@OneToMany`, `@ManyToOne`, `@ManyToMany`.

Есть два вида доступа к состоянию сущности:

Доступ по полю, когда аннотации стоят над полями.

В этом случае провайдер, например, Hibernate, обращается к полям класса напрямую, используя Reflection.

```
project lombok features page for @Data ↗ .
See Also: Getter, Setter,
RequiredArgsConstructor,
ToString, EqualsAndHashCode,
Value
```

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
@Entity
@Table(name = "users", schema = "public")
public class User {

    @Id
    private String username;
    private String firstname;
    private String lastname;
    private LocalDate birthDate;
    private Integer age;
}
```

Доступ по свойству (методу), когда аннотации стоят над методами-геттерами. В этом случае провайдер, например, Hibernate, обращается к полям класса через методы.

Требования к Entity классу в JPA

✓	<code>@Entity</code> XML конфиг	Entity класс должен быть помечен аннотацией <code>@Entity</code> или описан в XML файле конфигурации JPA.
✓	<code>NoArgsConstructor</code> <code>AllArgsConstructor</code>	Entity класс должен содержать <code>public</code> или <code>protected</code> конструктор без аргументов (он также может иметь конструкторы с аргументами).
✓	<code>top-level class</code>	Entity класс должен быть классом верхнего уровня (top-level class).
✓	<code>implements Serializable</code>	Если объект Entity класса будет передаваться по значению как отделённый от контекста персистентности объект (detached object), например через удаленный интерфейс (through a remote interface), то он также должен реализовывать интерфейс <code>Serializable</code> (чтобы объекты, которые достаются из базы могли сохраняться в кэше).
✓	<code>extends</code>	Как обычный, так и абстрактный класс может быть Entity. Entities могут наследоваться как от не Entity классов, так и от Entity классов. А не Entity классы могут наследоваться от Entity классов.
✓	<code>getter/setter</code>	Поля Entity класса должны быть объявлены <code>private</code> , <code>protected</code> или <code>package-private</code> , быть напрямую доступными только методам самого Entity класса и не должны быть напрямую доступны другим классам, использующим этот Entity. Другие классы должны обращаться только к специальным методам Entity класса, предоставляемым доступ к этим полям (<code>getter/setter</code> -методам или другим методам бизнес-логики в Entity классе).

✓ Primary key	Entity класс должен содержать первичный ключ, то есть атрибут или группу атрибутов, которые уникально определяют запись этого Entity класса в базе данных.
✗ Enum	Перечисление [enum] или интерфейс НЕ могут быть определены как сущность [Entity].
✗ Final class	Entity класс НЕ может быть финальным классом (final class).
✗ final methods final variables	Entity класс НЕ может содержать финальные поля или методы, если они участвуют в маппинге (persistent final methods or persistent final instance variables).

Требования к Entity классу в Hibernate МЯГЧЕ, т.е. Hibernate не так строг.

Отличия от JPA:

- Класс сущности должен иметь конструктор без аргументов, который может быть не только public или protected, но и package visibility (default), т.е. любой модификатор
- Класс сущности не обязательно должен быть классом верхнего уровня, т.е. может быть вложенным классом
- Технически Hibernate может сохранять финальные классы или классы с финальными методами (getter / setter). Однако, как правило, это не очень хорошая идея, так как это лишит Hibernate возможности генерировать прокси для отложенной загрузки сущности. Т.к. final class не может иметь наследников, то => Hibernate не сможет создавать PROXY объекты для LAZY загрузки.
- Hibernate не запрещает разработчику приложения открывать прямой доступ к переменным экземпляра и ссылаться на них извне класса сущности, однако обоснованность такого подхода спорна.

Аннотации

```
// @Entity javax.persistence.Entity javax.persistence.Entity (Класс является сущностью (entity bean) и должен иметь пустой конструктор (по умолчанию). Entity это легковесный хранимый объект бизнес логики (persistent domain object). Основная программная сущность, использует дополнительные классы, которые могут использоваться как вспомогательные классы или для сохранения состояния entity.)
@Table javax.persistence.Table (сообщаем, с какой именно таблицей необходимо связать (map) данный класс.)
@Column javax.persistence.Table (определяет к какому столбцу в таблице БД относится конкретное поле класса (атрибут класса).)
@Id javax.persistence.Id (указываем primary key (Primary Key) данного класса.)
@GeneratedValue javax.persistence.GeneratedValue (используется вместе с аннотацией @Id и определяет такие параметры, как strategy и generator)
@Version javax.persistence.Version (Version решает проблему с потерянными обновлениями.)
@OrderBy javax.persistence.OrderBy (При загрузке данных указываем порядок их загрузки. Например, @OrderBy(name = "group_name ASC, name DESC")
```

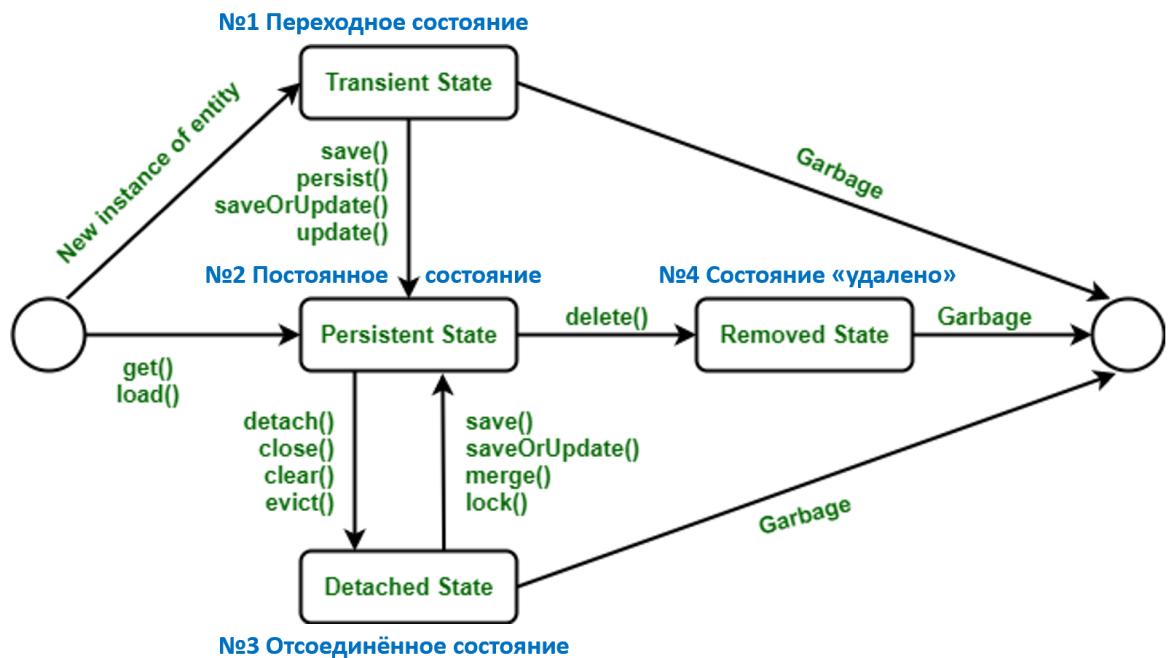
```
Configuration configuration = new Configuration();
configuration.addAnnotatedClass(User.class);
configuration.configure();

try (SessionFactory sessionFactory = configuration.buildSessionFactory();
Session session = sessionFactory.openSession()) {
    session.beginTransaction();

    User user = User.builder()
        .username("ivan@gmail.com")
        .firstname("Ivan")
        .lastname("Ivanov")
        .birthDate(LocalDate.of(year: 2000, month: 1, dayOfMonth: 19))
        .age(20)
        .build();
    session.save(user);

    session.getTransaction().commit();
}
}
```

11. Расскажи про жизненный цикл сущности Entity, перечисли четыре статуса ЖЦ Entity объекта (Entity Instance's Life Cycle).



Этап №1 New, переходное состояние (Transient State)

Объект создан, не имеет primary key, не является частью контекста персистентности (не управляет JPA);

Переходное состояние — это первое состояние объекта сущности. Когда мы создаем объект POJO класса с помощью оператора new, объект находится в переходном состоянии. Этот объект не связан ни с одной сессией, следовательно это состояние не связано ни с одной таблицей БД. Итак, если мы внесем какие-либо изменения в данные класса POJO, таблица базы данных не изменится. Временные объекты не зависят от Hibernate и существуют в куче памяти.

```
//Here, The object arrives in the transient state.
Employee e = new Employee();
e.setId(21);
e.setFirstName("Neha");
e.setMiddleName("Shri");
e.setLastName("Rudra");
```

Этап №2 Managed, постоянное состояние (Persistent State)

Объект создан, имеет primary key, является частью контекста персистентности (управляет JPA);

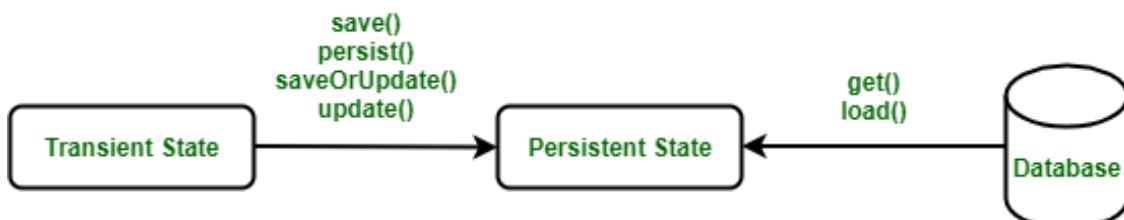
Как только объект подключается к сессии, он переходит в постоянное состояние.

Есть два способа преобразовать переходное состояние в постоянное состояние:

- Используя сессию, сохранить объект сущности в таблице базы данных.
- Используя сессию, загрузить объект сущности в таблицу базы данных.

```
//Persistent State
session.save(e);
```

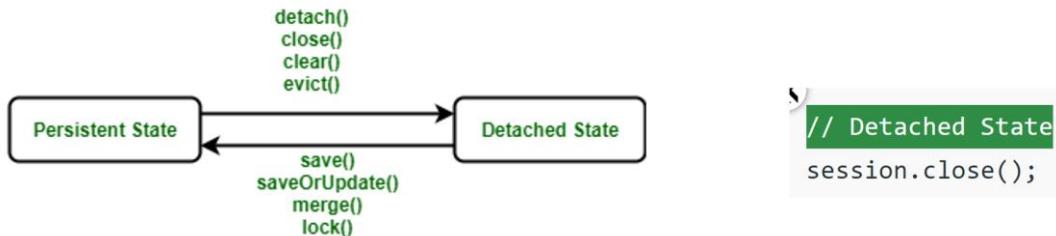
В этом состоянии, каждый объект представляет собой одну строку в таблице БД. Следовательно, если внести какие-либо изменения в данные, то hibernate обнаружит это и внесет изменения в таблицу базы данных.



Этап №3 Detached, отсоединенное состояние (Detached State)

Объект создан, имеет primary key, не является (или больше не является) частью контекста персistentности (не управляет JPA);

Для преобразования объекта из постоянного в отсоединенное состояние нужно либо закрыть сессию, либо очистить кеш. Поскольку сессия будет закрыта или кеш очищен, любые изменения, внесенные в данные, не повлияют на таблицу базы данных. При необходимости отсоединеный объект можно повторно подключить к новой сессии.



Этап №4 Removed, состояние «удалено» (Removed State)

Объект создан, является частью контекста персистентности (управляется JPA), будет удален при commit-е транзакции (закрытии сессии или вызове метода flush()).

В жизненном цикле спящего режима это последнее состояние. Когда объект сущности удаляется из базы данных, объект сущности находится в удаленном состоянии. Это делается путем вызова операции `delete()`. Поскольку Entity объект находится в удаленном состоянии, любое изменение данных не повлияет на таблицу базы данных.



12. Как влияют операции persist, remove, merge, refresh, detach на Entity объекты каждого из четырех статусов?

	New	Managed	Detached	Removed
<code>persist</code>	<code>new → managed</code> , объект будет сохранен в базу при commit-е транзакции или в результате flush-операции.	<code>managed →</code> операция игнорируется, однако связанные entity могут поменять статус на <code>managed</code> , если у них есть аннотации каскадных изменений.	<code>detached → exception</code> сразу или на этапе commit-а транзакции (так как у <code>detached</code> уже есть первичный ключ).	<code>removed → managed</code> .
<code>remove</code>	<code>new →</code> операция игнорируется, однако связанные entity могут поменять статус на <code>removed</code> , если у них есть аннотации каскадных изменений и они имели статус <code>managed</code> .	<code>managed → removed</code> , и запись в базе данных будет удалена при commit-е транзакции (также произойдут операции <code>remove</code> для всех каскадно зависимых объектов).	<code>detached → exception</code> сразу или на этапе commit-а транзакции.	<code>removed →</code> операция игнорируется
<code>merge</code>	<code>new →</code> будет создана новая <code>managed entity</code> , в которую будут скопированы данные объекта.	<code>managed →</code> операция игнорируется, однако операция <code>merge</code> сработает на каскадно зависимых entity, если их статус не <code>managed</code> .	<code>detached →</code> либо данные будут скопированы в существующую БД <code>managed entity</code> с тем же первичным ключом, либо создана новая <code>managed entity</code> , в которую скопируются данные.	<code>removed → Exception</code> сразу или на этапе commit-а транзакции
<code>refresh</code>	Exception	<code>managed →</code> будут восстановлены все изменения из базы данных данного entity, также произойдет <code>refresh</code>	Exception	Exception

		всех каскадно зависимых объектов.		
detach	new → операция игнорируется.	managed → detached.	detached → операция игнорируется.	removed → detached.

13. Может ли абстрактный класс быть Entity?

Абстрактный класс может быть Entity классом. Абстрактный Entity класс отличается от обычных Entity классов только тем, что **нельзя создать объект этого класса**.

Имена абстрактных классов могут использоваться в запросах.

Абстрактные Entity классы используются в наследовании, когда их потомки наследуют поля абстрактного класса:

```

@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Employee {
    @Id
    @GeneratedValue
    private long id;
    private String name;
    .....
}

@Entity
@Table(name = "FULL_TIME_EMP")
public class FullTimeEmployee extends Employee {
    private int salary;
    .....
}

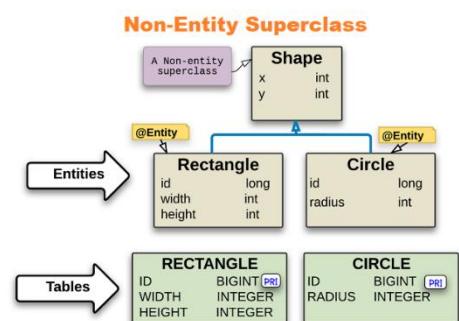
@Entity
@Table(name = "PART_TIME_EMP")
public class PartTimeEmployee extends Employee {
    private int hourlyRate;
    .....
}

```

14. Может ли Entity класс наследоваться от не Entity классов (non-entity classes)?

Да, сущности могут наследоваться от НЕ Entity классов, которые, в свою очередь, могут быть как абстрактными, так и обычными.

Состояние (поля) НЕ Entity суперкласса НЕ является персистентным, то есть не хранится в БД и не обрабатывается провайдером (Hibernate), поэтому любое такое состояние (поля), унаследованное Entity классом, также НЕ будет отображаться в БД.



Не Entity суперклассы не могут участвовать в операциях EntityManager или Query. Любые маппинги или аннотации отношений в не Entity суперклассах игнорируются.

15. Может ли Entity класс наследоваться от других Entity классов? Да, может.

16. Может ли не Entity класс наследоваться от Entity класса? Да, может.

Родитель/Наследник	Entity класс	не Entity класс
Entity класс	+	+
не Entity класс	+	+ (обычное наследование в Java)

17. Что такое встраиваемый (Embeddable) класс?

Как можно сопоставить одну сущность, содержащую встроенные свойства, с одной таблицей базы данных?

Можно также применять это **для денормализации БД** (ускорений запросов к БД). Для этой цели используют аннотации `@Embeddable` и `@Embedded`, предоставляемые Java Persistence API (JPA). Тестировать производительность в продакшне (!).

Пример. Определим контекст модели данных.

Создадим таблицу `company`. В ней будет храниться основная информация: название компании, адрес и телефон, а также информация о контактном лице (рис. слева). Контактное лицо должно быть абстрагировано в отдельный класс. Однако мы не хотим создавать отдельную таблицу, что можно сделать?

JPA предоставляет аннотацию `@Embeddable`, чтобы объявить, что класс будет внедрен другими объектами. Определим класс для абстрагирования сведений о контактном лице. Аннотация JPA `@Embedded` используется для встраивания типа в другой объект.

Изменим класс Company. Добавим аннотации JPA, а также изменим использование ContactPerson вместо отдельных полей:

```
public class Company {
    private Integer id;
    private String name;
    private String address;
    private String phone;
    private String contactFirstName;
    private String contactLastName;
    private String contactPhone;
    // standard getters, setters
}

@Embeddable
public class ContactPerson {
    private String firstName;
    private String lastName;
    private String phone;
    // standard getters, setters
}

@Entity
public class Company {
    @Id
    @GeneratedValue
    private Integer id;
    private String name;
    private String address;
    private String phone;
    @Embedded
    private ContactPerson contactPerson;
    // standard getters, setters
}
```

В результате есть сущность Company, содержащая данные контактного лица и сопоставление с одной таблицей базы данных. Но все еще есть еще одна проблема, и это то, как JPA будет отображать эти поля в столбцы базы данных.

Переопределение атрибутов.

Поля назывались так: `contactFirstName` в исходном классе Company, а теперь `firstName` в классе ContactPerson. Таким образом, JPA захочет сопоставить их с `contact_first_name` и `first_name` соответственно. Кроме того, колонка `phone` теперь дублируется.

Необходимо использовать `@AttributeOverrides` и `@AttributeOverride` для переопределения свойств столбца встроенного типа. Добавим это в поле «Контактное лицо» в сущности «Компания» (рис. ниже).

Поскольку эти аннотации помещаются в поле, то могут быть разные переопределения для каждого включающего объекта.

```
@Embedded
@GeneratedValue({
    @AttributeOverride( name = "firstName", column = @Column(name = "contact_first_name") ),
    @AttributeOverride( name = "lastName", column = @Column(name = "contact_last_name") ),
    @AttributeOverride( name = "phone", column = @Column(name = "contact_phone") )
})
private ContactPerson contactPerson;
```

Итак, **Embeddable класс** – это класс, который не используется сам по себе, а только как часть одного или нескольких Entity классов. Entity класс может содержать как одиночные встраиваемые классы, так и коллекции таких классов. Также такие классы могут быть использованы как ключи или значения Map.

Во время выполнения каждый встраиваемый класс принадлежит только одному объекту Entity класса и не может быть использован для передачи данных между объектами Entity классов (то есть такой класс не является общей структурой данных для разных объектов).

В целом, такой класс служит для того, чтобы выносить определение общих атрибутов для нескольких Entity, можно считать, что JPA просто встраивает в Entity вместо объекта такого класса те атрибуты, которые он содержит.

Hibernate называет эти классы компонентами, а JPA встраиваемыми. В любом случае, **концепция одна и та же: композиция значений**.

Встраиваемый класс помечается аннотацией `@Embeddable`. Встраиваемый класс может быть встроен в несколько классов-сущностей, но встроенный объект с конкретным состоянием принадлежит исключительно владеющей им сущности и не может использоваться одновременно другими сущностями, он НЕ является общим для нескольких сущностей.

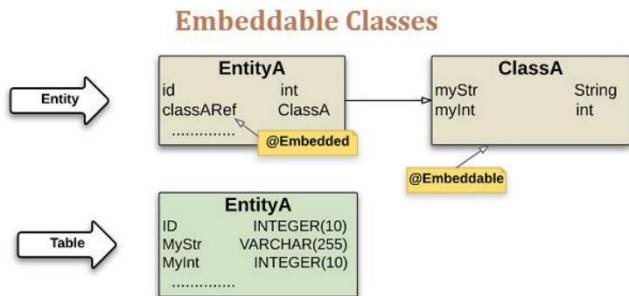
Особенности встраиваемых классов:

- все поля встраиваемого класса, даже коллекции, станут полями класса, в который происходит встраивание
- встраиваемые классы могут быть встроены в одну и ту же сущность несколько раз, нужно только поменять имена полей
- экземпляры встраиваемых классов, в отличие от экземпляров сущностей, не имеют собственного персистентного состояния, вместо этого они существуют только как часть состояния объекта, которому они принадлежат
- встраиваемые классы могут использовать в качестве полей:
 - базовые типы;
 - коллекции базовых типов (с аннотацией `@ElementCollection`);
 - другие встраиваемые классы;
 - коллекции других встраиваемых классов (с аннотацией `@ElementCollection`);
 - сущности;
 - коллекции сущностей;
- сущность может использовать в качестве полей одиночные встраиваемые классы и коллекции встраиваемых классов;
- встраиваемые классы могут использоваться в качестве ключей и значений Map.

18. Какие требования JPA устанавливает к встраиваемым (Embeddable) классам?

Должны соответствовать требованиям для сущностей (за исключением того, что у встраиваемых классов не ставится аннотация `@Entity` и может отсутствовать первичный ключ (`@Id`)).

Должны быть аннотированы `@Embeddable`.



19. Для чего нужны аннотации `@Embedded` и `@Embeddable`?

@Embeddable Аннотация JPA, размещается над классом для указания того, что класс является встраиваемым в другие классы и будет внедрен другими сущностями, то есть поля этого встраиваемого класса будут добавляться к полям других сущностей и будут представлять столбцы в таблице этой сущности.

Так, во встраиваемый класс мы можем выделить общие поля для разных сущностей, не создавая для него таблицу. **Встраиваемый класс сам не является сущностью.**

@Embedded аннотация JPA, используется для размещения над полем в классе-сущности для указания того, что внедряется встраиваемый класс.

```

@Embeddable
public class ContactPerson {
    private String firstName;
    private String lastName;
    private String phone;
    // standard getters, setters
}

@Entity
public class Company {
    @Id
    @GeneratedValue
    private Integer id;
    private String name;
    private String address;
    private String phone;
    @Embedded
    private ContactPerson contactPerson;
    // standard getters, setters
}

```

Таблица Company будет выглядеть так:

id	name	address	phone	firstName	lastName	phone
----	------	---------	-------	-----------	----------	-------

```

@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
@Embeddable
public class PersonalInfo {

    private String firstname;
    private String lastname;
    @Column(name = "birth_date")
    private Birthday birthDate;
}

```

```

@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
@Entity
@Table(name = "users", schema = "public")
@TypeDef(name = "gmdev", typeClass = JsonBinaryType.class)
public class User {

    @Id
    private String username;

    @Embedded
    private PersonalInfo personalInfo;

    @Type(type = "gmdev")
    private String info;

    @Enumerated(EnumType.STRING)
    private Role role;
}

```

```

public static void main(String[] args) throws SQLException {
    User user = User.builder()
        .username("petr@gmail.com")
        .personalInfo(PersonalInfo.builder()
            .lastname("Petrov")
            .firstname("Petr")
            .build())
        .build();
    log.info("User entity is in transient state, object: {}", user);
}

```

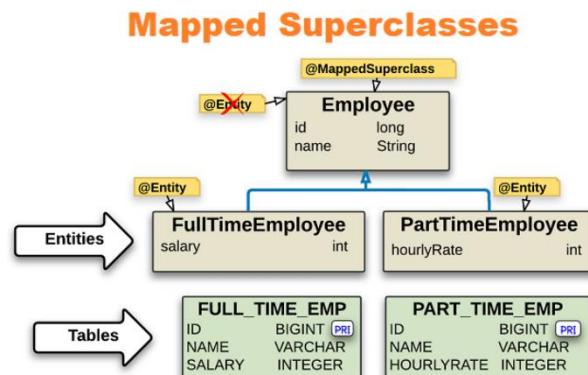
20. Что такое Mapped Superclass?

<https://www.logicbig.com/tutorials/java-ee-tutorial/jpa.html>

Это стратегия наследования полей, когда базовый класс НЕ является отдельной Entity сущностью Hibernate и НЕ имеет своей таблицы в базе данных.

Mapped Superclass (сопоставленный суперкласс) – это класс, от которого наследуются Entity, он может содержать аннотации JPA, однако сам такой класс **не является Entity**, ему не обязательно выполнять все требования, установленные для Entity (например, он может не содержать первичного ключа).

Эти суперклассы чаще всего используются, когда есть общая для нескольких классов сущностей информация о состоянии и отображении, которую можно вынести в Mapped Superclass.



```

@MappedSuperclass
public class Employee {
    @Id
    @GeneratedValue
    private long id;
    private String name;
}

@Entity
@Table(name = "FULL_TIME_EMP")
public class FullTimeEmployee extends Employee {
    private int salary;
}

@Entity
@Table(name = "PART_TIME_EMP")
public class PartTimeEmployee extends Employee {
    private int hourlyRate;
}

```

В этом примере в БД будут таблицы FULLTIMEEMPLOYEE и PARTTIMEEMPLOYEE, но таблицы EMPLOYEE не будет.

Особенности Mapped Superclass:

- Должен быть помечен аннотацией `@MappedSuperclass` или описан в xml файле.
- Не может использоваться в операциях EntityManager или Query, вместо этого нужно использовать классы-наследники.
- Не может состоять в отношениях с другими сущностями, т.е. в Entity нельзя создать поле с типом сопоставленного суперкласса.
- Может быть абстрактным.
- Не имеет своей таблицы в БД.

Основным недостатком использования сопоставленного суперкласса является то, что полиморфные запросы невозможны, то есть мы не можем загрузить всех наследников Mapped Superclass.

Mapped Superclass v.s. Embeddable class

Сходства:

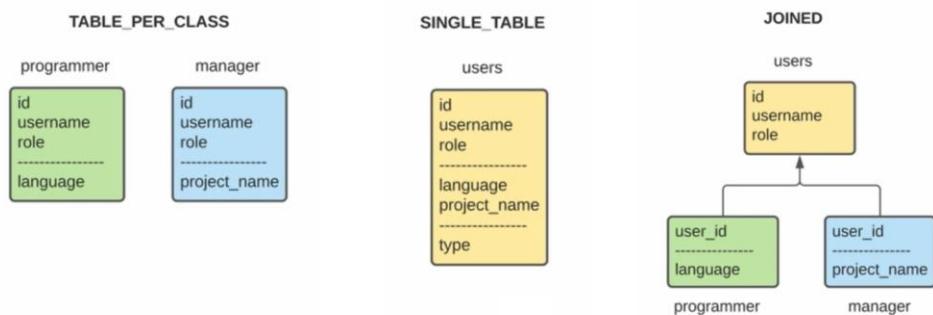
- не являются сущностями и могут иметь все аннотации, кроме @Entity;
- не имеют своих таблиц в БД;
- не могут использоваться в операциях EntityManager или Query.

Различия:

- **MappedSuperclass – это наследование, а Embeddable class – это композиция** (экземпляр «части» может входить только в одно целое (или никуда не входить));
- Поля из Mapped Superclass могут быть у сущности в одном экземпляре, а полей из Embeddable class может быть сколько угодно (встроив в сущность Embeddable class несколько раз и поменяв имена полей);
- В Entity нельзя создать поле с типом сопоставленного суперкласса, а с Embeddable можно и нужно.

21. Какие стратегии маппинга иерархии наследования (Inheritance Mapping Strategies) описаны в JPA?

Стратегии наследования нужны для того, чтобы дать понять провайдеру (Hibernate) **КАК отображать в БД сущности-наследники**. Для этого нужно декорировать родительский класс аннотацией @Inheritance и указать один из типов отображения: SINGLE_TABLE, TABLE_PER_CLASS, JOINED.

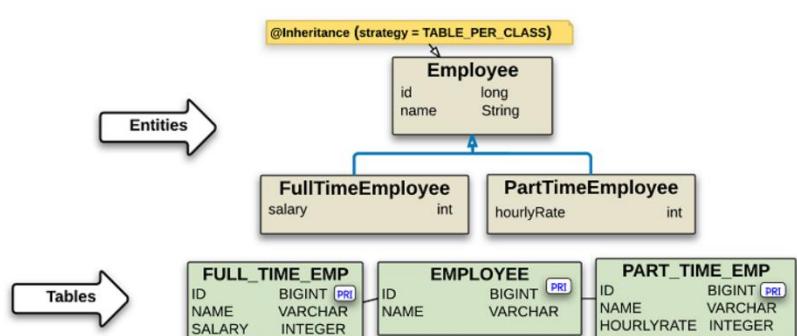


TABLE_PER_CLASS (отдельная таблица для каждого конкретного класса сущностей)

Table Per Class Inheritance Strategy

Каждый класс-наследник имеет свою таблицу.

Во всех таблицах подклассов хранятся все поля этого класса плюс те, которые унаследованы от суперкласса.



```

@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
@Entity
public class Employee {
    @Id
    @GeneratedValue
    private long id;
    private String name;
    .....
}

@Entity
@Table(name = "FULL_TIME_EMP")
public class FullTimeEmployee extends Employee {
    private int salary;
    .....
}

@Entity
@Table(name = "PART_TIME_EMP")
public class PartTimeEmployee extends Employee {
    private int hourlyRate;
    .....
}

```

МИНУСЫ:

Плохая поддержка полиморфизма (polymorphic relationships), т.к. для выборки всех классов иерархии потребуется большое количество отдельных sql-запросов для каждой таблицы-наследника или **использование UNION запроса для соединения таблиц** всех наследников в одну таблицу.

Повторение одних и тех же атрибутов в таблицах. При TABLE PER CLASS **не работает стратегия генератора первичных ключей IDENTITY**, поскольку может быть несколько объектов подкласса, имеющих один и тот же идентификатор, и запрос базового класса приведет к получению объектов с одним и тем же идентификатором (даже если они принадлежат разным типам).

ПЛЮСЫ:

Если нужно получить отдельную сущность, то можно обратиться к соответствующей таблице (не нужно искать во всех таблицах).

Сохранение и загрузка данных:

```

-- Persisting entities --
FullTimeEmployee{id=0, name='Sara', salary=100000}
PartTimeEmployee{id=0, name='Robert', hourlyRate='60'}

-- Loading entities --
PartTimeEmployee{id=2, name='Robert', hourlyRate='60'}
FullTimeEmployee{id=1, name='Sara', salary=100000}

-- Native queries --
>Select * from Employee' // no data
>Select * from FULL_TIME_EMP'
[1, Sara, 100000]
>Select * from PART_TIME_EMP'
[2, Robert, 60]

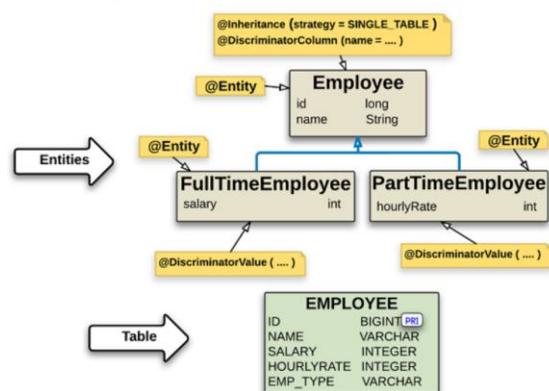
```

SINGLE_TABLE (одна таблица на всю иерархию классов)

Стратегия используется по умолчанию, когда аннотация @Inheritance не указана в родительском классе или указана, но без конкретной стратегии.

Все Entity, со всеми наследниками записываются в **ОДНУ** таблицу.

Для идентификации типа entity (наследника) определяется специальная колонка “discriminator column”.

Single Table Inheritance Strategy

Например, если есть entity Employee с классами-потомками FullTimeEmployee и PartTimeEmployee, то при такой стратегии все FullTimeEmployee и PartTimeEmployee записываются в таблицу Employee, и при этом в таблице появляется дополнительная колонка с именем DTYPEn, в которой будут записаны значения, определяющие принадлежность к классу.

По умолчанию эти значения формируются из имён классов, в этом примере либо «FullTimeEmployee», либо «PartTimeEmployee». Но можно их поменять в аннотации у каждого класса-наследника: `@DiscriminatorValue("F")`. Если хотим поменять имя колонки, то необходимо указать её новое имя в параметре аннотации у класса-родителя: `@DiscriminatorColumn(name=EMP_TYPE)`.

```
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@Entity
@DiscriminatorColumn(name = "EMP_TYPE")
public class Employee {
    @Id
    @GeneratedValue
    private long id;
    private String name;
    .....
}

@Entity
@DiscriminatorValue("F")
public class FullTimeEmployee extends Employee {
    private int salary;
    .....
}

@Entity
@DiscriminatorValue("P")
public class PartTimeEmployee extends Employee {
    private int hourlyRate;
    .....
}
```

Эта стратегия обеспечивает **хорошую поддержку полиморфных отношений** между сущностями и запросами, которые охватывают всю иерархию классов сущностей:

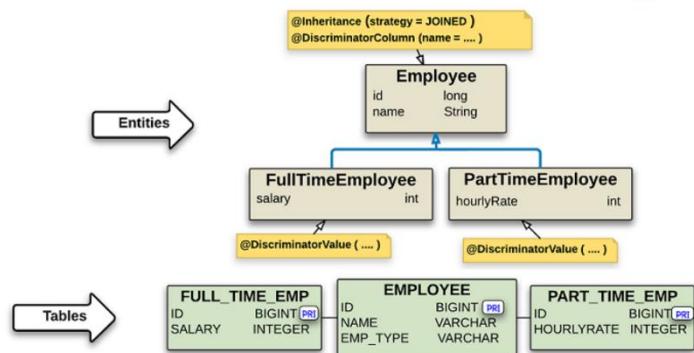
<pre>-- Persisting entities -- FullTimeEmployee{id=0, name='Sara', salary=100000} PartTimeEmployee{id=0, name='Tom', hourlyRate='60'} . . . -- Loading entities -- FullTimeEmployee{id=1, name='Sara', salary=100000} PartTimeEmployee{id=2, name='Tom', hourlyRate='60'}</pre>	<pre>-- Native queries -- 'Select * from Employee' [F, 1, Sara, null, 100000] [P, 2, Tom, 60, null]</pre>
---	---

JOINED (стратегия соединения)

В этой стратегии корневой класс иерархии представлен отдельной таблицей, а каждый наследник имеет свою таблицу, в которой отображены только поля этого класса-наследника.

То есть таблица подкласса не содержит столбцы для полей, унаследованных от родительского класса, за исключением поля для первичного ключа `@Id`, который должен быть определен только в родительской таблице.

Joined Subclass Inheritance Strategy



Столбец первичного ключа в таблице подкласса служит внешним ключом первичного ключа таблицы суперкласса. Также **в таблице родительского класса** добавляется столбец DiscriminatorColumn с DiscriminatorValue для определения типа наследника.

```

@Inheritance(strategy = InheritanceType.JOINED)
@Entity
@DiscriminatorColumn(name = "EMP_TYPE")
public class Employee {
    @Id
    @GeneratedValue
    private long id;
    private String name;
    .....
}

@Entity
@DiscriminatorValue("F")
@Table(name = "FULL_TIME_EMP")
public class FullTimeEmployee extends Employee {
    private int salary;
    .....
}

@Entity
@DiscriminatorValue("P")
@Table(name = "PART_TIME_EMP")
public class PartTimeEmployee extends Employee {
    private int hourlyRate;
    .....
}

```

Эта стратегия обеспечивает **хорошую поддержку полиморфных отношений**, но требует выполнения одной или нескольких операций соединения таблиц при создании экземпляров подклассов сущностей. В глубоких иерархиях классов это **может привести к недопустимому снижению производительности**. Точно так же запросы, которые покрывают всю иерархию классов, требуют операций соединения между таблицами подклассов, что приводит к снижению производительности:

```

-- Persisting entities --
FullTimeEmployee{id=0, name='Sara', salary=100000}
PartTimeEmployee{id=0, name='Robert', hourlyRate='60'}

-- Loading entities --
FullTimeEmployee{id=1, name='Sara', salary=100000}
PartTimeEmployee{id=2, name='Robert', hourlyRate='60'}
```

```
-- Native queries --
'Select * from Employee'
[F, 1, Sara]
[P, 2, Robert]
'Select * from FULL_TIME_EMP'
[100000, 1]
'Select * from PART_TIME_EMP'
[60, 2]
```

22. Как мапятся Enum -ы?

По порядковым номерам

Если мы сохраняем в БД сущность, у которой есть поле-перечисление (Enum), то в таблице этой сущности создаётся колонка для значений этого перечисления и по умолчанию в ячейки сохраняется порядковый номер этого перечисления (ordinal).

В JPA типы Enum могут быть помечены аннотацией @Enumerated, которая может принимать в качестве атрибута **EnumType.ORDINAL** или **EnumType.STRING**, определяющий, отображается ли перечисление (enum) на столбец с типом Integer или String соответственно.

```

public enum MyEnum {
    ConstA, ConstB, ConstC
}

@Entity
public class MyEntity {
    @Id
    private long myId;
    private MyEnum myEnum;
    public MyEntity() {
    }
    public MyEntity(long myId, MyEnum myEnum) {
        this.myId = myId;
        this.myEnum = myEnum;
    }
    .....
}
```

@Enumerated(EnumType.ORDINAL) – это значение по умолчанию, говорит о том, что в базе будут храниться порядковые номера Enum (0, 1, 2...). Проблема с этим типом отображения возникает, когда нам нужно изменить наш Enum.

Если мы добавим новое значение в середину или просто изменим порядок перечисления, мы сломаем существующую модель данных. Такие проблемы могут быть трудно уловимыми, и нам придется обновлять все записи базы данных.

По именам

@Enumerated(EnumType.STRING) означает, что в базе будут храниться имена Enum.

Так мы можем безопасно добавлять новые значения перечисления или изменять порядок перечисления. Однако переименование значения enum все равно нарушит работу базы данных.

Кроме того, даже несмотря на то, что это представление данных гораздо более читаемо по сравнению с параметром @Enumerated(EnumType.ORDINAL), оно потребляет намного больше места, чем необходимо. Это может оказаться серьезной проблемой, когда нам нужно иметь дело с большим объемом данных.

@PostLoad и @PrePersist

Другой вариант – это использование стандартных методов обратного вызова из JPA.

Можно сmapить перечисления в БД и обратно в методах с аннотациями @PostLoad и @PrePersist.

Идея состоит в том, чтобы в Entity иметь не только поле с Enum, но и вспомогательное поле.

```
@Entity
public class Article {
    @Id
    private int id;
    private String title;
    @Enumerated(EnumType.ORDINAL)
    private Status status;
    @Enumerated(EnumType.STRING)
    private Type type;
    @Basic
    private int priorityValue;
    @Transient
    private Priority priority;
    @PostLoad void fillTransient() {
        if (priorityValue > 0) {
            this.priority = Priority.of(priorityValue);
        }
    }
    @PrePersist
    void fillPersistent() {
        if (priority != null) {
            this.priorityValue = priority.getPriority();
        }
    }
}
```

```
public enum Priority {
    LOW(100), MEDIUM(200), HIGH(300);

    private int priority;

    private Priority(int priority) {
        this.priority = priority;
    }

    public int getPriority() {
        return priority;
    }

    public static Priority of(int priority) {
        return Stream.of(Priority.values())
            .filter(p -> p.getPriority() == priority)
            .findFirst()
            .orElseThrow(IllegalArgumentException::new);
    }
}
```

Поле сEnum аннотируем @Transient, а в БД будет храниться значение из вспомогательного поля. Создадим Enum с полем priority, содержащем числовое значение приоритета (см. код слева).

Здесь добавили метод **Priority.of()**, чтобы упростить получение экземпляра Priority на основе его значения int. Теперь, чтобы использовать его в классе Article, нужно добавить два атрибута и реализовать методы обратного вызова.

Несмотря на то, что этот вариант дает нам большую гибкость по сравнению с ранее описанными решениями, он не идеален.

Просто кажется неправильным иметь в entity целых два атрибута, представляющих одно перечисление.

Кроме того, если мы используем этот вариант, мы не сможем использовать значение Enum в запросах JPQL.

Converter

В JPA с версии 2.1 можно использовать Converter для конвертации Enum -а в некое его значение для сохранения в БД и получения из БД. Все, что нужно сделать, это создать новый класс, который реализует интерфейс `javax.persistence.AttributeConverter` и аннотировать его с помощью `@Converter`.

```
public enum Category {
    SPORT("S"), MUSIC("M"), TECHNOLOGY("T");

    private String code;

    private Category(String code) {
        this.code = code;
    }
    public String getCode() {
        return code;
    }
}

@Entity
public class Article {
    @Id
    private int id;
    private String title;
    @Basic
    private int priorityValue;
    @Transient
    private Priority priority;
    private Category category;
}
```

```
@Converter(autoApply = true)
public class CategoryConverter implements AttributeConverter<Category, String> {
    @Override
    public String convertToDatabaseColumn(Category category) {
        if (category == null) {
            return null;
        }
        return category.getCode();
    }
    @Override
    public Category convertToEntityAttribute(String code) {
        if (code == null) {
            return null;
        }
        return Stream.of(Category.values())
            .filter(c -> c.getCode().equals(code))
            .findFirst()
            .orElseThrow(IllegalArgumentException::new);
    }
}
```

Тут установили `@Converter(autoApply=true)`, чтобы JPA автоматически применял логику преобразования ко всем сопоставленным атрибутам типа Category.

В противном случае пришлось бы поместить аннотацию `@Converter` непосредственно над полем Category у каждой сущности, где оно имеется. В результате в столбце таблицы будут храниться значения: "S", "M" или "T".

То есть можно просто установить собственные правила преобразования перечислений в соответствующие значения базы данных, если использовать **интерфейс AttributeConverter**.

Более того, можно безопасно добавлять новые значения enum или изменять существующие, не нарушая уже сохраненные данные. Это решение просто в реализации и устраняет все недостатки с `@Enumerated(EnumType.ORDINAL)`, `@Enumerated(EnumType.STRING)` и методами обратного вызова.

23. Как мапятся даты (до Java 8 и старше)?

При работе с датами рекомендуется установить определенный часовой пояс для драйвера JDBC. Таким образом, приложение будет независимым от текущего часового пояса системы.

Другой способ – настроить свойство `hibernate.jdbc.time_zone` в файле свойств Hibernate, который используется для создания фабрики сессий. Таким образом, можно указать часовой пояс **один раз для всего приложения**.

java.sql

Hibernate позволяет отображать различные классы даты/времени из Java в таблицах баз данных. Стандарт SQL определяет три типа даты/времени:

- **DATE** представляет календарную дату путем хранения лет, месяцев и дней.
Эквивалентом JDBC является `java.sql.Date`.
- **TIME** представляет время дня и хранит часы, минуты и секунды.
Эквивалентом JDBC является `java.sql.Time`.
- **TIMESTAMP** хранит как DATE, так и TIME плюс наносекунды.
Эквивалентом JDBC является `java.sql.Timestamp`.

Поскольку эти типы соответствуют SQL, их сопоставление относительно простое. Можно использовать аннотацию `@Basic` или `@Column`

→

```
@Entity
public class TemporalValues {
    @Basic
    private java.sql.Date sqlDate;
    @Basic
    private java.sql.Time sqlTime;
    @Basic
    private java.sql.Timestamp sqlTimestamp;
}
```

Далее можно установить соответствующие значения:

```
temporalValues.setSqlDate(java.sql.Date.valueOf("2017-11-15"));
temporalValues.setSqlTime(java.sql.Time.valueOf("15:30:14"));
temporalValues.setSqlTimestamp(
    java.sql.Timestamp.valueOf("2017-11-15 15:30:14.332"));
```

Использование типов `java.sql` для полей сущностей не всегда может быть хорошим выбором. Эти классы специфичны для JDBC и содержат множество устаревших функций.

Чтобы избежать зависимостей от пакета `java.sql`, начали использовать классы даты/времени из пакета `java.util` вместо классов `java.sql.Timestamp` и `java.sql.Time`.

java.util

Точность представления времени составляет одну миллисекунду. Для большинства практических задач этого более чем достаточно, но иногда хочется иметь точность повыше. Поскольку классы в данном API изменяемые (не `immutable`), использовать их в многопоточной среде нужно с осторожностью. В частности `java.util.Date` можно признать «эффективно» поток безопасным, если вы не вызываете у него устаревшие методы.

java.util.Date

Тип `java.util.Date` содержит информацию о дате и времени с точностью до миллисекунд. Но так как классы из этого пакета не имели прямого соответствия типам данных SQL, приходилось использовать над полями `java.util.Date` аннотацию `@Temporal`, чтобы дать понять SQL, с каким конкретно типом данных она работает.

```
@Basic
@Temporal(TemporalType.DATE)
private java.util.Date utilDate;

@Basic
@Temporal(TemporalType.TIME)
private java.util.Date utilTime;

@Basic
@Temporal(TemporalType.TIMESTAMP)
private java.util.Date utilTimestamp;
```

Для этого у аннотации `@Temporal` нужно было указать параметр `TemporalType`, который принимал одно из трёх значений: `DATE`, `TIME` или `TIMESTAMP`, что позволяло указать базе данных с какими конкретными типами данных она работает.

Тип `java.util.Date` имеет точность до миллисекунд, и недостаточно точен для обработки SQL-значения `Timestamp`, который имеет точность вплоть до наносекунд. Поэтому, когда мы извлекаем сущность из базы данных, неудивительно, что в этом поле мы находим экземпляр `java.sql.Timestamp`, даже если изначально мы сохранили `java.util.Date`. Но это не страшно, так как `Timestamp` наследуется от `Date`.

java.util.Calendar

Как и в случае `java.util.Date`, тип `java.util.Calendar` может быть сопоставлен с различными типами SQL, поэтому мы должны указать их с помощью `@Temporal`. Разница лишь в том, что Hibernate не поддерживает отображение (маппинг) `Calendar` на `TIME` →

```
@Basic  
@Temporal(TemporalType.DATE)  
private java.util.Calendar calendarDate;  
  
@Basic  
@Temporal(TemporalType.TIMESTAMP)  
private java.util.Calendar calendarTimestamp;
```

java.time

Начиная с Java 8, доступен новый API даты и времени для работы с временными значениями. Этот API-интерфейс устраниет многие проблемы классов `java.util.Date` и `java.util.Calendar`.

Все классы в новом API неизменяемые (immutable) и, как следствие, поток безопасные. Точность представления времени составляет одну наносекунду, что в миллион раз точнее чем в пакете `java.util`.

Типы данных из пакета `java.time` напрямую отображаются (мапятся) на соответствующие типы SQL и поэтому нет необходимости явно указывать аннотацию `@Temporal`:

- `LocalDate` соответствует `DATE`.
- `LocalTime` и `OffsetTime` соответствуют `TIME`.
- `Instant`, `LocalDateTime`, `OffsetDateTime` и `ZonedDateTime` соответствуют `TIMESTAMP`.

Это означает, что можно пометить эти поля только аннотацией `@Basic` (или `@Column`), например →

Каждый временной класс в пакете `java.time` имеет статический метод `parse()` для анализа предоставленного значения типа `String` с использованием соответствующего формата.

Вот как можно установить значения полей сущности:

```
temporalValues.setLocalDate(LocalDate.parse("2017-11-15"));  
temporalValues.setLocalTime(LocalTime.parse("15:30:18"));  
temporalValues.setOffsetTime(OffsetTime.parse("08:22:12+01:00"));  
temporalValues.setInstant(Instant.parse("2017-11-15T08:22:12Z"));  
temporalValues.setLocalDateTime(  
    LocalDateTime.parse("2017-11-15T08:22:12"));  
temporalValues.setOffsetDateTime(  
    OffsetDateTime.parse("2017-11-15T08:22:12+01:00"));  
temporalValues.setZonedDateTime(  
    ZonedDateTime.parse("2017-11-15T08:22:12+01:00[Europe/Paris]"));
```

```
@Basic  
private java.time.LocalDate localDate;  
@Basic  
private java.time.LocalTime localTime;  
@Basic  
private java.time.OffsetTime offsetTime;  
@Basic  
private java.time.Instant instant;  
@Basic  
private java.time.LocalDateTime localDateTime;  
@Basic  
private java.time.OffsetDateTime offsetDateTime;  
@Basic  
private java.time.ZonedDateTime zonedDateTime;
```

```
@Temporal(TemporalType.TIMESTAMP)  
private Date date;  
Устаревшие аналоги до Java 8)  
private LocalDateTime localDateTime;  
  
private LocalDate localDate;  
  
private LocalTime localTime;
```

24. Как Hibernate работает с разными типами?

Подробно о типах Hibernate читаем по ссылке: <http://hibernate-refdoc.3141.ru/ch6.Types>

Для экземпляров классов ТИПОМ считается **название класса**, например при объявлении экземпляра TextView используется TextView tv = findViewById(R.id.some_id);

Важно

Tип *Hibernate* не является ни Java-типом, ни типом данных SQL; он предоставляет информацию об обоих. Когда вы сталкиваетесь с термином *тип* в отношении Hibernate, имейте в виду, что он может подразумевать как тип Java, так и тип SQL/JDBC или тип Hibernate.

Тип *Hibernate* описывает различные аспекты поведения типа Java, такие как «как проверяется равенство?» или «как клонируются значения?».

Hibernate классифицирует типы на ДВЕ группы высокого уровня:

- СУЩНОСТИ (у них есть id и состояние ЖЦ Entity)
- ЗНАЧЕНИЯ (у них нет id и нет состояния, они сами не определяют свой ЖЦ, а лишь принадлежат сущностям)

💡 Типы сущностей

Определение сущностей подробно описано в [главе 4 «Постоянные классы»](#). В рамках текущего обсуждения достаточно сказать, что **сущности являются классами (обычно специфичными для приложения)**, которые коррелируют с строками в таблице. В частности, они соотносятся с строкой с помощью уникального идентификатора. Из-за этого уникального идентификатора сущности существуют независимо и определяют свой собственный жизненный цикл. В качестве примера, когда мы удаляем Membership, сущности User и Group сохраняются.



Заметка

Это понятие независимости сущности может быть изменено разработчиком приложения с использованием концепции каскадов. Каскады позволяют некоторым операциям продолжать выполняться (каскадом) через объединение от одной сущности к другой. Каскады подробно описаны в [главе 8 «Ассоциативные отображения»](#).

💡 Типы значений

Основной отличительной характеристикой типов значения является тот факт, что они не определяют свой собственный жизненный цикл. Мы говорим, что они **принадлежат** чему-то другому (в частности, сущности, как мы увидим позже), которая определяет их жизненный цикл. Типы значений далее подразделяются на три подкатегории: основные типы (см. раздел [«6.1.1 Основные типы значений»](#)), составные типы (см. раздел [«6.1.2 Составные типы»](#)) и типы коллекций (см. раздел [«6.1.3 Типы коллекций»](#)).

Есть также базовые типы (они же основные), встраиваемые типы и коллекции.

К базовым типам относятся:

- примитивные типы и их обертки
- String, массивы
- java.net.URL
- BigDecimal и BigInteger
- java.util.Date и java.sql.Date (Timestamp, Time, Date, Calendar)
- java.util.TimeZone
- java.util.Currency
- java.util.Locale
- java.lang.Class
- java.util.UUID
- java.io.Serializable

К встраиваемым типам относятся:

Компоненты представляют **агрегацию значений в один тип Java**. Например, у вас есть класс Address, который объединяет информацию о улицах, городах, штатах и т. д. Или класс Name, который объединяет части имени человека. Во многих случаях компонент выглядит точно, как сущность. Они оба (вообще говоря) классы, написанные специально для приложения. Оба они могут иметь ссылки на другие классы в приложении, а также на коллекции и простые типы JDK. Как обсуждалось ранее, единственной отличительной особенностью является тот факт, что **компонент не имеет собственного жизненного цикла и не определяет идентификатор**.

Базовые типы (любые типы с реализацией Serializable), для которых и используется аннотация **@Basic**, соответствуют одному столбцу в БД. Эта аннотация НЕ обязательная.

Составные типы



Заметка

Java Persistence API называет их **встроенные (embedded) типы**, тогда как Hibernate традиционно называет их компонентами. Просто имейте в виду, что оба термина используются и означают одно и то же когда обсуждается Hibernate.

Типы коллекций



Важно

Очень важно понимать, что мы имеем в виду **саму коллекцию, а не ее содержимое**. Содержимое коллекции, в свою очередь, может быть базовым типом, составным типом или типом сущностным (хотя и не коллекциями), но принадлежать самой коллекции.

25. Как сохранять в базе данных коллекции базовых типов?

Если у сущности есть поле с коллекцией, то мы привыкли ставить над ним аннотации `@OneToOne` либо `@ManyToMany`. Но данные аннотации применяются в случае, когда это **коллекция других сущностей (entities)**.

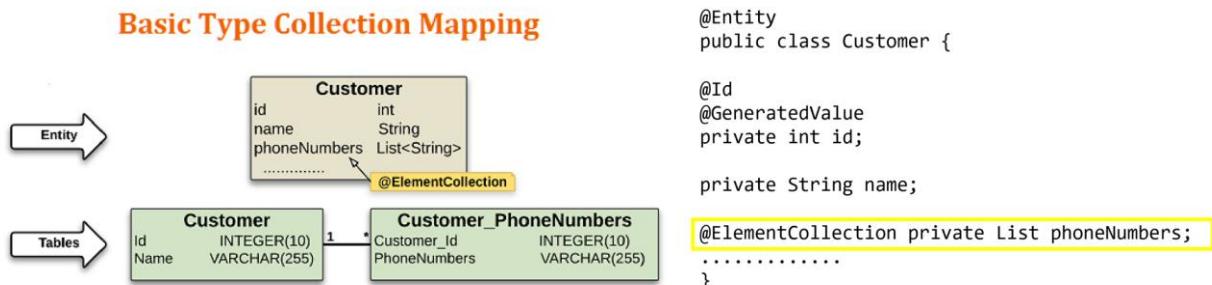
Но что, если у сущности коллекция не других сущностей, а **базовых или встраиваемых (embeddable) типов**, то есть коллекция элементов? См. Как Hibernate работает с разными типами?

Для этих случаев в JPA имеется специальная аннотация `@ElementCollection`, которая указывается в классе сущности над полем коллекции базовых или встраиваемых типов.

Все записи коллекции хранятся в отдельной таблице, то есть в итоге получаем ДВЕ таблицы:

- одну для сущности
- вторую для коллекции элементов

Конфигурация для таблицы коллекции элементов указывается с помощью аннотации `@CollectionTable`, которая используется для указания имени таблицы коллекции и `@JoinColumn`, который ссылается на первичную таблицу.



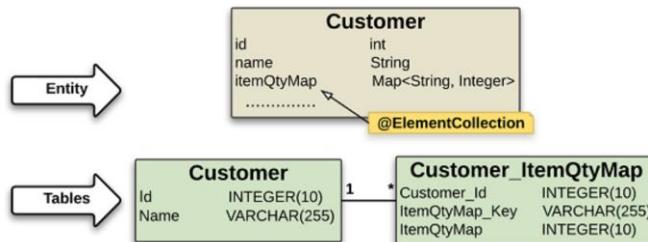
Аннотация `@ElementCollection` похожа на отношение `@OneToMany`, за исключением того, что **целью являются базовые и встраиваемые типы, а не сущности**.

Можно использовать аннотации `@AttributeOverrides` и `@AttributeOverride` для настройки отображения в таблице полей базовых или встраиваемых типов.

Коллекции могут иметь тип `java.util.Map`, которые состоят из ключа и значения. Для этого типа коллекций применяются следующие правила:

- Ключ или значение `Map` может быть базовым типом языка программирования Java, встраиваемым классом или сущностью.
- Если значение Мап является встраиваемым классом или базовым типом, используйте аннотацию `@ElementCollection`

Persisting `java.util.Map`



- Если значение Мап является сущностью, используйте аннотацию `@OneToMany` или `@ManyToMany`
- Использовать тип **Мар только на одной стороне двунаправленной связи**.

Аннотация `@MapKeyColumn` позволяет настроить столбец «ключ» в таблице Мар.
Аннотация `@Column` позволяет настроить столбец «значение» в таблице Мар.

Использование коллекций элементов имеет один большой недостаток: элементы коллекции не имеют идентификатора, и Hibernate не может обращаться индивидуально к каждому элементу коллекции.

Когда нужно добавить новый объект в коллекцию или удалить из коллекции существующий элемент, Hibernate удаляет все строки из таблицы элементов и вставляет новые строки по одной для каждого элемента в коллекции.

То есть при добавлении одного элемента в коллекцию, **Hibernate** не добавит одну строку в таблицу коллекции, а очистит её и заполнит по новой всеми элементами.

Поэтому **коллекции элементов следует использовать только для очень маленьких коллекций**, чтобы Hibernate не выполнял слишком много операторов SQL. Во всех других случаях рекомендуется использовать коллекции сущностей с `@OneToMany`

26. Какие есть виды связей (маппинг ассоциаций)?

Вид связи	Описание	Пример	Тип загрузки по умолчанию
OneToOne	один экземпляр Entity может быть связан не больше, чем с одним экземпляром другого Entity	Один гражданин – одно место рождения	EAGER
OneToMany	один экземпляр Entity может быть связан с несколькими экземплярами других Entity (коллекцией сущностей)	Один гражданин – много паспортов (гражданский и парочка заграничных)	LAZY
ManyToOne	Несколько экземпляров Entity (коллекция сущностей) могут быть связаны с одним экземпляром другого Entity	Много студентов учатся на оной кафедре	EAGER
ManyToMany	экземпляры Entity могут быть связаны с несколькими экземплярами друг друга	Много студентов учатся у разных (многих) преподавателей	LAZY

Первое слово относится к классу Entity, второе к полю в этом классе (ассоциации).

Например, много отзывов об одной книге. В классе `Review` над полем `book` будет аннотация `@ManyToOne`. А в классе `Book` аннотация `@OneToMany` над полем-коллекцией `reviews`.

Направления в отношениях сущностей: unidirectional и bidirectional

Однонаправленные (uni-directional associations): одна сторона знает про отношения, а другая нет. Однонаправленные отношения имеют только одного владельца связи.

Сотрудники – это список (таблица) сотрудников компании.

Детали – это таблица с персональными данными сотрудников.

Поле **details_id** – это FK в таблице **employees** (ссылается на id таблицы details).

Связь односторонняя. Владелец связи – details (т.к. на него ссылается employees).

Сотрудники «знают» о своих деталях (приватное поле **empDetail** в классе Employees).

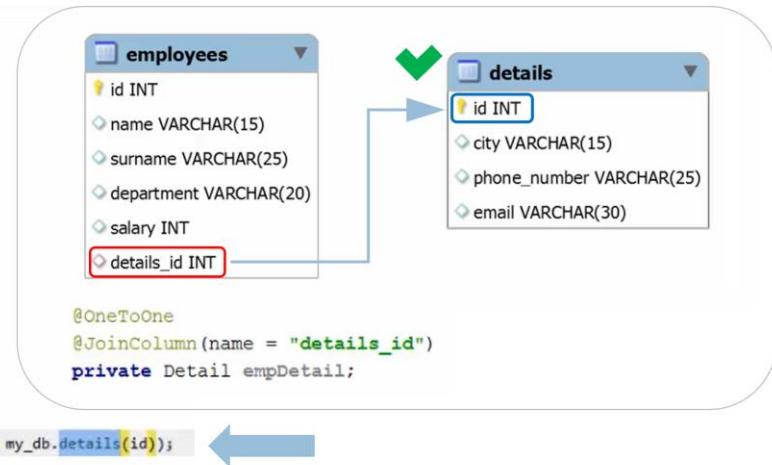
Детали ничего о сотрудниках «не знают».

Пример @OneToOne (uni-directional)

```

1 • CREATE TABLE my_db.details (
2     id INT NOT NULL AUTO_INCREMENT,
3     city VARCHAR(15),
4     phone_number VARCHAR(25),
5     email VARCHAR(30),
6     PRIMARY KEY (id)
7 );
8
9 • CREATE TABLE my_db.employees (
10    id INT NOT NULL AUTO_INCREMENT,
11    name VARCHAR(15),
12    surname VARCHAR(25),
13    department VARCHAR(20),
14    salary INT,
15    details_id INT
16    , PRIMARY KEY (id)
17    , FOREIGN KEY (details_id) REFERENCES my_db.details(id));

```



Пример @OneToMany (uni-directional)

Когда связь **One-to-Many** (один департамент, много сотрудников), то **Foreign Key** всегда будет в таблице, которая отвечает за **Many**. Поле **department_id** – это FK в таблице **employees** (target table), ссылается на id таблицы **departments** (source table).

Связь односторонняя и владелец связи – это сторона One (департамент, т.к. на него ссылка).

Департамент «знает» о сотрудниках (приватное поле-коллекция **emps**), сотрудник «не знает» о деп-те.



При использовании связи One-to-Many в аннотации **@JoinColumn** name будет ссылаться на Foreign Key не из source, а из target таблицы.

Атрибут моделирует ассоциацию, а аннотация объявляет тип отношения.

Маппинг ассоциации зависит только от исходной таблицы (source table, владелец связи) и целевого объекта (target entity). Если у **отдела** (один) есть набор, **коллекция сотрудников** (много), а у сотрудника нет собственного отдела, тогда используем **unidirectional @OneToMany**.

В этом случае **@JoinColumn** должен быть в классе **ВЛАДЕЛЬЦА СВЯЗИ** (department). У нас это Департамент, т.к. на него ссылается внешний ключ класса Employees.

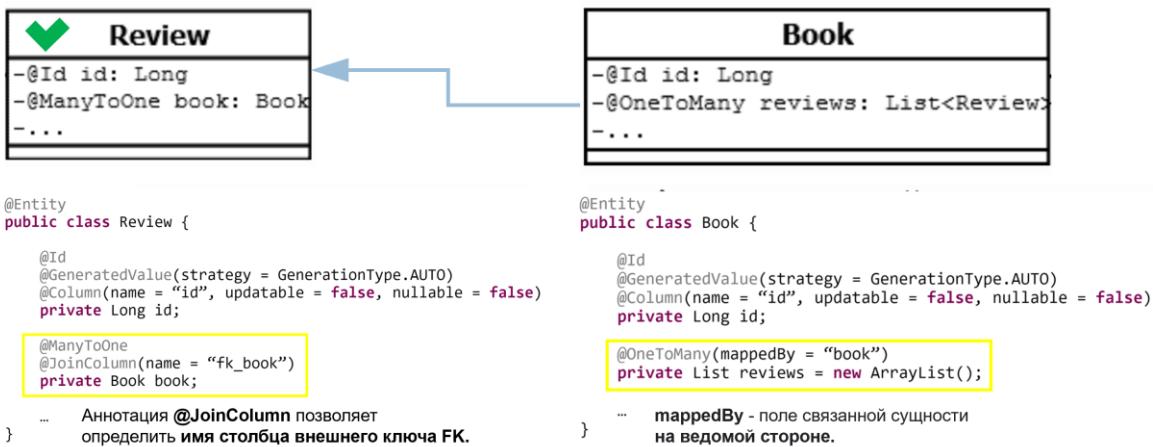
Двунаправленные (bi-directional associations): каждая сущность имеет поле, которое ссылается на другую сущность. Через это поле код первой сущности может получить доступ ко второй сущности, находящейся на другой стороне отношений.

Если у первой сущности есть поле, ссылающееся на вторую сущность, и наоборот, то в этом случае говорят, что обе сущности знают друг о друге, и что они состоят в двунаправленных отношениях.

Пример @ManyToOne (bi-directional)

Пример: Книга в интернет-магазине может иметь несколько отзывов. Объект Review имеет отношение @ManyToOne к объекту Book, а Book имеет отношение @OneToMany к объекту Review.

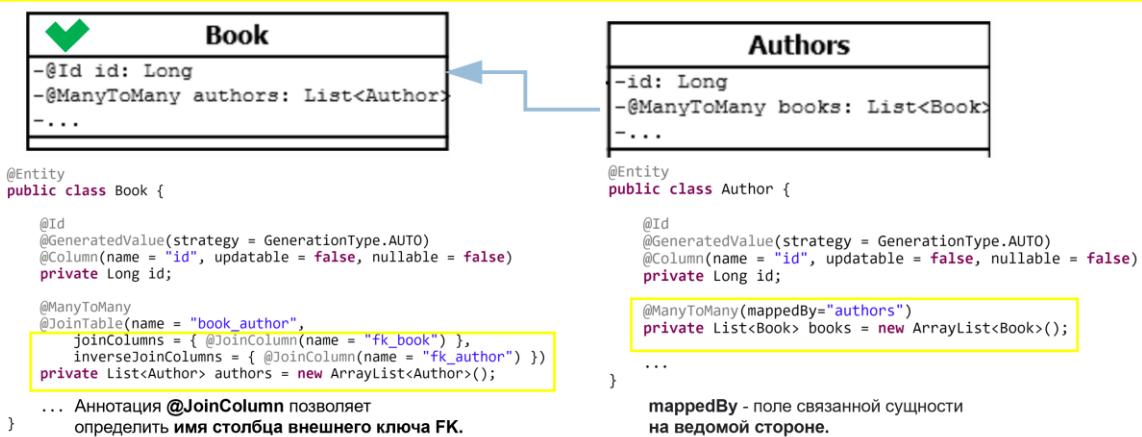
Связь двунаправленная, оба объекта знают друг о друге (в классе Review приватное поле **fk_book**, а в классе book приватное **поле-коллекция reviews**). **Объект Review является владельцем ассоциации**, т.к. он определяет ассоциацию, а сущность Book просто ссылается на нее (поле mappedBy на ведомой стороне)



Пример @ManyToMany (bi-directional)

Несколько авторов могут написать несколько книг, а книга может быть написана одним или несколькими авторами. В этом примере ассоциация **Many-to-Many для объекта Book и для объекта Author**.

Связь двунаправленная, оба объекта знают друг о друге (в классе Book приватное поле-коллекция **authors**, а в классе Author приватное **поле-коллекция books**). **Объект Book является владельцем ассоциации**, т.к. он определяет ассоциацию, а сущность Author просто ссылается на него (поле mappedBy на ведомой стороне)



27. Что такое владелец связи? код тут: <https://gist.github.com/thergbway/a5ebd11af6bd4dfa4600da32f50d37da>



Связи бывают односторонними и двусторонними: unidirectional и bidirectional.

Обозначения на UML: стрелка с направлением, либо одна соединительная линия.

В реляционных БД связи осуществляются с использованием внешнего ключа (1-1) или таблицы соединения (1-N, N-N).

Множественность связи (или кардинальность).

Связь с более чем одним объектом (Many) **является типом коллекции:** List, Set, Map.

Владелец связи. При односторонней связи владелец – это инициатор связи, при двусторонней владельца надо явно указывать.

Общий принцип: **на КОГО ссылаются, тот и владелец связи.**

При односторонней связи настраиваем **параметры внешнего ключа** таблицы во владеющей сущности.

Если связь двусторонняя, то владеющая сторона определяет параметры, а **противоположная указывает в поле mappedBy имя соответствующего параметра** владеющей связью сущности (атрибут mappedBy ВСЕГДА на ведомой стороне).

Каждая из связей односторонняя или двусторонняя может быть одним видом отношений.

Все возможные виды связей помнить не нужно (многие очень похожи друг на друга), поэтому достаточно запомнить лишь некоторые →

#Тип связи	#Направление
* OneToOne	Unidirectional
* OneToOne	Bidirectional
* OneToMany	Unidirectional
* OneToMany/ManyToOne	Bidirectional
* ManyToOne	Unidirectional
* ManyToMany	Unidirectional
* ManyToMany	Bidirectional

Разумно правильно определять тип загрузки сущности (fetch = LAZY or EAGER).

По дефолту:

- Для @OneToOne, @ManyToOne fetch = EAGER (т.к. загружается только одна сущность)
- Для @OneToMany, @ManyToMany fetch = LAZY (оптимизация, производительность)

Можно для сущности задавать упорядочивание элементов-коллекций:

- @OrderBy(attribute_name_1 DESC, attribute_name_2 ASC).
Упорядочение при выборке согласно указанным атрибутам сущности
- @OrderColumn(name = "column_index_name").
Коллекция будет упорядочена в соответствии с индексом элементов, записанных в отдельном столбце.
Есть потеря производительности, т.к. данный столбец надо держать актуальным.

28. Что такое каскады?

Если посмотреть документацию, то CascadeType – это Enum, в котором перечисления соответствуют этапам ЖЦ Entity

Как работают каскады, что за кулисами?

JPA позволяет распространять операции с сущностями (например, persist или remove) на связанные сущности.

Это означает, что при включенном каскадировании если сущность – ВЛАДЕЛЕЦ СВЯЗИ переходит в одно из своих состояний, то зависимая сущность **без явных команд** перейдёт в то же состояние (например, будет удалена):

Once: Java Persistence 1.0
public enum CascadeType {
Соответствует этапам ЖЦ Entity
Cascade all operations
ALL, BCE операции
Cascade persist operation
PERSIST, Сохранение Entity
Cascade merge operation
MERGE, Слияние Entity
Cascade remove operation
REMOVE, Удаление Entity
Cascade refresh operation
REFRESH, Обновление Entity
Cascade detach operation
Since: Java Persistence 2.0
DETACH Отсоединение
}

class User (владелец связи)

```
@ManyToOne(fetch = FetchType.EAGER, cascade = {CascadeType.ALL})
@JoinColumn(name = "company_id") // company_id
private Company company;
```

class Company

```
@Builder.Default
@OneToMany(mappedBy = "company", cascade = CascadeType.ALL)
private Set<User> users = new HashSet<>();
```

Удаляем компанию

```
var company : Company = session.get(Company.class, id: 3);
session.delete(company);
```

Hibernate удалит сначала всех юзеров, а потом компанию.

```
Hibernate:
delete
from
    public.users
where
    id=?
```

```
Hibernate:
delete
from
    company
where
    id=?
```

Как правило, **каскадные операции применяются от владельца связи к зависимым сущностям**, но они могут распространяться и в обратном направлении (хотя это совсем не нужно). Главное условие: между ними должно быть двунаправленное отношение, иначе каскадные операции выполняются только в одном направлении.

29. Какие два типа fetch стратегии в JPA вы знаете?

Для чтения связанных объектов из БД используются две стратегии загрузок (fetch type): EAGER и LAZY. В первом случае объекты коллекции сразу загружаются в память, во втором случае — только при обращении к ним. Оба этих подхода имеют достоинства и недостатки.

В случае **FetchType.EAGER** в памяти будут находиться все загруженные объекты, даже если нужен только один объект из десятка (сотен/тысяч). При использовании данной стратегии необходимо быть внимательным, поскольку при загрузке какого-нибудь корневого объекта, который связан со всеми остальными объектами и коллекциями, можно случайно попытаться загрузить в память и всю базу.

Согласно стратегии **FetchType.LAZY** связанные объекты загружаются только по мере необходимости, т.е. при обращении. Но при этом требуется, чтобы соединение с базой (или транзакция) сохранялись. Если точно, то требуется, чтобы объект был attached. Поэтому для работы с lazy объектами тратится больше ресурсов на поддержку соединений.

Тип связи и тип загрузки по умолчанию →

Вид связи	Тип загрузки по умолчанию
OneToOne	EAGER
OneToMany	LAZY
ManyToOne	EAGER
ManyToMany	LAZY

30. Для чего нужна аннотация Basic?

Аннотация **@Basic** для поля или свойства означает, что это **Basic Type*** (базовый тип), и Hibernate должен использовать стандартное сопоставление для его сохранения. Также в параметрах аннотации можно указать fetch стратегию доступа к полю и является ли это поле обязательным или нет. Однако, это необязательная аннотация.

* JPA поддерживает различные типы данных Java в качестве сохраняемых полей объекта, часто называемых **базовыми типами**. К ним относятся примитивы Java и их классы-оболочки, String, java.math.BigInteger и java.math.BigDecimal, различные доступные классы даты и времени, перечисления и **любой другой тип, который реализует интерфейс java.io.Serializable**.

31. Для чего нужна аннотация Column?

Аннотация говорит к какому именно столбцу в таблице БД мы привязываем поле класса.

Аннотация **@Column** сопоставляет поле класса столбцу таблицы, а её атрибуты определяют поведение в этом столбце, используется для генерации схемы базы данных.

Атрибут nullable аннотации @Column указывает, может ли соответствующий столбец в таблице быть null.

Аннотация @Column позволяет указать имя столбца в таблице и **ряд других свойств**:

insertable/updatable – можно ли добавлять/изменять данные в колонке (by default = true);
length – длина, для строковых типов данных, по умолчанию 255 символов.

32. Для чего нужна аннотация Access?

Аннотация **определяет тип доступа** (access type) для класса entity, суперкласса, embeddable или отдельных атрибутов, то есть **как JPA будет обращаться к атрибутам entity**, как к полям класса (FIELD, по умолчанию) или как к свойствам класса (PROPERTY), имеющим геттеры (getter) и сеттеры (setter).

 AccessType.FIELD (javax.persistence)	AccessType
 AccessType.PROPERTY (javax.persistence)	AccessType

33. Для чего нужна аннотация Cacheable?

Cacheable — позволяет **включить или выключить использование кэша второго уровня** (second-level cache) для данного Entity (если провайдер JPA поддерживает работу с кешированием и настройки кэша (second-level cache) стоят как ENABLE_SELECTIVE или DISABLE_SELECTIVE).

Аннотация `@Cacheable` размещается над классом Entity. Её действие распространяется на эту сущность и её наследников, если они не определили другое поведение.

*Каких провайдеров кэша второго уровня знаешь?

В Hibernate существует четыре стратегии одновременного доступа к объектам в кэше:

- Read-only (только чтение)
- Read-write (чтение-запись)
- Nonstrict-read-write (не строгое чтение-запись)
- Transactional (транзакционное)

См. также вопрос: как работать с кэшем второго уровня?

Чем менее строгую стратегию для кэша вы выбираете, тем большая производительность у кэша второго уровня. Hibernate имеет стратегию кэша по умолчанию, для этого нужно использовать в файле настроек hibernate.cfg.xml свойство: `hibernate.cache.default_cache_concurrency_strategy`

Например, сделаем стратегию кэша по умолчанию read-write:

```
<property name="hibernate.cache.default_cache_concurrency_strategy">read-write</property>
```

Рассмотрим несколько известных кэш провайдеров: какие стратегии они поддерживают?

Провайдер/Стратегия	Read-only	Nonstrict read-write	Read-write	Transactional
EHCache	✓	✓	✓	✓
HashTable (использовать только для тестирования)	✓	✓	✓	✗
Infinispan	✓	✗	✗	✓

Если выполняется запрос нескольких сущностей, при помощи критериев или языка запросов HQL, то действует кэш запросов, который также как и кэш второго уровня, нужно сначала включить в настройках hibernate и ему так же нужен сторонний кэш провайдер. Все сторонние кэш провайдеры совместимы с hibernate, реализуют интерфейс `org.hibernate.cache.spi.CacheProvider`, который нужно указать в файле настроек hibernate.cfg.xml для свойства `hibernate.cache.region.factory_class`, для простого понимания привожу таблицу.

Провайдер	Class провайдера	Поддерживает ли кластеры	Поддерживает ли кэш запросов
EHCache	<code>org.hibernate.cache.ehcache.EhCacheRegionFactory</code>	✓	✓
HashTable (использовать только для тестирования)	<code>org.hibernate.testing.cache.CachingRegionFactory</code>	✗	✓
Infinispan	<code>org.hibernate.cache.infinispan.InfinispanRegionFactory</code>	✓	✓

34. Как сmapить составной ключ?

Составной первичный ключ, также называемый составным ключом, представляет собой комбинацию из двух или более столбцов для формирования первичного ключа таблицы.

В соответствии с JPA, допустимые типы атрибутов для первичного ключа:

- примитивные типы и их обертки
- строки
- `BigDecimal` и `BigInteger`

- java.util.Date и java.sql.Date

В JPA есть требования к составному ключу:

- составной **КЛЮЧ** должен быть представлен классом первичного ключа, при этом используется одна из двух аннотаций: **@IdClass** и **@EmbeddedId**;
- **КЛАСС** первичного ключа должен быть public и иметь public конструктор без аргументов;
- **КЛАСС** первичного ключа должен имplementировать маркерный интерфейс Serializable;
- **КЛАСС** первичного ключа должен иметь методы equals и hashCode;
- **АТРИБУТЫ**, представляющие поля составного ключа, могут быть базовыми, составными и @ManyToOne, но НЕ могут быть коллекциями или **@OneToOne**.

Однако, первое правило имеется только в JPA.

Hibernate позволяет определять составные идентификаторы без «класса первичного ключа» с помощью нескольких атрибутов с аннотацией **@Id**.

@IdClass

Допустим, есть таблица Account, в ней два столбца: accountNumber и accountType, которые формируют составной ключ. Чтобы обозначить оба этих поля как части составного ключа мы должны создать класс, например, AccountId с этими полями:

```
public class AccountId implements Serializable {  
    private String accountNumber;  
    private String accountType;  
    // default constructor  
    public AccountId(String accountNumber, String accountType) {  
        this.accountNumber = accountNumber;  
        this.accountType = accountType;  
    }  
    // equals() and hashCode()  
}
```

Затем нужно аннотировать сущность Account аннотацией **@IdClass** и объявить поля из класса AccountId в entity Account с такими же именами и аннотировать их **@Id** →

```
@Entity  
@IdClass(AccountId.class)  
public class Account {  
    @Id  
    private String accountNumber;  
    @Id  
    private String accountType;  
    // other fields, getters and setters  
}
```

@EmbeddedId

Является альтернативой аннотации **@IdClass**. Рассмотрим пример: необходимо сохранить информацию о книге с заголовком и языком в качестве полей Primary Key.

В этом случае класс первичного ключа, BookId, должен быть аннотирован **@Embeddable**.

Затем нужно встроить этот класс в сущность Book, используя **@EmbeddedId**.

```
@Embeddable
public class BookId implements Serializable {
    private String title;
    private String language;
    // default constructor
    public BookId(String title, String language) {
        this.title = title;
        this.language = language;
    }
    // getters, equals() and hashCode() methods
}

@Entity
public class Book {
    @EmbeddedId
    private BookId bookId;
    // constructors, other fields, getters and setters
}
```

@IdClass v.s. @EmbeddedId

С аннотацией **@IdClass** пришлось указывать столбцы дважды - в AccountId и в Account. Но с **@EmbeddedId** мы этого не сделали;

JPQL-запросы с **@IdClass** проще. С **@EmbeddedId**, чтобы получить доступ к полю, нужно из сущности обратиться к встраиваемому классу и потом к его полю:

```
SELECT account.accountNumber FROM Account account // с @IdClass
SELECT book.bookId.title FROM Book book // с @EmbeddedId
```

@EmbeddedId более подробна, чем **@IdClass**, поскольку можно получить доступ ко всему объекту первичного ключа, используя метод доступа к полю в классе-сущности. Это также дает четкое представление о полях, которые являются частью составного ключа, поскольку все они агрегированы в классе, который доступен только через метод доступа к полям;

@IdClass может быть предпочтительным выбором по сравнению с **@EmbeddedId** в ситуациях, когда класс составного первичного ключа поступает из другого модуля или устаревшего кода, а также когда невозможно его изменить, например, чтобы установить аннотацию **@EmbeddedId**. Для таких сценариев, когда нельзя изменить класс составного ключа, аннотация @IdClass является единственным выходом;

Если нужно получить доступ к частям составного ключа по отдельности, можно использовать **@IdClass**, но в тех местах, где часто используется полный идентификатор в качестве объекта, @EmbeddedId предпочтительнее.

35. Для чего нужна аннотация ID?

@Id определяет простой (не составной) первичный ключ, состоящий из одного поля.

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

В соответствии с JPA, допустимые типы атрибутов для первичного ключа:

- примитивные типы и их обертки
- строки
- BigDecimal и BigInteger
- java.util.Date и java.sql.Date

36. Какие @GeneratedValue вы знаете?

GenerationType.AUTO – дефолтный тип. Выбор стратегии будет зависеть от типа базы, с которой мы работаем.

GenerationType.IDENTITY полагается на автоматическое увеличение столбца по правилам, прописанным в БД-х.

GenerationType.SEQUENCE полагается на работу Sequence, созданного в БД-х.

GenerationType.TABLE полагается на значение столбца таблицы БД-х. Цель такой таблицы – поддержка уникальности значений.

Чтобы значение первичного ключа генерировалось автоматически, нужно добавить первичному ключу, отмеченному аннотацией `@Id`, аннотацию `@GeneratedValue`.

Согласно спецификации JPA возможно 4 различных варианта стратегии генерации ключа: **AUTO, IDENTITY, SEQUENCE, TABLE**.

Если не указать значение явно, то типом генерации по умолчанию будет **AUTO**. Спецификация JPA строго не определяет поведение этих стратегий.

AUTO

Указывает, что Hibernate должен выбрать подходящую стратегию для конкретной базы данных, учитывая её диалект, так как у разных БД разные способы по умолчанию.

То, как провайдер должен реализовывать тип генерации AUTO, оставлено на усмотрение провайдера. Поведение по умолчанию – исходить из типа поля идентификатора. С версии Hibernate 5.0 для числовых значений генерация основана на SEQUENCE, и, если БД её не поддерживает, то на TABLE.

Пример AUTO, в котором значения первичного ключа будут уникальными на уровне базы данных →

Интересная особенность, представленная в Hibernate 5, это **UUIDGenerator**. Чтобы его использовать, необходимо объявить идентификатор типа UUID с аннотацией `@GeneratedValue`:

```
@Entity
public class Student {
    @Id
    @GeneratedValue
    private long studentId;
    ...
}
```

```
@Entity
public class Course {
    @Id
    @GeneratedValue
    private UUID courseId;
    ...
}
```

Hibernate сгенерирует идентификатор вида «8dd5f315-9788-4d00-87bb10eed9eff566».

IDENTITY

Указывает, что для генерации значения первичного ключа будет использоваться столбец **IDENTITY**, имеющийся в базе данных. Значения в столбце автоматически увеличиваются, что позволяет базе данных генерировать новое значение при каждой операции вставки.

С точки зрения базы данных это очень эффективно, поскольку столбцы с автоинкрементом хорошо оптимизированы и не требуют каких-либо дополнительных операторов. Процесс инкремента (получения следующего) первичного ключа происходит

вне текущей выполняемой транзакции, поэтому откат транзакции может в конечном итоге обнулить уже присвоенные значения (могут возникнуть пропуски значений).

Если используется Hibernate, то применение IDENTITY имеет существенный недостаток. Так как Hibernate нужен первичный ключ для работы с managed-объектом в persistence context, а мы не можем узнать значение первичного ключа ДО выполнения инструкции INSERT, то Hibernate должен немедленно выполнить оператор INSERT, чтобы получить этот самый первичный ключ, сгенерированный БД.

Только после этого у Hibernate будет возможность работать с сущностью в контексте персистентности, после чего выполнить операцию persist.

Но Hibernate, в соответствии со своей идеологией, использует стратегию **“транзакционная запись-после”** (transactional write behind), согласно которой он пытается максимально отложить сброс данных в БД из контекста персистентности, чтобы не делать много обращений к БД.

Так как поведение при IDENTITY противоречит идеологии и стратегии “транзакционная запись-после”, Hibernate отключает пакетные вставки (batching inserts) для объектов, использующих генератор IDENTITY. Однако, пакетные обновления и удаления (batching updates и batching deletes) всё же поддерживаются.

IDENTITY является самым простым в использовании типом генерации, **но не самым лучшим с точки зрения производительности.** Стратегия генератора первичных ключей IDENTITY не работает при TABLE PER CLASS, поскольку может быть несколько объектов подкласса, имеющих один и тот же идентификатор, и запрос базового класса приведет к получению объектов с одним и тем же идентификатором (даже если они принадлежат разным типам).

SEQUENCE

Указывает, что для получения значений первичного ключа Hibernate должен использовать имеющиеся в базе данных механизмы генерации последовательных значений (Sequence). Но если БД не поддерживает тип SEQUENCE, то Hibernate автоматически переключится на тип TABLE.

SEQUENCE – это объект базы данных, который генерирует инкрементные целые числа при каждом последующем запросе. **SEQUENCE намного более гибкий**, чем IDENTITY, т.к.:

- не содержит таблиц, и одну и ту же последовательность можно назначить нескольким столбцам или таблицам
- может предварительно распределять значения для улучшения производительности
- может определять шаг инкремента, что позволяет воспользоваться «объединенным» алгоритмом Hi/Lo <https://www.baeldung.com/hi-lo-algorithm-hibernate>
- не ограничивает пакетные вставки JDBC в Hibernate
- не ограничивает модели наследования Hibernate.

При SEQUENCE для получения следующего значения из последовательности базы данных требуются дополнительные операторы SELECT, но это не влияет на производительность для большинства приложений. И если приложению необходимо сохранить огромное количество новых сущностей можно использовать некоторые специфичные для Hibernate оптимизации, чтобы уменьшить количество операторов.

Для работы с этой стратегией Hibernate использует свой класс **SequenceStyleGenerator**.

[SEQUENCE – тип генерации, рекомендуемый документацией Hibernate](#) и самый простой способ задать безымянную генерацию последовательности:

```
@Entity(name = "Product")
public static class Product {
    @Id
    @GeneratedValue(
        strategy = GenerationType.SEQUENCE // Имя последовательности не
        // определено, поэтому Hibernate будет использовать последовательность
        // hibernate_sequence для всех сущностей
    )
    private Long id;

    @Column(name = "product_name")
    private String name;

    //Getters and setters are omitted for brevity
}
```

Для всех сущностей с безымянной последовательностью Hibernate будет использовать одну и ту же **hibernate_sequence**, из которой будет брать для них id -шник.

Используя аннотацию **@SequenceGenerator**, можно указать конкретное имя последовательности для таблицы, а также иные параметры. Также можно настроить под себя несколько разных последовательностей (SEQUENCE-генераторов), указав, например, имя последовательности и начальное значение (код справа)

```
@Entity
public class User {
    @Id
    @GeneratedValue(generator = "sequence-generator")
    @GenericGenerator(
        name = "sequence-generator",
        strategy = "org.hibernate.id.enhanced.SequenceStyleGenerator",
        parameters = {
            @Parameter(name = "sequence_name", value = "user_sequence"),
            @Parameter(name = "initial_value", value = "4"),
            @Parameter(name = "increment_size", value = "1")
        }
    )
    private long userId;
    // ...
}
```

В этом примере установили имя последовательности и начальное значение, поэтому генерация первичного ключа начнется с 4. Для каждой последовательности генерированные значения являются уникальными.

Можно назначать разные последовательности разным сущностям, и они будут брать id -шники из этой последовательности.

В зависимости от требований приложения, можно иметь один генератор на всё приложение, по генератору на каждую сущность или несколько генераторов, которыми пользуются несколько сущностей.

Например, есть 10 сущностей, для трех из них создадим последовательность с именем first_sequence, из которой они будут брать id -шники. Для пяти других сущностей создадим последовательность с именем second_sequence, из которой они будут брать свои id -шники. А для оставшихся двух сущностей можем задать безымянную последовательность, и в этом случае id -шники для них будут браться по умолчанию из hibernate_sequence.

TABLE

В настоящее время GenerationType.TABLE используется редко.

Hibernate должен получать первичные ключи для сущностей из специальной создаваемой для этих целей таблицы, способной содержать несколько именованных сегментов значений для любого количества сущностей.

Основная идея заключается в том, что подобная таблица (например, `hibernate_sequence`) может содержать несколько сегментов со значениями идентификаторов для разных сущностей. Это требует использования пессимистических блокировок, которые помещают все транзакции по получению идентификаторов в очередь.

Это замедляет работу приложения. Третья стратегия, `GenerationType.TABLE`, не зависит от поддержки конкретной базой данных и хранит счётчики значений в отдельной таблице.

С одной стороны это более гибкое и настраиваемое решение, с другой стороны более медленное и требующее большей настройки.

Вначале требуется создать (вручную!) и проинициализировать (!) таблицу для значений ключей. Затем создать генератор и связать его со идентификатором, используя аннотацию

`@TableGenerator` можно настроить этот тип генерации →

```
@Entity
public class Department {
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE,
        generator = "table-generator")
    @TableGenerator(name = "table-generator",
        table = "dep_ids",
        pkColumnName = "seq_id",
        valueColumnName = "seq_value")
    private long depId;

    // ...
}
```

37. Расскажите про аннотации `@JoinColumn` и `@JoinTable`? Где и для чего они используются?

Аннотация `@JoinColumn` используется для указания столбца FOREIGN KEY, используемого при установлении связей между сущностями или коллекциями.

Только сущность-владелец связи может иметь внешние ключи от другой сущности (ведомой). Однако, можно указать аннотацию `@JoinColumn` как во владеющей таблице, так и в ведомой, но столбец с внешними ключами всё равно появится во владеющей таблице.

Особенности использования:

`@OneToOne` означает, что появится столбец `addressId` в таблице сущности-владельца связи **Office**, который будет содержать внешний ключ, ссылающийся на первичный ключ ведомой сущности **Address**.

```
@Entity
public class Office {
    @OneToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "addressId")
    private Address address;
}
```

`@OneToMany` или `@ManyToOne`: можно использовать атрибут `mappedBy` для того, чтобы столбец с внешними ключами находился на владеющей стороне `ManyToOne` (в таблице `Email`):

```
@Entity
public class Employee {
    @Id
    private Long id;

    @OneToMany(fetch = FetchType.LAZY, mappedBy = "employee")
    private List emails;
}
```

```
✓ @Entity
public class Email {
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "employee_id")
    private Employee employee;
}
```

В этом примере таблица `Email` (владелец связи) имеет столбец `employee_id`, в котором хранится значение идентификатора и внешний ключ к таблице `Employee`. Если не указать `mappedBy`, то будет создана сводная (третья) таблица с первичными ключами из двух основных таблиц.

@JoinColumns (множественное число)

Аннотация `@JoinColumns` используется для группировки нескольких аннотаций `@JoinColumn`, которые используются при установлении связей между сущностями или коллекциями, у которых составной первичный ключ и требуется несколько колонок для указания внешнего ключа.

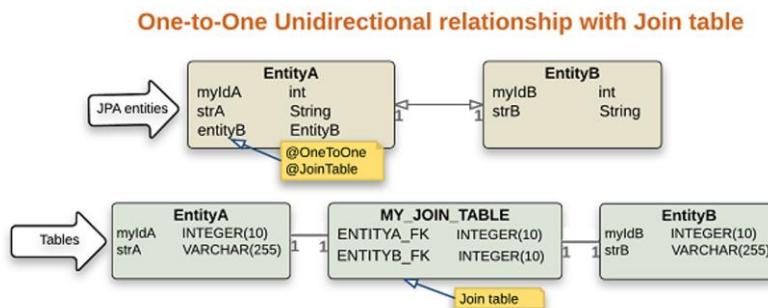
В каждой аннотации `@JoinColumn` должны быть указаны элементы `name` и `referencedColumnName`:

```
@ManyToOne
@JoinColumns({
    @JoinColumn(name="ADDR_ID", referencedColumnName="ID"),
    @JoinColumn(name="ADDR_ZIP", referencedColumnName="ZIP")
})

public Address getAddress() {
    return address;
}
```

@JoinTable

Аннотация `@JoinTable` используется для указания связывающей (сводной, третьей) таблицы между двумя другими таблицами.



Хороший пример: задача на ПП Rest Security (слева поле в классе User, справа data.sql)

```
@ManyToMany(fetch = FetchType.LAZY)
@JoinTable(name = "users_roles",
    joinColumns = @JoinColumn(name = "user_id"),
    inverseJoinColumns = @JoinColumn(name = "role_id"))
private Collection<Role> roles;
```

```
insert into rest_db.users_roles (user_id, role_id)
values  (1, 1),
        (1, 2),
        (2, 1),
        (3, 1),
```

38. Для чего нужны аннотации `@OrderBy` и `@OrderColumn`, чем они отличаются?

<https://www.logicbig.com/tutorials/java-ee-tutorial/jpa/order-by-annotation.html>

Эти аннотации служат для установки порядка выдачи элементов коллекций Entity.

Аннотация `@OrderBy` указывает порядок, в соответствии с которым должны располагаться элементы коллекций сущностей, базовых или встраиваемых типов при их извлечении из БД. Эта аннотация может использоваться с аннотациями `@ElementCollection`, `@OneToMany`, `@ManyToMany`.

Если коллекция содержит элементы базового типа, то упорядочивание будет по значению базовых объектов. Например, номера телефонов будут упорядочены в их естественном порядке →

```
@ElementCollection
@OrderBy
private List<String> phoneNumbers;
```

Если коллекция относится к типу **@Embeddable**, то нотация с точкой (".") используется для ссылки на атрибут внутри встроенного атрибута. Например, тут адреса будут упорядочены по названиям стран →

```
@ElementCollection  
@OrderBy("city.country DESC")  
private List<Address> addresses;
```

Где адрес определяется как:

```
@Embeddable  
public class Address {  
    ...  
    @Embedded  
    private City city  
    ...  
}
```

Можно использовать дополнительно **ASC** или **DESC**, чтобы указать, является ли порядок восходящим или нисходящим.
По умолчанию — **ASC (восходящий)**.

Когда @OrderBy используется с отношением

@OrderBy работает только с прямыми свойствами, если используется с отношением @OneToMany или @ManyToMany. Например, так (рис. 1) для сущности Task (рис. 2).

Рис. 1

```
@ManyToMany  
@OrderBy("supervisor")  
private List<Task> tasks;
```

Рис. 2

```
@Entity  
public class Task {  
    ....  
    @OneToOne  
    private Employee supervisor;  
    ...  
}
```

Рис. 3

```
@ManyToMany  
@OrderBy  
private List<Task> tasks;
```

Доступ через точку ("") не работает в случае отношений. Попытка использовать вложенное свойство, например. `@OrderBy("supervisor.name")` приведет к исключению во время выполнения.

Если элемент упорядочивания не указан для ассоциации объектов (т. е. аннотация используется без какого-либо значения), предполагается упорядочивание по первичному ключу связанного объекта (рис. 3). В приведенном случае сбор задач будет упорядочен по номеру задачи (`id`-шник).

39. Для чего нужна аннотация Transient?

Вспоминаем тему Сериализация =)

Иногда бывает так, что в объекте есть поля, которые не нужно сохранять.

Это может быть какой-то кэш или другие данные, которые легко вычисляются по тому, что есть в объекте.

Их потеря ничем не грозит, зато позволит сократить размер памяти, занимаемой объектом при сериализации.

Отметить поля **ключевым словом Transient** и JVM будет их игнорировать при записи.

Свойства класса, помеченные модификатором **transient**, не сериализуются.

Ключевое слово `transient` говорит о том, что **объект нельзя сериализовать** или де-сериализовать.

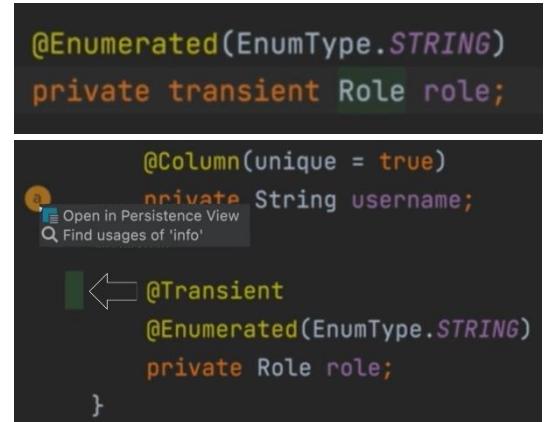
Если добавить к полю модификатор transient, то после восстановления его значение будет null.

У Hibernate есть JPA аннотация **@transient**.
Она исключает поля и свойства Entity из маппинга (property or field is not persistent).

Добавили аннотацию, в идеале пропал жёлтый кружок слева, значит поле не будет мапиться.

На практике не стоит использовать, т.к. в Entity лучше хранить те поля, которые мапятся на соответствующие поля в таблице базы данных.

40. Какие шесть видов блокировок (lock) описаны в спецификации JPA (или какие есть значения у enum LockModeType в JPA)? <https://youtu.be/C-wEZjEOhWc?t=1952>



КОРОТКИЙ ОТВЕТ

У JPA есть шесть видов блокировок, **повышающих производительность**.

Простыми словами:

Суть оптимистичной блокировки: к одному ресурсу вряд ли будут конкурирующие обращения и поэтому транзакции НЕ блокируют друг друга. Ставим тогда, когда к одним и тем же данным редко одновременно обращаются разные транзакции. ОПТИМИСТ. И версионность тут нужна, чтобы понимать, были ли изменения и если «да», то будет выброшено исключение. Дополнительный контроль.

Суть пессимистичной блокировки: к одним и тем же данным возможны частые запросы и поэтому используется **блокирующий подход**. ПЕССИМИСТ.

Перечислим их в порядке увеличения надежности (от самого ненадежного и быстрого, до самого надежного и медленного):

NONE — без блокировки (по умолчанию)

OPTIMISTIC (устаревший READ, оставшийся от JPA 1) — оптимистическая блокировка.

OPTIMISTIC_FORCE_INCREMENT (устаревший WRITE, оставшийся от JPA 1) —

оптимистическая блокировка с принудительным увеличением поля версионности.

PESSIMISTIC_READ — пессимистичная блокировка на чтение.

PESSIMISTIC_WRITE — пессимистичная блокировка на запись (и чтение).

PESSIMISTIC_FORCE_INCREMENT — пессимистичная блокировка на запись (и чтение) с принудительным увеличением поля версионности.

ПОДРОБНЕЕ (по урокам Матвиенко)

Вспоминаем тему «транзакции и блокировки» из SQL модуля =)

Решение конфликтов транзакций на уровне Java кода (оптимистические блокировки).

Оптимистичный подход предполагает, что **параллельно выполняющиеся транзакции редко обращаются к одним и тем же данным** и позволяет им спокойно и свободно выполнять любые чтения и обновления данных.

Но при окончании транзакции производится проверка, изменились ли данные в ходе выполнения данной транзакции и, если да, транзакция обрывается и выбрасывается исключение. Оптимистичное блокирование в JPA **реализовано** путём **внедрения в сущность специального поля версии**.

Поле, аннотирование @Version, может быть целочисленным или временным.

При завершении транзакции, если сущность была оптимистично заблокирована, будет проверено, не изменилось ли значение @Version кем-либо ещё, после того как данные были прочитаны, и, если изменилось, будет выкинуто **OptimisticLockException**.

Использование этого поля позволяет отказаться от блокировок на уровне базы данных и сделать всё на уровне JPA, улучшая уровень конкурентности.

В КОДЕ: Обе блокировки ставятся путём вызова метода lock() у EntityManager, в который передаётся сущность, требующая блокировки и уровень блокировки:

```
EntityManager em = entityManagerFactory.createEntityManager();
em.lock(company1, LockModeType.OPTIMISTIC);
em.lock(company2, LockModeType.OPTIMISTIC_FORCE_INCREMENT);
```

ЗА КУЛИСАМИ (для тех, кто хочет разобраться, НО на собеседовании не углубляемся, а то закопают)):

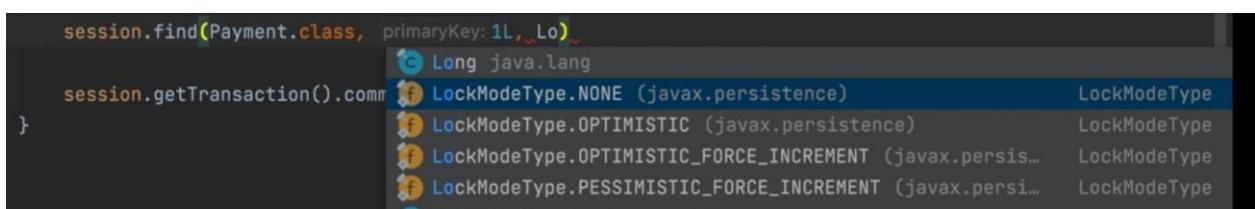
Теоретически в проперти файле в Hibernate можно устанавливать уровень изолированности транзакций, но **на практике не стоит повышать уровень**, который используется по умолчанию в СУБД проекта (по умолчанию уровень 2: read committed).

```
<property name="hibernate.connection.isolation">8</property>-->
```

Есть лучший вариант решения проблемы параллельного выполнения транзакций, а именно **оптимистические и пессимистические блокировки**.

Первые решаются с помощью **Java кода** (выполнение необходимых SQL запросов, решающих конфликты транзакций на уровне приложения), а вторые с помощью **СУБД** (более жёсткий вариант).

В любом случае это будет лучше, чем самый высокий уровень изолированности (8 в проперти, см. скрин выше), а именно Serializable т.к. он выполняет транзакции практически последовательно, «убивая» производительность. Но выход есть!



```
session.find(Payment.class, primaryKey: 1L, Lo)
    c Long java.lang
    session.getTransaction().comm f LockModeType.NONE (javax.persistence) LockModeType
}
f LockModeType.OPTIMISTIC (javax.persistence) LockModeType
f LockModeType.OPTIMISTIC_FORCE_INCREMENT (javax.persistence) LockModeType
f LockModeType.PESSIMISTIC_FORCE_INCREMENT (javax.persistence) LockModeType
```

Since: Java Persistence 1.0

```
public enum LockModeType {
    READ,
    OPTIMISTIC,
    PESSIMISTIC_READ,
    PESSIMISTIC_WRITE,
    NONE
}
```

Synonymous with OPTIMISTIC. ←
OPTIMISTIC is to be preferred for new applications.

Synonymous with OPTIMISTIC_FORCE_INCREMENT. ←
OPTIMISTIC_FORCE_INCREMENT is to be preferred for new applications.

Optimistic lock.
Since: Java Persistence 2.0

Optimistic lock, with version update.
Since: Java Persistence 2.0

LockModeType – это Enum.
Представлены два вида блокировок: оптимистические и пессимистические.

Pessimistic read lock.
Since: Java Persistence 2.0

PESSIMISTIC_READ,

Pessimistic write lock.
Since: Java Persistence 2.0

PESSIMISTIC_WRITE,

Pessimistic write lock, with version update.
Since: Java Persistence 2.0

PESSIMISTIC_FORCE_INCREMENT,

No lock.
Since: Java Persistence 2.0

NONE

Сначала в классе Entity добавляем аннотацию @OptimisticLocking и @Version

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
@Entity
@OptimisticLocking(type = OptimisticLockType.VERSION)
public class Payment implements BaseEntity<Long> {

    @Entity
    @OptimisticLocking(type = OptimisticLockType.VERSION)
    public class Payment implements BaseEntity<Long> {
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Long id;
    }
}
```

OptimisticLockType
OptimisticLockType
OptimisticLockType
OptimisticLockType

If VERSION, then it is necessary to specify a field that will indicate the version of the row in the table. That is, each row in the table will have a column in which the current version of the row will be specified and when the row is changed, the version will change.

If we use VERSION, then it is necessary to specify a field that will indicate the version of the row in the table. That is, each row in the table will have a column in which the current version of the row will be specified and when the row is changed, the version will change.

Now if changes occur inside a transaction (increase the payment amount by 10), then Hibernate will add not only WHERE, but also VERSION.

```
session.beginTransaction();

var payment : Payment = session.find(Payment.class, primaryKey: 1L, LockModeType.OPTIMISTIC);
payment.setAmount(payment.getAmount() + 10);

session.getTransaction().commit();
```

Hibernate:
update
payment
set
amount=?,
receiver_id=?,
version=?
where
id=?
and version=?

That is, it will be set the previous version (zero) + increment (0+1=1).
If the version was zero, then the same number, then CRUD operations => changes were not made.

А если номер версии не совпал (операции в строчке с id -шником были), то будет выброшено исключение **OptimisticLockException**. Можно его ловить в try-catch и логгировать: «номер версии забронирован».

Прежде чем транзакция выполнит обновление, она снова проверяет свойство версии. Если за это время значение изменилось, создается исключение OptimisticLockException. В противном случае транзакция фиксирует обновление и увеличивает значение свойства версии.

Таким образом решается проблема Last Commit Wins, получаем First Commit Wins. В БД увидим изменённое число в колонке version.

	id	amount	version	receiver_id
13	14	300	0	5
14	1	110	1	1

Так работает оптимистическая блокировка. По умолчанию имеем OPTIMISTIC и номер версии изменится только тогда, когда будет выполнена одна из операций.

А вариант FORCE_INCREMENT в любом случае инкрементирует версию.

```
var payment : Payment = session.find(Payment.class, primaryKey: 1L, LockModeType.OPTIMISTIC_FORCE_INCREMENT);
payment.setAmount(payment.getAmount() + 10);
```

Стратегии ALL и DIRTY (обе эти стратегии не используют поле version).

```
@Entity
@OptimisticLocking(type = OptimisticLockType.ALL)
@DynamicInsert org.hibernate.annotations
@DynamicUpdate org.hibernate.annotations
```

Суть ALL в том, что в WHERE нужно указать ВСЕ поля, которые есть в классе. Операция update теперь **ДИНАМИЧЕСКАЯ** по условию (добавляем соответствующую аннотацию). Под капотом Hibernate парсит аннотации @Table, @Column и т.д. и создаёт CRUD операции для каждой из сущностей. Чтобы каждый раз это не делать, проще один раз распарсить и закэшировать (т.к. они одни и те же данные). А дальше динамически отслеживать условия.

Какая разница между ALL и DIRTY?

DIRTY смотрит лишь «грязный контекст» и в WHERE добавляет только те поля, которые изменились. На практике DIRTY не рекомендуют использовать (есть подводные камни?).

Лучше использовать OPTIMISTIC с версионностью!

Решение конфликтов транзакций на уровне СУБД (пессимистические блокировки).

Пессимистичный подход напротив, ориентирован на транзакции, которые постоянно или достаточно часто КОНКУРИРУЮТ за одни и те же данные и поэтому блокирует доступ к данным превентивно, в тот момент, когда читает их.

Другие транзакции останавливаются, когда пытаются обратиться к заблокированным данным и ждут снятия блокировки (или кидают исключение).

Пессимистичное блокирование **выполняется на уровне базы** и поэтому не требует вмешательств в код сущности. Так же, как и в случае с оптимистичным блокированием, поддерживаются блокировки чтения и записи.

В КОДЕ: накладываются пессимистичные блокировки так же как и оптимистичные, вызовом метода **lock()**, а снимаются они тоже автоматически, по завершению транзакции.

```
em.lock(company1, LockModeType.PESSIMISTIC_READ);
em.lock(company2, LockModeType.PESSIMISTIC_WRITE);
em.lock(company3, LockModeType.PESSIMISTIC_FORCE_INCREMENT);
```

ЗА КУЛИСАМИ

Не поднимая уровень изолированности транзакций в проперти (не ухудшая производительность), решаем проблему.

```
var payment : Payment = session.find(Payment.class, primaryKey: 1L, LockModeType.PESSIMISTIC_READ);
payment.setAmount(payment.getAmount() + 10);
```

READ: берём запись, изменяем её и пока идёт транзакция, берём ту же самую запись в другой транзакции и изменяем её.

```
session1.getTransaction().commit();
session.getTransaction().commit();
```

Приложение подвиснет, т.к. первая сессия будет ждать пока блокировка не снимется другой транзакцией. И так будет ждать, пока не случится timeout.

В консоли можно увидеть **for share**. Блокируются только те строки, которые Hibernate выбрал в SELECT (обращайтесь в документацию той СУБД, которая используется у вас).

```
Hibernate:
select
    payment0_.id as id1_4_0_,
    payment0_.amount as amount2_4_0_,
    payment0_.receiver_id as receiver3_4_0_
from
    payment payment0_
where
    payment0_.id=? for share!
```

Документация Postgres

13.3. Explicit Locking

13.3.1. Table-Level Locks

блок таблицы

13.3.2. Row-Level Locks

блок строки

13.3.3. Page-Level Locks

13.3.4. Deadlocks

13.3.5. Advisory Locks

PESSIMISTIC_WRITE: если изменим тип на WRITE, то **в консоли увидим for update** (а не for share). Он еще более строгий, чем for share.

PESSIMISTIC_FORCE_INCREMENT – это обычная пессимистическая блокировка, но дополнительно нужна еще и версионность (а форс инкремент, значит в любом случае версия изменяется).

```
Hibernate:
    select
        payment0_.id as id1_4_0_,
        payment0_.amount as amount2_4_0_,
        payment0_.receiver_id as receiver4_4_0_,
        payment0_.version as version3_4_0_
    from
        payment payment0_
    where
        payment0_.id=? for update
        nowait
```

```
Hibernate:
    update
        payment
    set
        version=?
    where
        id=?
        and version=?
```

При обновлении добавлена версионность

Чтобы решить проблему с зависаниями нужно выставлять `timeout`-ы.
Hint указывает, на сколько мс будет задержана блокировка.

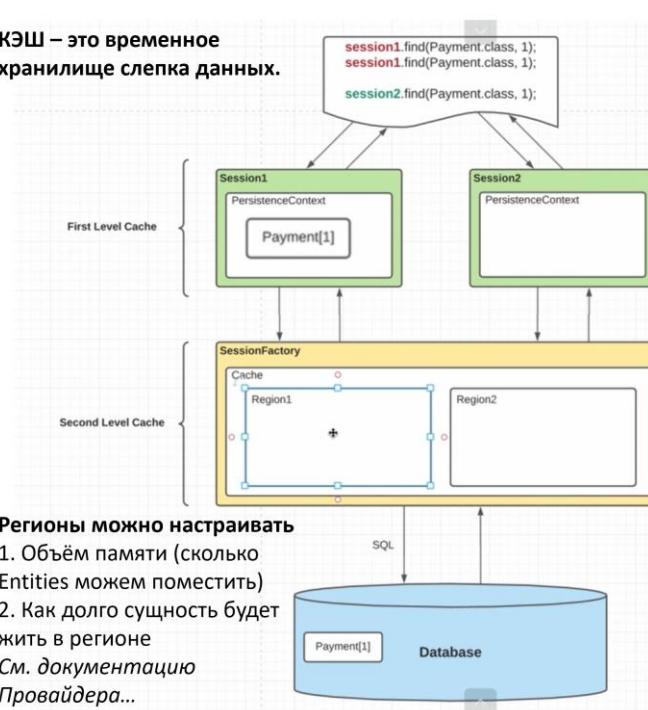
Это нужно, чтобы не останавливать транзакцию для всего приложения в целом.

```
session.createQuery(queryString: "select p from Payment p", Payment.class)
    .setLockMode(LockModeType.PESSIMISTIC_FORCE_INCREMENT)
    .setHint(hintName: "javax.persistence.lock.timeout", value: 5000)
    .list();
```

Такие hints можно найти в классе:
`org.hibernate.cfg.AvailableSettings`

41. Какие два вида кэшей (cache) вы знаете в JPA и для чего они нужны?

КЭШ – это временное хранилище слепка данных.



КЭШИРОВАНИЕ

Пример: есть две сессии (зелёные) и они пытаются получить payment по ID. Т.к. сессии разные, то и контекст у них разный.

Первая уже получила объект и сохранила в свой кэш, поместила в контекст.

Вторая сессия ничего в своём кэше не видит и поэтому получает, а потом кэширует объект из БД.

Это 1й уровень кэша.....

2й уровень по умолчанию ВЫКЛЮЧЕН (жёлтый).

Кэш разбит на регионы (R1, R2...), это аналог Persistent Context –a.

Там будет Map (по КЛЮЧУ поле id будет лежать ЗНАЧЕНИЕ – это Entity).

В случае кэша Entity будет не в виде объекта, а в виде сериализованного значения. При получении из кэша происходит сериализация и десериализация.

JPA говорит о двух видах кэшей (cache):

- first-level cache (кэш первого уровня) — кэширует данные одной транзакции
- second-level cache (кэш второго уровня) — кэширует данные дольше чем одна транзакция.

Провайдер JPA может, но не обязан реализовывать работу с кэшем второго уровня. Такой вид кэша позволяет сэкономить время доступа и **улучшить производительность**, однако оборотной стороной является возможность получить устаревшие данные.

42. Как работать с кэшем 2 уровня?

Каким способом можно в коде работать с кэшем второго уровня (удалять все или определенные Entity из кэша, узнать закэшировалась ли сущность Entity и т.п.)?

Для работы с кэшем второго уровня (second level cache) в JPA описан **Cache интерфейс**, содержащий большое количество методов по управлению кэшем второго уровня.

Если он поддерживается провайдером JPA, конечно. Объект данного интерфейса можно получить с помощью **метода getCache()** у **EntityManagerFactory**.

JPA сообщает о пяти значениях shared-cache-mode из persistence.xml, который определяет как будет использоваться second-level cache:

- **ALL** — все Entity могут кэшироваться в кеше второго уровня
- **NONE** — кеширование отключено для всех Entity
- **ENABLE_SELECTIVE** — кеширование работает только для тех Entity, у которых установлена аннотация Cacheable(true) или её xml эквивалент, для всех остальных кеширование отключено
- **DISABLE_SELECTIVE** — кеширование работает для всех Entity, за исключением тех у которых установлена аннотация Cacheable(false) или её xml эквивалент
- **UNSPECIFIED** — кеширование не определено, каждый провайдер JPA использует свою значение по умолчанию для кэширования.

43. Что такое JPQL/HQL и чем он отличается от SQL?

JPQL (Java Persistence query language) это язык запросов, практически такой же как SQL, однако вместо имен и колонок таблиц базы данных, он **использует имена классов Entity и их атрибуты**. В качестве параметров запросов так же используются типы данных атрибутов Entity, а не полей баз данных.

В отличии от SQL в JPQL есть автоматический полиморфизм (см. следующий вопрос). Также в JPQL используется функции, которых нет в SQL: такие как KEY (ключ Мар'ы), VALUE (значение Мар'ы), TREAT (для приведения суперкласса к его объекту-наследнику, downcasting), ENTRY и т.п.

*Что означает полиморфизм (polymorphism) в запросах JPQL (Java Persistence query language) и как его «выключить»?

В отличии от SQL в запросах JPQL есть автоматический полиморфизм, то есть каждый запрос к Entity возвращает не только объекты этого Entity, но также объекты всех его классов-потомков, независимо от стратегии наследования (например, запрос select * from Animal, вернет не только объекты Animal, но и объекты классов Cat и Dog, которые унаследованы от Animal).

Чтобы исключить такое поведение используется функция TYPE в where условии (например `select * from Animal a where TYPE(a) IN (Animal, Cat)` уже не вернет объекты класса Dog).

44. Что такое Criteria API и для чего он используется?

Criteria API это тоже **язык запросов**, аналогичным JPQL (Java Persistence query language), однако запросы основаны на методах и объектах, то есть запросы выглядят так:

```

1 CriteriaBuilder cb = ...
2 CriteriaQuery<Customer> q = cb.createQuery(Customer.class);
3 Root<Customer> customer = q.from(Customer.class);
4 q.select(customer);

```

```

CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<PersonCase1> q = cb.createQuery(PersonCase1.class);
Root<PersonCase1> c = q.from(PersonCase1.class);
q.select(c);
q.orderBy(cb.asc(c.get("name")), cb.desc(c.get("surname")));

```

Преимущество использования языка Criteria – это динамические запросы, написанные на Java. Это, пожалуй, единственное преимущество. В остальном трудно читаемый код. Чаще используется HQL, JPQL – интуитивно понятный (т.к. похож на SQL).

*В чем разница в требованиях к Entity в Hibernate, от требований к Entity, указанных в спецификации JPA?

1) Конструктор без аргументов в Hibernate НЕ обязан быть public или protected, рекомендуется чтобы он был хотя бы package видимости, однако это только рекомендация, если настройки безопасности Java позволяют доступ к приватным полям, то он может быть приватным.

2) JPA категорически требует не использовать **final классы**, Hibernate лишь рекомендует не использовать такие классы чтобы он мог создавать прокси для ленивой загрузки, однако позволяет либо выключить прокси Proxy(lazy=false), либо использовать в качестве прокси интерфейс, содержащий все методы маппинга для данного класса (аннотацией Proxy(proxyClass = интерфейс.class)).

45. Расскажите про проблему N+1 Select и путях ее решения.

КОРОТКО:

Проблема более не актуальна, она решена самим Hibernate (выше 6 версии =) шутка

ПОДРОБНЕЕ:

Вместо того, чтобы получить ОДИН запрос, получаем в нагрузку N количество.

Одним запросом достали всех пользователей и потом на каждого инициализируется какая-либо связь сущностей.

Проблема N+1 связана с производительностью. Если делаем множественный запрос (пользователь, платёж, компания), то Hibernate делает дополнительный SELECT запрос.

```

public static void main(String[] args) throws SQLException {
    try (SessionFactory sessionFactory = HibernateUtil.buildSessionFactory();
        Session session = sessionFactory.openSession()) {
        session.beginTransaction();

        var user : User = session.get(User.class, 1L);
        System.out.println(user.getPayments().size());
        System.out.println(user.getCompany().getName());

        session.getTransaction().commit();
    }
}

```

```

Hibernate:
select
    payments0_.receiver_id as receiver3_4_0_,
    payments0_.id as id1_4_0_,
    payments0_.id as id1_4_1_,
    payments0_.amount as amount2_4_1_,
    payments0_.receiver_id as receiver3_4_1_
from
    payment payments0_
where
    payments0_.receiver_id=?
```

3

```

Hibernate:
select
    company0_.id as id1_2_0_,
    company0_.name as name2_2_0_
from
    company company0_
where
    company0_.id=?
```

Microsoft

Правило: использовать везде, где можно, ленивую загрузку.

Например, если поставить на трёх классах EAGER загрузку, то выскочит **MultipleBagFetchException**, т.к. нельзя одним запросом получать сразу несколько ассоциаций, если это коллекция (Bag – прокси для коллекций), которая не упорядочена, т.е. не поддерживает сортировку. Поэтому Hibernate не может их сmapить (не понимает, какие строки относятся к каким таблицам).

PersistentSet (как и любой другой Set) - не зависит от порядка.
Главное правильно переопределить `equals` & `hashCode`

Если поставим Set (нет порядка), то Hibernate выполнит Left Outer Join и получим декартово произведение записей на три сущности).

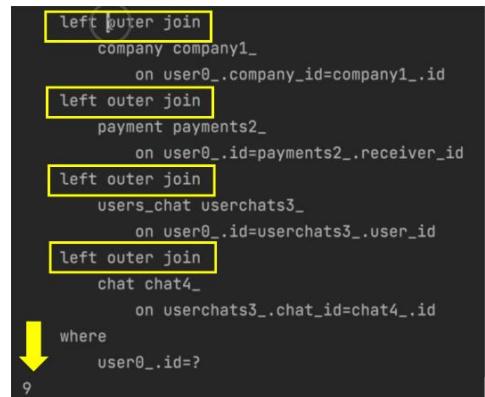
Получаем кучу дублирующей информации, которая не нужна и никак не отсортирована.

Кроме того, нет возможности использовать `LIMIT`, `OFFSET` и агрегирующие функции.

```

left outer join
    company company1_
        on user0_.company_id=company1_.id
left outer join
    payment payments2_
        on user0_.id=payments2_.receiver_id
left outer join
    users_chat userchats3_
        on user0_.id=userchats3_.user_id
left outer join
    chat chat4_
        on userchats3_.chat_id=chat4_.id
where
user0_.id=?
9

```



Пути решения проблемы N+1

<https://www.youtube.com/watch?v=XH9KMY4jMSQ>

Сводная таблица решений N+1 проблемы и кейсов по ней на OneToMany

solution	UNIdirectional OneToMany		UNIdirectional ManyToOne		Bidirectional ManyToOne, OneToMany	
	JPQL	native	JPQL	native	JPQL	native
join fetch	+	—	+	—	+	—
FetchMode SUBSELECT	+	—	—	—	+ / — **	—
BatchSize	+	+	—	—	+ / — **	+ / — **
Entity Graph	+	—	+	—	+	—
SqlResultSetMapping	—	—	—	+	—	— / + ***
HibernateSpecificMapping	—	— *	—	+	—	+

* если не используем аннотацию `@JoinColumn` и оставляем связанную третью таблицу

** работает только при выборке владельца связи с коллекцией зависимых сущностей

*** работает только при выборке зависимой сущности со ссылкой на Entity владельца связи

ВЫВОДЫ:

- Лучшим вариантом решения N+1 проблемы для простых запросов (1-3 уровня вложенности связанных объектов) будет **join fetch** и **JPQL** запрос. Следует придерживаться тактики, когда выбираем из JPQL и наивного запроса JPQL
- Если имеется нативный запрос и мы не заботимся о слабой связанныности кода, то хорошим вариантом будет использование **Hibernate Specific Mapping**. В противном случае стоит использовать **@SqlResultSetMapping**
- В случаях, когда нужно получить по-настоящему много данных и используется JPQL запрос, лучше всего использовать **Entity Graph**
- Если мы знаем примерное кол-во коллекций, которые будут использоваться в любом месте приложения, то можно использовать **@BatchSize**

Сначала бизнес-логика, потом оптимизация (DMDEV)

1. Avoid `@OneOnOne bidirectional` ❌
2. Use fetch type `LAZY` everywhere ✅
3. Don't prefer `@BatchSize`, `@Fetch` ❌
4. Use query `fetch` (HQL, Criteria API, QueryDSL) ✅
5. Prefer `EntityGraph API` than `@FetchProfile` ✅

избегайте отношение `OneToOne, Bidirectional`
 используйте `LAZY` ленивую загрузку везде, где возможно
 избегайте аннотаций `@BatchSize, @Fetch`
 используйте `Query Fetch` запрос (HQL, Criteria)
 предпочитайте `EntityGraph API` (вместо `@FetchProfile`)

JOIN FETCH

И при `FetchType.EAGER` и при `FetchType.LAZY` поможет **JPQL-запрос с JOIN FETCH**. Опцию «**FETCH**» можно использовать в `JOIN` (`INNER JOIN` или `LEFT JOIN`) для выборки связанных объектов в одном запросе вместо дополнительных запросов для каждого доступа к ленивым полям объекта.

```
var users : List<User> = session.createQuery(
    queryString: "select u from User u join fetch u.payments where 1 = 1", User.class)
    .list();
users.forEach(user -> System.out.println(user.getPayments().size()));
users.forEach(user -> System.out.println(user.getCompany().getName()));

session.getTransaction().commit();
```

```
List<PostComment> comments = entityManager.createQuery("""
    select pc
    from PostComment pc
    join fetch pc.post p
    """, PostComment.class)
    .getResultList();

for(PostComment comment : comments) {
    LOGGER.info(
        "The Post '{}' got this review '{}'",
        comment.getPost().getTitle(),
        comment.getReview()
    );
}
```

На этот раз Hibernate выполнит одну инструкцию SQL:

```
SELECT
    pc.id as id1_1_0_,
    pc.post_id as post_id3_1_0_,
    pc.review as review2_1_0_,
    p.id as id1_0_1_,
    p.title as title2_0_1_
FROM
    post_comment pc
    INNER JOIN
        post p ON pc.post_id = p.id
```

Использование `LEFT JOIN FETCH` аналогично `JOIN FETCH`, только будут загружены все сущности из таблицы `PostComment`, даже те, у которых нет связанной сущности `Post` (в нашем случае пример не логичный, но понятный).

EntityGraph

В случаях, когда нужно получить по-настоящему много данных, и используется JPQL запрос, то лучше всего использовать `EntityGraph`.

```
var userGraph : RootGraph<User> = session.createEntityGraph(User.class);
userGraph.addAttributeNodes(...names: "company", "userChats");
var userChatsSubgraph : SubGraph<UserChat> = userGraph.addSubgraph(name: "userChats", UserChat.class);
userChatsSubgraph.addAttributeNodes(...names: "chat");
```

@Fetch(FetchMode.SUBSELECT)

Это Аннотация Hibernate, в JPA её нет. Можно использовать **только с коллекциями**.

Будет сделан один SQL запрос для получения корневых сущностей и, если в контексте персистентности будет обращение к ленивым полям-коллекциям, то выполнится еще один запрос для получения связанных коллекций:

```
@Fetch(FetchMode.SUBSELECT)
@OneToMany(mappedBy = "receiver")
private List<Payment> payments = new ArrayList<>();
```

Available for collections only. When accessing a non-initialized collection, this fetch mode will trigger loading all elements of all collections of the same role for all owners associated with the persistence context using a single secondary select.
SUBSELECT

HibernateSpecificMapping, SqlResultSetMapping

Для нативных запросов рекомендуется использовать именно их.

Batch Fetching

Это аннотация Hibernate, в JPA её нет.

Указывается над классом сущности или над полем коллекции с ленивой загрузкой.

```
@OneToMany(mappedBy = "customer")
@Fetch(value = FetchMode.SELECT)
@BatchSize(size=5)
private Set<Order> orders = new HashSet<>();
```

Будет сделан один sql-запрос для получения корневых сущностей и, если в контексте персистентности будет обращение к ленивым полям-коллекциям, то выполнится еще один запрос для получения связанных коллекций.

Например, в персистентный контекст загружено 12 сущностей Customer, у которых по одному полю-коллекции orders, но так как это @OneToMany, то у них ленивая загрузка по умолчанию и они не загружены в контекст персистентности из БД.

При первом обращении к какому-нибудь полю orders, нам бы хотелось, чтобы для всех 12 сущностей Customer были загружены их 12 коллекций Order, по одной для каждой.

Но так как @BatchSize(size=5), то Hibernate сделает 3 запроса: в первом и втором получит по пять коллекций, а в третьем получит две коллекции. Если знать примерное количество коллекций, которые будут использоваться в любом месте приложения, то можно использовать @BatchSize и указать нужное количество.

Также аннотация @BatchSize может быть указана у класса.

Рассмотрим пример, где есть сущность Order, у которой есть поле типа Product (не коллекция). Выгрузили в контекст персистентности 27 объектов Order. При обращении к полям Product у объектов Order будет инициализировано до 10 ленивых прокси сущностей Product одновременно:

```
@Entity
class Order {
    @OneToOne(fetch = FetchType.LAZY)
    private Product product;
    ...
}

@Entity
@BatchSize(size=10)
class Product {
    ...
}
```

Хотя использовать @BatchSize лучше, чем столкнуться с проблемой запроса N+1, в большинстве случаев гораздо лучшей альтернативой является использование DTO или JOIN FETCH, поскольку они позволяют получать необходимые данные одним запросом.

46. Что такое EntityGraph? Как и для чего их использовать? <https://youtu.be/b2a4rVR5hiQ?t=311>

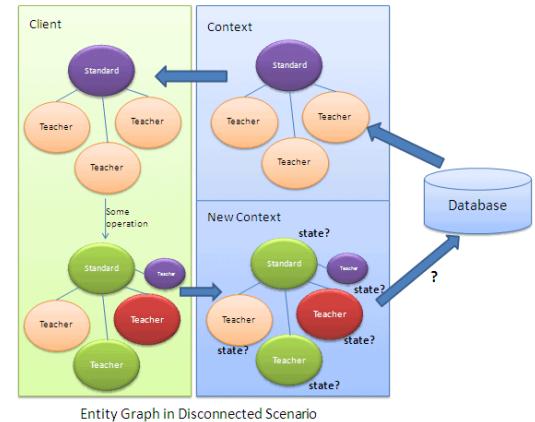
<https://www.baeldung.com/jpa-entity-graph>

Справка: Граф – нелинейная структура, состоящая из конечного множества вершин, соединенных между собой ребрами. Порядок соединения может быть любым. Количество вершин определяет порядок графа, а количество ребер – его размер.

Основная цель JPA Entity Graph состоит в том, чтобы **улучшить производительность** во время выполнения при загрузке связанных ассоциаций и основных полей сущности.

JPA 2.1 представила функцию Entity Graph как более **сложный метод работы с нагрузкой на производительность.**

Это позволяет определить шаблон, сгруппировав связанные поля сохраняемости, которые мы хотим получить, и позволяет нам выбрать тип графика во время выполнения.



КАК изменить настройки fetch стратегии любых атрибутов Entity для отдельных запросов (query) или методов поиска (find), если у Entity есть атрибут с fetchType = LAZY, но для конкретного запроса его требуется сделать EAGER или наоборот?

Для этого существует EntityGraph API, используется он так:

С помощью аннотации **NamedEntityGraph** для Entity, создаются именованные EntityGraph объекты, которые содержат список атрибутов, у которых нужно поменять fetchType на EAGER, а потом данное имя указывается в hits запросов или метода find().

В результате fetchType атрибутов Entity меняется, но только для этого запроса.

Существует две стандартных property для указания EntityGraph в hit:

- javax.persistence.fetchgraph — все атрибуты перечисленные в EntityGraph меняют fetchType на EAGER, все остальные на LAZY
- javax.persistence.loadgraph — все атрибуты перечисленные в EntityGraph меняют fetchType на EAGER, все остальные сохраняют свой fetchType (то есть если у атрибута, не указанного в EntityGraph, fetchType был EAGER, то он и останется EAGER).

С помощью **NamedSubgraph** можно также изменить fetchType вложенных объектов Entity.

<T> EntityGraph<T>	createEntityGraph(Class<T> rootType)
Return a mutable EntityGraph that can be used to dynamically create an EntityGraph.	
EntityGraph<?>	createEntityGraph(String graphName)
Return a mutable copy of the named EntityGraph.	

*Можно ли в JDBC реализовать кэш?

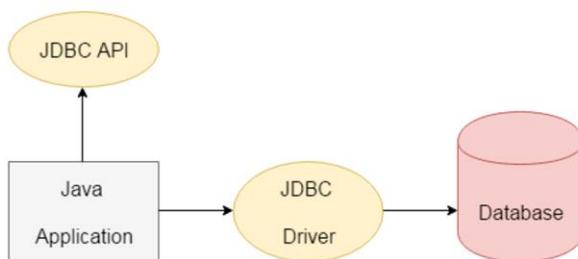
Нет, т.к. JDBC пишет запросы сразу в БД и другой функции у него нет.

*[Какие интерфейсы, классы есть в JDBC?](#)

Интерфейсы JDBC API содержит два основных типа интерфейсов:

- первый – для разработчиков приложений
- второй (более низкого уровня) – для разработчиков драйверов.

Соединение с базой данных описывается классом, реализующим интерфейс java.sql



Список популярных интерфейсов JDBC

- Driver interface
- Connection interface
- Statement interface
- PreparedStatement interface
- CallableStatement interface
- ResultSet interface
- ResultSetMetaData interface
- DatabaseMetaData interface
- RowSet interface

Список популярных классов JDBC API

- DriverManager class
- Blob class
- Clob class
- Types class

*Если установить стратегию генерации ключа AUTO, то какую из типов стратегии Hibernate САМ никогда не выберет и почему?

Hibernate для работы с Entity нужно знать их id -шник. А чтобы id узнать, нужно обратиться (сходить) в базу. А Hibernate ленивый, он «не хочет» лишний раз ходить в БД, поэтому IDENTITY он никогда не выберет.