



Lesson 25

24.06.2021

```
public class Test1 implements Runnable {  
  
    @Override  
    public void run() {  
        System.out.println(3);  
    }  
  
    public static void main(String[] args) throws InterruptedException {  
        Thread thread = new Thread(new Test1());  
        thread.start();  
        System.out.println(1);  
        thread.join();  
        System.out.println(2);  
    }  
}
```

```
public class Test2 {  
    public static void main(String[] args) {  
        List<String> list_one = new ArrayList<>();  
        list_one.add("one");  
        list_one.add("two");  
        list_one.add("one");  
        list_one.add("three");  
        list_one.add("four");  
        list_one.add("five");  
  
        List<String> list_two = new ArrayList<>();  
        list_two.add("one");  
  
        list_one.removeAll(list_two);  
  
        for (String str : list_one)  
            System.out.printf(str + " ");  
    }  
}
```

```
public class Test3 {  
    public static void main(String[] args) {  
        List<String> list = new ArrayList();  
        list.add("one");  
        list.add("two");  
        list.add("three");  
  
        Iterator<Integer> iter = list.iterator();  
  
        while (iter.hasNext()) {  
            System.out.printf(iter.next() + " ");  
        }  
    }  
}
```

```
public class Test4 {  
    public static void main(String[] args) {  
        int[] x = {120, 200, 016};  
        for (int i = 0; i < x.length; i++) {  
            System.out.printf(x[i] + " ");  
        }  
    }  
}
```

ORM (Object Relational Mapping)

ORM (англ. Object-Relational Mapping, рус. объектно-реляционное отображение, или преобразование) — технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая «виртуальную объектную базу данных».

O/R Mapping





Задачи ORM:

- Необходимо обеспечить работу с данными в терминах классов, а не таблиц данных и напротив, преобразовать термины и данные классов в данные, пригодные для хранения в СУБД.
- Необходимо также обеспечить интерфейс для CRUD-операций над данными. В общем, необходимо избавиться от необходимости писать SQL-код для взаимодействия в СУБД.





JPA

Java Persistence API



HIBERNATE

eclipselink



MyBatis

Java Persistence API (JPA) — спецификация API Java EE, предоставляет возможность сохранять в удобном виде Java-объекты в базе.



Преимущества ORM над JDBC:

- Позволяет нашим 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:

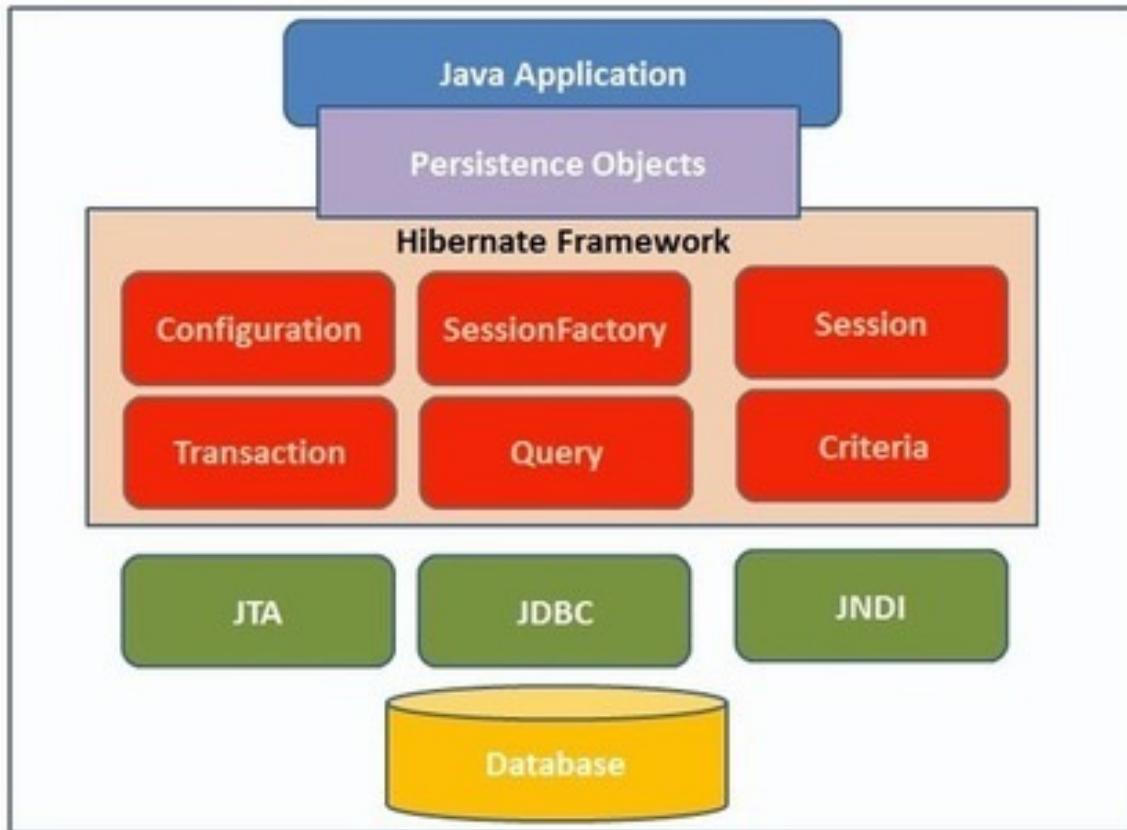
- API, который реализует базовые операции (СОЗДАНИЕ, ЧТЕНИЕ, ИЗМЕНЕНИЕ, УДАЛЕНИЕ) объектов-моделей.
- Средства настройки метаданных связывания
- Технику взаимодействия с транзакциями, которая позволяет реализовать такие функции, как dirty checking, lazy association fetching и т.д.

Hibernate:

Hibernate создаёт связь между таблицами в базе данных (далее – БД) и Java-классами и наоборот. Это избавляет разработчиков от огромного количества лишней, рутинной работы, в которой крайне легко допустить ошибку и крайне трудно потом её найти..



Hibernate Architecture



Configuration

Этот объект используется для создания объекта SessionFactory и конфигурирует сам Hibernate с помощью конфигурационного XML-файла, который объясняет, как обрабатывать объект Session.

SessionFactory

Самый важный и самый тяжёлый объект (обычно создаётся в единственном экземпляре, при запуске приложения). Нам необходима как минимум одна SessionFactory для каждой БД, каждый из которых конфигурируется отдельным конфигурационным файлом.

Session

Сессия используется для получения физического соединения с БД. Обычно, сессия создаётся при необходимости, а после этого закрывается. Это связано с тем, что эти объекты крайне легковесны. Чтобы понять, что это такое, можно сказать, что создание, чтение, изменение и удаление объектов происходит через объект Session.

Query

Этот объект использует HQL или SQL для чтения/записи данных из/в БД. Экземпляр запроса используется для связывания параметров запроса, ограничения количества результатов, которые будут возвращены и для выполнения запроса.

Transaction

Этот объект представляет собой рабочую единицу работы с БД. В Hibernate транзакции обрабатываются менеджером транзакций.

Criteria

Используется для создания и выполнения объекто-ориентированных запроса для получения объектов.



Configuration

обычно, вся эта информация помещена в отдельный файл, либо XML-файл – **hibernate.cfg.xml**, либо – **hibernate.properties**.

hibernate.dialect	Указывает Hibernate диалект БД. Hibernate генерирует необходимые SQL-запросы
hibernate.connection.driver_class	Указывает класс JDBC драйвера.
hibernate.connection.url	Указывает URL (ссылку) необходимой нам БД.
hibernate.connection.username	Указывает имя пользователя БД
hibernate.connection.password	Укашывает пароль к БД
hibernate.connection.pool_size	Ограничивает количество соединений, которые находятся в пуле соединений Hibernate.
hibernate.connection.autocommit	Указывает режим autocommit для JDBC-соединения.

<https://docs.jboss.org/hibernate/orm/3.5/api/org/hibernate/dialect/class-use/Dialect.html>



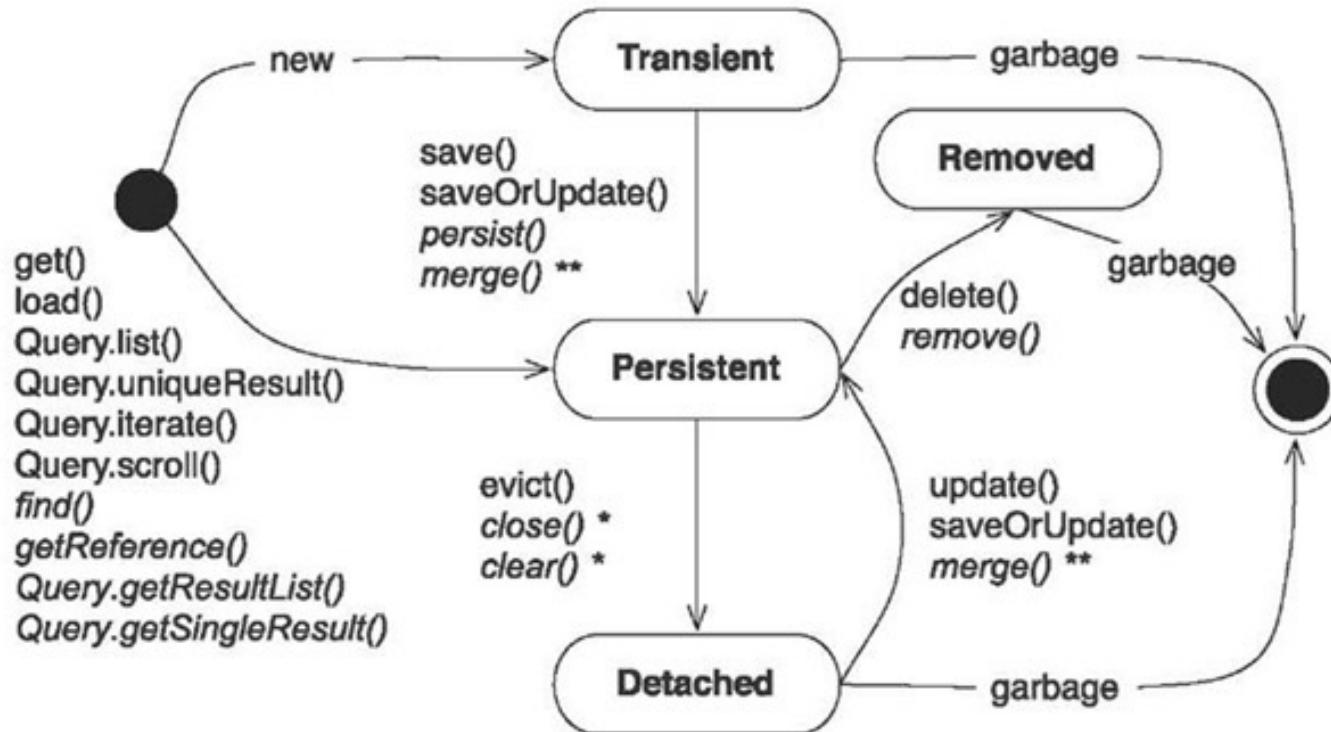
Session

Сессия используется для получения физического соединения с базой данных (далее – БД). Благодаря тому, что сессия является легковесны объектом, его создают (открывают сессию) каждый раз, когда возникает необходимость, а потом, когда необходимо, уничтожают (закрывают сессию). Мы создаём, читаем, редактируем и удаляем объекты с помощью сессий.

Мы стараемся создавать сессии при необходимости, а затем уничтожать их из-за того, что ни не являются потоко-защищёнными и не должны быть открыты в течение длительного времени.

Экземпляр класса может находиться в одном из трёх состояний:

- **transient**
- **persistent**
- **detached**



* Hibernate & JPA, affects all instances in the persistence context

** Merging returns a persistent instance, original doesn't change state



Transient – это новый экземпляр класса, который не привязан к сессии и ещё не представлен в БД. Он не имеет значения, по которому может быть идентифицирован.

Persistent - мы можем создать переходный экземпляр класса, связав его с сессией. Устойчивый экземпляр класса представлен в БД, а значение идентификатора связано с сессией.

Detached - после того, как сессия закрыта, экземпляр класса становится отдельным, независимым экземпляром класса.

Ключевая функция Hibernate заключается в том, что мы можем взять значения из нашего Java-класса и сохранить их в таблице базы данных. С помощью конфигурационных файлов или аннотаций мы указываем Hibernate как извлечь данные из класса и соединить с определённым столбцами в таблице БД.

Если мы хотим, чтобы экземпляры (объекты) Java-класса в будущем созранялся в таблице БД, то мы называем их persistent class. Для того, чтобы сделать работу с Hibernate максимально удобной и эффективной, мы следует использовать программную модель Plain Old Java Object – POJO.

Существуют определённые требования к POJO классам. Вот самые главные из них:

- Все классы должны иметь ID для простой идентификации наших объектов в БД и в Hibernate. Это поле класса соединяется с первичным ключём (primary key) таблицы БД.
- Все POJO – классы должны иметь конструктор по умолчанию (пустой).
- Все поля POJO – классов должны иметь модификатор доступа **private** иметь набор getter-ов и setter-ов в стиле JavaBean.
- POJO – классы не должны содержать бизнес-логику.

Но в Hibernate предусмотрена возможность конфигурирования приложения с помощью аннотаций.

@Entity

Эта аннотация указывает Hibernate, что данный класс является сущностью (entity bean). Такой класс должен иметь конструктор по-умолчанию (пустой конструктор).

@Table

С помощью этой аннотации мы говорим Hibernate, с какой именно таблицей необходимо связать (тар) данный класс. Аннотация **@Table** имеет различные атрибуты, с помощью которых мы можем указать имя таблицы, каталог, БД и уникальность столбцов в таблице БД.

```
1  @Entity  
2  @Table(name = "cat_table")  
3  public class Cat implements Serializable {  
4  ...  
5  }
```

@Id

С помощью аннотации **@Id** мы указываем *первичный ключ (Primary Key)* данного класса.

@GeneratedValue

Эта аннотация используется вместе с аннотацией **@Id** и определяет параметры, **strategy** и **generator**.

```
1  @Entity
2  @Table(name = "cat_table")
3  public class Cat implements Serializable {
4
5      @Id
6      @GeneratedValue(strategy = IDENTITY) //AUTO, SEQUENCE, TABLE
7      @Column(name = "id")
8      private int id;
9
10     ...
11 }
```

@Column

Аннотация **@Column** определяет к какому столбцу в таблице БД относится конкретное поле класса (атрибут класса).

Наиболее часто используемые атрибуты аннотации **@Column** такие:

- **name**

Указывает имя столбца в таблице

- **unique**

Определяет, должно ли быть данное значение уникальным

- **nullable**

Определяет, может ли данное поле быть NULL, или нет.

- **length**

Указывает, какой размер столбца (например количество символов, при использовании String).

@Version

управление версией в записи сущности. При изменении записи увеличится на 1.

```
1  @Entity  
2  @Table(name = "cat_table")  
3  public class Cat implements Serializable {  
4  
5      @Version  
6      @Column(name = "version")  
7      private int version;  
8  
9      ...  
10 }
```

@OrderBy

указание сортировки. В примере множество кошек будет отсортировано по имени по возрастанию.

```
1  @Entity  
2  @Table(name = "cat_table")  
3  public class Cat implements Serializable {  
4  
5      @OrderBy("firstName asc")  
6      private Set catsSet;  
7  
8  }
```

@Transient

указывает, что свойство не нужно записывать. Значения под этой аннотацией не записываются в базу данных (так же не участвуют в сериализации). static и final переменные экземпляра всегда transient.

```
1  @Entity  
2  @Table(name = "cat_table")  
3  public class Cat implements Serializable {  
4  
5      @Transient  
6      public boolean isNew() {  
7          return id == null;  
8      }  
9  
10     ...  
11 }
```

@Lob

указание на большие объекты.



@OneToOne

указывает на связь между таблицами «один к одному».

`orphanRemoval` — позволяет удалять объекты сироты. При удалении родительского объекта удаляется и дочерний.

`mappedBy` — обратная сторона связи сущности. Поле под этим атрибутом не сохраняется как часть исходной сущности в базе данных, но будет доступна по запросу. (см. ниже `@JoinColumn`)

`cascade = CascadeType.ALL` — означает, что операция, например, записи должна распространяться и на дочерние таблицы.

```
1  @Entity
2  @Table(name = "contactDetail")
3  public class ContactDetail implements Serializable {
4
5      @Id
6      @Column(name = "id")
7      @GeneratedValue
8      private int id;
9
10     @OneToOne
11     @MapsId
12     @JoinColumn(name = "contactId")
13     private Contact contact;
14
15     ...
16 }
17
18 @Entity
19 @Table(name = "contact")
20 public class Contact implements Serializable {
21
22     @Id
23     @Column(name = "ID")
24     @GeneratedValue
25     private Integer id;
26
27     @OneToOne(mappedBy = "contact", cascade = CascadeType.ALL)
28     private ContactDetail contactDetail;
29
30     ....
31 }
```

@OneToMany

```
1  @Entity  
2  public class Troop {  
3      @OneToMany(mappedBy="troop")  
4      public Set<Soldier> getSoldiers() {  
5          ...  
6      }  
7  
8  @Entity  
9  public class Soldier {  
10     @ManyToOne  
11     @JoinColumn(name="troop_fk")  
12     public Troop getTroop() {  
13         ...  
14     }
```

@ManyToMany

```
1  @Entity
2  @Table(name = "contact")
3  public class Contact implements Serializable {
4
5      private Set<Hobby> hobbies = new HashSet<Hobby>();
6      //...
7
8      @ManyToMany
9      @JoinTable(name = "contact_hobby_detail",
10         joinColumns = @JoinColumn(name = "CONTACT_ID"),
11         inverseJoinColumns = @JoinColumn(name = "HOBBY_ID"))
12     public Set<Hobby> getHobbies() {
13         return this.hobbies;
14     }
15 ...
16
17  @Entity
18  @Table(name = "hobby")
19  public class Hobby implements Serializable {
20
21      private Set<Contact> contacts = new HashSet<Contact>();
22      //...
23
24      @ManyToMany
25      @JoinTable(name = "contact_hobby_detail",
26         joinColumns = @JoinColumn(name = "HOBBY_ID"),
27         inverseJoinColumns = @JoinColumn(name = "CONTACT_ID"))
28     public Set<Contact> getContacts() {
29         return this.contacts;
30     }
31 }
```

Стратегии загрузки коллекций в JPA

Стратегии загрузки определяют какой граф объектов будет выгружен на этапе компиляции через маппинги (XML или аннотации). Будем различать стратегии, распространяемые на все приложение такие, как «ленивая» загрузка (lazy loading) или «жадная» загрузка (eager loading) и стратегии, распространяемые на методы и запросы такие, как явные джойны.

```
1  @Entity
2  @Table(name = "usr")
3  public class User {
4
5      @OneToMany(fetch = FetchType.EAGER)
6      private Set<UserLogins> logins;
7 }
```



Lazy loading

Поведение JPA по умолчанию такое, что только корневой объект, которые явно был запрошен из БД, будет загружен. Все остальные объекты в графе будут «ленивыми». Они будут представлены в виде прокси и загружены в момент обращения к ним. Однако, необходимым условием является то, что сессия Entity manager должны быть открыта. Когда сессия открыта? По умолчанию сессия привязана к жизненному циклу транзакции. Это означает, что если Вы не имеете какой-нибудь открытой транзакции, то сессия будет закрыта после того как Вы выполнили свою операцию с БД.

Преимущества:

- Время начальной загрузки намного меньше, чем при другом подходе
- Меньше потребление памяти, чем в другом подходе

Недостатки:

- Отложенная инициализация может повлиять на производительность

Eager loading

В отличие от «ленивой» загрузки в этой стратегии объекты загружаются сразу. Стратегия называется «жадная» загрузка и ее можно включить через маппинг объекта. Для каждого атрибута сущности, указывающего на другую сущность (через аннотации `@OneToOne`, `@ManyToOne`, `@OneToMany` или `@ManyToMany`) Вы можете указать, чтобы эта сущность загрузилась вместе с исходной.

Преимущества:

Отсутствие влияния на производительность при отсроченной инициализации

Недостатки:

Длительное время начальной загрузки

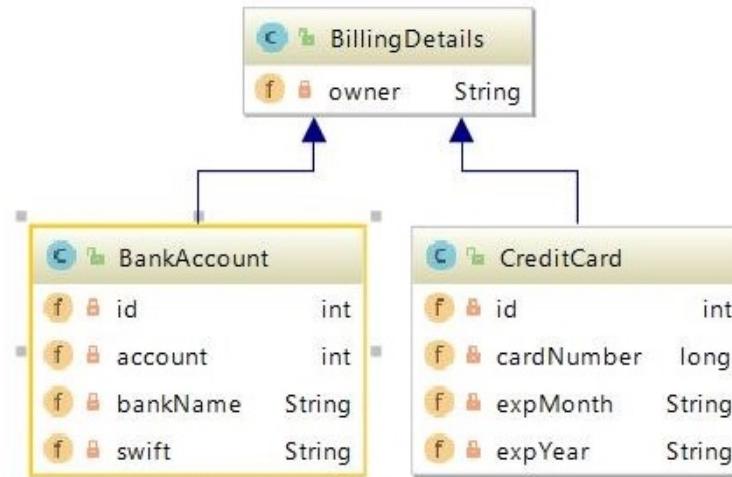
Загрузка слишком большого количества ненужных данных может повлиять на производительность

Наследование

Классы в Java могут вступать в наследственные отношения и эти отношения должны как-то сохраняться и при переносе классов в базы данных, в которых наследования, за исключением некоторых реализаций, как бы нет. JPA предлагает целых четыре решения по заполнению этой пропасти между классами и таблицами.

- 1) Использовать одну таблицу для каждого класса и полиморфное поведение по умолчанию.
- 2) Одна таблица для каждого конкретного класса, с полным исключением полиморфизма и отношений наследования из схемы SQL (для полиморфного поведения во время выполнения будут использоваться UNION-запросы)
- 3) Единая таблица для всей иерархии классов. Возможна только за счет денормализации схемы SQL. Определять суперкласс и подклассы будет возможно посредством различия строк.
- 4) Одна таблица для каждого подкласса, где отношение “is a” представлено в виде «has a», т.е. – связь по внешнему ключу с использованием JOIN.

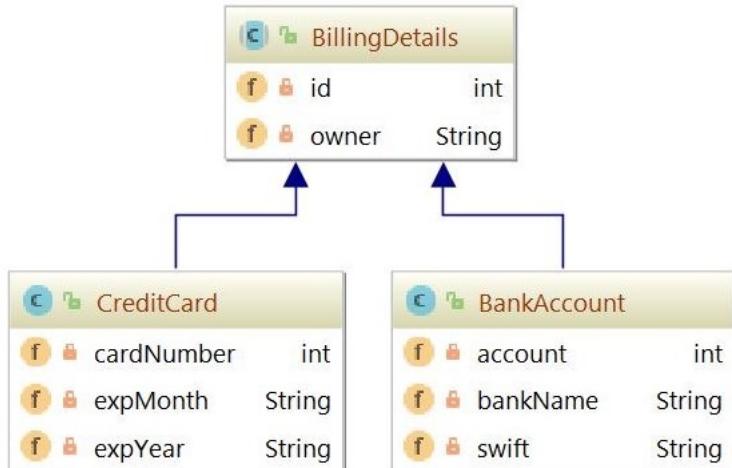
Одна таблица для каждого класса



CREDIT_CARD	
🔑	ID: INTEGER
👤	CC_OWNER: VARCHAR(20)
👤	CARD_NUMBER: VARCHAR(018)
👤	EXP_MONTH: VARCHAR(9)
👤	EXP_YEAR: VARCHAR(4)

BANK_ACCOUNT	
🔑	ID: INTEGER
👤	OWNER: VARCHAR(20)
👤	ACCOUNT: VARCHAR(20)
👤	BANK_NAME: VARCHAR(30)
👤	SWIFT: VARCHAR(15)

Единая таблица для всей иерархии классов



BILLING_DETAILS	
ID: INTEGER	Discriminator
BD_TYPE: VARCHAR(2)	
OWNER: VARCHAR(20)	
CARD_NUMBER: INTEGER	
EXP_MONTH: VARCHAR(20)	
EXP_YEAR: VARCHAR(9)	
ACCOUNT: INTEGER	
BANK_NAME: VARCHAR(20)	
SWIFT: VARCHAR(20)	

Одна таблица для каждого класса с использованием соединений (JOIN)

