



# Lesson 32

07.09.2023



```
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() + " ");  
        }  
    }  
}
```

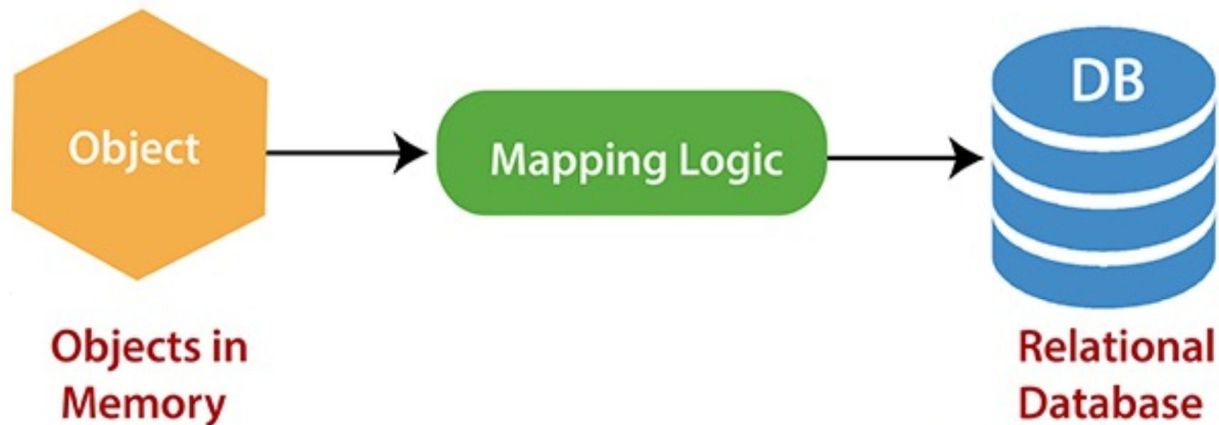
```
public class Test4 {  
    public static void main(String[] args) {  
        int[] x = {120, 200, 016};  
        for (int i =0; i < x.length; i++){  
            System.out.printf(x[i] + " ");  
        }  
    }  
}
```



## ORM (Object Relational Mapping)

**ORM** (англ. Object-Relational Mapping, об'єктно-реляційне відображення, або перетворення) - технологія програмування, яка пов'язує бази даних з концепціями об'єктно-орієнтованих мов програмування, створюючи «віртуальну об'єктну базу даних».

### O/R Mapping



### Student class in object model

```
@Entity(name="STUDENT")
public class Student {

    @Id
    @Column(name = "ID")
    private Long studentId;

    @Column(name = "FNAME")
    private String firstName;

    @Column(name = "LNAME")
    private String lastName;

    @Column(name = "CONTACT_NO")
    private String contactNo;
```

ORM  
Implementation  
Eg: Hibernate

### Student table in relational model

**student**

- ID INT(11)
- FNAME VARCHAR(45)
- LNAME VARCHAR(45)
- CONTACT\_NO VARCHAR(45)

Indexes

PRIMARY

## Завдання ORM:

- Необхідно забезпечити роботу з даними у термінах класів, а не таблиць даних і наоборот, перетворити терміни та дані класів на дані, придатні для зберігання у СУБД.
- Необхідно також забезпечити інтерфейс для CRUD-операцій над даними. Загалом, необхідно позбутися необхідності писати SQL-код для взаємодії у СУБД.







# JPA

Java Persistence API



# HIBERNATE

eclipse) link



# MyBatis

## Переваги ORM над JDBC:

- ❖ Дозволяє нашим бізнес методам звертатися не до БД, а до Java-класів
- ❖ Прискорює розробку програми
- ❖ Заснований на JDBC
- ❖ Відокремлює SQL-запити від ОО моделі
- ❖ Дозволяє не думати про реалізацію БД
- ❖ Сутності засновані на бізнес-завданнях, а не на структурі БД
- ❖ Управління транзакціями

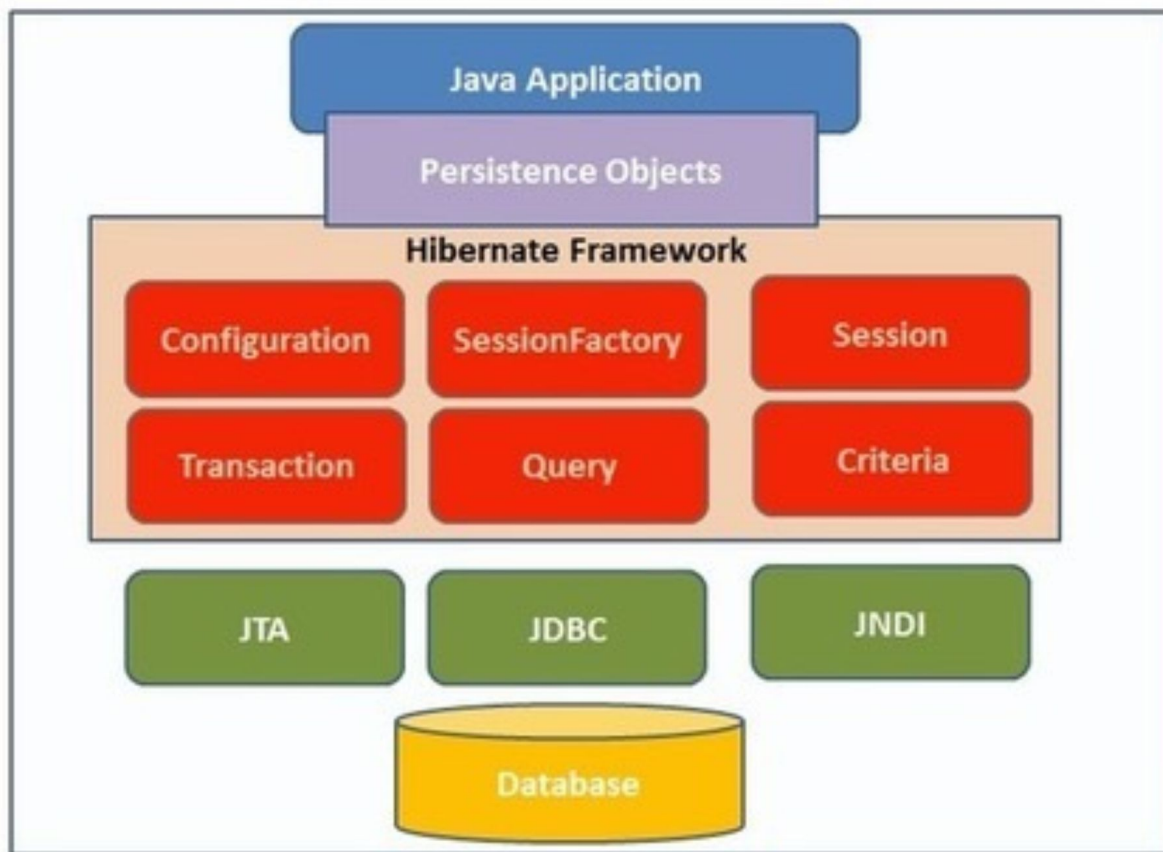


## Hibernate:

Hibernate створює зв'язок між таблицями у базі даних (далі – БД) та Java класами і навпаки. Це позбавляє розробників від величезної кількості зайвої, рутинної роботи, в якій дуже легко припуститися помилки і вкрай важко потім її знайти.



# Hibernate Architecture





## Configuration

Цей об'єкт використовується для створення об'єкта SessionFactory і конфігурує сам Hibernate за допомогою конфігураційного XML-файлу, який пояснює, як обробляти об'єкт Session.

## SessionFactory

Найважливіший і найважчий об'єкт (зазвичай створюється в єдиному еземплярі, при запуску приладу). Нам необхідна щонайменше одна SessionFactory кожної БД, кожен із яких конфігурується окремим конфігураційним файлом.

## Session

Сесія використовується для отримання фізичної сполуки з БД. Зазвичай, сесія створюється за необхідності, а потім закривається. Це з тим, що ці об'єкти вкрай легковагні. Щоб зрозуміти, що це таке, модно сказати, що створення, читання, зміна та видалення об'єктів відбувається через об'єкт Session.

## Query

Цей об'єкт використовує HQL або SQL для читання/запису даних з БД. Примірник запиту використовується для зв'язування параметрів запиту, обмеження кількості результатів, які будуть повернуті та виконання запиту.

## Criteria

Використовується для створення та виконання об'єктно-орієнтованих запитів для отримання об'єктів.

## Transaction

Цей об'єкт є робочою одиницю роботи з БД. Hibernate транзакції обробляються менеджером транзакцій.

# Configuration

Як правило, вся ця інформація поміщена в окремий файл, або XML-файл - `hibernate.cfg.xml`, або - `hibernate.properties`.

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

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



## Session

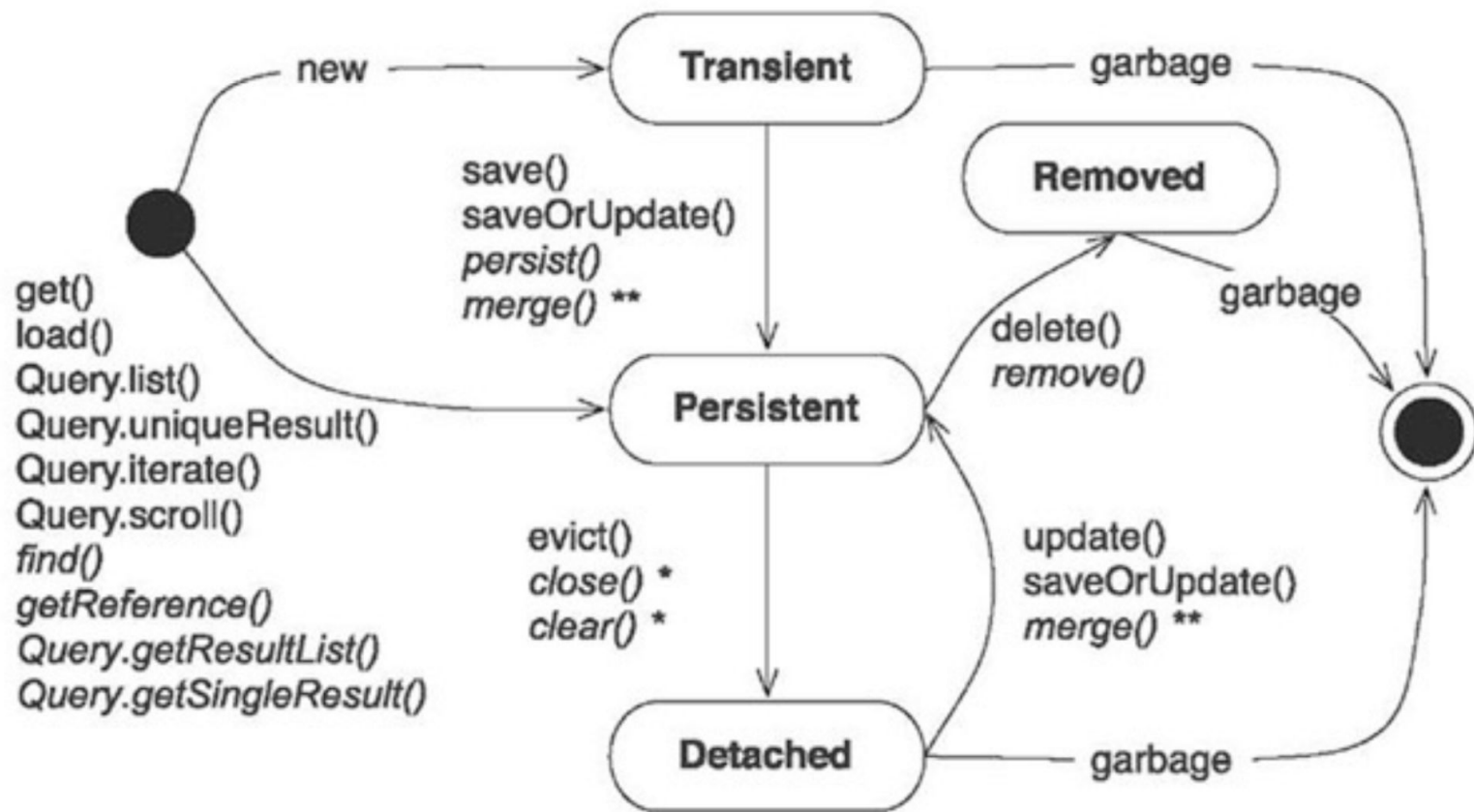
Сесія використовується для отримання фізичного з'єднання з базою даних (далі – БД). Завдяки тому, що сесія є легковісним об'єктом, його створюють (відкривають сесію) щоразу, коли виникає необхідність, а потім, коли необхідно, знищують (закривають сесію). Ми створюємо, читаємо, редагуємо та видаляємо об'єкти за допомогою сесій. Ми намагаємося створювати сесії при необхідності, а потім знищувати їх через те, що вони не є потоко-захищеними і не повинні бути відкриті протягом тривалого часу.

Екземпляри класу можуть бути в одному з трьох станів:

➤ transient

➤ persistent

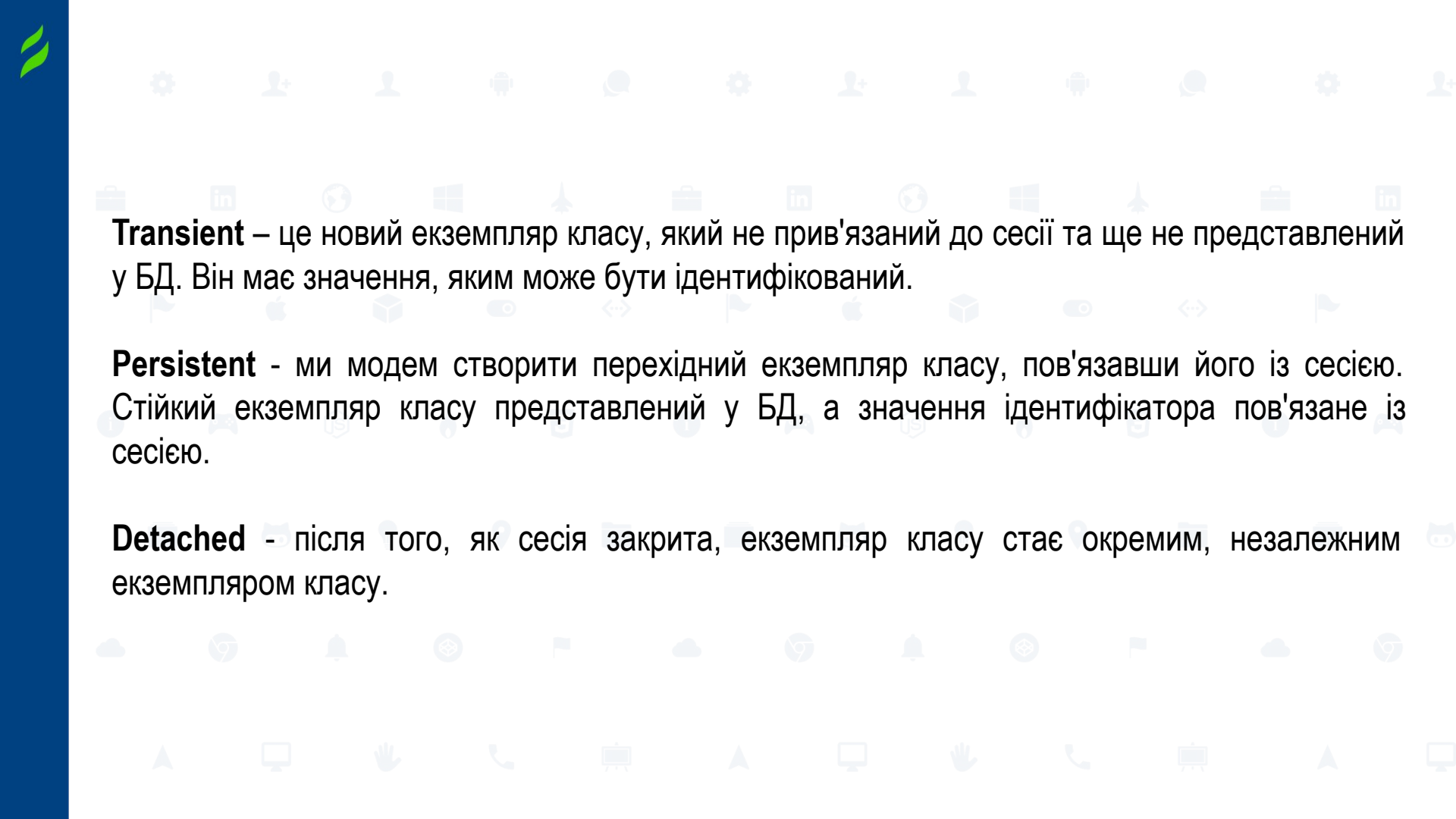
➤ detached



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

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






**Transient** – це новий екземпляр класу, який не прив'язаний до сесії та ще не представлений у БД. Він має значення, яким може бути ідентифікований.

**Persistent** - ми можемо створити перехідний екземпляр класу, пов'язавши його із сесією. Стійкий екземпляр класу представлений у БД, а значення ідентифікатора пов'язане із сесією.

**Detached** - після того, як сесія закрита, екземпляр класу стає окремим, незалежним екземпляром класу.



Ключова функція Hibernate полягає в тому, що ми можемо взяти значення нашого Java-класу і зберегти їх у таблиці бази даних. За допомогою конфігураційних файлів або анотацій ми вказуємо Hibernate як отримати дані з класу і з'єднати з певними стовпцями в таблиці БД. Якщо хочемо, щоб екземпляри (об'єкти) Java-класу у майбутньому зберігався у таблиці БД, ми називаємо їх `persistent class`. Для того, щоб зробити роботу з Hibernate максимально зручною та ефективною, слід використовувати програмну модель Plain Old Java Object – POJO.

Існують певні вимоги до класів POJO. Ось найголовніші з них:

- Усі класи повинні мати ID для простої ідентифікації наших об'єктів у БД і в Hibernate. Це поле класу поєднується з первинним ключем (`primary key`) таблиці БД.
- Усі POJO – класи повинні мати конструктор за замовчуванням (порожній).
- Усі поля POJO – класів повинні мати модифікатор доступу `private` мати набір `getter`-ів та `setter`-ів у стилі `JavaBean`.
- POJO – класи не повинні містити бізнес-логіки.

## JPA Annotations

1. **@Entity**: Specifies that a class is an entity and is mapped to a database table.
2. **@Table**: Specifies the table name associated with an entity.
3. **@Id**: Marks a field as the primary key of an entity.
4. **@GeneratedValue**: Specifies the strategy for generating primary key values.
5. **@Column**: Specifies the mapping for a database column.
6. **@Transient**: Excludes a field from being persisted to the database.
7. **@OneToOne**: Defines a one-to-one relationship between two entities.
8. **@OneToMany**: Defines a one-to-many relationship between two entities.
9. **@ManyToOne**: Defines a many-to-one relationship between two entities.
10. **@ManyToMany**: Defines a many-to-many relationship between two entities.
11. **@JoinColumn**: Specifies the foreign key column for a relationship.
12. **@Embedded**: Specifies a persistent field or property of an entity whose value is an instance of an embeddable class.
13. **@NamedQuery**: Declares a named query for an entity.
14. **@NamedNativeQuery**: Declares a named native SQL query for an entity.
15. **@Version**: Specifies the version field for optimistic locking.

## Hibernate Annotations

1. **@Cascade**: Specifies the cascade behavior for associations.
2. **Fetch**: Specifies the fetching strategy for associations.
3. **@LazyToOne**: Specifies the lazy loading behavior for a to-one association.
4. **@LazyCollection**: Specifies the lazy loading behavior for a collection association.
5. **@BatchSize**: Specifies the batch size for loading a collection association.
6. **@Cacheable**: Enables caching for an entity or a collection.
7. **@Cache**: Specifies the cache region and cache strategy for an entity or a collection.
8. **@Formula**: Defines a computed property using an SQL formula.
9. **@NaturalId**: Marks a property as a natural identifier.
10. **@Filter**: Defines a filter condition to be applied to a collection association.
11. **@Where**: Specifies a SQL WHERE condition to be applied to a collection association.
12. **@Type**: Specifies the Hibernate type for a property.
13. **@Any**: Maps a polymorphic association to any entity type.
14. **@TypeDef**: Defines a custom Hibernate type.