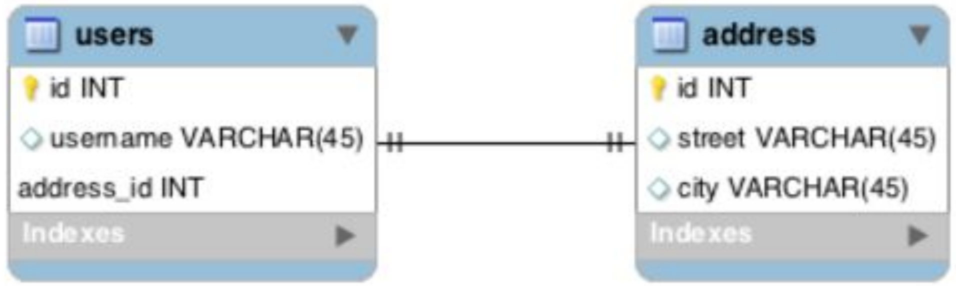
ORM - JPA - Hibernate - extended concepts

Codecool, 2021

CODECOOL

JOINS

@OneToOne

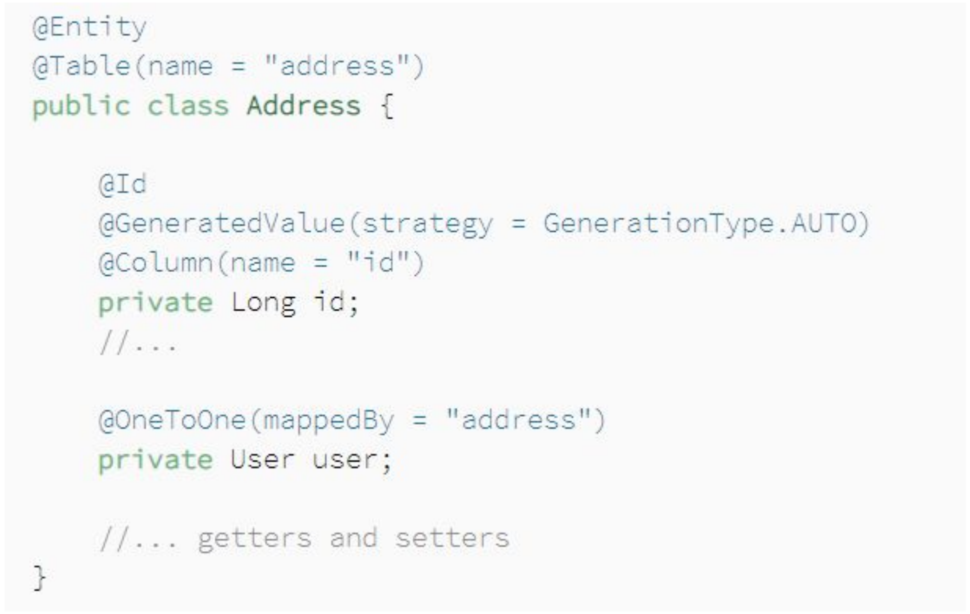

users
id INT
username VARCHAR(45)
address_id INT
Indexes
address
id INT
street VARCHAR(45)
city VARCHAR(45)
Indexes

@OneToOne

```java
@Entity
@Table(name = "users")
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id")
    private Long id;
    //...

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "address_id", referencedColumnName = "id")
    private Address address;

    // ... getters and setters
}
```

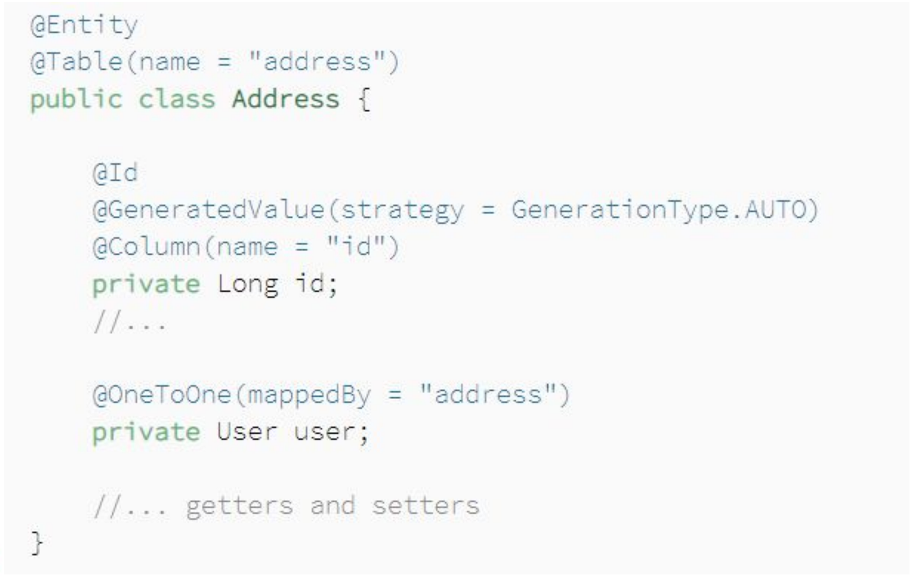
@OneToOne

whoever owns the foreign key column gets the @JoinColumn annotation.

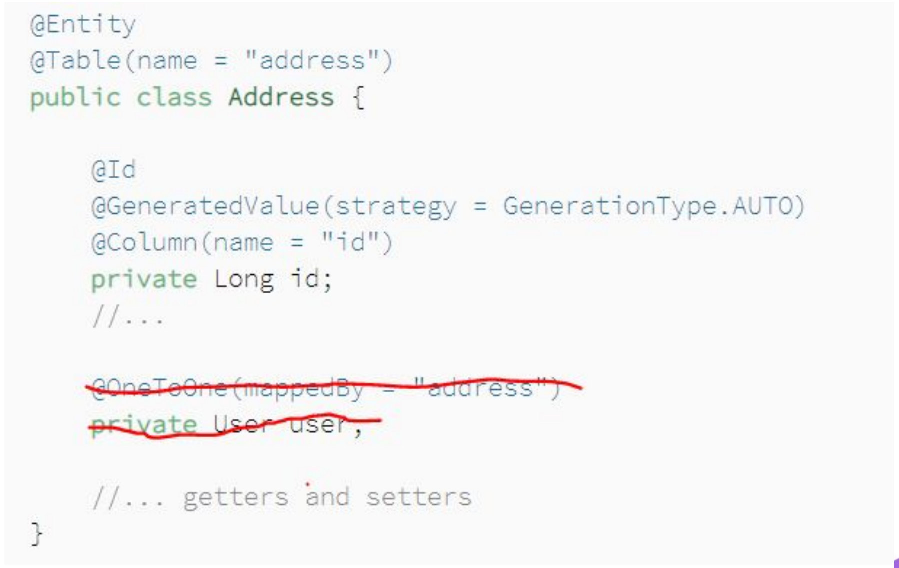


```java
@Entity
@Table(name = "address")
public class Address {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id")
    private Long id;
    //...

    @OneToOne(mappedBy = "address")
    private User user;

    //... getters and setters
}
```

Bidirectional vs Unidirectional

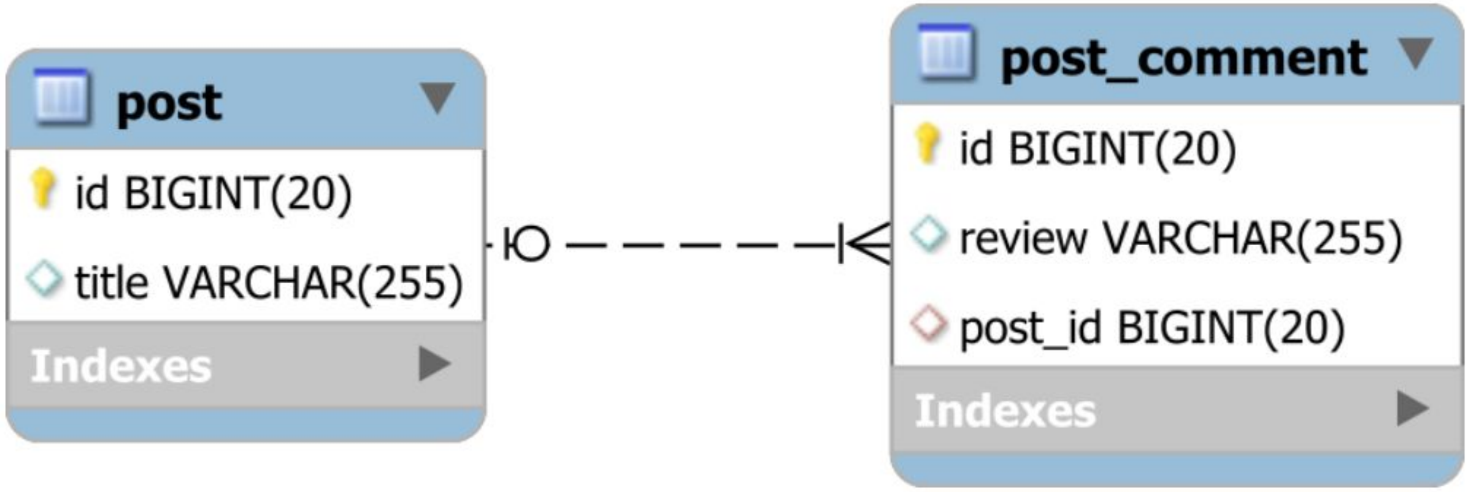


```
@Entity
@Table(name = "address")
public class Address {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id")
    private Long id;
    //...

    @OneToOne(mappedBy = "address")
    private User user;

    //... getters and setters
}
```



```
@Entity
@Table(name = "address")
public class Address {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id")
    private Long id;
    //...

    ~~@OneToOne(mappedBy = "address")~~
    ~~private User user;~~

    //... getters and setters
}
```

@OneToMany

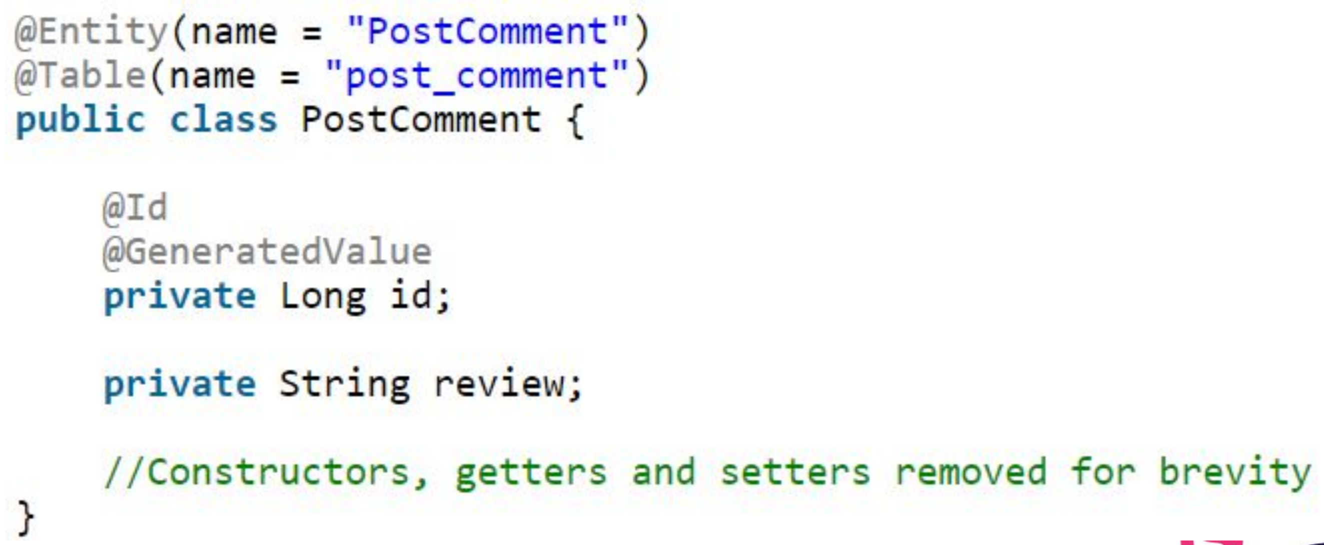

post
id BIGINT(20)
title VARCHAR(255)
Indexes
post_comment
id BIGINT(20)
review VARCHAR(255)
post_id BIGINT(20)
Indexes

CODECOOL

@OneToMany

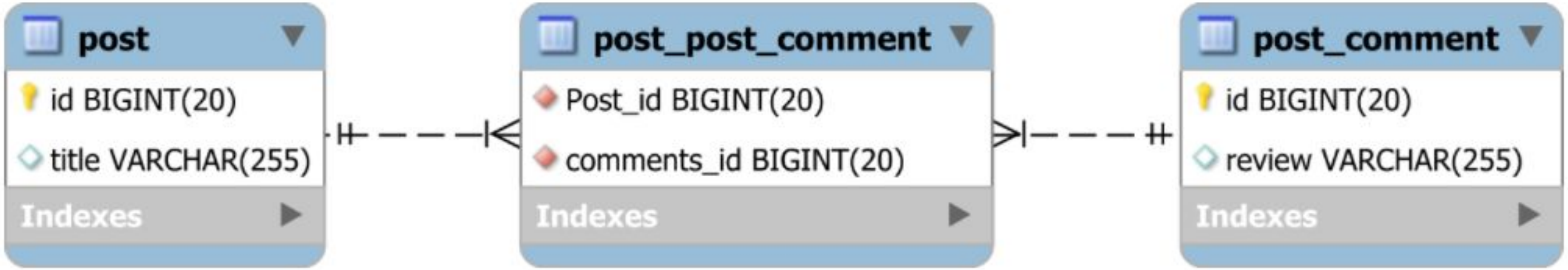
```
@Entity(name = "Post")
@Table(name = "post")
public class Post {

    @Id
    @GeneratedValue
    private Long id;

    private String title;

    @OneToMany(
        cascade = CascadeType.ALL,
        orphanRemoval = true
    )
    private List<PostComment> comments = new ArrayList<>();

    //Constructors, getters and setters removed for brevity
}
```

@OneToMany (or maybe we should say @ManyToOne?)



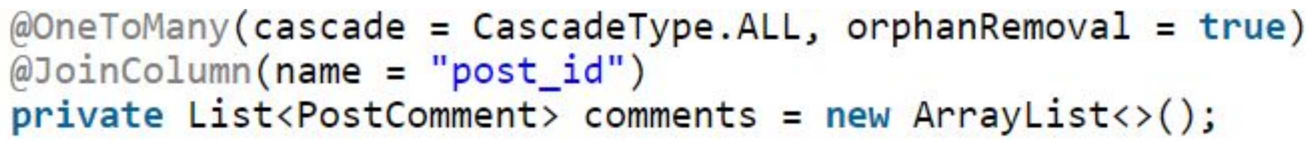
```
@Entity(name = "PostComment")
@Table(name = "post_comment")
public class PostComment {

    @Id
    @GeneratedValue
    private Long id;

    private String review;

    //Constructors, getters and setters removed for brevity
}
```

But wait, what happened?



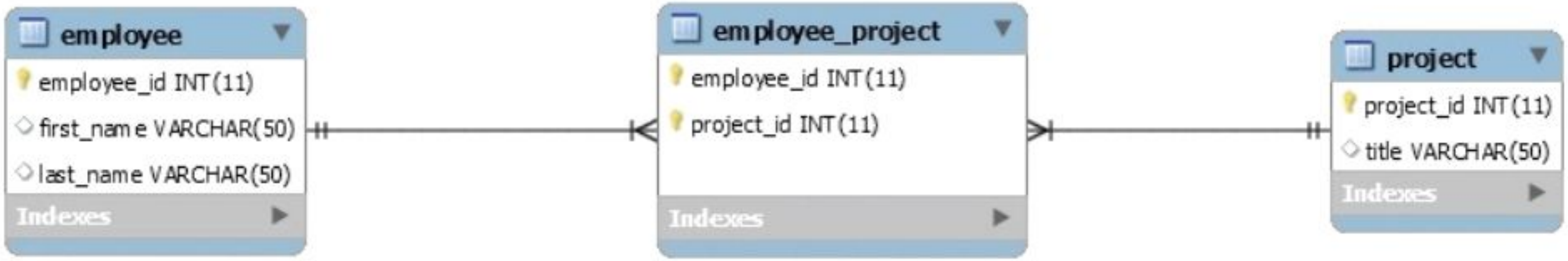
How to fix it?

To fix the aforementioned extra join table issue, we just need to add the @JoinColumn in the mix:

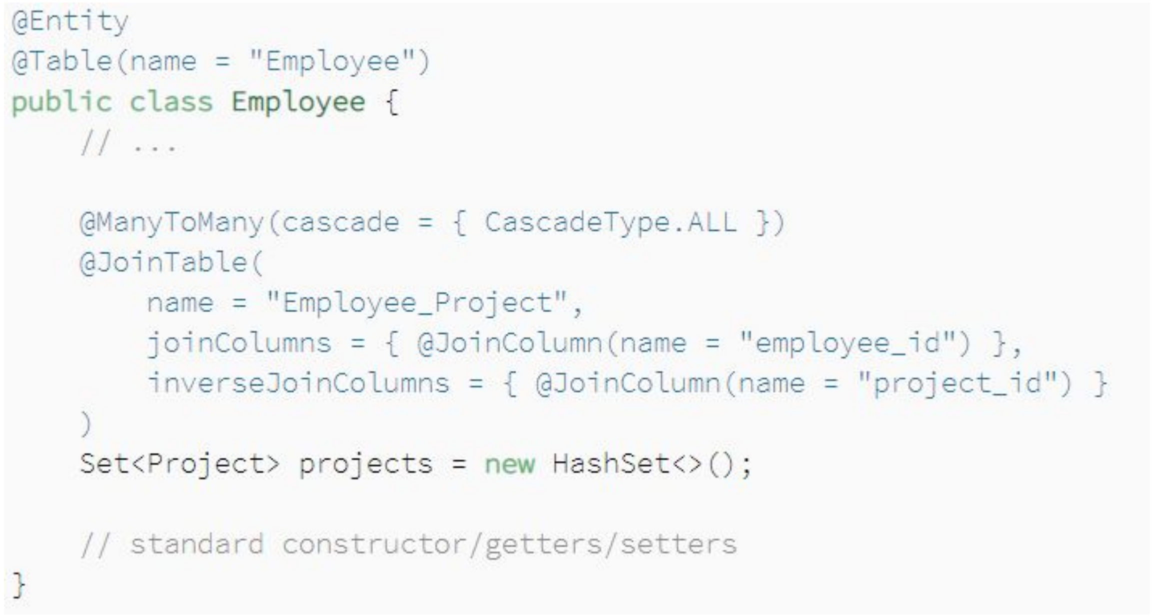


```java
@OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
@JoinColumn(name = "post_id")
private List<PostComment> comments = new ArrayList<>();
```

@ManyToMany


employee
employee_id INT(11)
first_name VARCHAR(50)
last_name VARCHAR(50)
Indexes
employee_project
employee_id INT(11)
project_id INT(11)
Indexes
project
project_id INT(11)
title VARCHAR(50)
Indexes

@ManyToMany

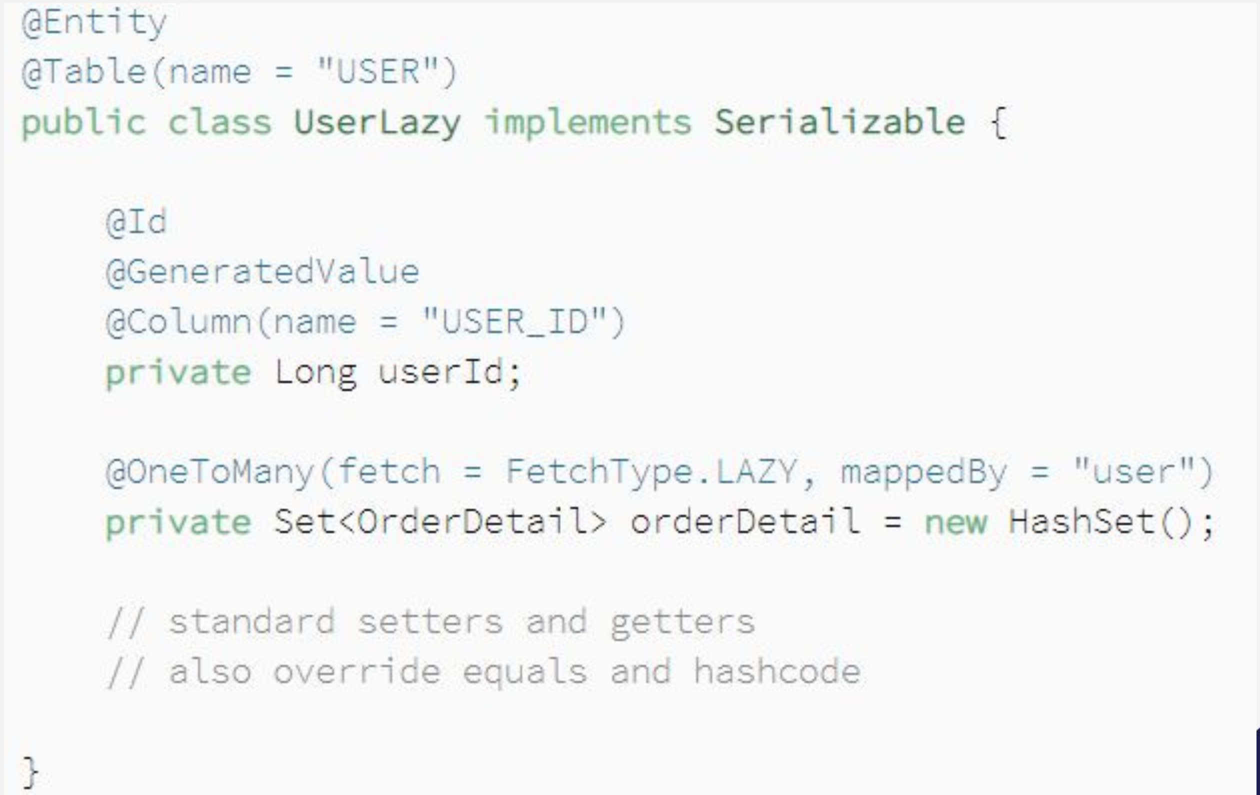


```java
@Entity
@Table(name = "Employee")
public class Employee {
    // ...

    @ManyToMany(cascade = { CascadeType.ALL })
    @JoinTable(
        name = "Employee_Project",
        joinColumns = { @JoinColumn(name = "employee_id") },
        inverseJoinColumns = { @JoinColumn(name = "project_id") }
    )
    Set<Project> projects = new HashSet<>();

    // standard constructor/getters/setters
}
```

CODECOOL

FETCH

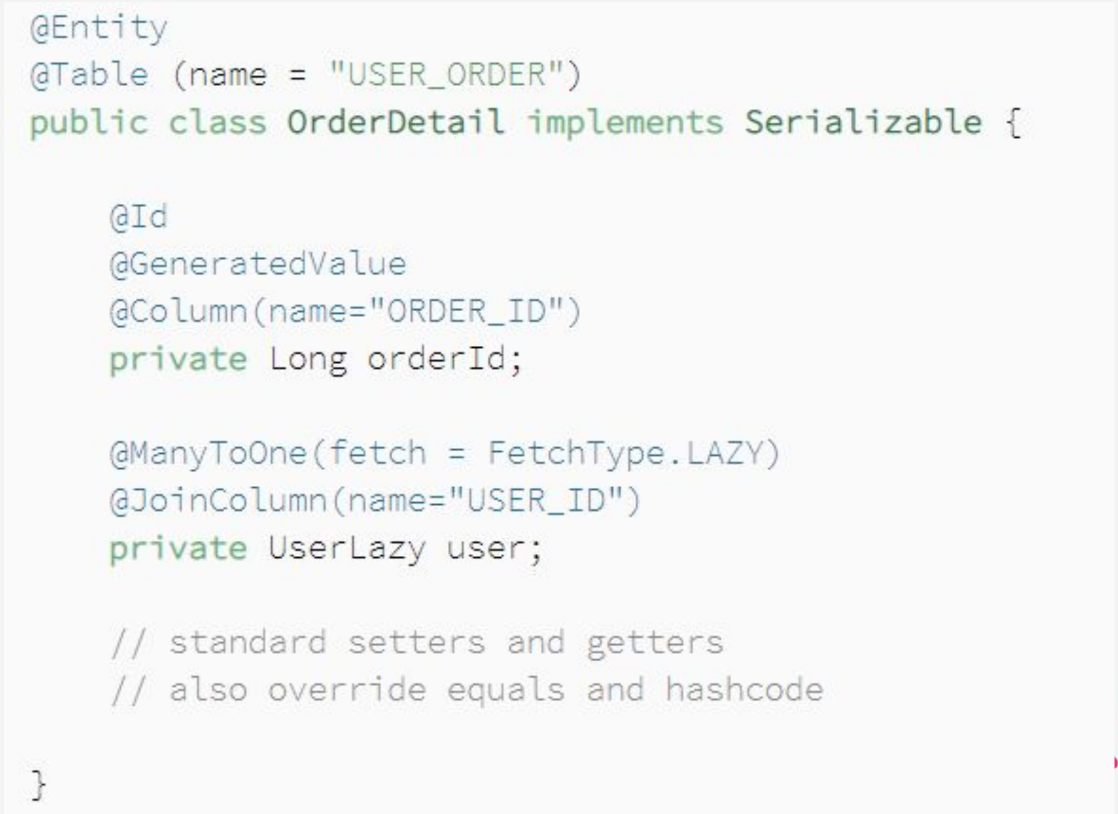
ADD FETCH



```
@Entity
@Table(name = "USER")
public class UserLazy implements Serializable {

    @Id
    @GeneratedValue
    @Column(name = "USER_ID")
    private Long userId;

    @OneToMany(fetch = FetchType.LAZY, mappedBy = "user")
    private Set<OrderDetail> orderDetail = new HashSet();

    // standard setters and getters
    // also override equals and hashcode

}
```

CODECOOL

ADD FETCH



```
@Entity
@Table (name = "USER_ORDER")
public class OrderDetail implements Serializable {

    @Id
    @GeneratedValue
    @Column(name="ORDER_ID")
    private Long orderId;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name="USER_ID")
    private UserLazy user;

    // standard setters and getters
    // also override equals and hashcode

}
```

EAGER vs LAZY

- `fetch = FetchType.LAZY`

Advantages:

- Initial load time much smaller than in the other approach
- Less memory consumption than in the other approach

Disadvantages:

- Delayed initialization might impact performance during unwanted moments
- In some cases you need to handle lazily-initialized objects with a special care or you might end up with an exception

- `fetch = FetchType.EAGER`

Advantages:

- No delayed initialization related performance impacts

Disadvantages:

- Long initial loading time
- Loading too much unnecessary data might impact performance

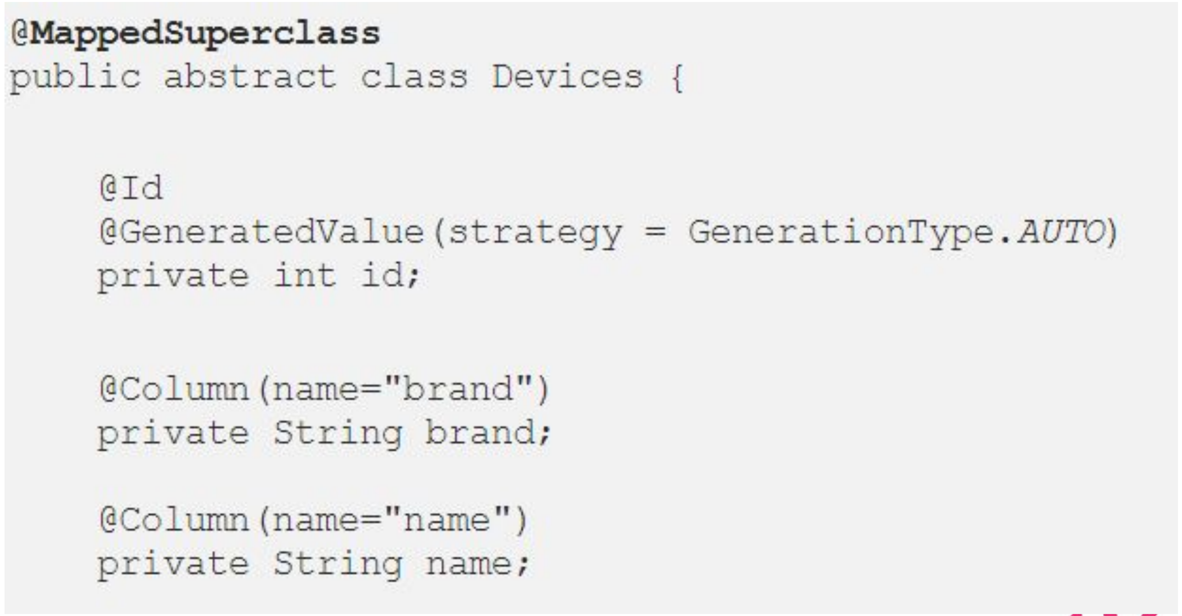
Default Fetch Type

- OneToMany: LAZY
- ManyToOne: EAGER
- ManyToMany: LAZY
- OneToOne: EAGER

Inheritance

@MappedSuperclass

non-entity super class and you can create entity subclasses that inherit this super class. With this annotation, a separate table for each subclass is created. However, a table is not created for the super class itself.



```
@MappedSuperclass
public abstract class Devices {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;

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

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

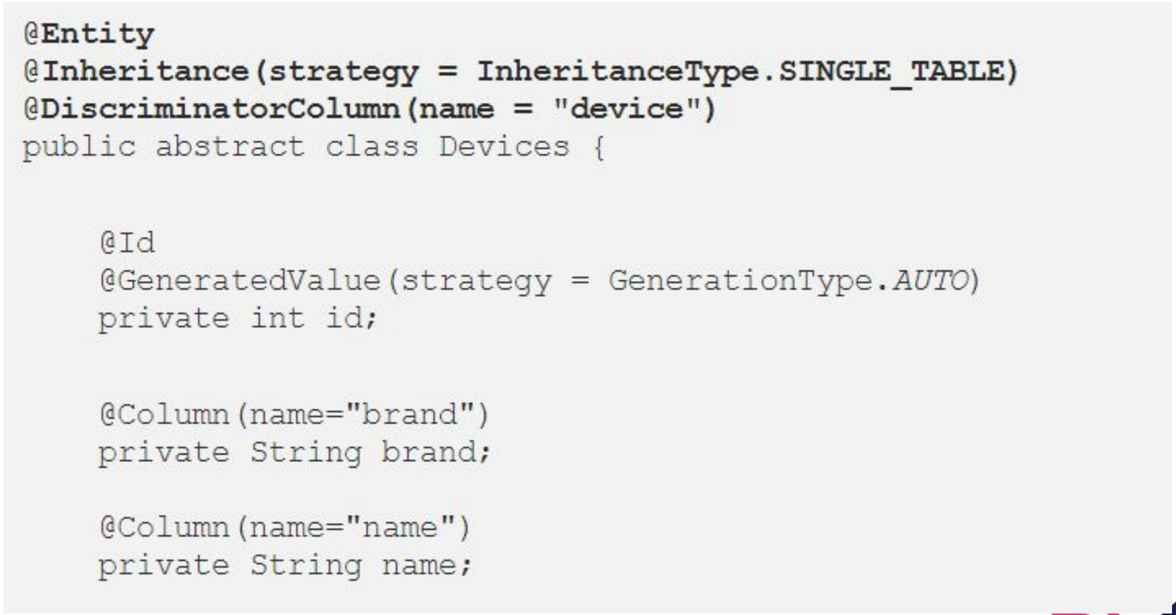
@MappedSuperclass

non-entity super class and you can create entity subclasses that inherit this super class. With this annotation, a separate table for each subclass is created. However, a table is not created for the super class itself.

```
@Entity
@Table(name = "computer")
public class Computer extends Devices {
   //Any new object and field here


}


-----------------------------------------------


@Entity
@Table(name = "mobile_phone")
public class MobilePhone extends Devices{
   //Any new object and field here
}
```

Single Table

create a single table as the main entity. It does not create separate tables for each entity. We add the @DiscriminatorValue notation to distinguish each entity. If we do not add it, Hibernate uses the entity name as a distinguishing value by default. Unused columns are created in an entity table and they take null values.



```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "device")
public abstract class Devices {


    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;


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

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

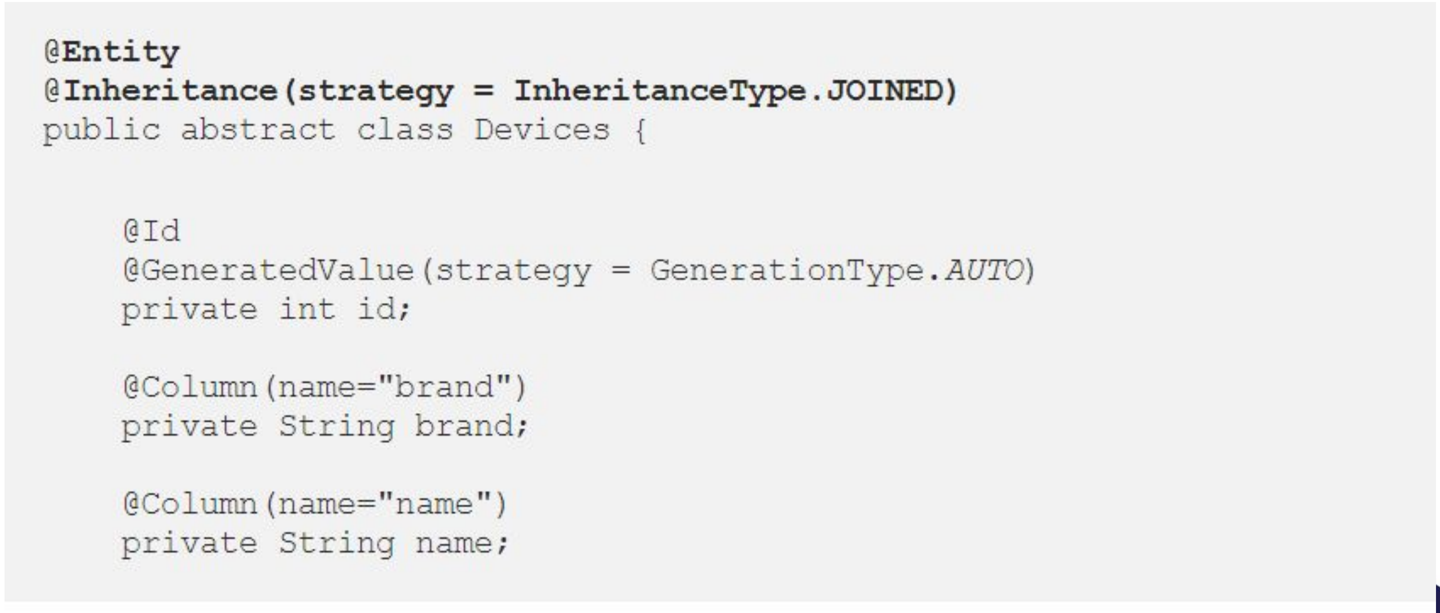
Single Table

create a single table as the main entity. It does not create separate tables for each entity. We add the @DiscriminatorValue notation to distinguish each entity. If we do not add it, Hibernate uses the entity name as a distinguishing value by default. Unused columns are created in an entity table and they take null values.

```
@Entity
@DiscriminatorValue("computer")
public class Computer extends Devices {
   private String oS;


-----------------------------------------------


@Entity
@DiscriminatorValue("mobilephone")
public class MobilePhone extends Devices{
   private String color;
```

Joined Table

Separate tables are created for super classes and subclasses. However, the objects defined in the super class are not included in the tables of the subclass. The primary key of the super class constitutes the foreign key of the subclasses. In addition, with @PrimaryKeyJoinColumn, we can specify a primary key for subclasses which is a foreign key of the super-class.



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


    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;

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

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

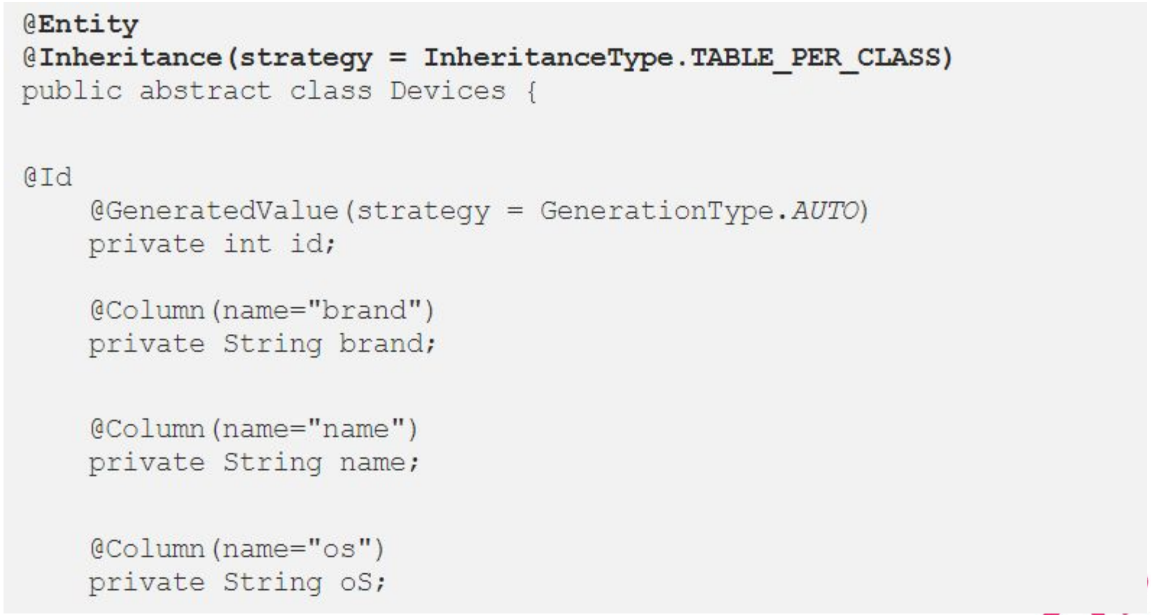
Joined Table

Separate tables are created for super classes and subclasses. However, the objects defined in the super class are not included in the tables of the subclass. The primary key of the super class constitutes the foreign key of the subclasses. In addition, with @PrimaryKeyJoinColumn, we can specify a primary key for subclasses which is a foreign key of the super-class.

```
@Entity
@PrimaryKeyJoinColumn(name = "computerId")
public class Computer extends Devices {
   private String oS;


-------------------------------------------


@Entity
@PrimaryKeyJoinColumn(name = "mobilephoneId")
public class MobilePhone extends Devices{
   private String color;
```

Table Per Class

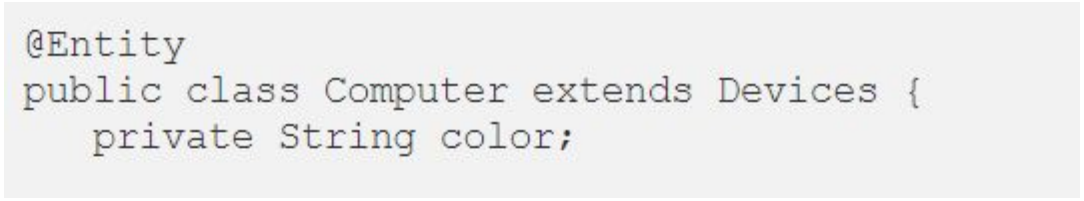
Similar to MappedSuperclass, but, the super class is also an entity. A table will also be created for the super class in the database. This inheritance structure allows us to create relationships with polymorphic queries and subclasses. But when querying, all subclasses are scanned, which slows down the performance very much. Therefore, this method should be avoided in jobs that require a lot of queries.



```java
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Devices {


@Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;

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


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


    @Column(name="os")
    private String oS;
```

Table Per Class

Similar to MappedSuperclass, but, the super class is also an entity. A table will also be created for the super class in the database. This inheritance structure allows us to create relationships with polymorphic queries and subclasses. But when querying, all subclasses are scanned, which slows down the performance very much. Therefore, this method should be avoided in jobs that require a lot of queries.



```
@Entity
public class Computer extends Devices {
   private String color;
```

CODECOOL