

OR Mappings

Java Persistence API (JPA)



Contents



Hibernate
JPA

- Introduction
- JPA Managed Classes
- OR Mappings
- Attribute Converters

Introduction

- Java မှာအသုံးပြုနေသော Object များနှင့် Relational Database မှာအသုံးပြုနေသော Relation တို့မှာ သဘောသဘာဝခြင်း မတူညီကြပါဘူး
- ORM ဆိုတာကတော့ Object တွေနဲ့ Relation တွေရဲ့ မတူညီတဲ့ အပိုင်းတွေကို Map လုပ်ထားခြင်းအားဖြင့် OOP Language တွေကနေ Database Operation တွေကို အလွယ်တကူလုပ်ဆောင်နိုင်စေနိုင်တဲ့ Tools တစ်ခုဖြစ်ပါတယ်
- Relational Database ထဲက Table တွေနဲ့ Map လုပ်ထားတဲ့ Java Object တွေကို Entity လို့ ခေါ်ဆိုကြပါတယ်
- ORM ကိုအသုံးပြုခြင်းအားဖြင့် SQL တွေကို တိုက်ရိုက်ရေးသားစရာမလိုတော့ပဲ၊ DBMS Specific ဖြစ်တဲ့ ရေးသားပုံတွေကို မရေးရတော့တဲ့အတွက် Database တွေပြောင်းသုံးတဲ့ အခါမှာ အလွယ်တကူဆောင်ရွက်နိုင်မှာ ဖြစ်ပါတယ်

JPA Managed Classes

- JPA Framework က Manage လုပ်နေတဲ့ Class ဖြစ်မှသာ OR Mapping တွေကို ရေးသား အသုံးပြုနိုင်မှာ ဖြစ်ပါတယ်
- Entity Class
Database Table တွေနဲ့ Map လုပ်ထားတဲ့ Class တွေဖြစ်ကြပါတယ်
- Embeddable
Embeddable Class ထဲက Field တွေနဲ့ Database Column တွေကို Map လုပ်ပြီး Entity တွေမှာ Embedded ဒါမှ မဟုတ် EmbeddedId အနေနဲ့ အသုံးပြုနိုင်ပါတယ်
- MappedSuperClass
Super Class ထဲမှာရှိတဲ့ Fields တွေကို Child Entity တွေမှာ Inheritance လုပ်ပြီး အသုံးပြုလိုတဲ့ အခါမှာ ရေးသား အသုံးပြုနိုင်ပါတယ် (Inheritance Mapping အရောက်မှာ ဖော်ပြသွားပါမယ်)

Entity Class

- Java Beans Technology အား လိုက်နာပြီး ရေးသားထားတဲ့ POJO Class တွေဖြစ်ရပါမယ်
- Top Level Class ဖြစ်ရန် လိုအပ်ပါတယ်
- Final Class တွေကို အသုံးပြုလို့မရပါ
- Default Constructor တော့ မဖြစ်မနေရှိနေဖို့ လိုအပ်ပါတယ်
- @Entity ကို Class မှာရေးသားထားပြီး @Id Field တစ်ခုကို ရေးသားထားရပါမယ်
- Default အတိုင်း အသုံးပြုမည် ဆိုပါက၊ အထွေအထူး Mapping များ မလိုအပ်ပါ
- Customize လုပ်ဖို့လိုတော့မှ Mapping တွေကို အသုံးပြုရေးသားရမှာ ဖြစ်ပါတယ်

Data Type of Entity Class


- Java primitive types
- java.lang.String
- Enumerated types
- Other entities and/or collections of entities
- Embeddable classes

JPA 2.2 ကနေစပြီး Java 8 ရဲ့ Date & Time API

ကို Default အတိုင်း အသုံးပြုလာနိုင်ပါတယ်

- Other serializable types
 - Wrappers of Java primitive types
 - java.math.BigInteger
 - java.math.BigDecimal
 - java.util.Date
 - java.util.Calendar
 - java.sql.Date
 - java.sql.Time
 - java.sql.Timestamp
 - User-defined serializable types
 - byte[]
 - Byte[]
 - char[]
 - Character[]

Sample Code



```
@Entity
public class Student implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    private int id;
    private String address;
    private Date birthDate;
    private String mail;
    private String name;
    private String nrcNumber;
    private String occupation;
    private String phone;

    public Student() {
    }

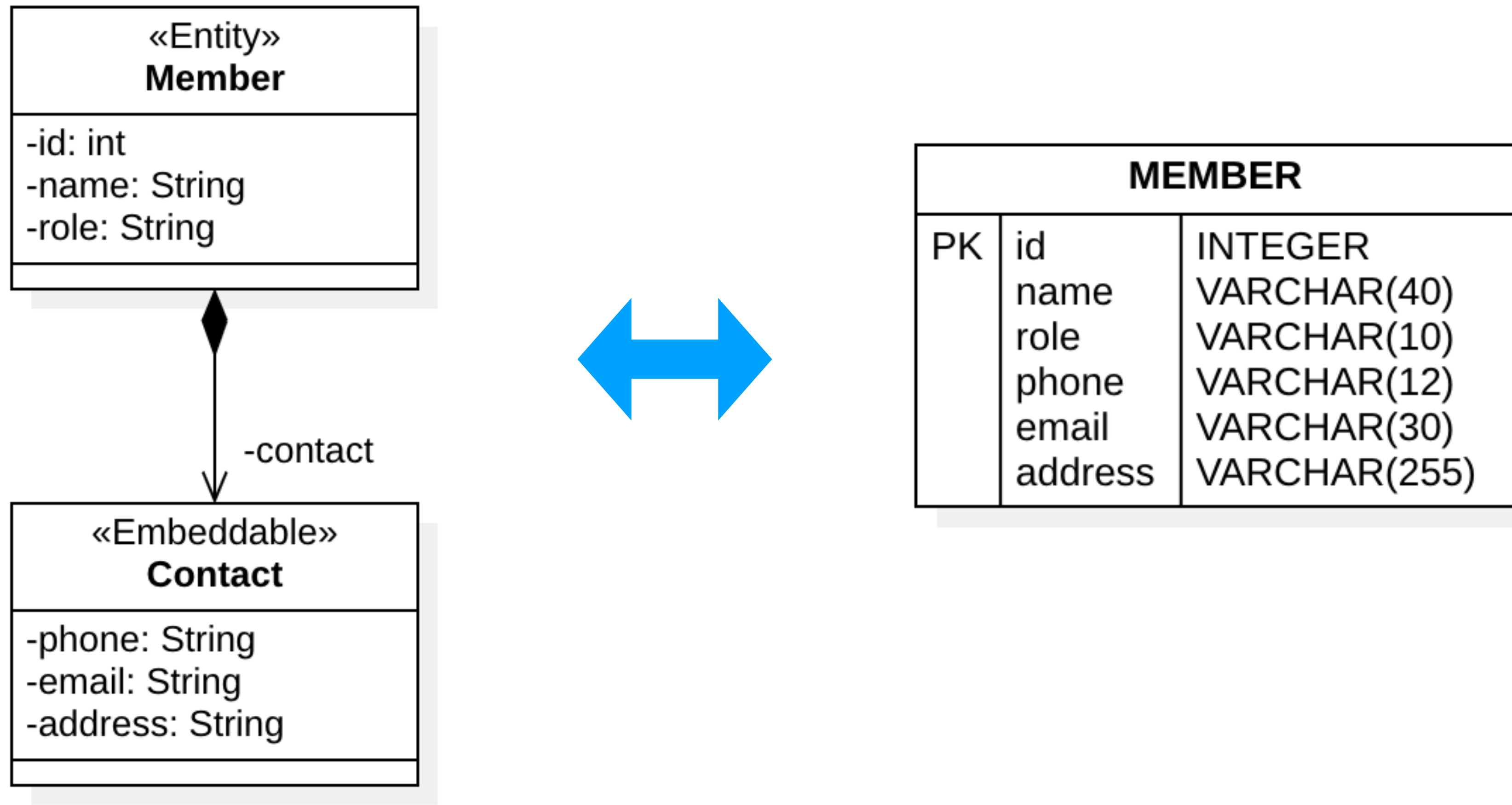
    // getter and setter

}
```

Embeddable Classes

- Embeddable ဟာ Entity လိုပဲ JPA က Manage လုပ်တဲ့ Class တစ်မျိုးဖြစ်ပါတယ်
- Entity တွေထဲမှာ Embeddable ကို ထည့်သွင်းရေးသားနိုင်ပြီး၊ Embeddable ထဲမှာပါတဲ့ Fields တွေအားလုံးဟာ Entity နဲ့ Map လုပ်ထားတဲ့ Table ထဲက Column တွေနဲ့ Map လုပ်ပေးသွားနိုင်မှာဖြစ်ပါတယ်
- Embeddable Class ထဲမှာရှိတဲ့ Fields တွေမှာလဲ Mapping တွေကို ရေးသားနိုင်ပါတယ်
- Embeddable တွေကို သုံးတဲ့အခါ Field Name တွေ တူသွားနိုင်တဲ့အတွက် သတိပြုသင့်ပါတယ်

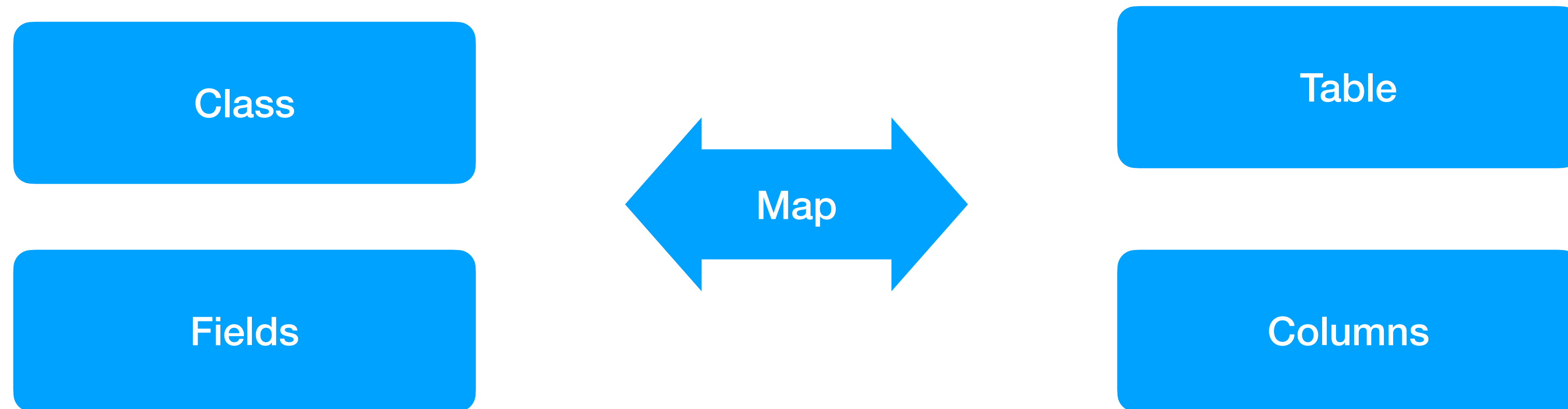
Embeddable Class



Writing Embeddable Class

- Embeddable Class တွေဟာ Top Level Class တစ်ခုဖြစ်ရပါမယ်
- Final Class မဟုတ်ရပါဘူး
- Default Constructor ကတော့ ရှိနေမှ ဖြစ်ပါတယ်
- Serializable Interface ကို Implement လုပ်ထားတာ ကောင်းပါတယ်
- Hash Code & Equals Method တွေကို Override လုပ်ထားသင့်ပါတယ်

OR Mapping



Java ထဲက Class တွေ Fields တွေနဲ့ Database ထဲက Table တွေ Column တွေကို Mapping လုပ်ပေးနိုင်ပါတယ်

OR Mappings

- Class Mapping

Entity Class တွေမှာ Default အတိုင်းဆိုရင် Class Name နဲ့ Database Table Name ကို Mapping လုပ်ပေးနိုင်ပြီး၊ လိုအပ်ပါက Custom Mapping တွေကို ရေးသားအသုံးပြုနိုင်ပါတယ်

- Fields Mapping

Entity Class ရဲ့ Field တွေကို Default အတိုင်းဆိုပါက Database Column ဖြင့် Map လုပ်ပေးနိုင်ပြီး၊ Primary Key ဖြင့် Map လုပ်လိုတဲ့ အခါ၊ ဒါမှဟုတ် Default အတိုင်း Map လုပ်လို့မရတဲ့ အခါမျိုးမှာ Custom Mapping တွေကို ရေးသားနိုင်ပါတယ်

- Relationship Mapping

Relational DBMS တွေမှာ Relationship တွေကို Foreign Key တွေကို အသုံးပြုပြီး သတ်မှတ်နိုင်ပါတယ်။ Table တွေရဲ့ Relationship နဲ့ ပတ်သက်တာတွေကို Specified လုပ်ချင်တဲ့အခါမျိုးတွေမှာ Relationship Mapping တွေကို ရေးသားနိုင်ပါတယ်

- Inheritance Mapping

OOP Language တွေမှာ Object တွေအချင်းချင်းချင်း Inheritance လုပ်လို့ရပါတယ်။ Inheritance လုပ်တဲ့အခါမှာ Table တွေကို ဘယ်လို Join မလဲ ဆိုတာကို Specified လုပ်ချင်တဲ့ အခါမျိုးတွေမှာ Inheritance Mapping ကို ရေးသားနိုင်ပါတယ်

Class Mappings

Mapping	Descriptions
	Table နဲ့ပတ်သက်တဲ့ Mapping တွေကို ရေးသားနိုင်ပါတယ်။
@Table	Table ရဲ့ catalog, name, schema, uniqueConstrients တွေကို Custom သတ်မှတ်လိုတဲ့ အခါမှာ အသုံးပြုနိုင်ပါတယ်။
@SecondaryTable	Entity Class ထဲက Field တွေကို အခြား Table ထဲက Column တွေနဲ့ Map လုပ်လိုတဲ့ အခါမှာ သတ်မှတ်ရေးသားနိုင်ပါတယ်
@SecondaryTables	Secondary Table တွေကို တစ်ခုထက်မက ရေးသားအသုံးပြုလိုတဲ့ အခါမှာ အသုံးပြုနိုင်ပါတယ်

@Table Mapping

```
@Entity
@Table(
    name = "MEMBER_TBL",
    indexes = {
        @Index(columnList = "role")
    },
    uniqueConstraints = {
        @UniqueConstraint(columnNames = {"email"})
    }
)
public class Member implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue
    private int id;
    private String name;

    @Embedded
    private Contact contact;

    // Getters & Setters

}
```

Table နဲ့ပတ်သက်တဲ့
Customization တွေကို ရေးသားလိုတဲ့
အခါ အသုံးပြုနိုင်ပါတယ်

@SecondaryTable Mappings

```
@Entity
@Table(
    name = "MEMBER_TBL"
)
@SecondaryTables({
    @SecondaryTable(
        name = "LOGIN_INFO",
        indexes = {
            @Index(columnList = "role")
        }
    ),
    @SecondaryTable(
        name = "CONTACT_INFO",
        uniqueConstraints = @UniqueConstraint(
            columnNames = "email"
        )
    )
})
public class Member implements Serializable {
    // Fields
    // Getters & Setters
}
```

တစ်ခုထက်မကသော
Table ထဲက Column တွေ့ Entity
ကို Map လုပ်ချင်တဲ့အခါမှာ သုံးနိုင်
ပါတယ်

Fields Mappings

Field Level Mapping	Descriptions
Primary Keys	Database ထဲက Primary Key Columns တွေနဲ့ Map လုပ်နိုင်တဲ့ Mapping တွေဖြစ်ပါတယ်
Columns	Database Column နှင့် Java Class ထဲက Fields တွေရဲ့ မတူညီတာတွေကို Map လုပ်နိုင်တဲ့ Mapping တွေဖြစ်ပါတယ်
Collections	Java မှာက Collection တွေကို Member အနေနဲ့ အသုံးပြုနိုင်ပြီး၊ Database Table မှာကတော့ Collection Type Column တွေမရှိတဲ့ အတွက် Collection တွေကို သုံးမယ်ဆိုရင် Mapping လုပ်ပေးဖို့လိုအပ်ပါတယ်

Primary Key Mappings

- **ID Key**

Entity Class တစ်ခုတွင် ID Key အနည်းဆုံး တစ်ခုလိုအပ်ပြီး၊ Table ၏ Primary Key Column နှင့် Mapping လုပ်ပေးပါသည်

- **Composite ID**

Key အကယ်၍ Primary Key Column အများရှိသော Table များဆိုပါက Composite ID Key အား အသုံးပြုရန် လိုအပ်ပါသည်

@Id Mapping

- Entity တိုင်းမှာ Primary Key နဲ့ Map လုပ်ထားတဲ့ ID Mapping တစ်ခုတော့ လိုအပ်ပါတယ်
- Database Table ထဲက Primary Key Column နှင့် Map လုပ်လိုတဲ့ Field မှာ @Id Annotation ဖြင့် Map လုပ်နိုင်ပါတယ်
- Number Type @Id Fields တွေမှာလဲ Generated Value ကို အသုံးပြုနိုင်အောင် Mapping တွေ ရေးသားနိုင်ပါတယ်
- Generation Type အလိုက် Custom Generator များကိုလဲ ရေးသားနိုင်ပါတယ်

@GeneratedValue

```
@Entity
public class Member {

    @Id
    @GeneratedValue
    private int id;
    private String name;
    private String phone;
    private String email;

    // Getters & Setters

}
```

ID Value ကို Generate လုပ်
မယ်လို့ သတ်မှတ်နိုင်

Generated Value Strategy

Strategy	Default	Description
IDENTITY		Auto Increment (Identity Column) ကို အသုံးပြုပြီး ID Column အတွက် Value ကို Generate လုပ်ပြီး အသုံးပြုပေးပါမယ်
SEQUENCE		Sequence Object ကို တည်ဆောက်ပြီး ID Column အတွက် Value ကို Generate လုပ်ပြီး အသုံးပြုပေးပါမယ်
TABLE		Sequence Table ကို တည်ဆောက်ပြီး ID Column အတွက် Value ကို Generate လုပ်ပြီး အသုံးပြုပေးပါမယ်
AUTO	✓	JPA Engine ကနေ SEQUENCE ဒါမှမဟုတ် TABLE ကို သင့်တော်သလို အသုံးပြုပေးပါမယ်

Custom Generators

- Generated Strategy တွေမှာ SEQUENCE ဒါမှမဟုတ် TABLE ကို အသုံးပြုထားရင် JPA Engine က Default အတိုင်း တည်ဆောက် အသုံးပြုသွားမှာ ဖြစ်ပါတယ်
- တစ်ခုထက်မကသော Entity တွေမှာ SEQUENCE တွေ TABLE တွေကို သုံးနေရင် တစ်ခုထဲသော SEQUENCE ဒါမှမဟုတ် TABLE ကနေ Sequence Number ကို Share လုပ်ပြီး သုံးနေမိကြမှာ ဖြစ်ပါတယ်
- JPA မှာ Custom Sequence Generator တွေကို တည်ဆောက်နိုင်အောင်လဲပြင်ပေးထားပါတယ်

Table Generator

```

@Entity
public class Member implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(generator = "member_seq")
    @TableGenerator(name = "member_seq", allocationSize = 1, initialValue = 2000)
    private int id;
    private String name;

    // Other Fields
    // Getters & Setters
}

```

@GeneratedValue ရဲ့ generator တွေမှာ Custom TableGenerator တွေကို သတ်မှတ်ရေးသားပြီး
Entity တစ်ခုအတွက် သီးသန့် Sequence Table တွေကို အသုံးပြုနိုင်ပါတယ်

Sequence Generator



```
@Entity
public class Company {

    @Id
    @GeneratedValue(generator = "company_seq")
    @SequenceGenerator(name = "company_seq", allocationSize = 1, initialValue = 1001)
    private int id;
    private String name;

    // Other Fields
    // Getters & Setters
}
```

@GeneratedValue ရဲ့ generator တွေမှာ Custom SequenceGenerator တွေကို သတ်မှတ်ရေးသားပြီး
Entity တစ်ခုအတွက် သီးသန့် Sequence Object တွေကို အသုံးပြုနိုင်ပါတယ်

Composite ID

- Primary Key Column တွေကို တစ်ခုထက်မက အသုံးပြုလိုတဲ့ အခါမှာ Composite ID Key များအား အသုံးပြုနိုင်ပါတယ်
- Composite ID Key ကိုပုံစံနှစ်မျိုးဖြင့် ရေးသားနိုင်ပါတယ်
 - @Embeddable နှင့် @EmbeddedId Annotation အား အသုံးပြုနည်း
 - @IdClass နှင့် @Id Annotation အား အသုံးပြုနည်း

Embedded ID

```
@Embeddable
public class TransactionId
    implements Serializable{

    @Temporal(TemporalType.TIMESTAMP)
    private Date date;
    private long customer;
    private long item;
    // getter and setter
}
```

```
@Entity
public class Transaction
    implements Serializable{

    @EmbeddedId
    private TransactionId id;
    private int count;
    @Temporal(TemporalType.DATE)
    private Date dilivary;
    // getter and setter
}
```

Embedded Id အဖြစ် အသုံးပြုလိုသည့် Class အား @Embeddable Annotation အား အသုံးပြု၍ ရေးသားရန် လိုအပ်ပါသည်။

Embeddable Class သည် Serializable Interface အား Implements လုပ်ထားရန် လိုအပ်ပြီး၊ Top Level Class တစ်ခု ဖြစ်ရန် လိုအပ်ပါသည်။ hashCode နှင့် equals method များအား Override လုပ်၍ ရေးသားရန် လိုအပ်ပါသည်။

Embeddable Class အတွင်းမှ Member များသည် Entity နှင့် Mapping လုပ်ထားသည့် Table အတွင်း Primary Key များအနေနှင့် တည်ဆောက်သွားမည် ဖြစ်ပါသည်။

Id Class

- Id Class အဖြစ်အသုံးပြုလိုသော Class သည် Serializable Interface အား Implements လုပ်ထားရန် လိုအပ်ပြီး၊ Top Level Class တစ်ခု ဖြစ်ရန် လိုအပ်ပါသည်။ hashCode နှင့် equals method များအား Override လုပ်၍ ရေးသားရန် လိုအပ်ပါသည်။
- Id Class အား အသုံးပြုလိုသည့် Entity တွင် @IdClass အား ရေးသားထားရန် လိုအပ်ပြီး ၎င်းတွင် အသုံးပြုလိုသည့် Class အမည်အား တန်ဖိုးအဖြစ် ဖော်ပြထားရန် လိုအပ်ပါသည်။
- ထို့အပြင် Id Class အတွင်းမှ Attribute များအားလုံးကိုလည်း Entity အတွင်းတွင် @Id Annotation တပ်၍ ရေးသားရပါမည်။

Using @IdClass

```
public class TransactionId
    implements Serializable{

    private Date date;
    private long customer;
    private long item;
    // getter and setter
}
```

```
@Entity
@IdClass(value=TransactionId.class)
public class Transaction
    implements Serializable{

    @Id
    @Temporal(TemporalType.TIMESTAMP)
    private Date date;
    @Id
    private long customer;
    @Id
    private long item;
    private int count;
    @Temporal(TemporalType.DATE)
    private Date dilivary;
    // getter and setter
}
```

အကယ်၍ ရှိပြီးသား Class တစ်ခုအား Composite ID အနေနှင့် အသုံးပြုလိုပြီး၊ ၎င်းအား ပြုပြင်၍ မရနိုင်သောအခါ @IdClass Annotation အား အသုံးပြုနိုင်ပါသည်။

Columns Mappings

Annotations	Parameters	Descriptions
@Basic	fetch optional	အရိုးရှင်းဆုံး Map လုပ်ခြင်း
@Column	name length unique nullable precision scale insertable updatable table columnDefination	Column ရဲ့ထူးခြားချက်များနှင့် Attribute အား Map လုပ်လိုသည့်အခါ
@Transient		Map မလုပ်လိုသည့် Attribute အတွက်
@Temporal	value	Date သို့ Claender ကို သုံးသည့်အခါ
@Enumerated	value	Enum ကို အသုံးပြုလိုသည့်အခါ
@Lob		Binary Data ပြောင်းသိမ်းလိုတဲ့အခါမှာ

Sample Codes

```
@Entity
public class Blog
    implements Serializable{

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

    @Column(name="BLOG_TITLE",
        nullable=false)
    private String title;
    @Lob
    @Basic(fetch=FetchType.LAZY)
    private String body;
    @Temporal(TemporalType.DATE)
    private Date date;
    @Enumerated
    private Status status;
    @Transient
    private boolean check;
    // getter and setter
}
```

```
CREATE TABLE BLOG (
    ID BIGINT
        GENERATED BY DEFAULT AS IDENTITY
        NOT NULL,

    BODY CLOB(2147483647),
    DATE DATE,
    STATUS INTEGER,
    BLOG_TITLE VARCHAR(255) NOT NULL,
    PRIMARY KEY (ID))
```

Collection Mappings

- Entity Class အတွင်းတွင် Collections များအားလည်း Attribute အနေနှင့် အသုံးပြုနိုင်ပါသည်။
- အသုံးပြုနိုင်သော Collection များမှာ အောက်ပါအတိုင်း ဖြစ်ကြသည်
 - java.util.Set
 - java.util.List
 - java.util.Map
- Collection အတွင်း ထည့်သွင်း အသုံးပြုနိုင်ကြသည်များမှာ အခြားသော Entity များ၊ Embeddable များနှင့် Basic Java Type များ ဖြစ်ကြသည်။
- Entity Class ထဲမှာ Collection Type Fields တွေကို ရေးထားပြီးဆိုရင် @ElementCollection ကို အသုံးပြုပြီး Map လုပ်ပေးရမှာ ဖြစ်ပါတယ်

@ElementCollection

```

@Entity
public class Product implements Serializable{

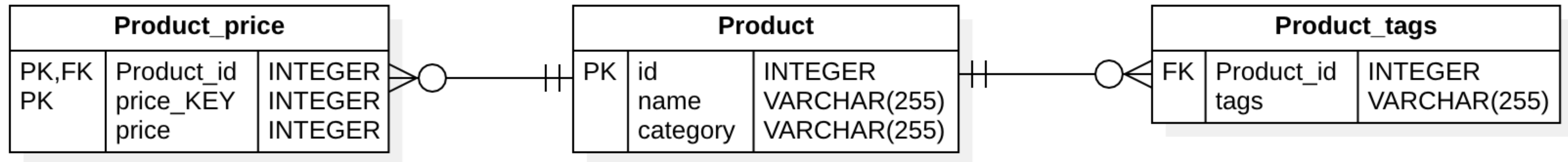
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;
    private String category;
    @ElementCollection
    private Map<PriceType, Integer> price;
    @ElementCollection
    private List<String> tags;

    // Getters & Setters

}
```


Collections Mapping



@ElementCollection ဖြင့် Map ကို အသုံးပြု
ထားရင် Owner Table ထဲက Primary Key ကို
Join Table ရဲ့ Primary Key , Foreign Key
အနေနဲ့ အသုံးပြုပြီး၊ Map Key ကို Primary
Key အနေနဲ့ သုံးပြီး Map လုပ်ပေးပါမယ်

@ElementCollection ဖြင့် Set ဒါမှမဟုတ်
List ကို အသုံးပြုထားရင် Owner Table ထဲက
Primary Key ကို Join Table ရဲ့ Foreign Key
အနေနဲ့ အသုံးပြုပြီး Map လုပ်ပေးပါမယ်

Other Mappings for Collections

Mappings	Definition
@CollectionTable	Collection Table နဲ့ပတ်သက်တဲ့ Table Name, Join Column Information တွေကို ပြင်ရေးလိုတဲ့အခါမှာ အသုံးပြုနိုင်ပါတယ်
@MapKeyColumn	Collection Table ရဲ့ Map Key Column Name ကို ပြင်ရေးလိုတဲ့အခါမှာ အသုံးပြုနိုင်ပါတယ်
@MapKeyEnumerated	Map Key ရဲ့ Type ဟာ Enum ဖြစ်နေပြီး Collection Table ရဲ့ Map Key Column Type ကို ပြင်ရေးလိုတဲ့အခါမှာ အသုံးပြုနိုင်ပါတယ်
@OrderBy	Collection ထဲက Element ရဲ့ Order ကို သတ်မှတ်လိုတဲ့အခါမှာ အသုံးပြုနိုင်ပါတယ်
@OrderColumn	Collection ထဲက Element တွေရဲ့ Order ကို ထိမ်းသိမ်းဖို့အတွက် Column တစ်ခုကို အသုံးပြုလိုတဲ့ အခါမျိုးမှာ ရေးသားနိုင်ပါတယ်။ @OrderColumn ကို ရေးထားရင် @OrderBy က အလုပ်မလုပ်ပါဘူး

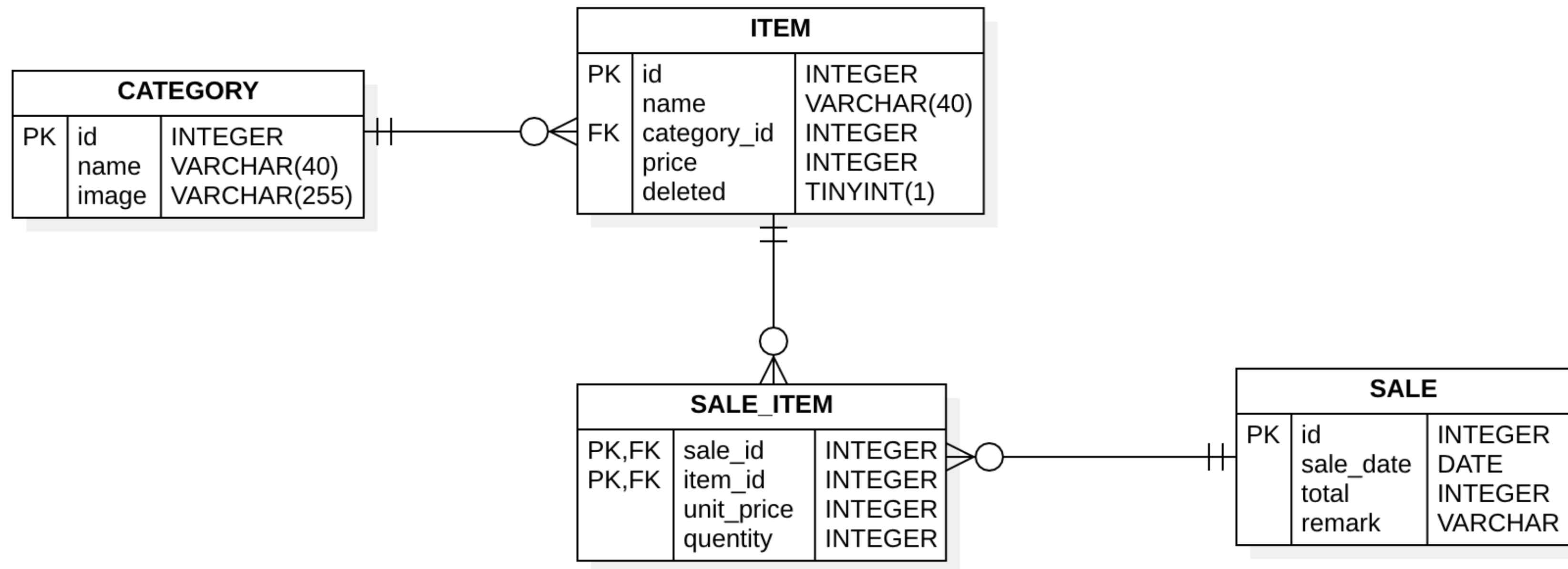
Embeddable & Element Collections

- Embeddable Type တွေကို Collection အနေနဲ့ Entity ထဲမှာ ရေးသားလို့ရနိုင်ပါတယ်
- တဖန် Embeddable Class တွေထဲမှာလဲ Element Collection ကို အသုံးပြုပြီး ရေးသားနိုင်ပါတယ်
- ဒါပေမဲ့ Collection အနေနဲ့ အသုံးပြုနေတဲ့ Embeddable Class တွေထဲမှာတော့ Element Collection ကို ရေးသားထားလို့ မရနိုင်ပါဘူး

Relationship Mappings

- Entity တစ်ခုထဲကနေ အခြား Entity တစ်ခုကို Reference လုပ်နေပြီဆိုရင် Relationship Annotation တွေကို အသုံးပြုပြီး Mapping တွေကို ရေးပေးရပါမယ်
- Reference လုပ်နေတဲ့ Entity က Single Object ဖြစ်နေရင်တော့ One နဲ့ ဆုံးတဲ့ Relationship တွေဖြစ်ကြတဲ့ @OneToOne ဒါမှမဟုတ် @ManyToOne တို့ကို သတ်မှတ်ပေးရမှာ ဖြစ်ပါတယ်
- Reference လုပ်နေတဲ့ Entity က Collection Object ဖြစ်နေရင်တော့ Many နဲ့ ဆုံးတဲ့ Relationship တွေဖြစ်ကြတဲ့ @OneToMany ဒါမှမဟုတ် @ManyToMany တို့ကို သတ်မှတ်ပေးရမှာ ဖြစ်ပါတယ်
- တဖန် Database Table တွေကို ဘယ်လို Join စေချင်တာလဲ ဆိုတာကိုလဲ Relationship Annotation တွေမှာ သတ်မှတ်ရေးသားပေးနိုင်ပါတယ်

Relationships in Database



Relational DB တွေမှာ Table တစ်ခုရဲ့ FK Column ကနေ အခြား Table ရဲ့ PK ကို Reference လုပ်ပြီး Relation တွေရဲ့ Relationship တွေကို သတ်မှတ်နေကြပါတယ်။ တဖန် Relationship ရှိတဲ့ Table တွေက Data တွေကို ရယူလိုတဲ့ အခါမှာ Reference တွေကနေ Column နဲ့ Join ပြီးရှာမလား၊ ကြားထဲမှာ Join Table ကို ခံပြီး Join မလား ဆိုပြီးလုပ်လေ့ရှိပါတယ်။

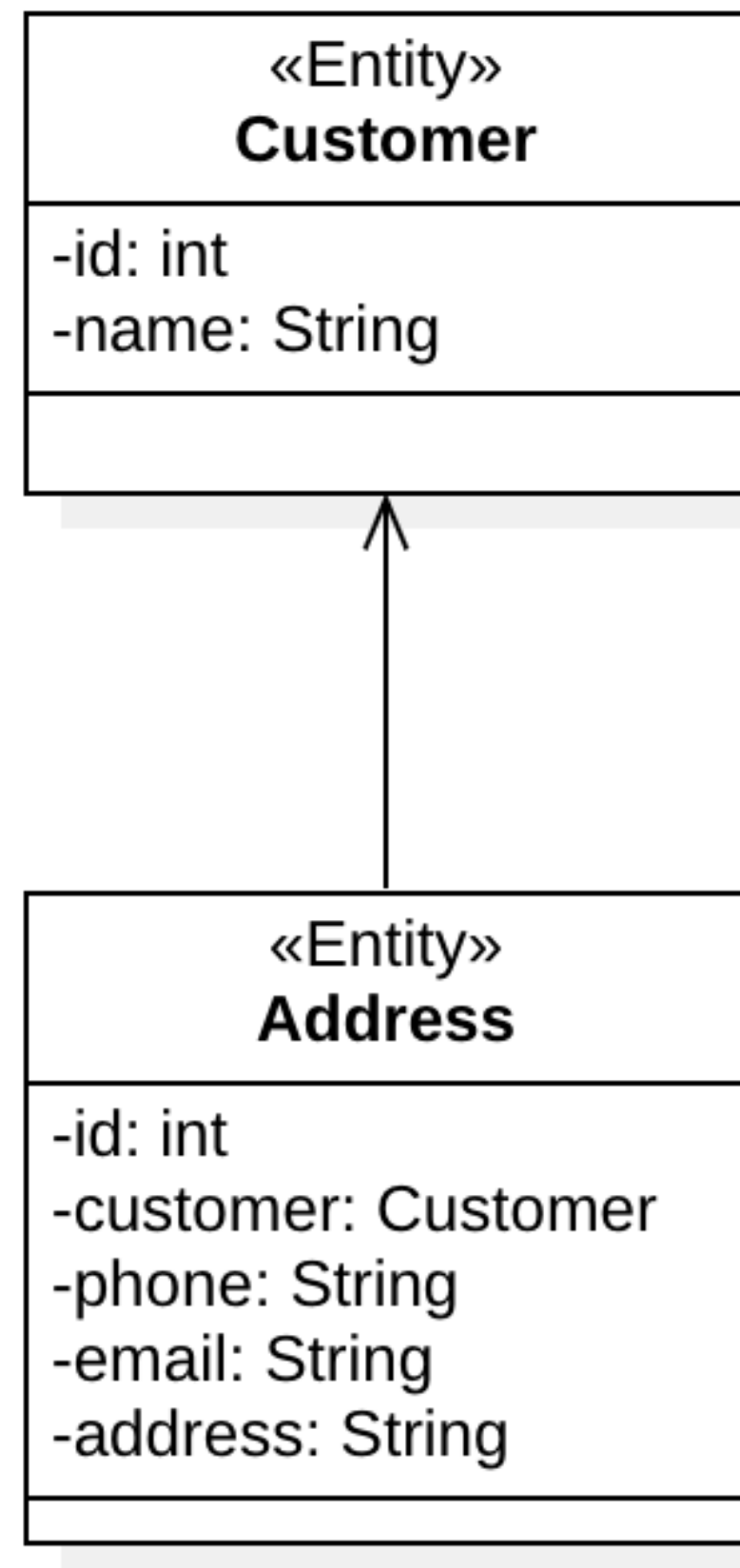
Relationship Mappings

Mappings	Default Join Strategy	Description
@OneToOne	@JoinColumn	Entity တစ်ခုက အခြား Entity တစ်ခုကို One to One Relationship နဲ့ Map လုပ်လိုတဲ့ အခါမှာ အသုံးပြုနိုင်ပါတယ်
@ManyToOne	@JoinColumn	Entity တစ်ခုက အခြား Entity တစ်ခုကို Many to One Relationship နဲ့ Map လုပ်လိုတဲ့ အခါမှာ အသုံးပြုနိုင်ပါတယ်
@OneToMany	@JoinTable	Entity တစ်ခုက တစ်ခုထက်မကသော အခြား Entity တစ်ခုကို One to Many Relationship နဲ့ Map လုပ်လိုတဲ့ အခါမှာ အသုံးပြုနိုင်ပါတယ်
@ManyToMany	@JoinTable	Entity တစ်ခုက တစ်ခုထက်မကသော အခြား Entity တစ်ခုကို Many to Many Relationship နဲ့ Map လုပ်လိုတဲ့ အခါမှာ အသုံးပြုနိုင်ပါတယ်

Join Strategy

	Join Columns	Join Table	Primary Key Join Columns	Map By
@OneToOne	✓	✓	✓	✓
@ManyToOne	✓	✓		
@OneToMany	✓	✓		✓
@ManyToMany		✓		✓

Relationship to Single Object



- Entity တစ်ခုထဲက Member Field တစ်ခုက အခြား Entity ဖြစ်နေပြီး Single Object ဖြစ်နေရင် One နဲ့ ဆုံးတဲ့ Relationship တစ်ခုခုဖြစ်ဖို့လိုပါတယ်
- @ManyToOne ကို သုံးသင့်သလား @OneToOne ကို သုံးသင့်သလားဆိုတာ ကတော့ မိမိရဲ့ Application သဘောတရားအပေါ်မှာပဲ မူတည်ပါတယ်
- Address Entity ကနေ Customer Entity ကို Reference လုပ်နေတယ် ဆိုကြ ပါစို့။ Customer တစ်ယောက်မှာ Address တစ်ခုသာရှိရမယ် ဆိုရင် @OneToOne ကို အသုံးပြုနိုင်ပေမဲ့၊ Customer တစ်ယောက်က Address အများကြီးရှိနေနိုင်တယ် ဆိုရင် @ManyToOne ပဲ ဖြစ်မှ ရပါမယ်။

@ManyToOne

- Entity တစ်ခုထဲက Member Field တစ်ခုက အခြား Entity ဖြစ်နေပြီး Single Object တစ်ခုကို @ManyToOne Relationship နဲ့ သတ်မှတ်လိုက်ပြီဆိုရင် Default အတိုင်းဆိုပါက Owner Table ထဲမှာ Foreign Key ကို တည်ဆောက်ပြီး Reference လုပ်နေတဲ့ Entity ရဲ့ Primary Key Column ကို Reference လုပ်တဲ့ ပုံစံနဲ့ Join ပေးပါမယ်
- @ManyToOne ရဲ့ Default Join Strategy ဟာ @JoinColumn ဖြစ်ပါမယ်
- Application ရဲ့ လိုအပ်ချက်အရ Join Table ကို အသုံးပြုပြီး Join လိုတယ်ဆိုရင်လဲ @JoinTable ကို အသုံးပြုပြီး Default Join Strategy ကို ပြောင်းလဲ သတ်မှတ်နိုင်ပါတယ်

Using @ManyToOne

```
@Entity
@Table(name = "fees")
public class CourseFees implements Serializable{

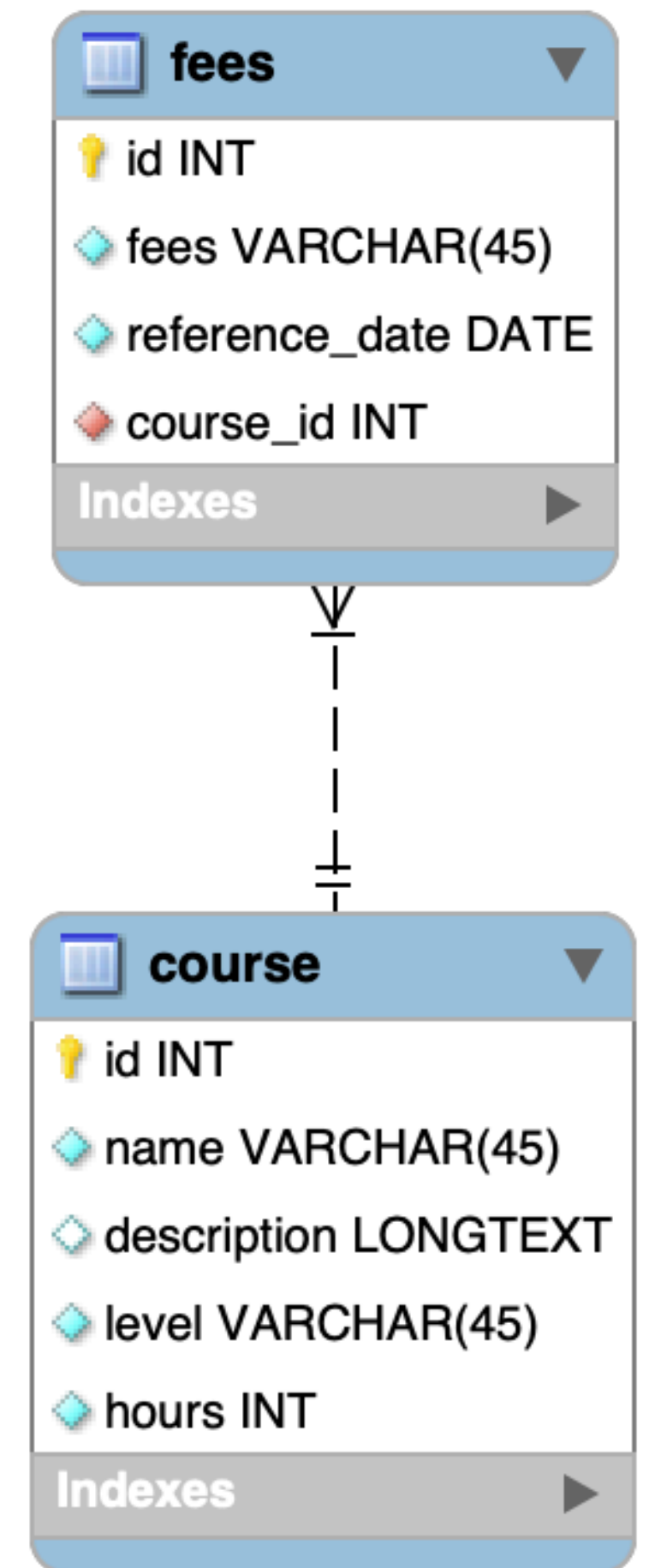
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    @Column(nullable = false)
    private int fees;
    @Column(name = "reference_date", nullable = false)
    private LocalDate refDate;

    @ManyToOne(optional = false)
    private Course course;

    // Getters and Setters

}
```



@OneToOne

- Entity တစ်ခုထဲက Member Field တစ်ခုက အခြား Entity ဖြစ်နေပြီး Single Object တစ်ခုကို @OneToOne Relationship နဲ့ သတ်မှတ်လိုက်ပြီဆိုရင် Default အတိုင်းဆိုပါက Owner Table ထဲမှာ Foreign Key ကို တည်ဆောက်ပြီး Reference လုပ်နေတဲ့ Entity ရဲ့ Primary Key Column ကို Reference လုပ်တဲ့ ပုံစံနဲ့ Join ပေးပါမယ်
- @OneToOne ရဲ့ Default Join Strategy ဟာ @JoinColumn ဖြစ်ပါမယ်
- @OneToOne မှာ Join Strategy အနေနဲ့ @JoinTable ကိုကော @PrimaryKeyJoinColumn ကိုပါ အသုံးပြုပြီး ရေးသားနိုင်ပါတယ်

@PrimaryKeyJoinColumn

- @PrimaryKeyJoinColumn ကိုတော့ @OneToOne Relationship တွေမှာပဲ ရေးသားအသုံးပြုနိုင်တာဖြစ်ပါတယ်
- Table ထဲက Primary Key ကို အသုံးပြုပြီး Reference Table ထဲက Primary Key ကို Reference လုပ်ပြီး Join ပေးမှာ ဖြစ်ပါတယ်
- @PrimaryKeyJoinColumn ကို အသုံးပြုမယ်ဆိုရင်တော့ Table နှစ်ခုစလုံးမှာ Primary Key ကို Share လုပ်ပြီး အသုံးပြုပေးဖို့လိုတယ် ဆိုတာကို သတိထားဖို့လိုအပ်ပါတယ်
- @MapsId Annotation ကို အသုံးပြုပြီး Reference Entity ထဲက Primary Key နဲ့ Owner Entity ရဲ့ ID ကို Map လုပ်ပြီးလဲ အသုံးပြုနိုင်ပါတယ်

Using @OneToOne

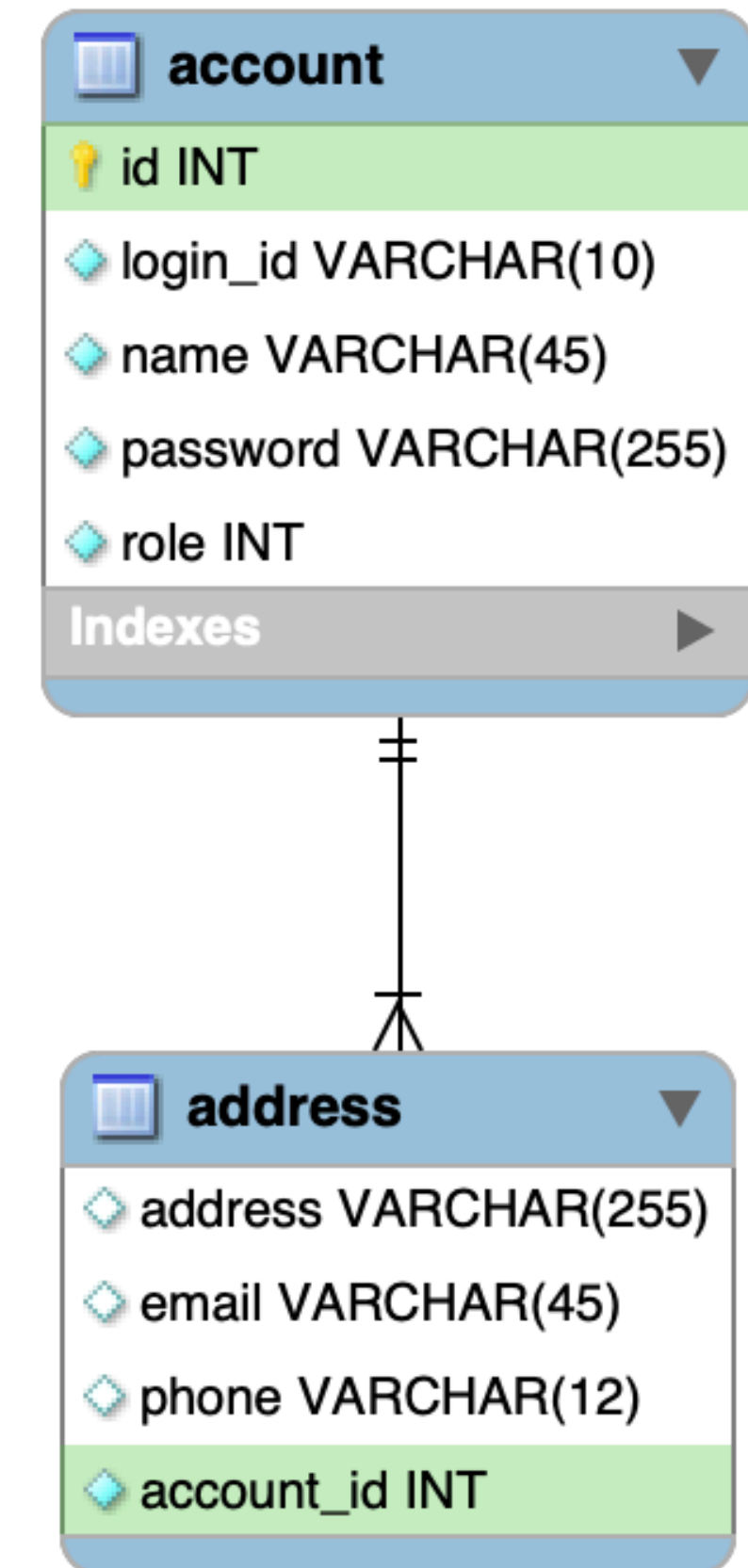
```
@Entity
@Table(name = "address")
public class Address implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    private int id;
    private String address;
    @Column(length = 12)
    private String phone;
    @Column(length = 45)
    private String email;

    @MapsId
    @OneToOne(optional = false)
    @PrimaryKeyJoinColumn
    private Account account;

    // Getters and Setters
}
```



Relationship to Collection of Entity

- Entity တစ်ခုထဲက Member Field တစ်ခုက အခြား Entity ဖြစ်နေပြီး Collection Object ဖြစ်နေရင် Many နဲ့ ဆုံးတဲ့ Relationship တစ်ခုဖြစ်ဖို့လိုပါတယ်
- Project Management System တစ်ခုကို ရေးကြပြီဆိုကြပါစို့။ Project တစ်ခုမှာ Task တွေက အများကြီးရှိနေနိုင်သလို၊ Project တစ်ခုမှာ Member တွေကလဲ အများကြီးရှိနေနိုင်ပါတယ်
- Project ဘက်က Task ကို ကြည့်မယ်ဆိုရင် Task တစ်ခုဟာ Project တစ်ခုနဲ့ သာဆိုင်ပြီး၊ Project တစ်ခုမှာ Task တွေ အများကြီးရှိနေနိုင်တဲ့ အတွက် @OneToMany Relationship ကို အသုံးပြုသင့်ပါတယ်
- Project နဲ့ Member ကိုကြည့်မယ်ဆိုရင် Project တစ်ခုမှာ Member တွေက အများကြီးရှိနေနိုင်ပြီး၊ Member တစ်ယောက်ဟာ Project တစ်ခုထက်မကမှာ ပါဝင်ပတ်သက်နေနိုင်တဲ့ အတွက် @ManyToMany Relationship ကို အသုံးပြုသင့်ပါတယ်

@OneToMany

- Entity တစ်ခုရဲ့ Member က အခြား Entity တစ်ခုရဲ့ Collection ဖြစ်နေရင် @OneToMany Relationship ကို သတ်မှတ်နိုင်ပါတယ်
- @OneToMany ရဲ့ Default Join Strategy ဟာ @JoinTable ဖြစ်ပါတယ်
- @OneToMany Relationship ရဲ့ Join Strategy အနေနဲ့ @JoinTable နဲ့ @JoinColumn တို့ကို ရွေးချယ် သတ်မှတ်နိုင်ပါတယ်
- @OneToMany ရဲ့ @JoinColumn က အခြား Relationship တွေနဲ့ မတူပါဘူး။ Foreign Key ကို Reference Entity ထဲမှာ တည်ဆောက်ပြီး Join ပေးတာ ဖြစ်ပါတယ်

Using @OneToMany

```

@Entity
@Table(name = "course")
public class Course implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    @Column(length = 45)
    private String name;
    @Enumerated(EnumType.STRING)
    private Level level;
    private int hours;

    @OneToMany
    private List<Section> sections;

    // Getters & Setters
}

```


@ManyToMany

- Entity တစ်ခုရဲ့ Member က အခြား Entity တစ်ခုရဲ့ Collection ဖြစ်နေရင် @ManyToMany Relationship ကို သတ်မှတ်နိုင်ပါတယ်
- @ManyToMany Relationship ကို @JoinTable Join Strategy နဲ့သာ သတ်မှတ်နိုင်မှာ ဖြစ်ပါတယ်
- Join Table ကို အသုံးပြုပြီး Join မှသာ @ManyToMany Relationship ကို သတ်မှတ် နိုင်မှာ ဖြစ်ပါတယ်

Using @ManyToMany

```

@Entity
@Table(name = "class")
public class Section implements Serializable {

    private static final long serialVersionUID = 1L;

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

    private LocalDate startDate;
    private LocalTime startTime;
    private LocalTime endTime;
    private double months;

    @ManyToMany
    private List<Account> teachers;

    // Getters & Setters
}
```

Bi-directional Relationship

- Entity တစ်ခုကနေ အခြား Entity တစ်ခုကို Relationship သတ်မှတ်ပြီး Reference လုပ်နေတာကို Uni-Directional Relationship လို့ ခေါ်ဆိုလေ့ရှိပါတယ်
- Relationship တစ်ခုအပေါ်မှာ Entity တစ်ခုနဲ့ အခြား Entity တစ်ခုက အပြန်အလှန် Reference လုပ်နေတာကို Bi-Directional Relationship လို့ ခေါ်လေ့ရှိကြပါတယ်
- အဓိကကတော့ ရှိပြီးသား Relationship ကို အသုံးပြုပြီး အခြားတစ်ဘက်က Map By နဲ့ သတ်မှတ်ထားမှသာ Bi-Directional Relationship ဖြစ်မှာ ဖြစ်ပါတယ်

Relationship Owner Class

```

@Entity
@Table(name = "fees")
public class CourseFees implements Serializable{

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    @Column(nullable = false)
    private int fees;
    @Column(name = "reference_date", nullable = false)
    private LocalDate refDate;

    @ManyToOne(optional = false)
    private Course course;

    // Getters & Setters
}
```

Mapped By Class

```

@Entity
@Table(name = "course")
public class Course implements Serializable {

    private static final long serialVersionUID = 1L;

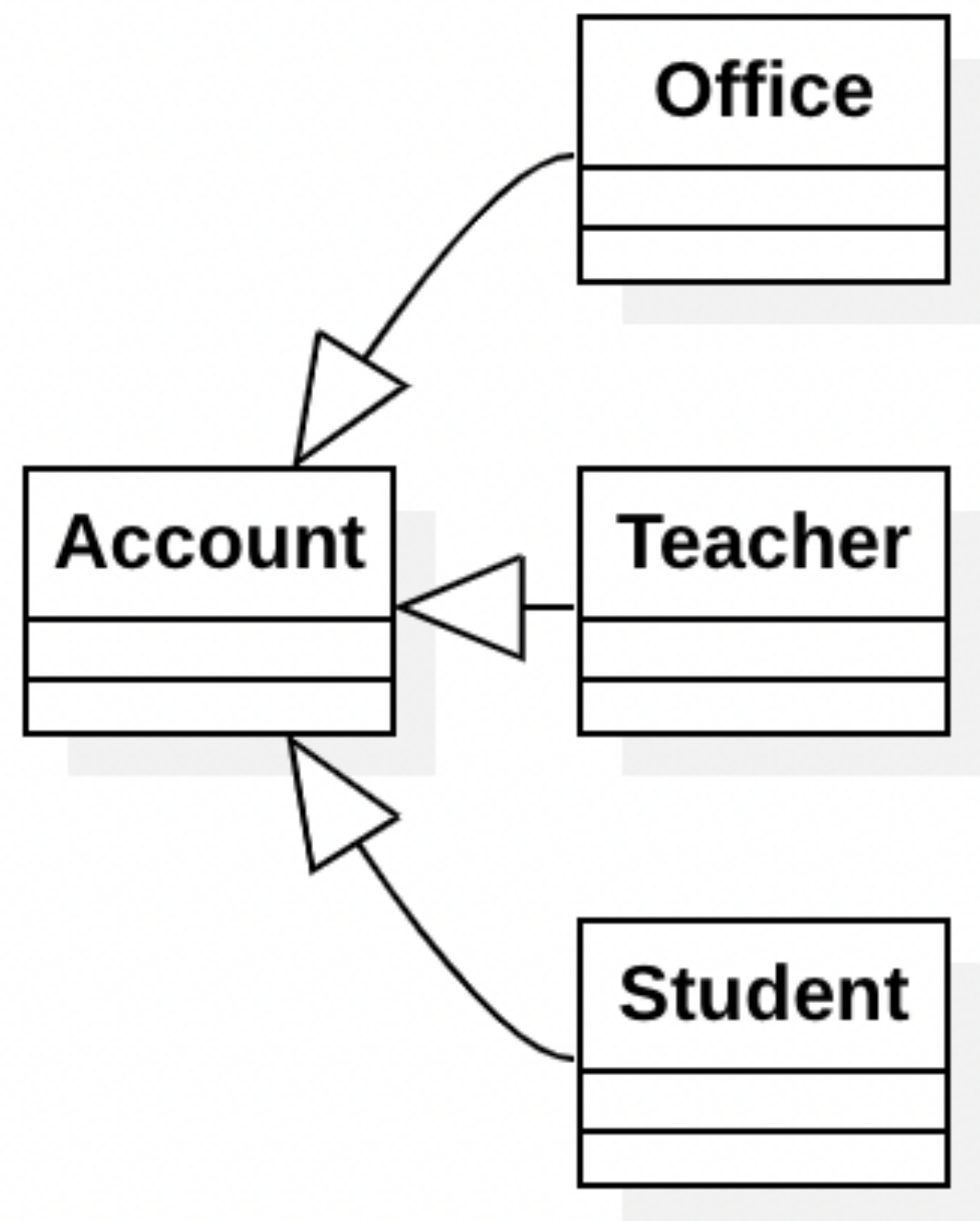
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    @Column(length = 45)
    private String name;
    private Level level;
    private int hours;

    @OneToMany(mappedBy = "course")
    private List<CourseFees> fees;

    // Getters & Setters

}
```

Inheritance Mapping



- OOP Language တွေမှာ Inheritance ဟာ အရေးပါတဲ့ Feature တစ်ခုဖြစ်ပြီး၊ IS-A Relationship ကို သတ်မှတ်ပေးနိုင်ပါတယ်
- ဒါပေမဲ့ Relational Database တွေမှာတော့ IS-A Relationship ကို သတ်မှတ်လို့ မရနိုင်ပါဘူး
- Inheritance ရှိတဲ့ နေရာတွေအတွက် @MappedSuperclasses နဲ့ @Inheritance Annotation ကို အသုံးပြုပြီး Map လုပ်ပေးနိုင်ပါတယ်

@MappedSuperClasses

- @MappedSuperClasses တွေကို Inheritance Hierarchy ထဲမှာရှိတဲ့ Supper Class တွေမှာ ရေးသားနိုင်ပါတယ်
- @MappedSuperClasses တွေဟာ Table တစ်ခုနဲ့ Map လုပ်ဖို့မဟုတ်ပဲ Child Entity တွေအတွက် Common Fields တွေကို Provide လုပ်ပေးနိုင်ပါတယ်
- @MappedSuperClasses တွေဟာ JPA ကနေ Manage လုပ်တဲ့ Class အမျိုးအစား တစ်မျိုးဖြစ်တဲ့ အတွက် OR Mapping တွေကို ရေးသားထားနိုင်ပါတယ်
- @MappedSuperClasses ရဲ့ Child Entity တွေမှာ ဘုံအနေနဲ့ တူညီတဲ့ Column တွေကို Inheritance လုပ်ပေးနိုင်ပေမဲ့ Database Level မှာတော့ အဲဒီ Table တွေအကြားမှာ ဘယ်လို Relationship မှ ရှိနေမှာ မဟုတ်ပါဘူး

Using @MappedSuperClass

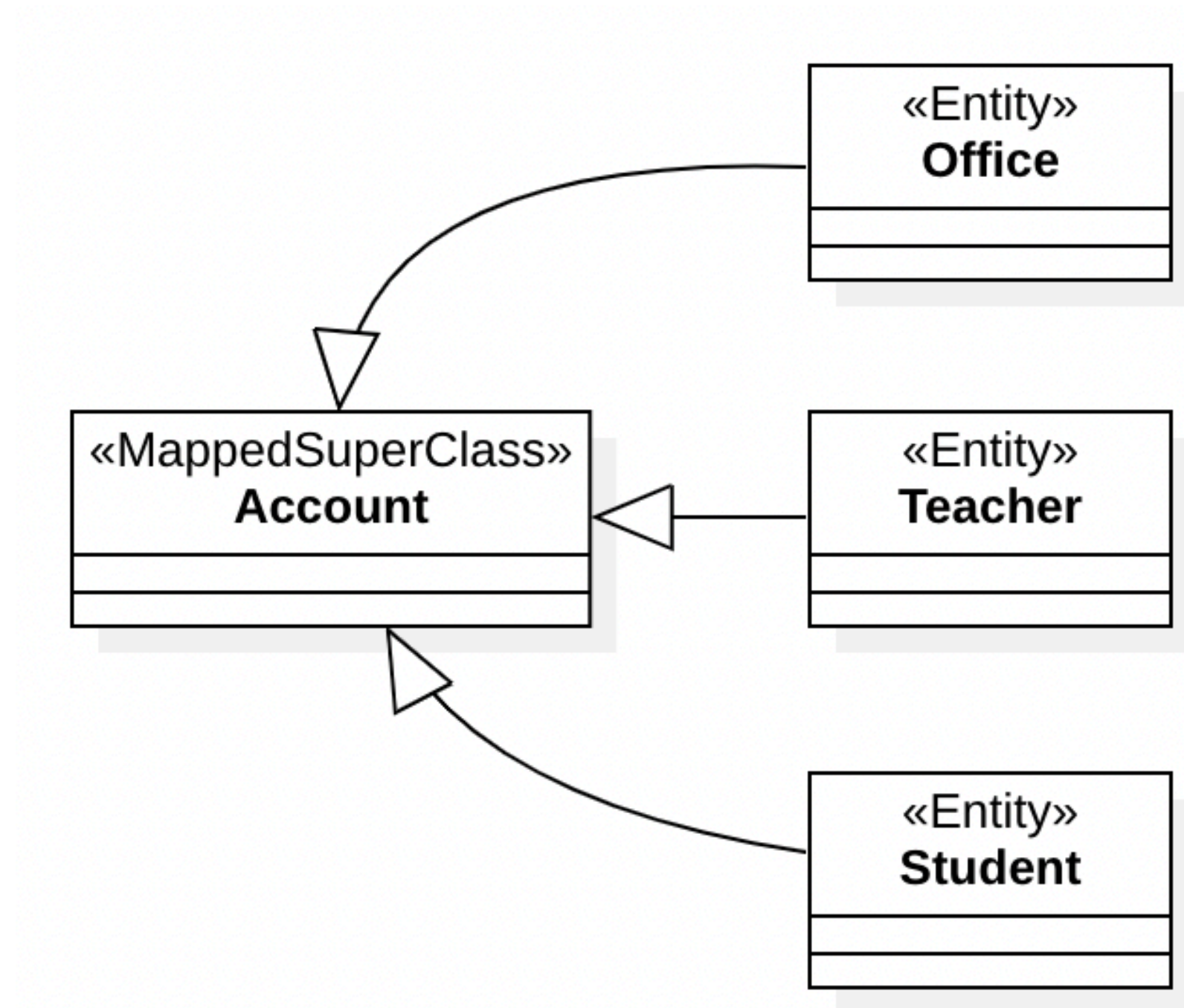
```
@MappedSuperclass
public abstract class Account implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue
    private int id;
    @Column(nullable = false, length = 45)
    private String name;
    @Column(nullable = false)
    private Role role;
    @Column(name = "login_id", nullable = false, length = 10)
    private String loginId;
    @Column(nullable = false)
    private String password;

    // Getters & Setters
}
```

Mapped Super Class



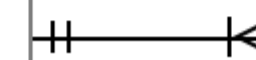
MappedSuperClass ရဲ့ Child Entity တွေနဲ့ Map လုပ်ထားတဲ့
Table တွေအကြားမှာ ဘယ်လို Relationship မှ ရှိမှာမဟုတ်ပါဘူး

office
id INT
login_id VARCHAR(10)
name VARCHAR(45)
password VARCHAR(255)
role INT
admin BIT(1)
Indexes

teacher
id INT
login_id VARCHAR(10)
name VARCHAR(45)
password VARCHAR(255)
role INT
Indexes

student
id INT
login_id VARCHAR(10)
name VARCHAR(45)
password VARCHAR(255)
role INT
Indexes

address
address VARCHAR(255)
email VARCHAR(45)
phone VARCHAR(12)
student_id INT
Indexes



@Inheritance

- Inheritance Hierarchy ထဲမှာရှိတဲ့ Supper Class တွေကို Table တစ်ခုအနေနဲ့ Map လုပ်ချင်တယ် ဆိုရင်တော့ @Entity နှင့်အတူ @Inheritance ကို အသုံးပြုပြီးရေးသားပေးရပါမယ်
- @Inheritance Mapping မှာ strategy Attribute ကို သတ်မှတ်ရေးသားနိုင်ပြီး Default အတိုင်းဆိုပါက SINGLE_TABLE ကို အသုံးပြုပါတယ်
- ရွေးချယ်နိုင်တဲ့ InheritanceType တွေကတော့ SINGLE_TABLE, JOINED, TABLED_PER_CLASS တို့ဖြစ်ကြပါတယ်

Using @Inheritance

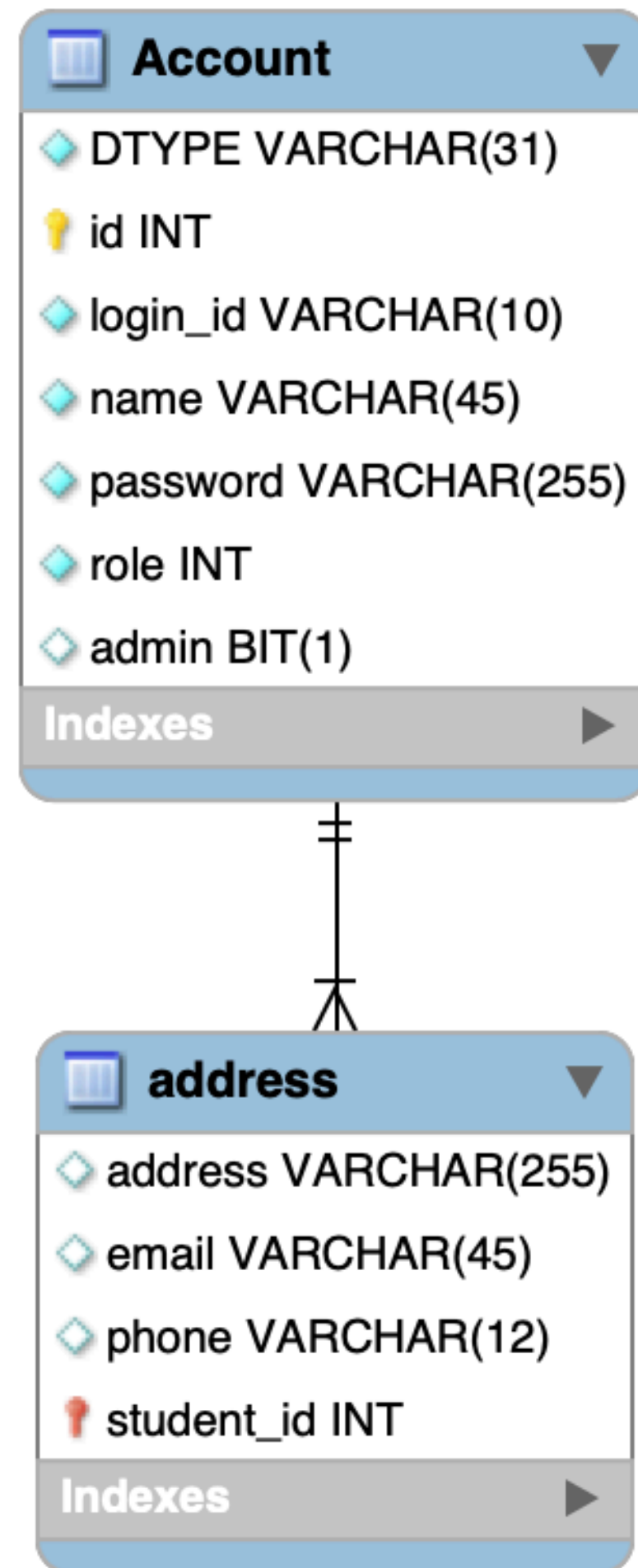
```
@Entity
@Inheritance
public abstract class Account implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue
    private int id;
    @Column(nullable = false, length = 45)
    private String name;
    @Column(nullable = false)
    private Role role;
    @Column(name = "login_id", nullable = false, length = 10)
    private String loginId;
    @Column(nullable = false)
    private String password;

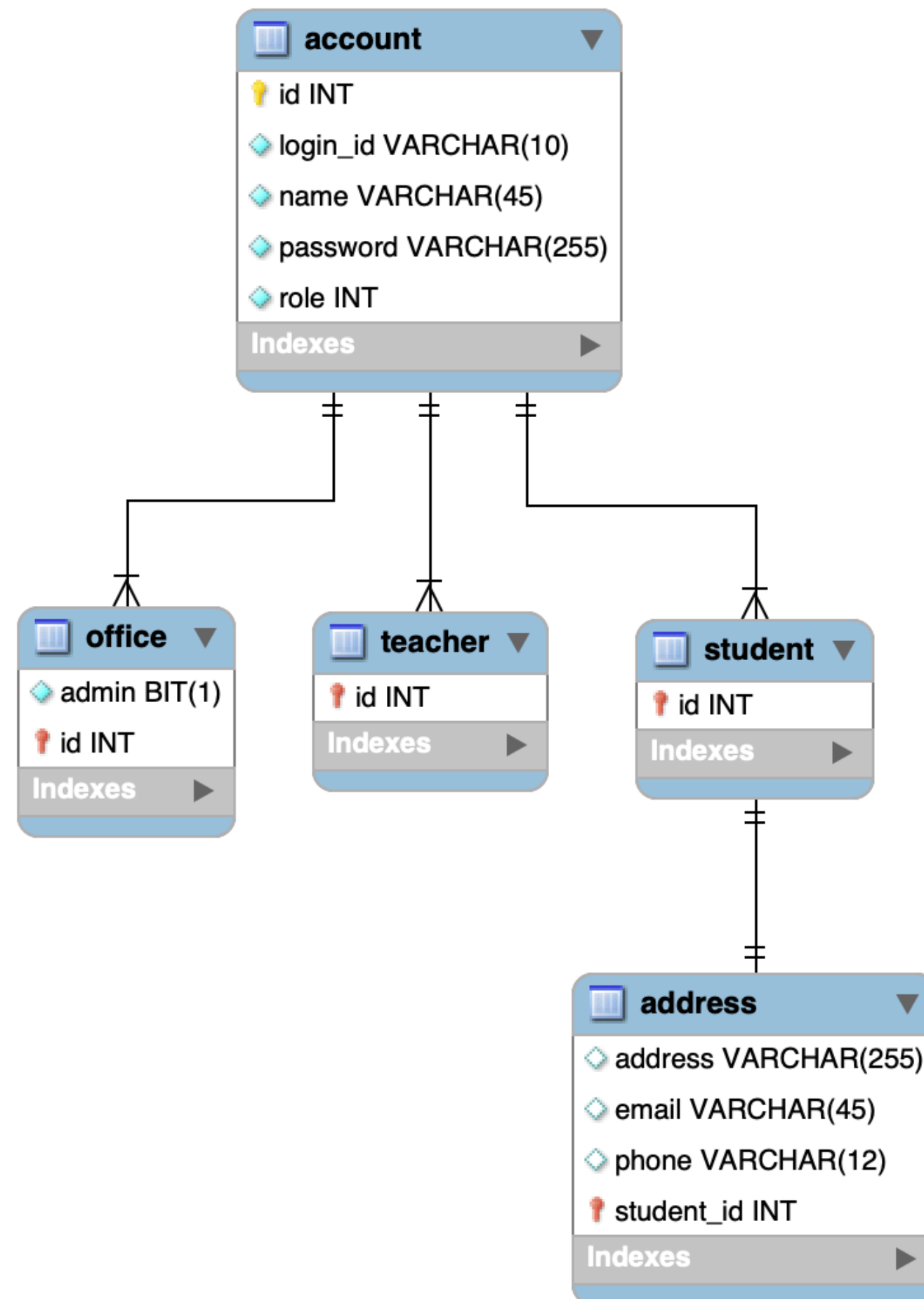
    // Getters & Setters
}
```

SINGLE_TABLE



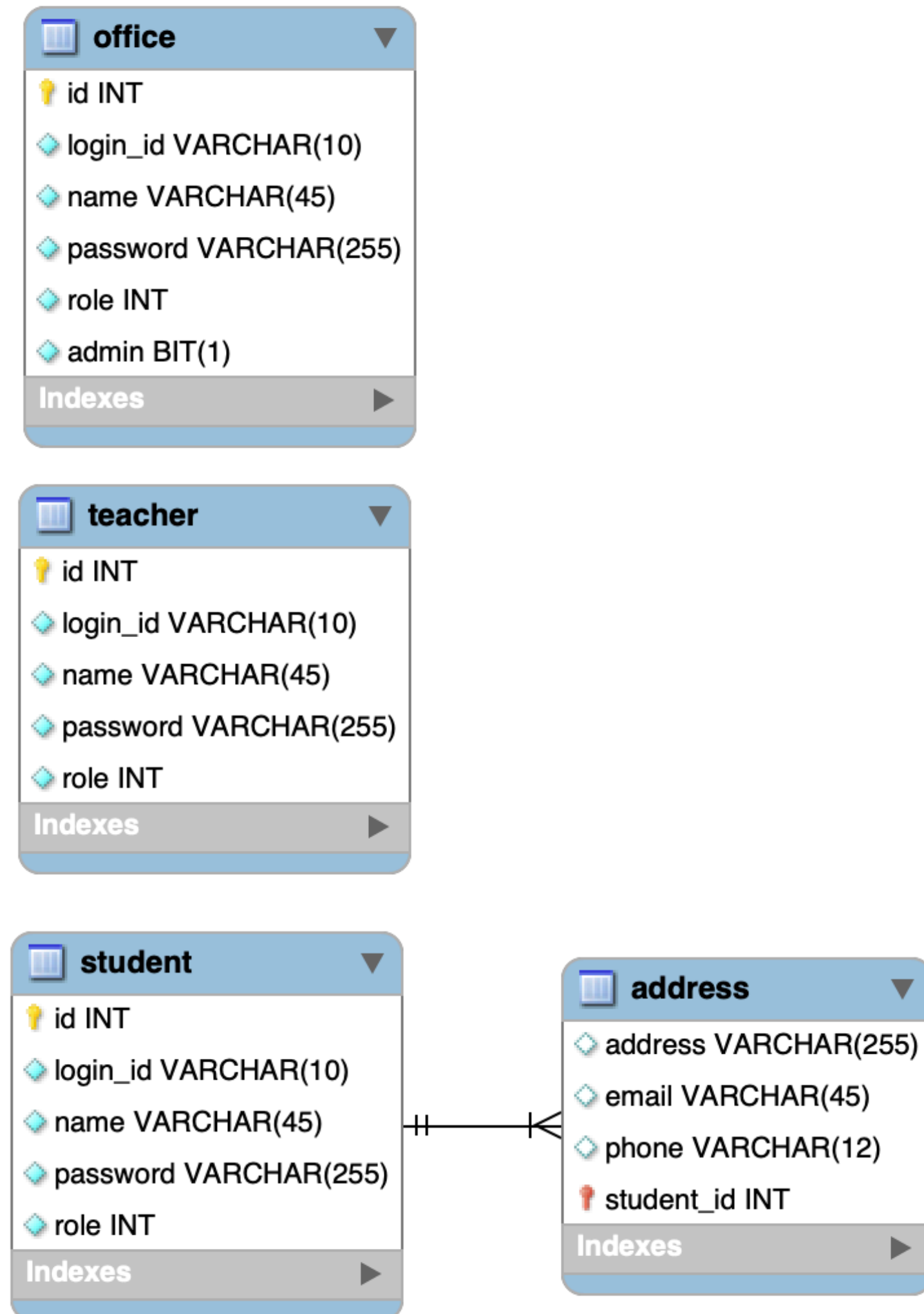
- @Inheritance Mapping ရဲ့ Default Inheritance Strategy ဖြစ်ပါတယ်
- Inheritance Hierarchy ထဲမှာရှိတဲ့ Entity Class တွေထဲက Field တွေ အားလုံးကို Table တစ်ခုထဲက Column တွေနဲ့ Map လုပ်ပေးမှာ ဖြစ်ပါတယ်
- Table တွေကို Join လုပ်ပြီး Fetch လုပ်စရာမလိုတဲ့အတွက် Inheritance Strategy တွေထဲမှာတော့ Performance အကောင်းဆုံး Strategy တစ်ခုဖြစ် ပါတယ်
- ဒါပေမဲ့ တစ်ချို့ Column တွေမှာ တန်ဖိုးမပါဘူးဆိုတာတွေရှိနေနိုင်တဲ့ အတွက် Data မှန်ကန်မှုအပိုင်းမှာတော့ အားနဲ့နေနိုင်ပါတယ်

JOINED



- Super Class အတွက်ကော Child Classes တွေအတွက်ပါ Table တစ်ခုစီနဲ့ Map လုပ်ပေးမှာဖြစ်ပါတယ်
- Child Class နဲ့ Map လုပ်ထားတဲ့ Table တွေရဲ့ Primary Key ကို Primary Key Foreign Key အနေနဲ့ အသုံးပြုပြီး Table တွေကို Join လုပ်ပေးပါတယ်
- Entity Class တွေမှာလဲ Relationship ရှိသလို၊ Database Table တွေအကြားမှာလဲ Relationship ရှိနေမှာ ဖြစ်ပါတယ်

TABLE_PER_CLASS



- Child Entity တစ်ခုကို Table တစ်ခုနဲ့ Map လုပ်ပြီးသွားမှာ ဖြစ်ပြီး အဲဒီ Table တွေကြားထဲမှာ ပတ်သက်မှုမရှိပါဘူး
- Hibernate JPA ကတော့ Support လုပ်ပေမဲ့ တစ်ချို့ JPA Provider တွေက Support မလုပ်ကြတဲ့အတွက် သတိထားပြီး အသုံးပြုသင့်ပါတယ်
- ရလဒ်အနေနဲ့ @MappedSuperClasses နဲ့ ဆင်တူတဲ့အတွက် @MappedSuperClasses ကို သုံးတာက ပိုပြီးစိတ်ချရမှာပါ

Discriminator Column

- Inheritance Type ကို SINGLE_TABLE လို့ သတ်မှတ်ထားတဲ့ အခါမှာ Child Entity တွေအားလုံးရဲ့ Data တွေအားလုံးကို Database Table တစ်ခုထဲမှာပဲ ထားပြီး သိမ်းပါတယ်
- Table တစ်ခုထဲမှာရှိတဲ့ Entity တွေရဲ့ Data တွေကို ခွဲခြားဖို့အတွက် JPA Engine က Mapping လုပ်တဲ့ Table ထဲမှာ Discriminator Colum တစ်ခုကို တည်ဆောက်ပြီး အသုံးပြုပါတယ်
- Entity အလိုက် Discriminator တွေကို ခွဲခြားထားပြီး၊ Table ထဲကို Insert လုပ်တဲ့ အခါမှာ Discriminator ကို ခွဲခြားပြီး သိမ်းပေးထားပါတယ်
- Entity Class တွေမှာ Discriminator Value ကို Customize လုပ်နိုင်သလို၊ Discriminator Column နဲ့ ပတ်သက်ပြီးလဲ Column Name, Data Type နှင့် Size တို့ကို လိုအပ်သလို Configure လုပ်နိုင်ပါတယ်

AttributeConverter

- ပုံမှန်အားဖြင့် JPA Entity ရဲ့ Fields တွေရဲ့ Type တွေကို JDBC Column Type တွေနဲ့ Map လုပ်နိုင်တဲ့ Type တွေကို သာ အသုံးပြုနိုင်ပါတယ်
- JPA 2.1 ကနေစပြီး Custom Data Type တွေကို JPA Entity ရဲ့ Fields အနေနဲ့ ရေးသားချင်ရင် ရေးသားနိုင်အောင် AttributeConverter Interface ကို ဖြည့်စွက်ခဲ့ကြပါတယ်
- AttributeConverter ကို Default အတိုင်း အသုံးပြုလို့မရနိုင်တဲ့ Custom Data Type တွေကို Entity Field တွေမှာ အသုံးပြုလိုတဲ့အခါနှင့် နှစ်သက်ရာ Database Column Data Type ဖြင့် Map လုပ်လိုတဲ့ အခါတွေမှာ အသုံးပြုနိုင်ပါတယ်
- AttributeConverter ကို အသုံးပြုမယ်ဆိုရင် အရင်ဆုံး AttributeConverter Interface ကို Implement လုပ်ထားတဲ့ Class မှာ @Converter Annotation နဲ့ ရေးသားထားရပါမယ်
- ပြီးရင် Convert လုပ်လိုတဲ့ Entity ရဲ့ Attribute မှာ @Convert Annotation နဲ့ အသုံးပြုလိုတဲ့ Converter ကို သတ်မှတ်ရေးသားပေးရမှာ ဖြစ်ပါတယ်

Writing AttributeConverter

```
@Converter
public class ColorConverter implements AttributeConverter<Color, String>{

    @Override
    public String convertToDatabaseColumn(Color color) {
        if(null != color) {
            return "%s,%s,%s,%s".formatted(
                color.getRed(), color.getGreen(), color.getBlue(), color.getAlpha());
        }
        return null;
    }

    @Override
    public Color convertToEntityAttribute(String dbData) {
        if(null != dbData) {
            var rgba = dbData.split(",");

            return new Color(getColorValue(rgba[0]),
                getColorValue(rgba[1]), getColorValue(rgba[2]), getColorValue(rgba[3]));
        }
        return null;
    }

    private float getColorValue(String str) {
        var factor = new BigDecimal(255);
        var color = new BigDecimal(str);
        return color.divide(factor, 12, RoundingMode.HALF_UP).floatValue();
    }
}
```


Using AttributeConverter

```

@Entity
@Table(name = "account")
public abstract class Account implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue
    private int id;
    private String name;
    private Role role;
    @Column(name = "login_id", nullable = false, length = 10)
    private String loginId;
    private String password;

    @Convert(converter = ColorConverter.class)
    private Color color;

    // Getters & Setters

}
```