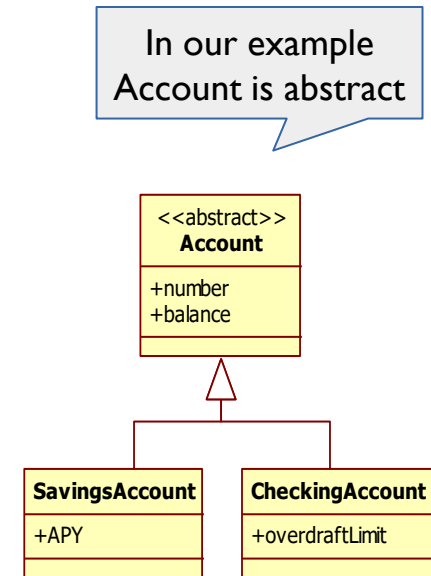# ORM Inheritance

Inheritance

- With Inheritance a class can extend another class
  - Inheriting its properties and methods
  - Often referred to as an 'IS-A' relationship

- Relational **does not have** inheritance
  - There are 3 ways to emulate it

In our example
Account is abstract

<> **Account**
+number +balance

SavingsAccount
+APY

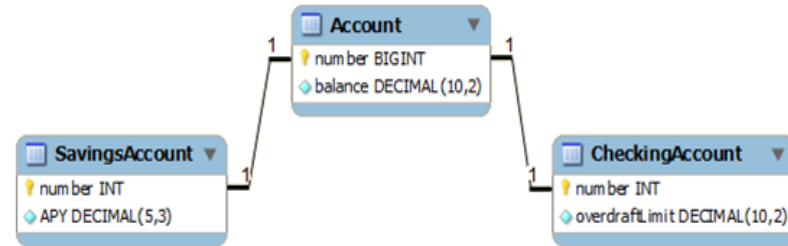| **CheckingAccount** |
|---|
| +overdraftLimit |
| |

Emulate Inheritance

- **Single Table** per Hierarchy
  - De-normalized schema
  - Fast DB operations
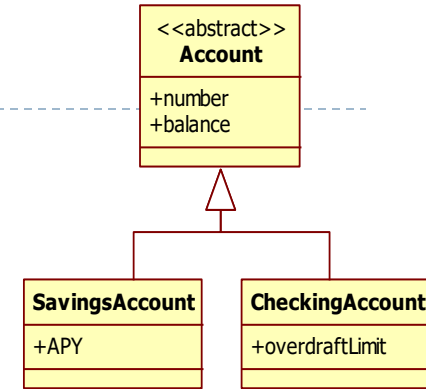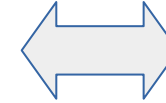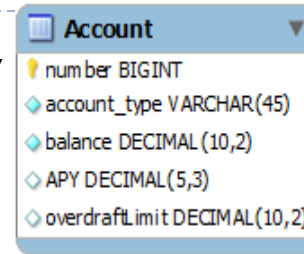
- **Joined** Tables
  - Normalized schema
  - Bit slower operations

- **Table per** Concrete Class
  - Uses UNION instead of JOIN
  - All needed columns in each

JPA does not require implementers to provide this strategy
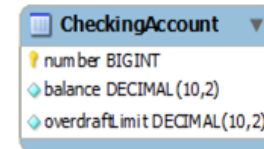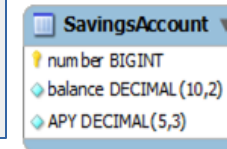
Polymorphism

- Polymorphism is the ability of a **subtype** to appear and behave **like its super type**

- This enables a person to have a list of account references, which can be any type of account

- A polymorphic query is a query for all objects in a hierarchy, independent of subtype
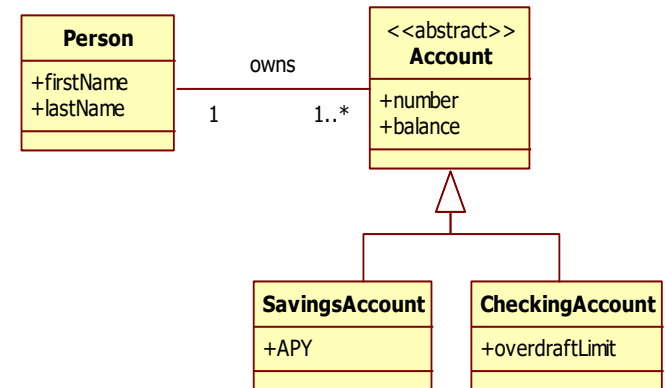
CS544 EA
Hibernate

Inheritance: Single Table

Single Table

- Single Table uses **one big table**
  - **Discriminator column** specifies actual type
  - Sub class properties added as nullable columns

Not normalized


```
<<abstract>>
Account

+number
+balance
```

```
SavingsAccount

+APY
```

```
CheckingAccount

+overdraftLimit
```

APY and overdraft are nullable

```
Account

id INT
account_type VARCHAR(45)
number BIGINT
balance DECIMAL(10,2)
APY DECIMAL(5,3)
overdraftLimit DECIMAL(10,2)
```

account_type specifies actual type
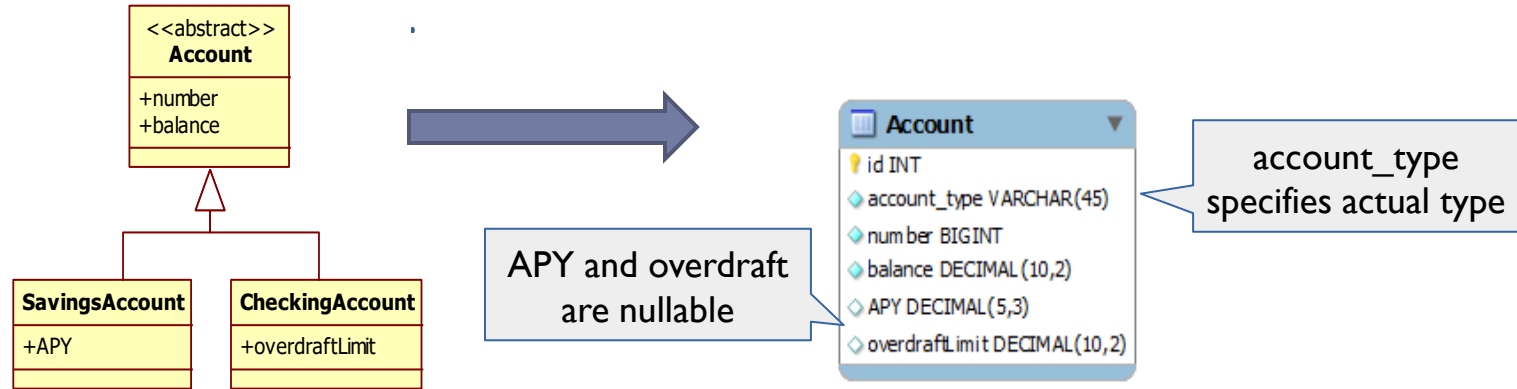
Single Table in Action

ACCOUNT_TYPE	NUMBER	BALANCE	OVERDRAFTLIMIT	APY
checking	1	500	200	
savings	2	100		2.3
checking	3	23.5	0	

APY null for checking
overdraft null for savings

✓ Simple, easy to implement

✓ Good performance on all queries (poly and not)

✗ Nullable columns / de-normalized

✗ Table may have to contain lots of columns

SQL for Single Table Query

```
select
        account0_.number as number0_,
        account0_.balance as balance0_,
        account0_.owner_id as owner6_0_,
        account0_.overdraftLimit as overdraf4_0_,
        account0_.APY as APY0_,
        account0_.account_type as account1_0_
from

        Account account0_
```

Single Table Mapping

Optional @Inheritance
Optional strategy
Default = SINGLE_TABLE

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="account_type",
        discriminatorType=DiscriminatorType.STRING)
public abstract class Account {
        @Id
        @GeneratedValue
        private Integer id;
        private Long number;
        private double balance;
```

Opional @DiscriminatorValue
Default value is class name

```
@Entity
@DiscriminatorValue("savings")
public class SavingsAccount extends Account {
        private double APY;
```

```
@Entity
@DiscriminatorValue("checking")
public class CheckingAccount extends Account {
        private double overdraftLimit;
```

Opional @DiscriminatorColumn
Default name is: DTYPE
Default type is: STRING

Important: subclasses do not have an **@Id**

```
<<abstract>>
Account
-----------
+number
+balance
```

```
SavingsAccount        CheckingAccount
----------------      -----------------
+APY                  +overdraftLimit
```
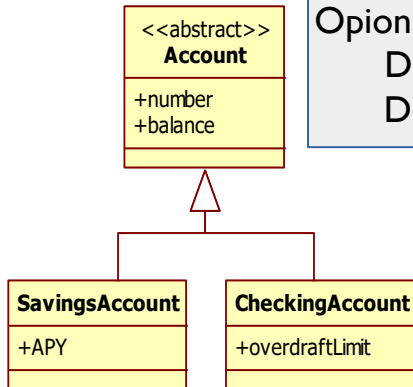
Account
- id INT
- number BIGINT
- balance DECIMAL(10,2)
- APY DECIMAL(5,3)
- overdraftLimit DECIMAL(10,2)
- account_type VARCHAR(45)

Defaults

- ## Works without extra annotations

  - ### Defaults to Single Table

```java
@Entity
public abstract class Account {
        @Id
        @GeneratedValue
        private Integer id;
        private Long number;
        private double balance;
```

```java
@Entity
public class SavingsAccount extends Account {
        private double APY;
```

```java
@Entity
public class CheckingAccount extends Account {
        private double overdraftLimit;
```

<>
Account

+number
+balance

SavingsAccount

+APY

CheckingAccount

+overdraftLimit

account ▼

◇ DTYPE VARCHAR(31)

🔑 id INT(11)

◇ balance DOUBLE

◇ number BIGINT(20)

◇ overdraftLimit DOUBLE

◇ APY DOUBLE

Indexes ▶
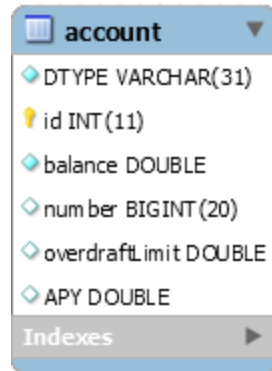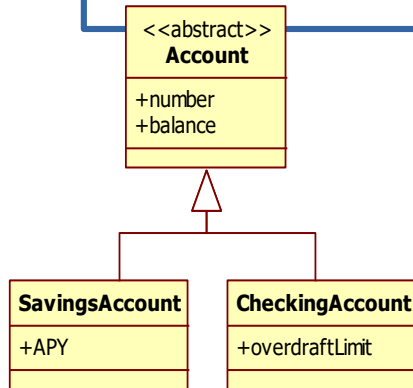
CS544 EA

Hibernate

Inheritance: Joined Tables

Joined Tables

- Uses FK '**has-a**' to emulate '**is-a**'
  - Uses **Shared Primary Key** as Foreign Key
  - Queries use joins to include needed tables


```
<<abstract>>
Account

+number
+balance
```

```
SavingsAccount

+APY
```

```
CheckingAccount

+overdraftLimit
```

Account
- number BIGINT
- balance DECIMAL(10,2)

Number is Shared PK

SavingsAccount
- number INT
- APY DECIMAL(5,3)

CheckingAccount
- number INT
- overdraftLimit DECIMAL(10,2)

Joined Tables in Action

Account Table

NUMBER	BALANCE
1	500
2	100
3	23.5

SavingsAccount

NUMBER	APY
2	2.3

CheckingAccount

NUMBER	OVERDRAFTLIMIT
1	200
3	0

✓ Normalized Schema

✓ Database view similar to domain

✗ Inserting or updating takes multiple statements

✗ Joins make queries slower

SQL for Joined Query

```
select
        account0_.number as number0_,
        account0_.balance as balance0_,
        account0_.owner_id as owner3_0_,
        account0_1_.overdraftLimit as overdraf1_1_
        account0_2_.APY as APY2_,
        case
            when account0_1_.number is not null then 1
            when account0_2_.number is not null then 2
            when account0_.number is not null then 0
        end as clazz_
    from
        Account account0_
    left outer join
        CheckingAccount account0_1_
            on account0_.number=account0_1_.number
    left outer join
        SavingsAccount account0_2_
            on account0_.number=account0_2_.number
```

Discriminator generated based on what is joined

Joined Mapping

- Simply set the **strategy to JOINED**

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public abstract class Account {
        @Id
        @GeneratedValue
        private Long number;
        private double balance;
```

```
@Entity
public class SavingsAccount extends Account {
        private double APY;
```

```
@Entity
public class CheckingAccount extends Account {
        private double overdraftLimit;
```
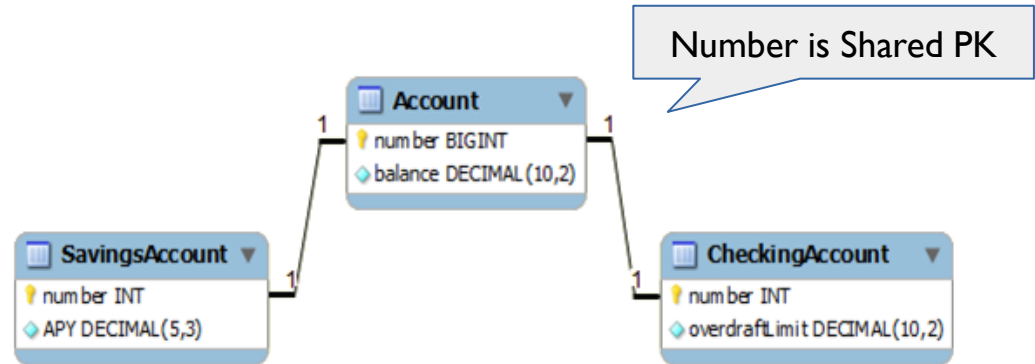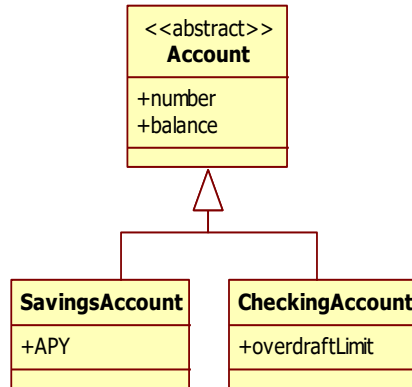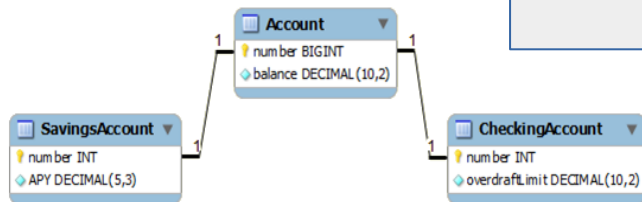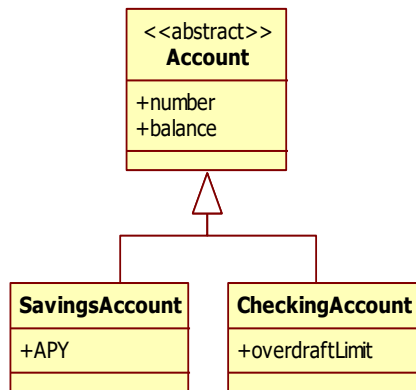

Remember: subclasses do not have an **@Id**

CS544 EA
Hibernate

Inheritance: Table Per Concrete Class

Table per Concrete

- Creates a **table for each** concrete class.
  - (each **class that is not abstract**)
  - Sublcass tables include all superclass properties
  - Polymorphic queries use UNION operator

account table only if account class not abstract

<>
Account

+number
+balance

SavingsAccount

+APY

CheckingAccount

+overdraftLimit

Account
- number INT
- balance DECIMAL (10,2)

SavingsAccount
- number BIGINT
- balance DECIMAL (10,2)
- APY DECIMAL (5,3)

CheckingAccount
- number BIGINT
- balance DECIMAL (10,2)
- overdraftLimit DECIMAL (10,2)
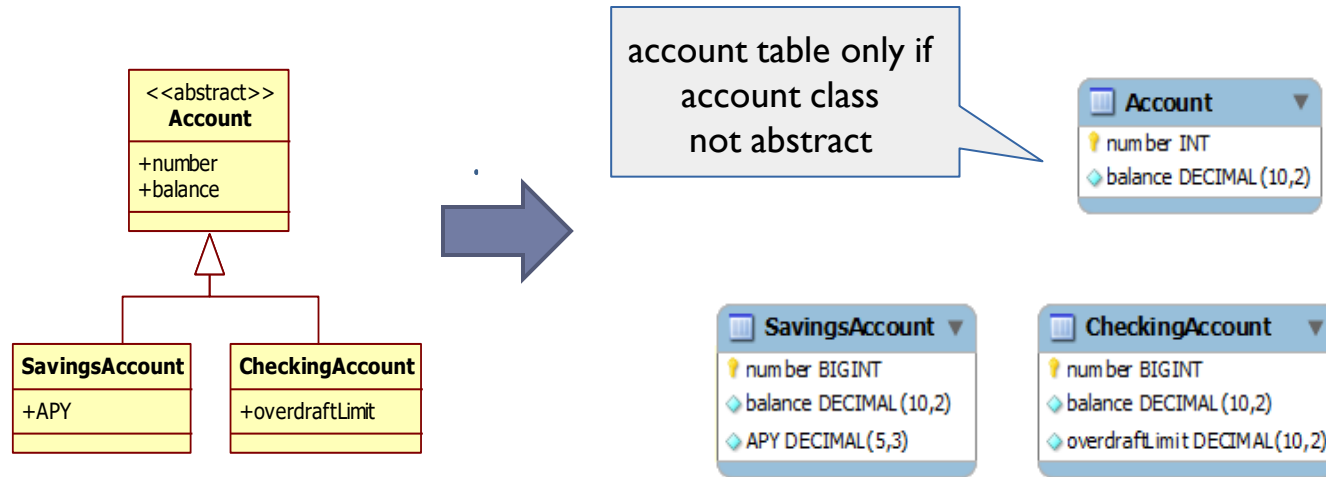
Table per Concrete in Action

SavingsAccount

NUMBER	BALANCE	APY
2	100	2.3

CheckingAccount

NUMBER	BALANCE	OVERDRAFTLIMIT
1	500	200
3	23.5	0

- ✓ Very efficient non-polymorphic
- ✓ Okay with polymorphic queries
- ✗ Cannot use identity column generation
- ✗ Not normalized
- ✗ Redundant columns in each concrete child table
- ✗ More work required to query across tables

SQL for Table per Concrete

```
select
        account0_.number as number0_,
        account0_.balance as balance0_,
        account0_.owner_id as owner3_0_,
        account0_.overdraftLimit as overdraf1_1_,
        account0_.APY as APY2_,
        account0_.clazz_ as clazz_
    from
        ( select
            number,
            balance,
            owner_id,
            overdraftLimit,
            cast(null as int) as APY,
            1 as clazz_
        from
            CheckingAccount
        union
        all select
            number,
            balance,
            owner_id,
            cast(null as int) as overdraftLimit,
            APY,
            2 as clazz_
        from
            SavingsAccount
    ) account0_
```

> Discriminator column generated
> based on which table
> is currently in UNION

Table per Concrete Mapping

- ## No identity column

Table per class strategy

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Account {
        @Id
        @GeneratedValue(strategy=GenerationType.TABLE)
        private Long number;
        private double balance;
```

```
@Entity
public class SavingsAccount extends Account {
        private Double APY;
```

```
@Entity
public class CheckingAccount extends Account {
        private Double overdraftLimit;
```

Table ID strategy
(for on MySQL)

Remember: subclasses do not have an **@Id**

```
<<abstract>>
Account

+number
+balance
```

```
SavingsAccount

+APY
```

```
CheckingAccount

+overdraftLimit
```

🔲 SavingsAccount ▼
🔑 number BIGINT
◇ balance DECIMAL (10,2)
◇ APY DECIMAL(5,3)

🔲 CheckingAccount ▼
🔑 number BIGINT
◇ balance DECIMAL(10,2)
◇ overdraftLimit DECIMAL(10,2)

Strategy Recommendation

- Generally **use Single Table**
  - Use JOINED if subclasses have many properties
  - Or if normalization is a priority

- Avoid Table per Concrete
  - Good to know it exists
  - But has some strange parts

Summary

- Inheritance can be emulated:
    - Single Table (with discriminator column)
    - Joined Tables (shared primary key)
    - Table per Concrete (properties in subclasses)
- The default is Single Table
    - And is also the one that is most recommended

CS544 EA

Hibernate

Complex Mapping

Complex Mapping

- Often you have no choice but to map to an **existing table structure**

  - Your mappings may not be very straight forward

- For example:

  - You can map a class to multiple tables

  - A table can contain multiple classes
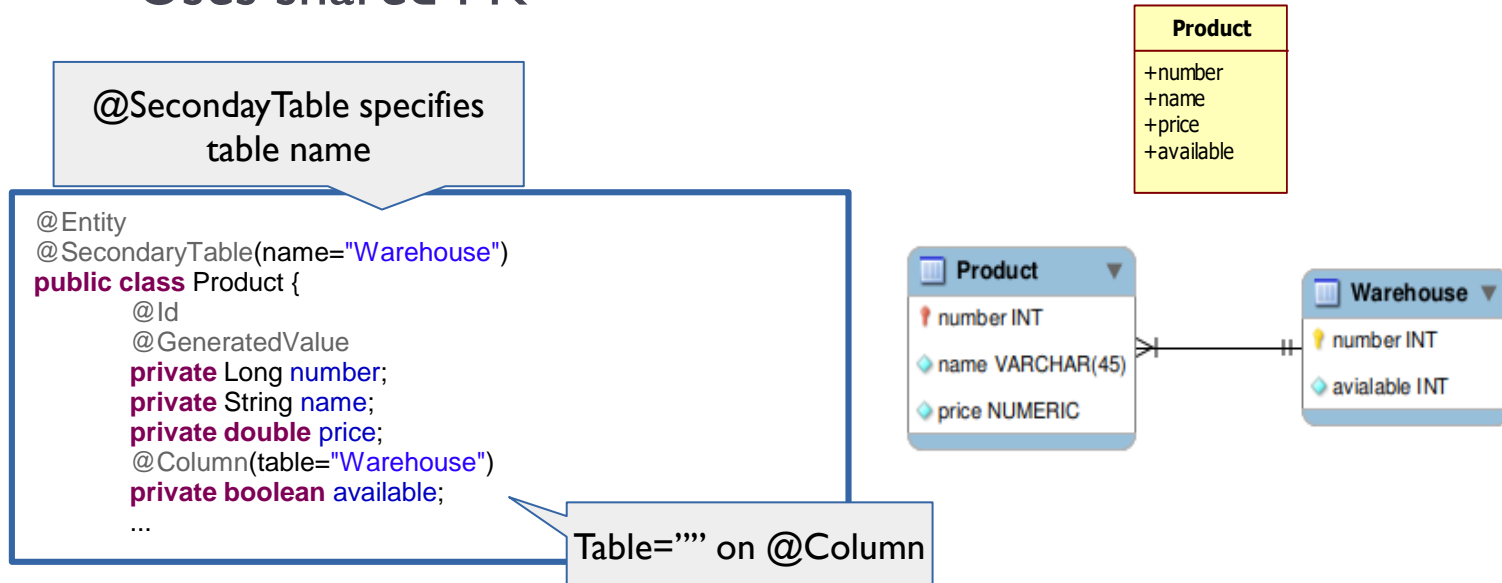
  - You may have composite natural keys

Hibernate

Complex: Secondary Table

Secondary Tables

- Secondary Tables can be used anywhere
  - **Moves properties** into a separate table
  - Uses shared PK

@SecondayTable specifies table name

```
@Entity
@SecondaryTable(name="Warehouse")
public class Product {
        @Id
        @GeneratedValue
        private Long number;
        private String name;
        private double price;
        @Column(table="Warehouse")
        private boolean available;
        ...
```

Table="" on @Column

Product

+number
+name
+price
+available

Product

number INT
name VARCHAR(45)
price NUMERIC

Warehouse

number INT
avialable INT

Many Options

- ## Can specify **multiple tables**

  - Each table can specify **multiple PK join columns**

  - **Column names** on each side can differ

```
@Entity
@SecondaryTables(
            @SecondaryTable(name="Warehouse", pkJoinColumns= {
                        @PrimaryKeyJoinColumn(name="product_id", referencedColumnName="number")
            })
)
public class Product {
        @Id
        @GeneratedValue
        private Long number;
        private String name;
        private double price;
        @Column(table="Warehouse")
        private boolean available;
        ...
```
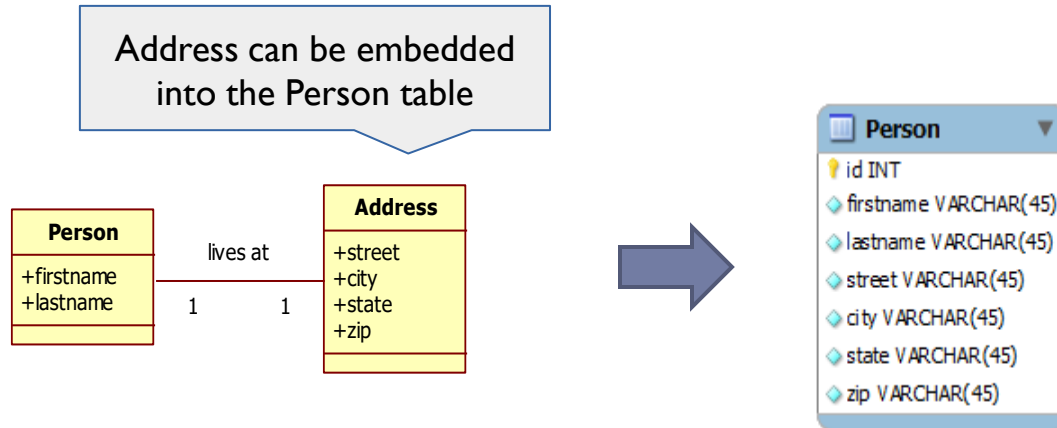
Product

+number
+name
+price
+available

Product

🔑 number INT
◆ name VARCHAR(45)
◆ price NUMERIC

1 ——— 1

Warehouse ▼

🔑 Product_id INT
◆ available INT

Hibernate

Complex: Embedded Classes

Embedded Classes

- Called **Value classes** instead of entity classes
    - Like a property value, can be embedded in entity
    - Useful for tight associations (like one to one)

Address can be embedded into the Person table

Person
+firstname
+lastname

lives at

1   1

Address
+street
+city
+state
+zip

Person
id INT
firstname VARCHAR(45)
lastname VARCHAR(45)
street VARCHAR(45)
city VARCHAR(45)
state VARCHAR(45)
zip VARCHAR(45)

Embeddable

- ## @Embeddable instead of @Entity
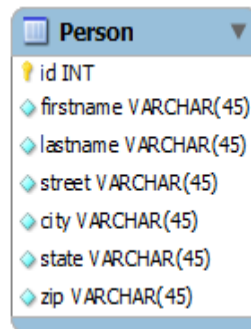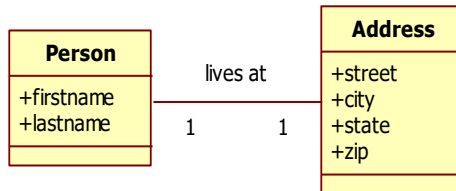 - No @Id inside an @Embedabble

```
@Embeddable
public class Address {
        private String street;
        private String city;
        private String state;
        private String zip;

        ...
```

```
@Entity
public class Person {
        @Id
        @GeneratedValue
        private Long id;
        private String firstName;
        private String lastName;
        @Embedded
        private Address address;

        ...
```

> @Embedded instead of @OneToOne

Person

+firstname
+lastname

lives at

1   1

Address

+street
+city
+state
+zip

Person
- id INT
- firstname VARCHAR(45)
- lastname VARCHAR(45)
- street VARCHAR(45)
- city VARCHAR(45)
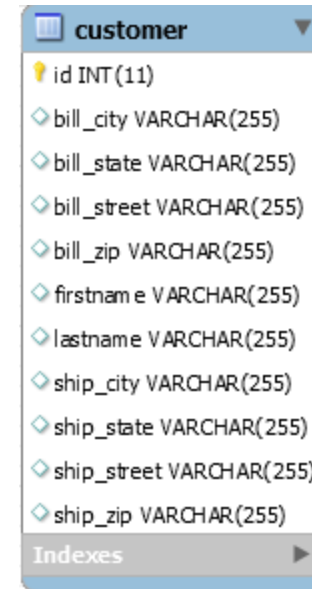- state VARCHAR(45)
- zip VARCHAR(45)

Multiple Embedded

```java
@Entity
public class Customer {
  @Id
  @GeneratedValue
  private int id;
  private String firstname;
  private String lastname;

  @Embedded
  @AttributeOverrides( {
   @AttributeOverride(name="street", column= @Column (name="ship_street")),
   @AttributeOverride(name="city", column=@Column(name="ship_city")),
   @AttributeOverride(name="state", column=@Column(name="ship_state")),
   @AttributeOverride(name="zip", column=@Column(name="ship_zip"))
  })
  private Address shipping;

  @Embedded
  @AttributeOverrides( {
   @AttributeOverride(name="street", column=@Column(name="bill_street")),
   @AttributeOverride(name="city", column=@Column(name="bill_city")),
   @AttributeOverride(name="state", column=@Column(name="bill_state")),
   @AttributeOverride(name="zip", column=@Column(name="bill_zip"))
  })
  private Address billing;
```

Use @AttributeOverrides to change the column names

customer

- id INT(11)
- bill_city VARCHAR(255)
- bill_state VARCHAR(255)
- bill_street VARCHAR(255)
- bill_zip VARCHAR(255)
- firstname VARCHAR(255)
- lastname VARCHAR(255)
- ship_city VARCHAR(255)
- ship_state VARCHAR(255)
- ship_street VARCHAR(255)
- ship_zip VARCHAR(255)

Indexes

| ID	FIRSTNAME	LASTNAME	SHIP_STREET	SHIP_CITY	SHIP_STATE	SHIP_ZIP	BILL_STREET	BILL_CITY	BILL_STATE	BILL_ZIP
1	Frank	Brown	45 N Main St	Chicago	Illinois	51885	100 W Adams St	Chicago	Illinois	60603

Hibernate

Complex: Composite Keys

Composite Keys

- Composite keys are **multi-column primary keys**
  - By definition natural keys, set by the app
  - Also create **multi-column foreign keys**
  - Generally found in legacy systems

Employee ▼
🔑 firstname VARCHAR(45)
🔑 lastname VARCHAR(45)
🔷 startDate DATE
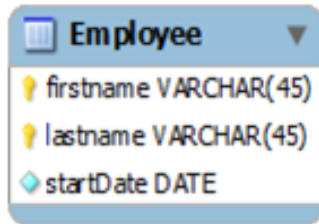
Composite Id

- @Embeddable can create a composite PK
  - Using **@EmbeddedId** annotation
  - -Must **implements Serializable**

```
@Embeddable
public class Name implements Serializable {
        private String firstName;
        private String lastName;
        ...
```
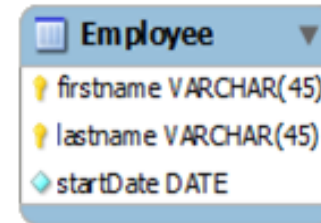
Requires
hashCode() and equals()

Java check for uniqueness

```
@Entity
public class Employee {
        @EmbeddedId
        private Name name;
        @Temporal(TemporalType.DATE)
        private Date startDate;
        ...
```

@EmbeddedID

Employee
- firstname VARCHAR(45)
- lastname VARCHAR(45)
- startDate DATE

.hashcode() & .equals()

```java
@Override
public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((firstName == null) ? 0 : firstName.hashCode());
        result = prime * result + ((lastName == null) ? 0 : lastName.hashCode());
        return result;
}
@Override
public boolean equals(Object obj) {
        if (this == obj)
                return true;
        if (obj == null)
                return false;
        if (getClass() != obj.getClass())
                return false;
        Name other = (Name) obj;
        if (firstName == null) {
                if (other.firstName != null)
                        return false;
        } else if (!firstName.equals(other.firstName))
                return false;
        if (lastName == null) {
                if (other.lastName != null)
                        return false;
        } else if (!lastName.equals(other.lastName))
                return false;
        return true;

}
```

Critical for checking uniqueness

IDE can generate these for you

Composite FK

- **Multiple columns** to refer to composite PK

```java
@Entity
public class Employee {
        @EmbeddedId
        private Name name;
        @Temporal(TemporalType.DATE)
        private Date startDate;
        @OneToMany(mappedBy="owner")
        private List<Project> projects
                = new ArrayList<>();
        ...
```

Normal mappedBy

Employee
- firstname VARCHAR(45)
- lastname VARCHAR(45)
- startDate DATE

1 ∞

Project
- id INT
- name VARCHAR(45)
- Emp_firstname VARCHAR(45)
- Emp_lastname VARCHAR(45)

```java
@Entity
public class Project {
        @Id
        @GeneratedValue
        private Long id;
        private String name;
        @ManyToOne
        @JoinColumns({
                @JoinColumn(name = "Emp_firstname", referencedColumnName = "firstname"),
                @JoinColumn(name = "Emp_lastname", referencedColumnName = "lastname")
        })
        private Employee owner;
        ...
```

Optional @JoinColumns

Defaults to:
owner_firstname
owner_lastname

Summary

- **Secondary Tables**

  – 1 class to multiple tables

- **Embedded classes**

  – 1 table, multiple classes

- **Composite Keys**

  – Multi-column PKs and Fks

  – By using an embedded class as Id

Main Point

Entities and objects relationships can be established through the different types of associations, creating a rich foundation that can represent a real world domain.

Science of Consciousness: *Seek the highest first, start with a good foundation and build rich relationships upon it.*