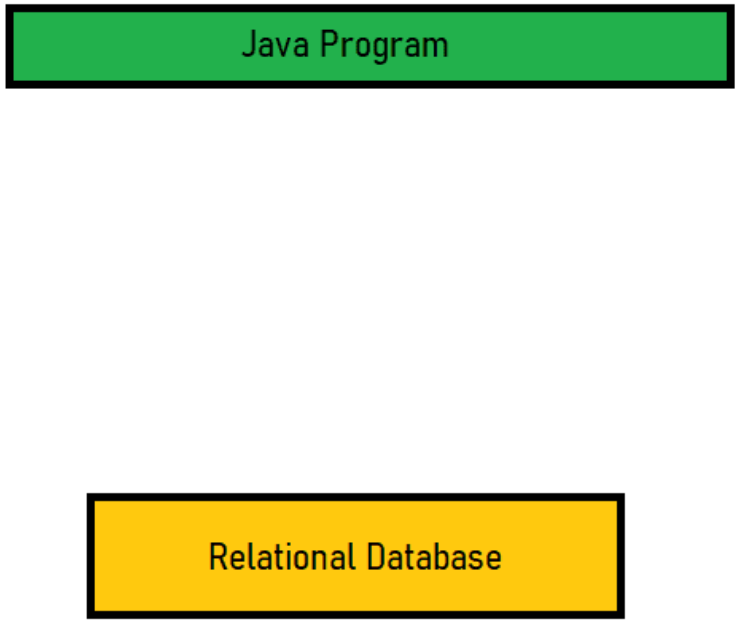
Persistence Intro

RUJUAN XING

Java Program vs Database

Java is one of the most powerful and popular server-side languages in the current scenario. One of the main features of a server-side language is the ability to communicate with the databases.

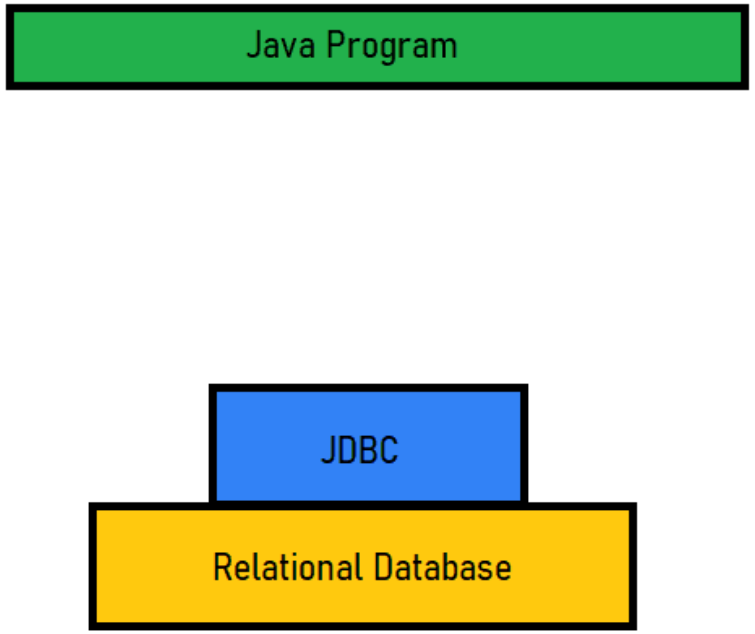
In this course, we'll learn the difference between two ways of connecting to the database (i.e.) JDBC, JPA, Hibernate and Spring Data.



JDBC

JDBC stands for **Java Database Connectivity**. It is a part of Java which was released by Sun Microsystems in 1997.

It is a java application programming interface to provide a connection between the Java programming language and a wide range of databases (i.e), it establishes a link between the two so that a programmer could send data from Java code and store it in the database for future use



MySQL Installation

Visual Studio 2019 Redistributable

- https://learn.microsoft.com/en-us/cpp/windows/latest-supported-vc-redist?view=msvc-170
- X64 version

MySQL Community Downloads: https://dev.mysql.com/downloads/

- MySQL Community Server: Version 8.0.34 – the latest 8.1.0 isn't compatible with workbench 8.0.34 which is the latest version by the time of creating this slide.
- MySQL Workbench : Version 8.0.34

N-Tier Architecture

Presentation [View & Controller] Tier

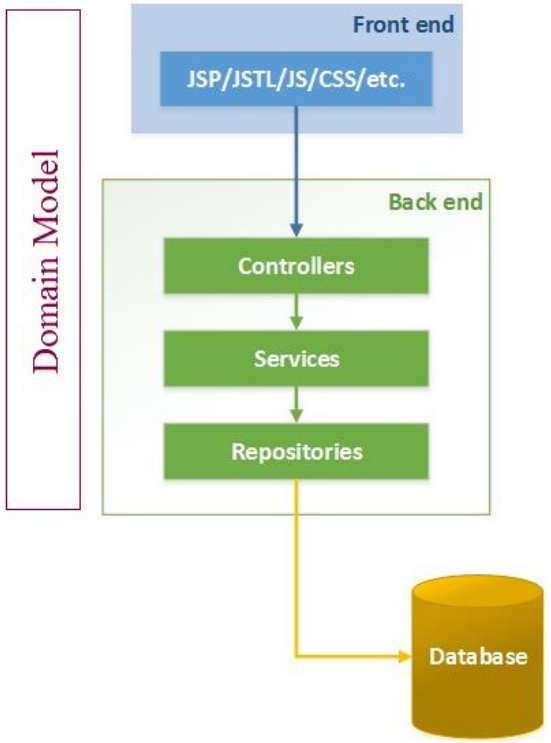
- Communication interface to external entities
- “View” in the model-view-controller

Business [Services] Tier

- Implements operations requested by clients through the presentation layer
- Represents the “business logic”

Persistence [Repository] Tier

- Deals with different data sources of an information system
- Responsible for storing and retrieving data



Service Tier “manages” Persistence

All access to Persistence through Services

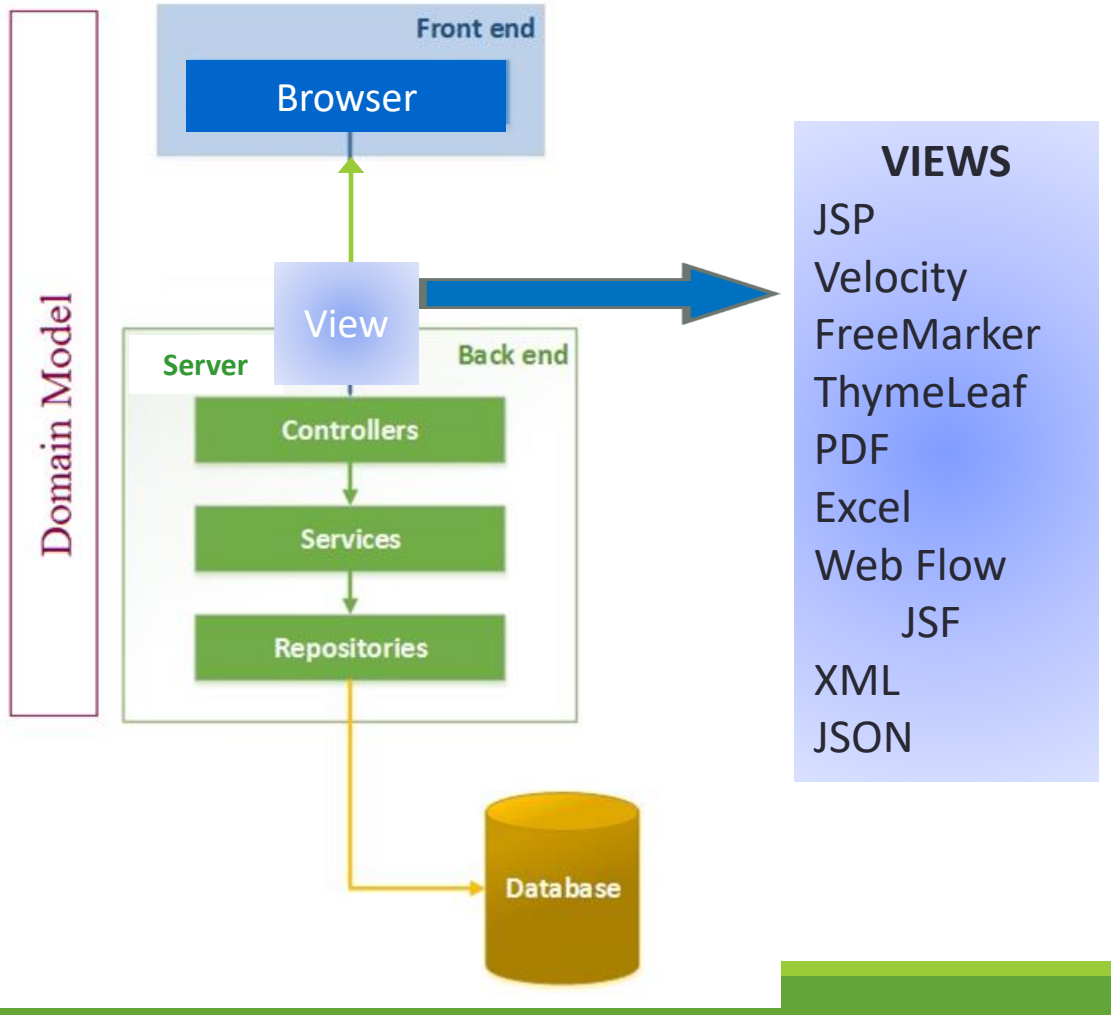
Services responsible for business Logic and data model composition

- Business logic does NOT belong in Persistence
- Business logic does NOT belong in Presentation

Spring/JPA/Persistence is designed with this architecture

Spring Layers – With Spring MVC Layer

Issue: not whether or not it is needed BUT what it contains



Domain Driven Design

- Primary focus - the core domain and domain logic.
- Complex designs based on a model of the domain.
- Collaboration between technical and domain experts to iteratively refine a conceptual model that addresses particular domain problems.
- Contains objects properly named after the nouns in the domain space
- Objects are connected with the rich relationships and structure that true domain models have.

- “Thin” Domain Model
  - Extreme case: Anemic Domain Model
  - Little or no behavior – bags of getters and setters
- GOAL: a Rich Domain Model

Service Layer

In a perfect world:

“Thin Layer”

With

“Rich Domain Model”

- No business rules or knowledge
- Coordinates tasks
- Delegates work to domain objects

“The Reality”

Quite often additional **“Domain”** Services exist - populated with “externalized” Business/Logic rules.

JDBC Example

```
@Repository
public class EmployeeDAO {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    public int addEmplyee(int id, String firstname, String lastname, String address) {
        return jdbcTemplate.update("INSERT INTO EMPLOYEE VALUES (?, ?, ?, ?)", id, firstname, lastname,
address);
    }

    public List<Employee> getAllEmployees() {
        return jdbcTemplate.query("SELECT * FROM EMPLOYEE", new EmployeeRowMapper());
    }

}
```

```
public class EmployeeRowMapper implements RowMapper<Employee> {

    @Override
    public Employee mapRow(final ResultSet rs, final int rowNum) throws
SQLException {
        final Employee employee = new Employee();

        employee.setId(rs.getInt("id"));
        employee.setFirstName(rs.getString("firstname"));
        employee.setLastName(rs.getString("lastname"));
        employee.setAddress(rs.getString("address"));

        return employee;
    }
}
```

JDBC Pros & Cons

Pros:

1. **Control:** JDBC offers fine-grained control over database interactions. Developers can write custom SQL queries and optimize them for specific database engines.
2. **Performance:** For simple queries and operations, JDBC can be faster than Hibernate because it eliminates the abstraction layer.
3. **Simplicity:** JDBC is relatively simple to learn and use, especially for developers who are already familiar with SQL.
4. **Flexibility:** JDBC allows you to work with stored procedures, database-specific features, and SQL optimizations directly.

Cons:

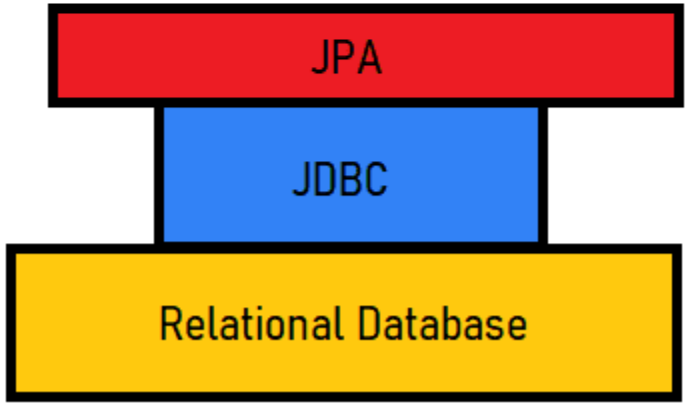
1. **Boilerplate Code:** JDBC requires writing a significant amount of boilerplate code for common database operations, which can be error-prone and time-consuming.
2. **Portability:** Code written using raw JDBC may not be easily portable to different database systems due to database-specific SQL queries.
3. **Maintenance:** Changes to the database schema may require extensive code modifications in JDBC-based applications.
4. **Complex Mapping:** Mapping between Java objects and database tables is not as straightforward as with Hibernate's ORM.

JPA

- Jakarta Persistence API (**JPA; formerly Java Persistence API**)
- JPA is a specification – not an implementation.
- JPA 1.0 (2006). JPA 2.0 (2009), JPA 2.2(2017).
- Standardizes interface across industry platforms
- Object/Relational Mapping
  - Specifically Persistence for RDBMS

Major Implementations [since 2006]:

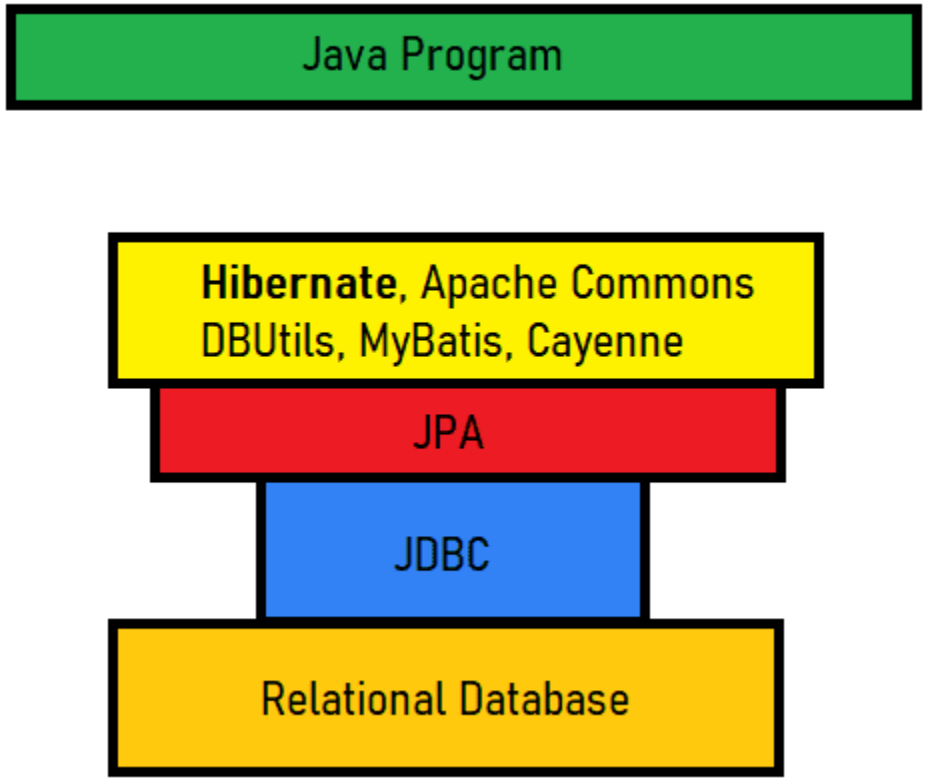
- Toplink - Oracle implementation [donated to Eclipse foundation for merge with Eclipselink 2008]
- Hibernate - Most deployed framework. Major contributor to JPA specification.
- OpenJPA - (openjpa.apache.org) which is an extension of Kodo implementation.



Hibernate

Hibernate is an open-source, non-invasive, lightweight java ORM(Object-relational mapping) framework to develop objects which are independent of the database software and make independent persistence logic in all JAVA, and JEE.

It simplifies the interaction of java applications with databases. Hibernate is an implementation of JPA.



Hibernate Demo

```
@Repository
public class EmployeeDAOImpl implements EmployeeDAO {

    @Autowired
    private EntityManagerFactory entityManagerFactory;

    @Override
    public void save(Employee p) {
        Session session = entityManagerFactory.unwrap(SessionFactory.class).openSession();
        Transaction tx = session.beginTransaction();
        session.persist(p);
        tx.commit();
        session.close();
    }


    @Override
    public List<Employee> getAllEmployees() {
        Session session = entityManagerFactory.unwrap(SessionFactory.class).openSession();
        List<Employee> personList = session.createQuery("from Employee").list();
        session.close();
        return personList;

    }
}
```

Hibernate Pros

Pros:

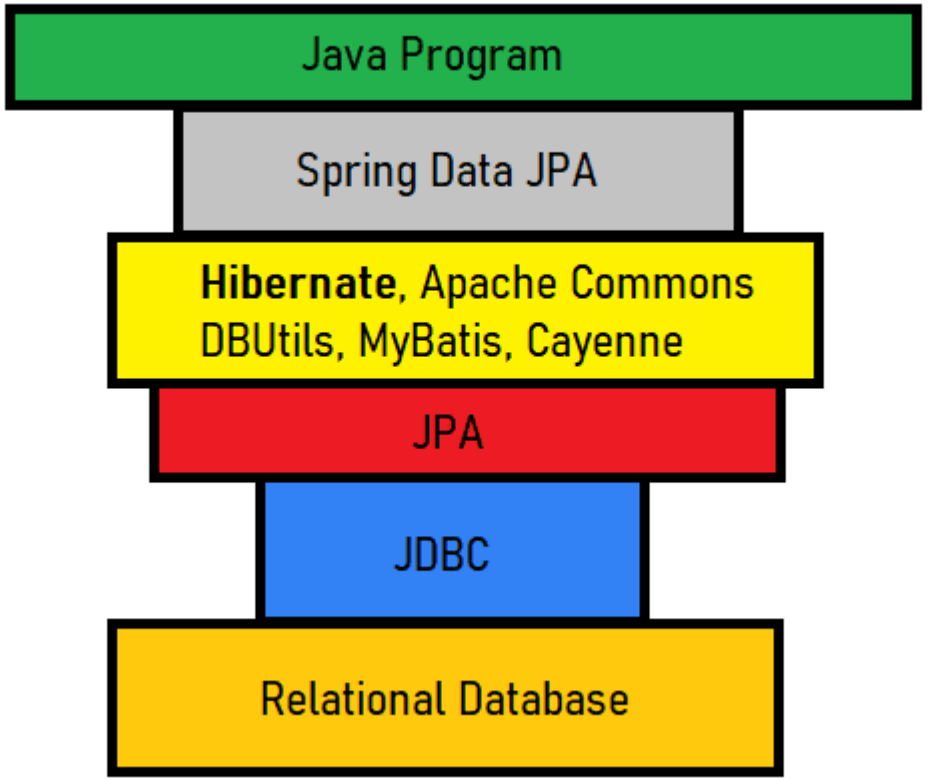
1. **Object-Relational Mapping (ORM):** Hibernate provides a powerful ORM framework that allows you to map Java objects to database tables. This abstraction simplifies database interactions and reduces the need for writing SQL queries manually.
2. **Productivity:** Hibernate reduces the amount of boilerplate code required for database operations, making development faster and more efficient. You can work with plain Java objects instead of dealing with ResultSet and PreparedStatement.
3. **Portability:** Hibernate is database-agnostic, meaning you can switch between different relational database management systems (RDBMS) without changing your application code. It abstracts the database-specific SQL syntax.
4. **Query Language (HQL and Criteria API):** Hibernate offers HQL (Hibernate Query Language) and Criteria API, which are more intuitive and readable than raw SQL. They allow for dynamic and type-safe queries.
5. **Caching:** Hibernate provides caching mechanisms, both at the first and second levels, to improve application performance by reducing database round-trips.
6. **Automatic Schema Generation:** Hibernate can generate database schemas automatically based on your entity mappings, which can save time during development.

Hibernate Cons

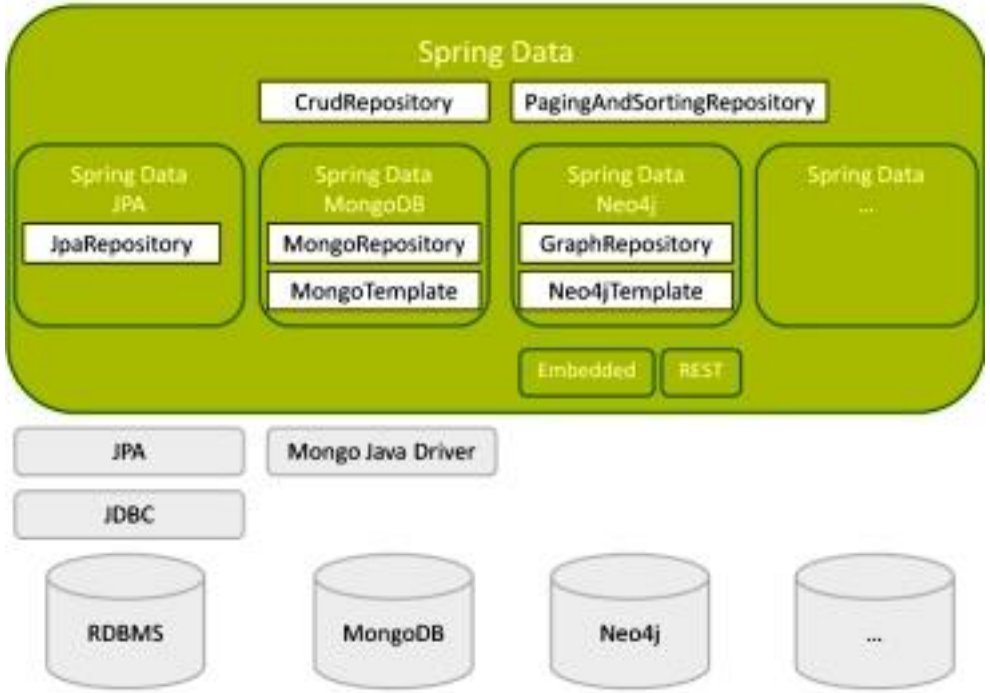
1. **Learning Curve:** Hibernate has a learning curve, especially for developers new to ORM concepts. It may take time to fully understand and use its features effectively.
2. **Performance Overhead:** Hibernate introduces some performance overhead due to its abstraction layer. While this overhead is usually negligible for most applications, it might be a concern for high-performance systems.
3. **Complexity:** Complex database models and queries may require a deep understanding of Hibernate configurations, and custom mappings can be challenging.
4. **Maintenance:** Hibernate can generate complex SQL queries, which can be difficult to optimize and tune for performance in some cases.

Spring Data

- High level SpringSource project whose purpose is to unify and ease the access to different kinds of persistence stores, both relational database systems and NoSQL data stores.
- It adds an extra layer of abstraction on the top of your JPA provider such as Hibernate.
- Significantly reduce the amount of boilerplate code required to implement data access layers
- Domain Object specific wrapper that provides capabilities on top of EntityManager
- Performs function of a Base Class DAO



Spring Data Project



Spring Data Example

```
@Repository
public interface EmployeeDAO extends CrudRepository<Employee, Integer> {
}
```

JPA Fundamentals

JPA ORM Fundamentals

Entity

- lightweight persistence domain object
- Annotation driven Entities - @Entity

Persistence Context ~= Session in Hibernate

- Like ***cache*** which contains a set of **persistent entities**
- Within the **persistence context**, the entity instances and their **lifecycle are managed.**

EntityManager

- Basically a CRUD Service -- { persist, find, remove}.
- Can Find entities by their primary key, and to query over all entities.
- Can participate in a transaction.

Transaction Manager

- Java Transaction API
- General API for managing transactions in Java
- Start, Close, Commit, Rollback operations

Entity

- Are POJOs (Plain Old Java Objects)
- Lightweight persistent domain object
- Typically represent a table in a relational database
- Each entity instance corresponds to one row in that table
- Have a persistent identity
- May have both, persistent and transient (non-persistent) state
  - Simple types (primitive data types, wrappers, enums)
  - Composite types (e.g., Address)
  - Non-persistent state (using identifier `transient` or `@Transient` annotation)

Hibernate Annotations

Annotations	Use of annotations
@Entity	Used for declaring any POJO class as an entity for a database
@Table	Used to change table details, some of the attributes are- ○ name – override the table name ○ schema ○ catalogue ○ enforce unique constraints
@Id	Used for declaring a primary key inside our POJO class
@GeneratedValue	Hibernate automatically generate the values with reference to the internal sequence and we don’t need to set the values manually.

Hibernate Annotations

Annotations	Use of annotations
@Column	It is used to specify column mappings. It means if in case we don't need the name of the column that we declare in POJO but we need to refer to that entity you can change the name for the database table. Some attributes are- ○ Name – We can change the name of the entity for the database ○ length – the size of the column mostly used in strings ○ unique – the column is marked for containing only unique values ○ nullable – The column values should not be null. It's marked as NOT
@Transient	Tells the hibernate, not to add this column
@Temporal	This annotation is used to format the date for storing in the database
@Lob	Used to tell hibernate that it's a large object and is not a simple object like image
@OrderBy	This annotation will tell hibernate to OrderBy as we do in SQL. For example – we need to order by student firstname in ascending order @OrderBy("firstname asc")

JPA Annoations Example

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

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

    @Column(name = "firstname", nullable = false)
    private String firstName;
    private String lastname;
    private String address;

    private LocalDate birthdate;

    @Transient
    private int serial;

    @ManyToOne
    @JoinColumn(name = "id")
    private Department department;

}
```

```java
@Entity
public class Department {

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

    private String name;

    @OneToMany(mappedBy = "department")
    @OrderBy("firstName ASC")
    private List<Employee> employees;
}
```

Associations

HIBERNATE

Associations

Java associations are made with **references**

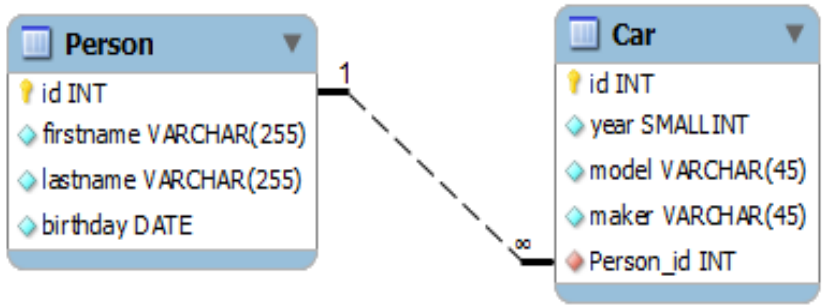
Person has cars collection of references

```
public class Person {
        private Long id;
        private String firstname;
        private String lastname;
        private List<Car> cars =
                new ArrayList<>();
```

```
public class Car {
        private Long id;
        private short year;
        private String model;
        private String maker;
        private Person owner;
```

Car has an owner reference back to Person

Relational association are made with **foreign keys**



Car has an Foreign Key to Person

ORM maps refs to FKs (and FKs to refs)

Directionality

OO has **uni-directional** and **bi-directional**

- Relational is always bi-directional (can emulate?)

```
public class Person {
        private Long id;
        private String firstname;
        private String lastname;
        private List<Car> cars =
                new ArrayList<>();
```

```
public class Car {
        private Long id;
        private short year;
        private String model;
        private String maker;
```

No ref to Person

Uni-directional
only peson has reference
to Car

```
public class Person {
        private Long id;
        private String firstname;
        private String lastname;
        private List<Car> cars =
                new ArrayList<>();
```

```
public class Car {
        private Long id;
        private short year;
        private String model;
        private String maker;
        private Person owner;
```

Bi-directional
can go either way

Types of Relationships

7 types of relationships: 4 uni, 3 bi-directional

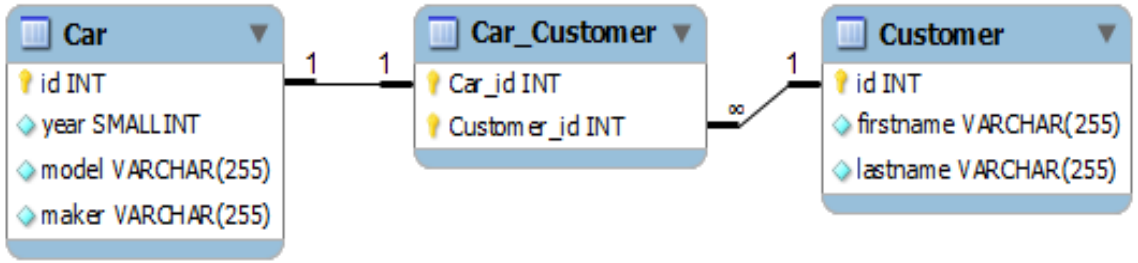
- ManyToOne and OneToMany are different sides of the same bi-directional relationship

Multiplicity	Uni-Directional	Bi-directional
One To One	Uni-Directional	Bi-Directional
Many To One	Uni-Directional	Bi-Directional
One To Many	Uni-Directional	
Many To Many	Uni-Directional	Bi-Directional

Join Table

Relational can use a table to hold foreign keys

- Required to make a many-to-many relationship
- Can also be used for **any relationship**



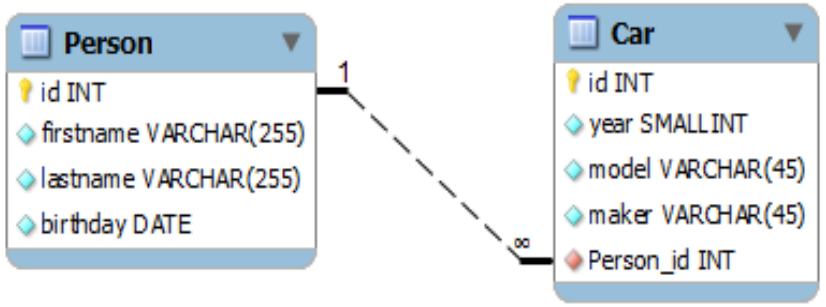
This concept has many names:

- Junction table, association table, link table, ...

Mapping Bi-directional

Relational has one FK for bi-directional

- Can be joined either direction



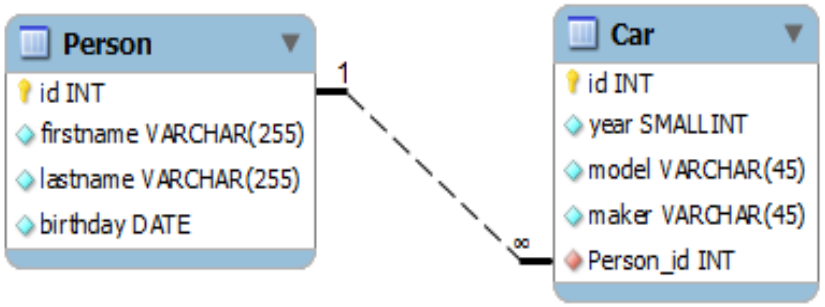
OO has two sides that both need reference(s)

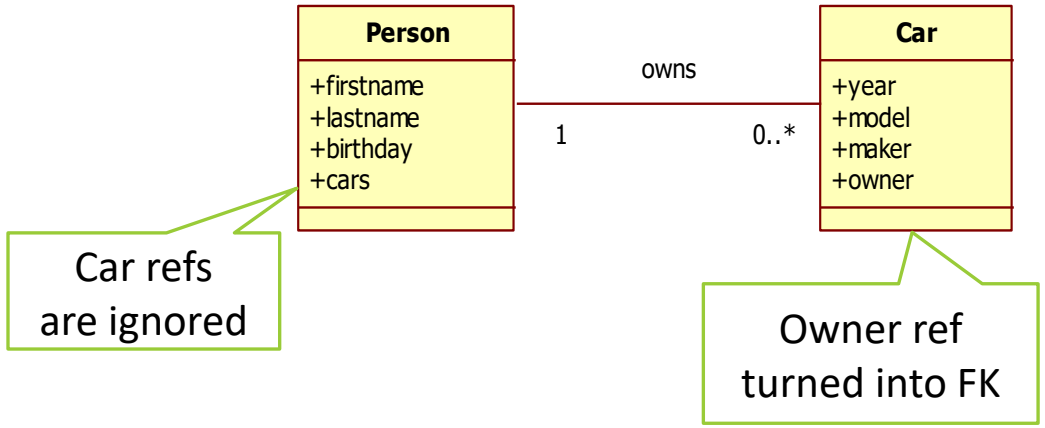
- One side will become the ‘owning side’

Owning Side

The owning side in a bi-directional association

- These references are turned into FK values
- **Other side** references are **ignored** when persisting
- In ManyToOne the many side is **natural owner**

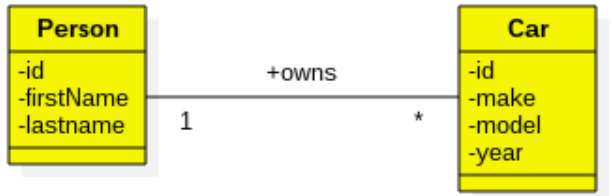


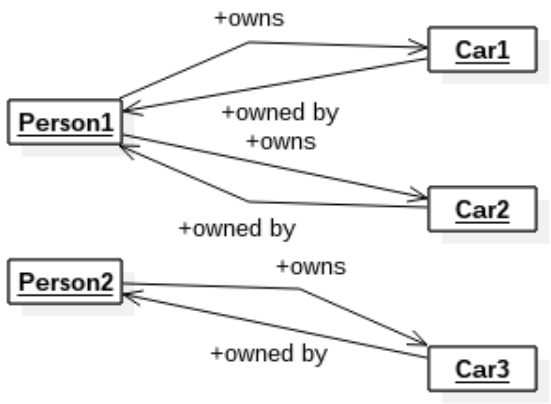


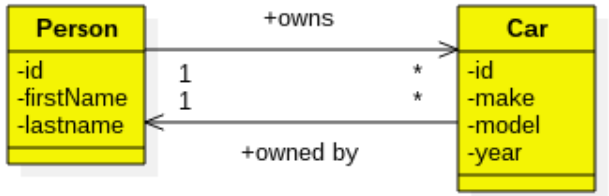
Bi-Directional VS 2 Uni-Directional

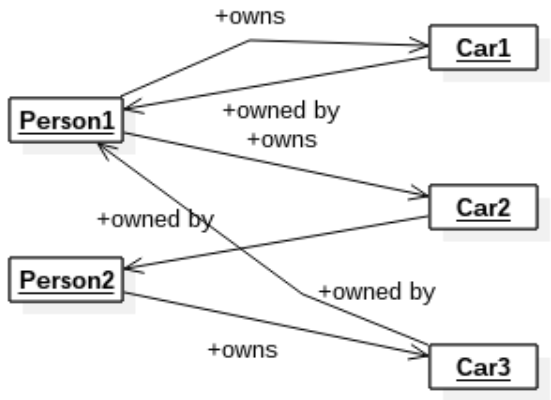
If you **do not specify** an owning side

- You get **two uni-directional** references!









Bi-Directional Convenience

Create convenience methods

- Properly **maintain bi-directional** association in Java

```
@Entity
public class Person {

        ...

        public boolean addCar(Car car) {
                if (cars.add(car)) {
                        car.setOwner(this);
                        return true;
                }
                return false;
        }

        public boolean removeCar(Car car) {
                if (cars.remove(car)) {
                        car.setOwner(null);
                        return true;
                }
                return false;
        }
```

Set both references

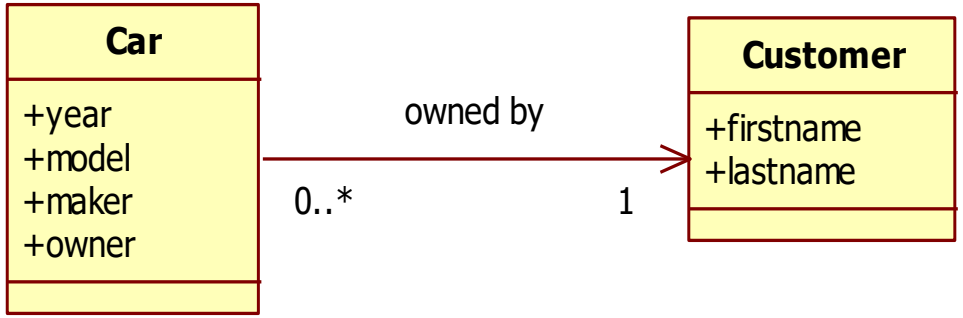
Unset both references

Association: ManyToOne

HIBERNATE

ManyToOne uni-directional

- OO:



A Car is owned by one Customer and a Customer can own many cars

- Relational:

FK is always on the many side

CAR table

ID	MAKER	MODEL	YEAR	CUSTOMER_ID
1	Honda	Acord	1996	1
2	Volvo	S80	1999	1

Car table has a Customer FK

CUSTOMER table

ID	FIRSTNAME	LASTNAME
1	Frank	Brown

```
@Entity
public class Car {
        @Id
        @GeneratedValue
        private Long id;
        private short year;
        private String model;
        private String maker;
        @ManyToOne
        @JoinColumn(name="customer_id")
        private Customer owner;

        ...
```

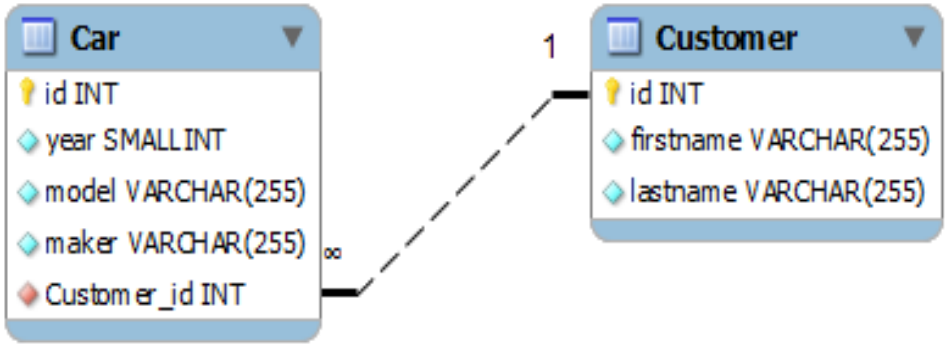
@ManyToOne

Optional @JoinColumn to name the column in DB

Default name: owner_id

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

        ...
```

Normally mapped Customer Entity



CAR table

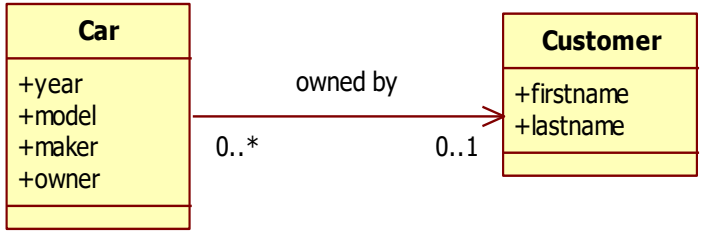
ID	MAKER	MODEL	YEAR	CUSTOMER_ID
1	Honda	Acord	1996	1
2	Volvo	S80	1999	1

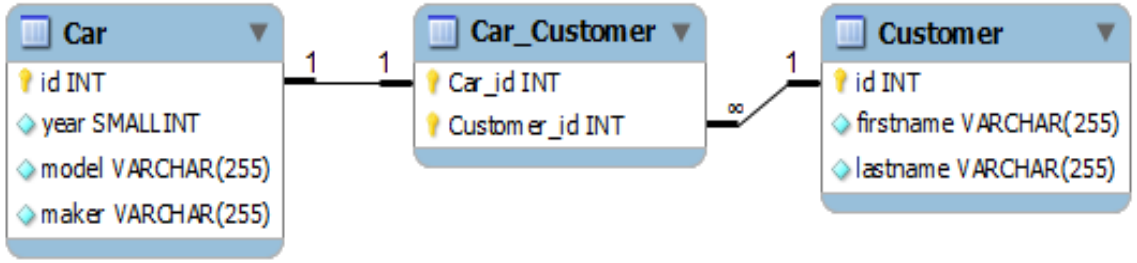
CUSTOMER table

ID	FIRSTNAME	LASTNAME
1	Frank	Brown

Join Table

- ManyToOne can be mapped with a JoinTable
  - Useful for optional (0..1) associations
  - Optional would require the FK to be nullable
    - Normalization does not like nullable columns



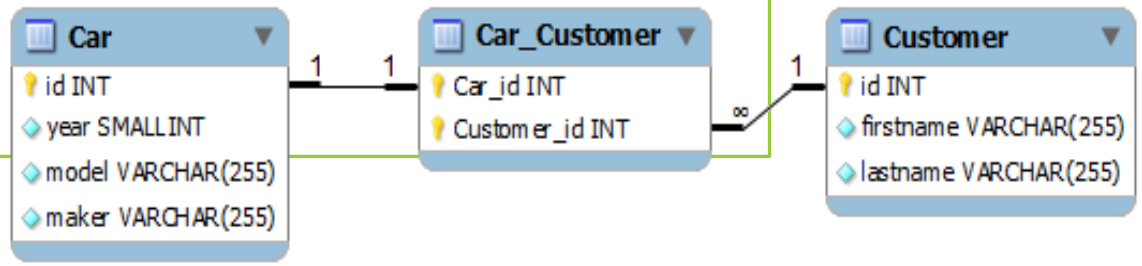


```
@Entity
public class Car {
        @Id
        @GeneratedValue
        private Long id;
        private short year;
        private String model;
        private String maker;
        @ManyToOne
        @JoinTable(name="car_customer",
         joinColumns = { @JoinColumn(name = "customer_id") },
         inverseJoinColumns = { @JoinColumn(name = "car_id") }
        )
        private Customer owner;

        ...
```

@JoinTable
and its name
are required

Column names
are optional

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

        ...
```



CAR table

ID	MAKER	MODEL	YEAR
1	Honda	Acord	1996
2	Volvo	S80	1999

CAR_CUSTOMER table

CUSTOMER_ID	ID
1	1

CUSTOMER table

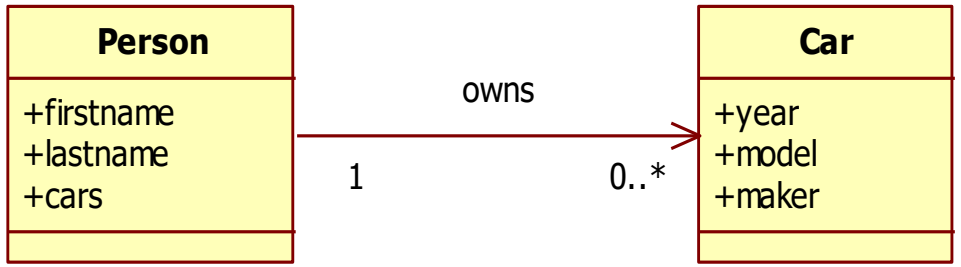
ID	FIRSTNAME	LASTNAME
1	Frank	Brown

Association: OneToMany

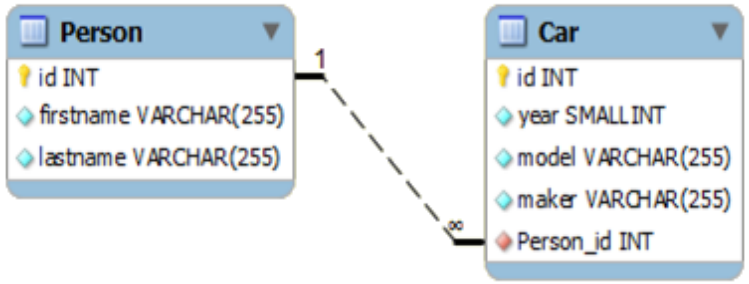
HIBERNATE

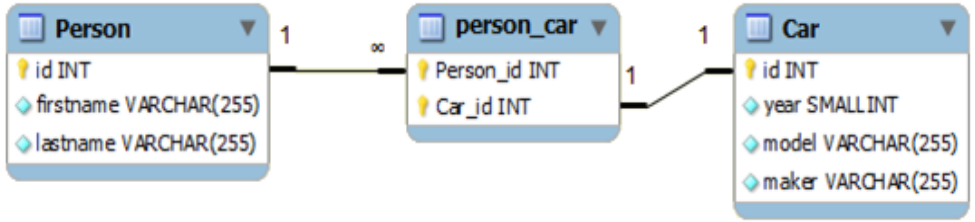
OneToMany

- OO:



- Relational
  - Again 2 options:

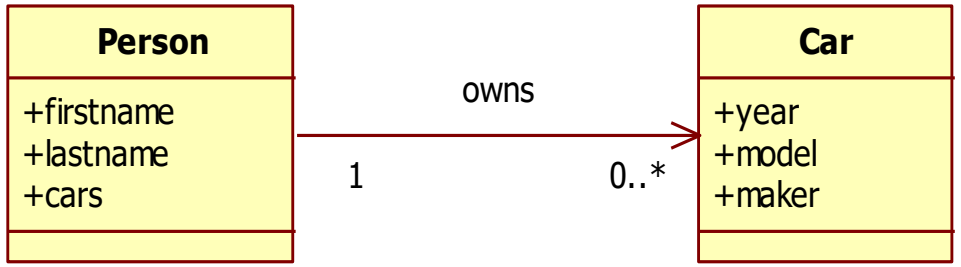


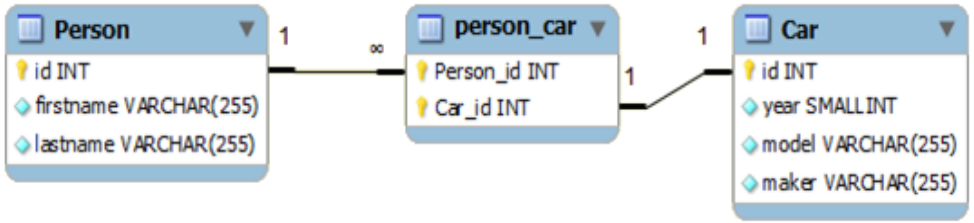


Default
is Join Table!

Why Default Join Table?

- **Uni-direct** OneToMany defaults to join table
  - OO: The many side **should not** have a reference
  - Relational: FK (like a ref) is **on many side**
    - Is it possible to have FK on other side?
    - Join Table only solution to this problem!





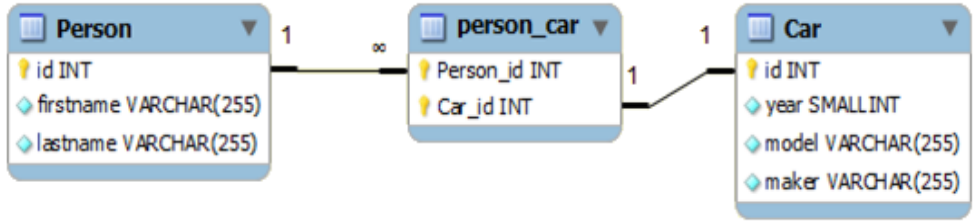
Uni-direct OneToMany

```
@Entity
public class Person {
        @Id
        @GeneratedValue
        private Long id;
        private String firstname;
        private String lastname;
        @OneToMany
        @JoinTable(name = "person_car", joinColumns =
@JoinColumn(name = "person_id"), inverseJoinColumns =
@JoinColumn(name = "car_id"))
        private List<Car> cars = new ArrayList<>();
```

```
@Entity
public class Car {
        @Id
        @GeneratedValue
        private Long id;
        private short year;
        private String model;
        private String maker;

        ...
```

Makes Join Table!

@JoinTable can be added to change table and column names



PERSON table

ID	FIRSTNAME	LASTNAME
1	Frank	Brown

PERSON_CAR table

PERSON_ID	CAR_ID
1	1
1	2

CAR table

ID	MAKER	MODEL	YEAR
1	Honda	Acord	1996
2	Volvo	S80	1999

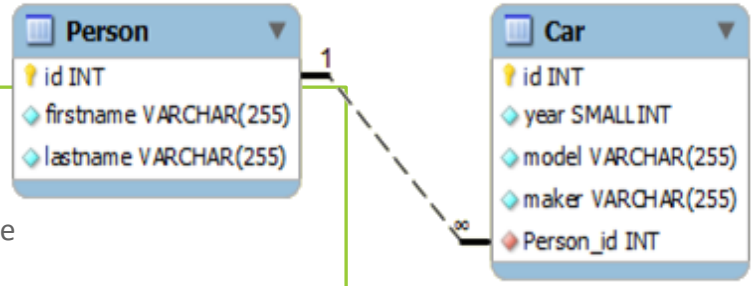
JoinColumn

- Does not match the spirit of Uni-Directional
  - Does work when specified

```
@Entity
public class Person {
        @Id
        @GeneratedValue
        private Long id;
        private String firstname;
        private String lastname;
        @OneToMany
        @JoinColumn(name="person_id")
        private List<Car> cars =
                new ArrayList<>();
```

@JoinColum
name defaults to:
cars_id

```
@Entity
public class Car {
        @Id
        @GeneratedValue
        private Long id;
        private short year;
        private String model;
        private String maker;

        ...
```



PERSON table

ID	FIRSTNAME	LASTNAME
1	Frank	Brown

CAR table

ID	MAKER	MODEL	YEAR	PERSON_ID
1	Honda	Acord	1996	1
2	Volvo	S80	1999	1

Association: Bi-Directional

HIBERNATE

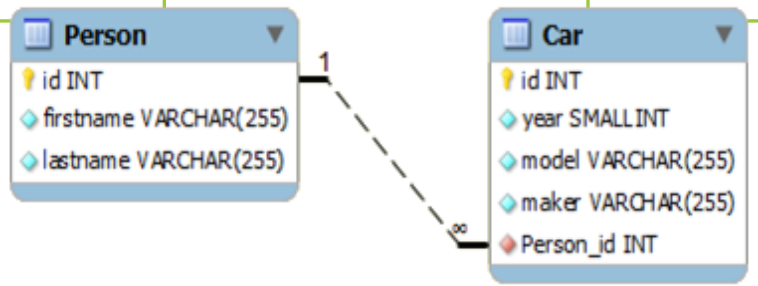
ManyToOne / OneToMany

- Bi-directional OneToMany == ManyToOne
  - Needs owning side → **mappedBy()**

```
@Entity
public class Person {
        @Id
        @GeneratedValue
        private Long id;
        private String firstname;
        private String lastname;
        @OneToMany(mappedBy="person")
        private List<Car> cars =
                new ArrayList<>();

        ...
```

Name of property on other side that has the mapping

```
@Entity
public class Car {
        @Id
        @GeneratedValue
        private Long id;
        private short year;
        private String model;
        private String maker;
        @ManyToOne
        private Person person;

        ...
```



Which Side?

- mappedBy() says other side mapped this association
  - Gives up control of the association
  - Says that the **other side is the owning side**

- For Bi-Directional OneToMany / ManyToOne:
  - Only @OneToMany has mappedBy() option

@ManyToOne
cannot say mappedBy()
Even if it wanted to!

Side with the FK,
Natural owner
of the association

Join Table

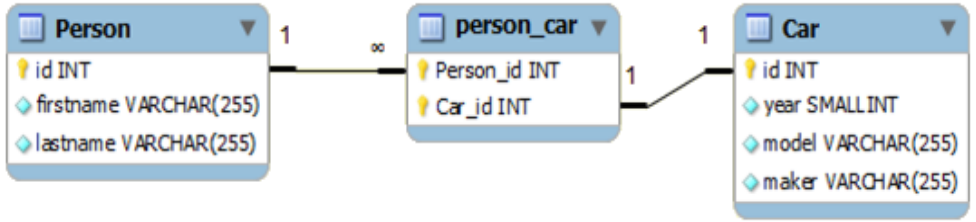
- You can use a Join Table
  - Annotation has to be on @ManyToOne

```
@Entity
public class Person {
        @Id
        @GeneratedValue
        private Long id;
        private String firstname;
        private String lastname;
        @OneToMany(mappedBy="person")
        private List<Car> cars =
                new ArrayList<>();

        ...
```

Because of mappedBy() extra mapping annotations throws AnnotationException

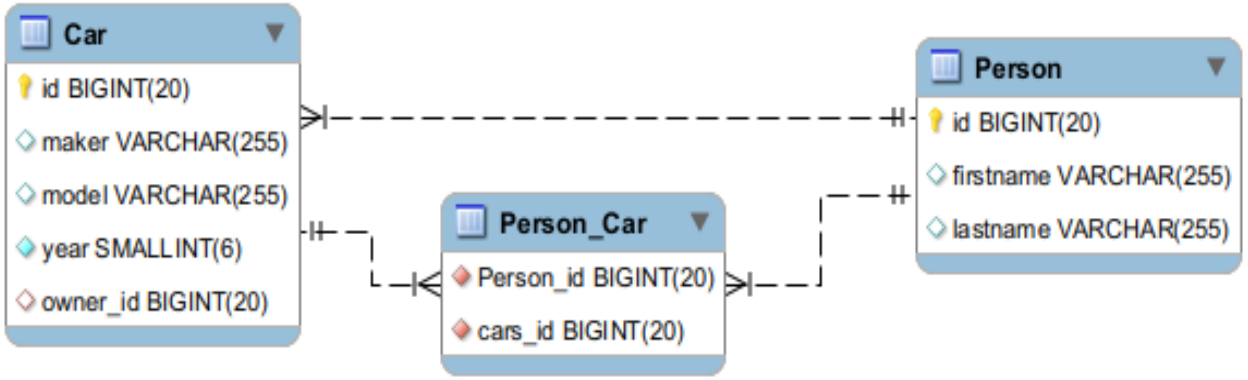
Only on this side do extra mapping annotations work

```
@Entity
public class Car {
        @Id
        @GeneratedValue
        private Long id;
        private short year;
        private String model;
        private String maker;
        @ManyToOne
        @JoinTable(name = "person_car",
                joinColumns = {
                        @JoinColumn(name = "car_id")
                }, inverseJoinColumns = {
                        @JoinColumn(name = "person_id")
                })
        private Person person;

        ...
```



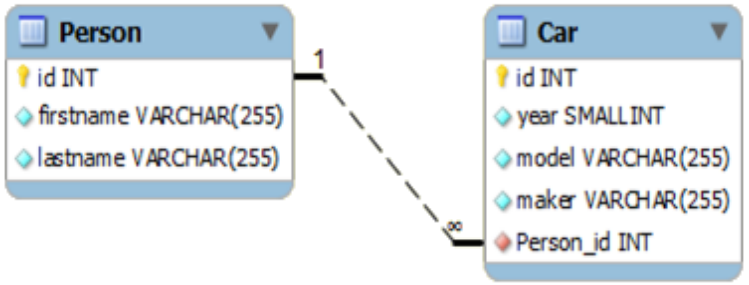
No mappedBy()

- What happens if you forget mappedBy()?
  - 2 uni-directional associations
  - Uni-directional @ManyToOne uses FK
  - Uni-directional @OneToMany uses Join Table



JoinColumn

- What if you forget mappedBy()
  - But specify @JoinColumn on @OneToMany
  - No join table, schema looks okay
  - By default, two FKs is generated



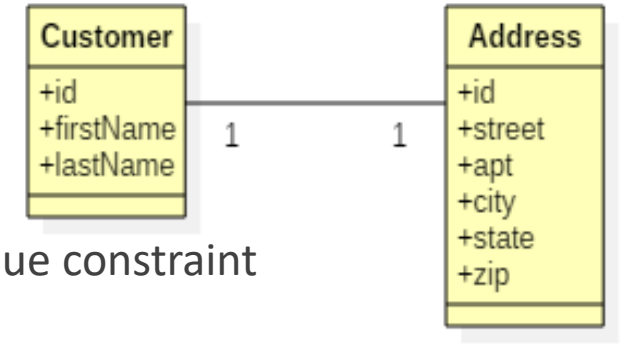
- Both sides will update the one FK
  - Creating a **race condition** (not sure which one wins)
  - Bad programming!

Association: OneToOne

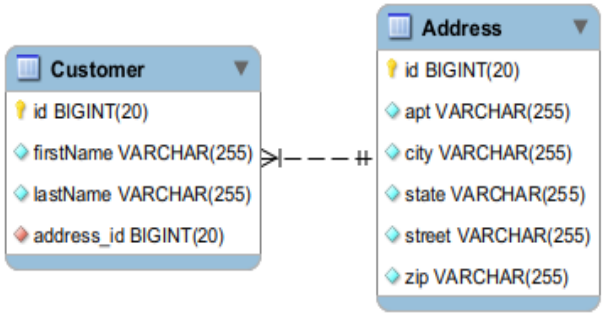
HIBERNATE

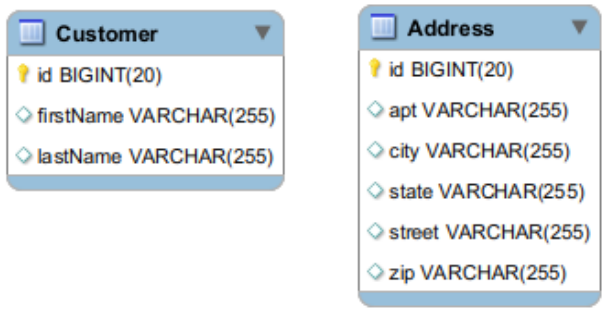
OneToOne

- OO: Customer and Address (if bi-directional) have a reference to each other



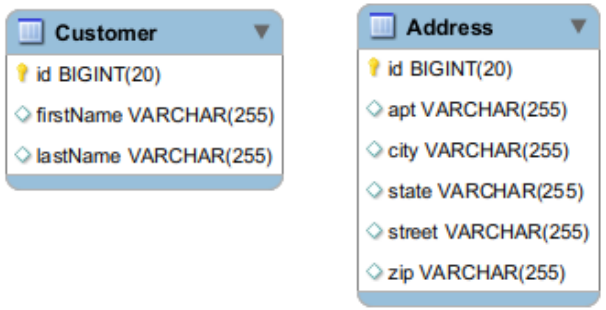
- Relational, **two options**:
  - FK (on one side) with unique constraint
  - Shared Primary Keys





Shared Primary Key

- Shared Primary Key uses the Primary Key as Foreign Key
  - By having the **same value** rows connect



CUSTOMER table

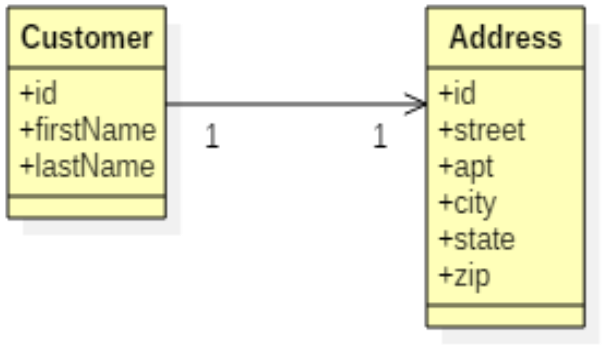
ID	FIRSTNAME	LASTNAME
1	John	Smith
2	Frank	Brown
3	Jane	Doe

ADDRESS table

ID	CITY	STATE	STREET	SUITEORAPT	ZIP
1	city1	state1	street1	suite1	zip1
3	city3	state3	street3	suite3	zip3

Uni-Directional FK

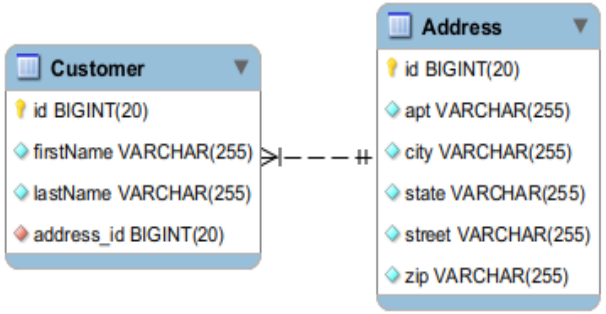
- Uni-directional use a FK
  - On the side that has the reference
  - Best match for spirit of uni-direct



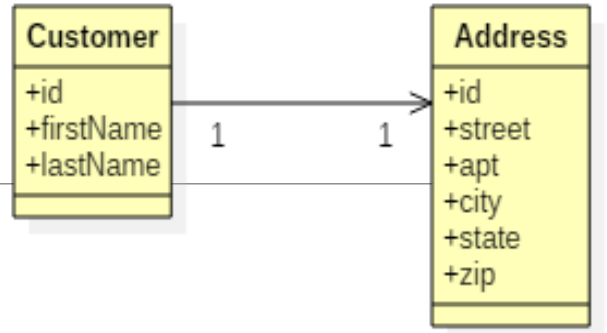

```
@Entity
public class Customer {
        @Id
        @GeneratedValue
        private Long id;
        private String firstName;
        private String lastName;
        @OneToOne
        private Address address;

        ...
```

```
@Entity
public class Address {
        @Id
        @GeneratedValue
        private Long id;
        private String street;
        private String apt;
        private String city;
        private String state;
        private String zip;

        ...
```

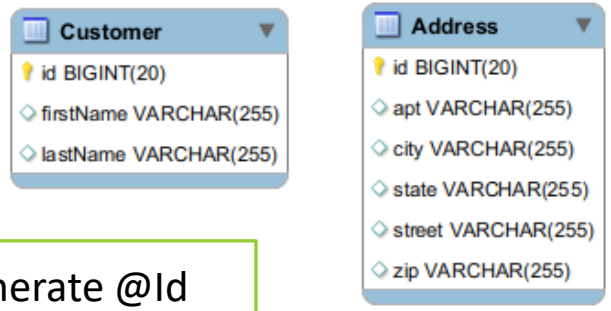
Simply place
@OneToOne
on the association



Uni-Directional Shared PK



- Not as 'in the spirit'
  - Works properly if you specify it
  - Remember to **assign the ID** for address!




```
@Entity
public class Customer {
        @Id
        @GeneratedValue
        private Long id;
        private String firstName;
        private String lastName;
        @OneToOne
        @PrimaryKeyJoinColumn
        private Address address;

        ...
```

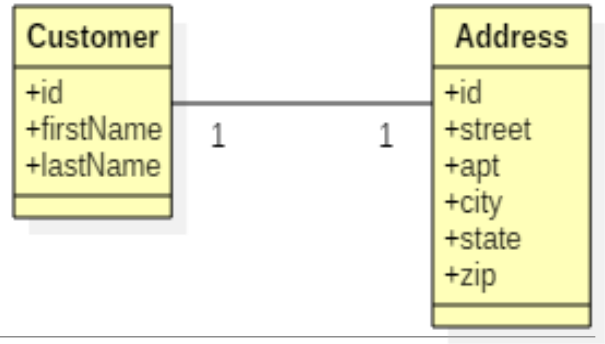
```
@Entity
public class Address {
        @Id
        private Long id;
        private String street;
        private String apt;
        private String city;
        private String state;
        private String zip;

        ...
```

Cannot generate @Id

The value has to be same as ID value of Customer
Programmer has to set it!

Add **@PrimaryKeyJoinColumn** to the association

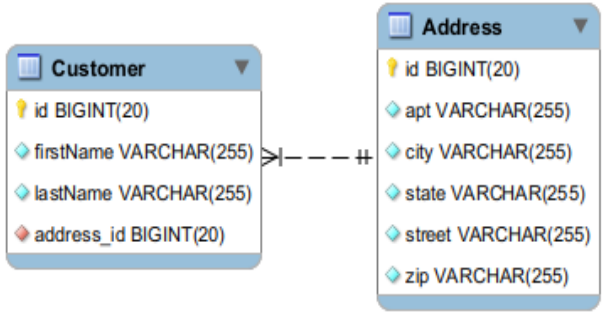
Bi-Directional FK



- A bi-directional associations based on a FK
  - Uses **@OneToOne** on both sides
  - One side has to give up control with **mappedBy()**

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

        ...
```

```
@Entity
public class Address {
        @Id
        @GeneratedValue
        private Long id;
        private String street;
        private String apt;
        private String city;
        private String state;
        private String zip;
        @OneToOne(mappedBy="address")
        private Customer customer;

        ...
```



From a business perspective Address is less important therefore it gives up ownership (says mappedBy)

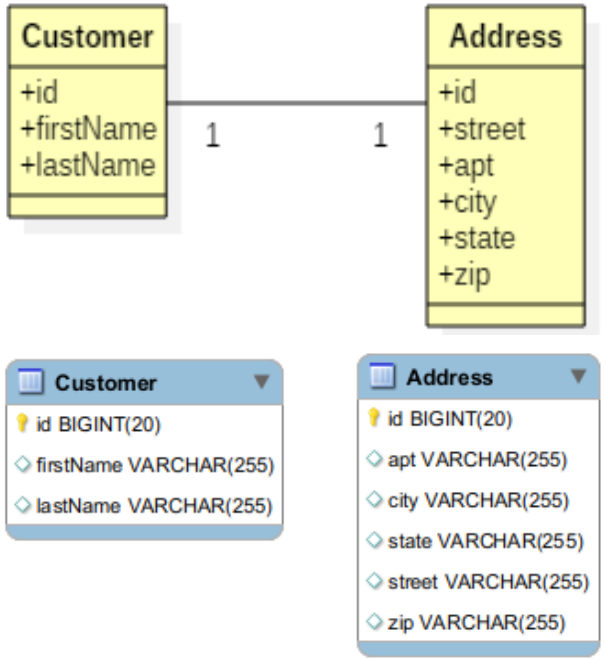
Bi-Directional Shared PK

- The 'owning side' generates the ID
  - Programmer **manually sets value** on the other side

```
@Entity
public class Customer {
        @Id
        @GeneratedValue
        private Long id;
        private String firstName;
        private String lastName;
        @OneToOne
        @PrimaryKeyJoinColumn
        private Address address;
```

```
@Entity
public class Address {
        @Id
        private Long id;
        private String street;
        private String apt;
        private String city;
        private String state;
        private String zip;
        @OneToOne
        @PrimaryKeyJoinColumn
        private Customer customer;

        ...
```

Both sides specify
@PrimaryKeyJoinColumn
No need for mappedBy



Association: ManyToMany

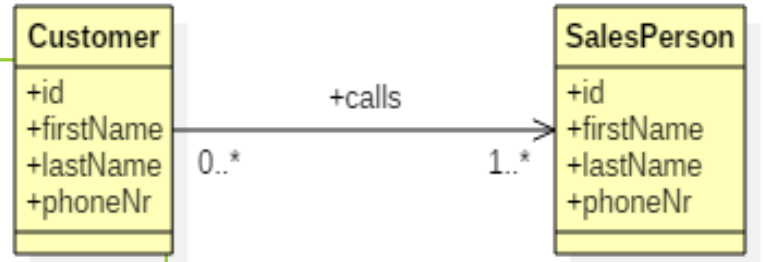
HIBERNATE

Uni Directional ManyToMany

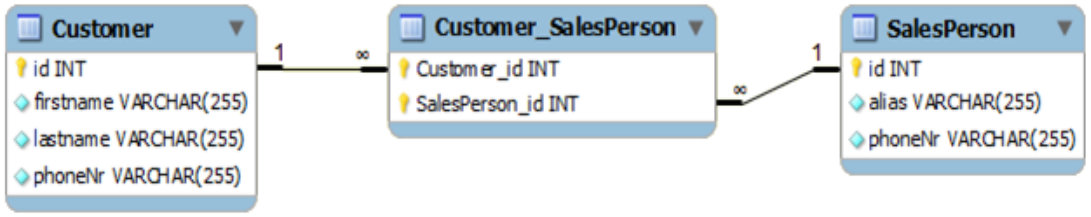
- **@ManyToMany** associations can only be implemented with a JoinTable

```
@Entity
public class Customer {
        @Id
        @GeneratedValue
        private Long id;
        private String firstName;
        private String lastName;
        @ManyToMany
        private List<SalesPerson> salesPeople
                = new ArrayList<>();

        ...
```

```
@Entity
public class SalesPerson {
        @Id
        @GeneratedValue
        private Long id;
        private String alias;
        private String phoneNr;

        ...
```



Optionally add @JoinTable to specify name of table and / or column names

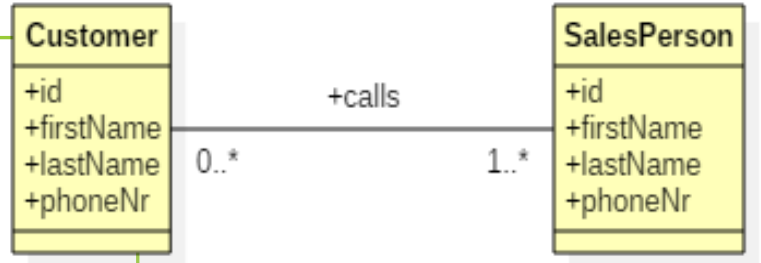


Bi-Directional ManyToMany

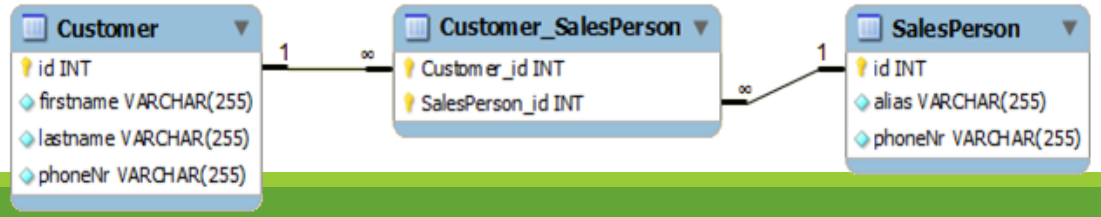
- **Choose** which side specifies **mappedBy**
  - Business may find one side more important (owner)

```
@Entity
public class Customer {
        @Id
        @GeneratedValue
        private Long id;
        private String firstName;
        private String lastName;
        @ManyToMany(mappedBy="customers")
        private List<SalesPerson> salesPeople
                = new ArrayList<>();

        ...
```

```
@Entity
public class SalesPerson {
        @Id
        @GeneratedValue
        private Long id;
        private String alias;
        private String phoneNr;
        @ManyToMany
        private List<Customer> customers
                = new ArrayList<>();
        ...
```

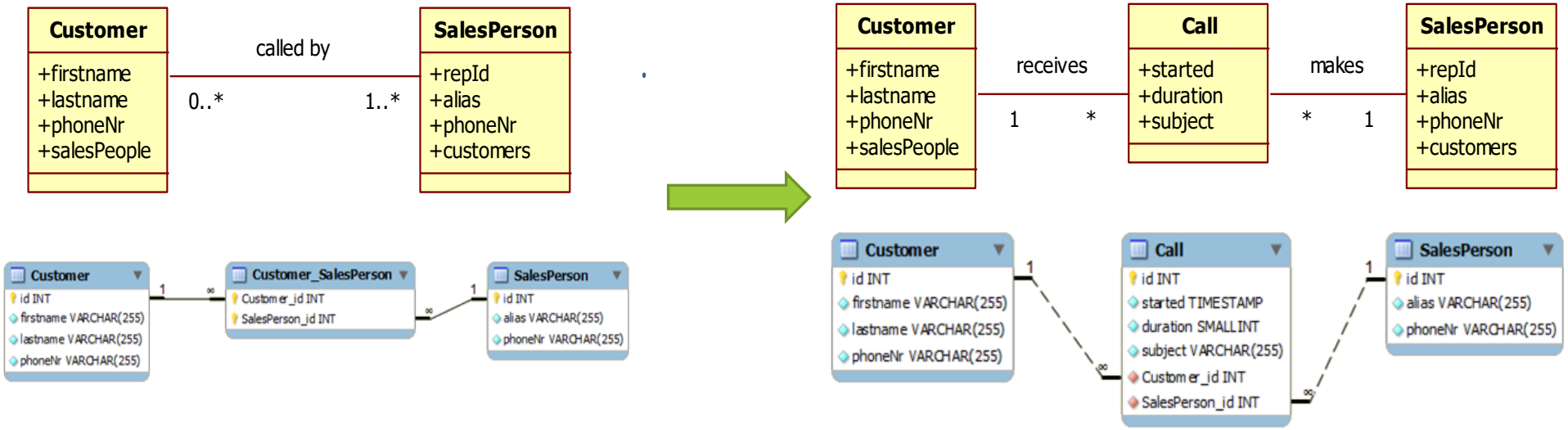


Both specify @ManyToMany
The owning side can also
specify @JoinTable to give names



Reconsider

- During Domain Analysis **consider changing** ManyToMany to two OneToMany



- ManyToMany are often interesting connections
  - Maybe you want to keep data on how / what connected
  - Turn JoinTable into an entity

Cascades

HIBERNATE

An Entity's Object States Relationship with the ORM Persistence Context

New, or Transient - the entity has just been instantiated and is not associated with a persistence context. It has no persistent representation in the database and no identifier value has been assigned.

Managed, or Persistent - the entity has an associated identifier and is associated with a persistence context.

Detached - the entity has an associated identifier, but is no longer associated with a persistence context (usually because the persistence context was closed or the instance was evicted from the context).

Removed - the entity has an associated identifier and is associated with a persistence context, however it is scheduled for removal from the database.

Cascade Configurable Parent-Child operations

Cascade Types

- ALL, PERSIST, MERGE, REMOVE, REFRESH, DETACH
- Default is none
- **Persist**
  - If the parent is persisted so are the children
- **Remove**
  - If the parent is “removed” so are the children
- **Merge [a detatched object]**
  - If the parent is merged so are the children
    - Merge modifications made to the detached object are merged into a corresponding **DIFFERENT** managed object

CascadeType

- You can specify **which operations** cascade
  - Every association has the cascade option

```
@Entity
public class Customer {
        @Id
        @GeneratedValue
        private Long id;
        private String firstName;
        private String lastName;
        @OneToMany(mappedBy="customer", cascade= {CascadeType.MERGE, CascadeType.PERSIST})
        private List<CreditCard> cards = new ArrayList<>();
        @OneToOne(cascade=CascadeType.ALL)
        private Address address;
```

Can be specified as a list

Or as a single value

Persisting a Customer now automatically also persists all linked SalesPerson and Address Objects

CascadeTypes
ALL
DETACH
MERGE
PERSIST
REFRESH
REMOVE

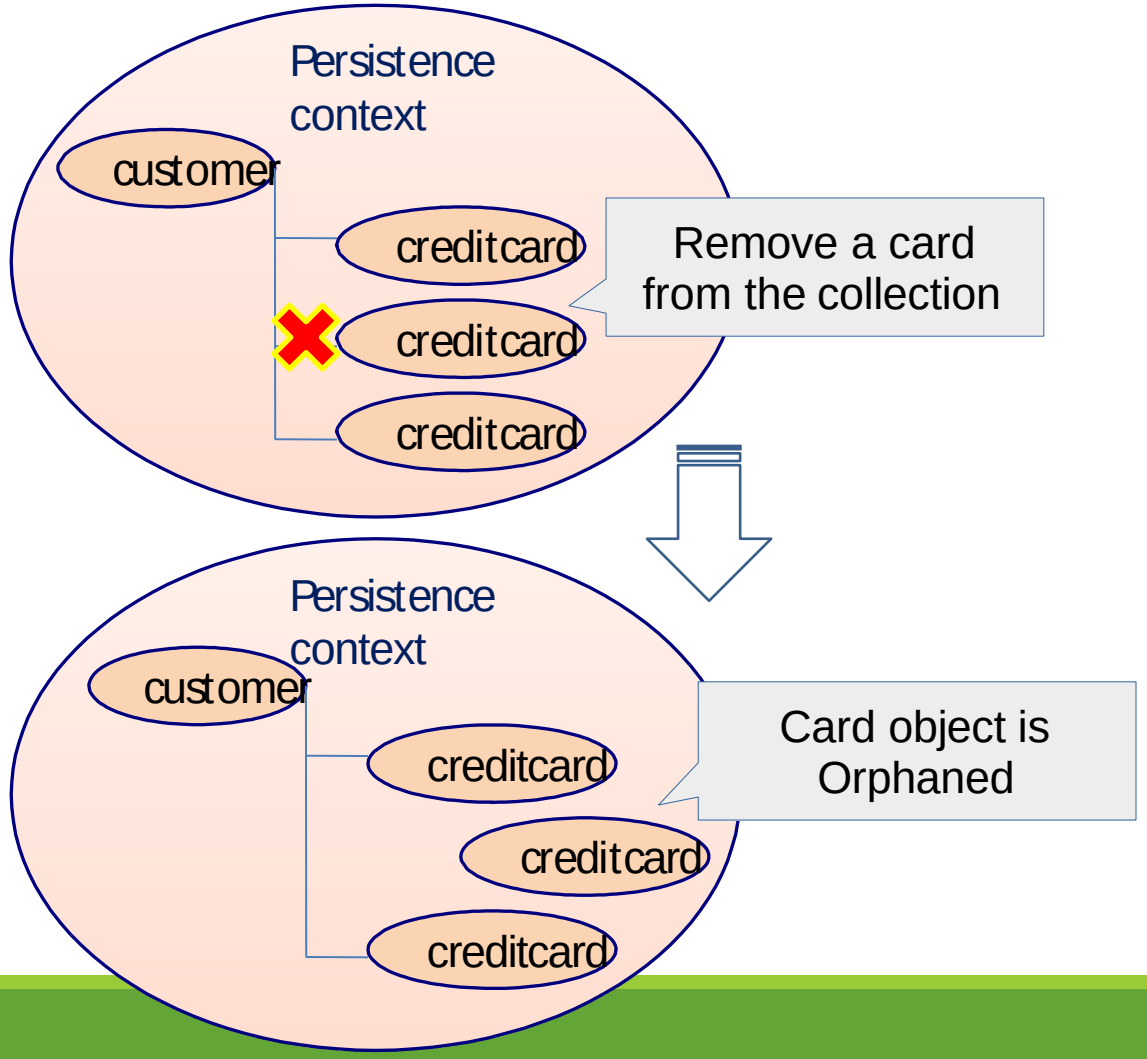
Orphan Removal

- Orphan removal is a topic related to cascades
  - Option on @OneToMany and @OneToOne
    - Both for Uni-directional and Bi-directional
  - When the connection / **reference is broken**, the entity that was referred to is **automatically removed**

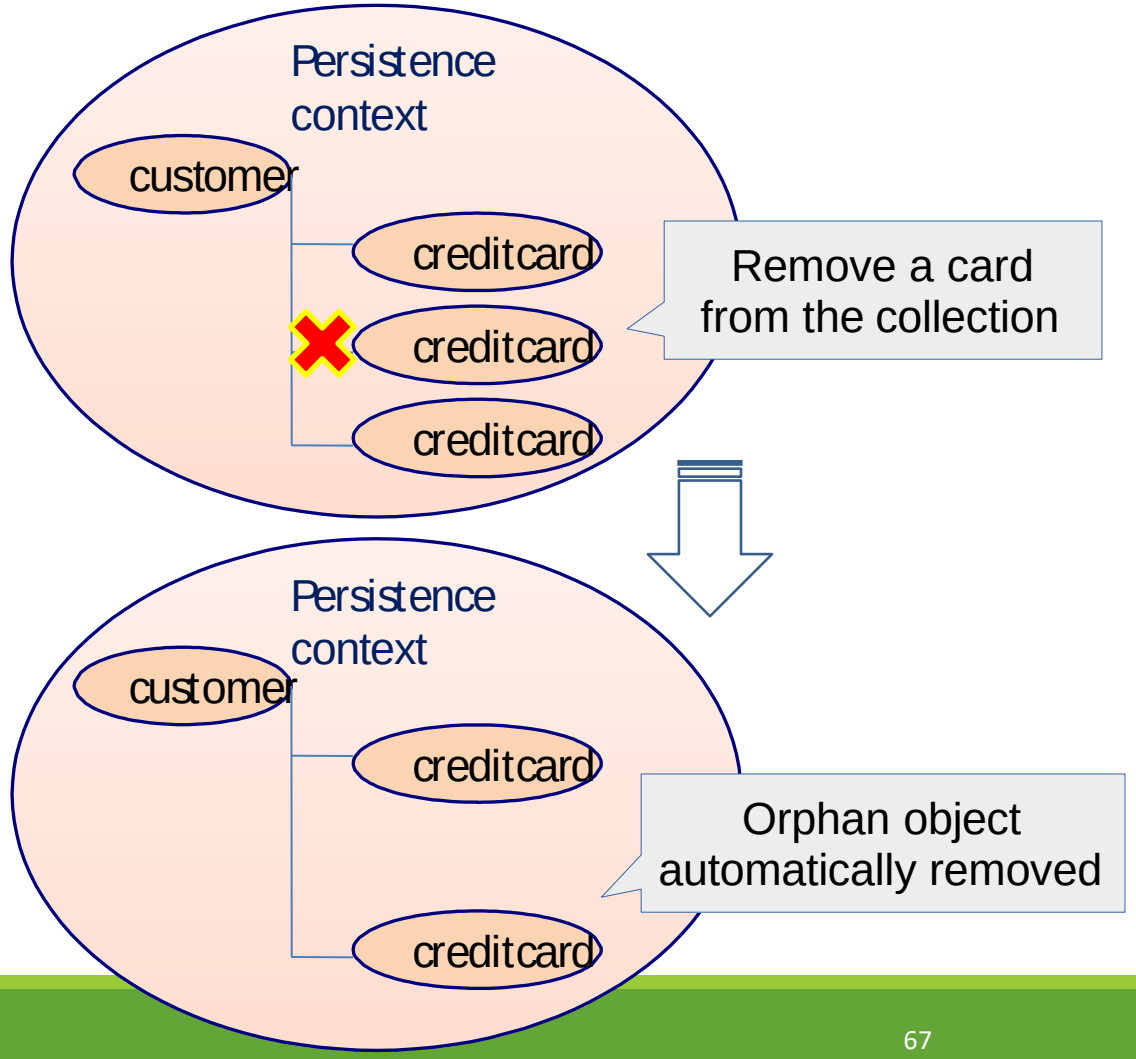
```
@Entity
public class Customer {
        @Id
        @GeneratedValue
        private Long id;
        private String firstname;
        private String lastname;
        @OneToMany(mappedBy="owner", orphanRemoval=true)
        private List<CreditCard> cards =
                new ArrayList<>();

        ...
```

Orphan Removal

One to Many without Orphan Removal



One to Many using Orphan Removal



Summary

- There are 7 types of associations
  - Bi-Directional associations need an owning side
  - Use mappedBy to give up control (not be owner)
- Mapping choices:
  - JoinTable of JoinColumn (OneToMany/ManyToOne)
  - Shared PK or FK (OneToOne)
- Cascades:
  - Allowing operations to follow references
- How connections are made is as important as the parts themselves The whole is greater than the sum of the parts

References

- https://dev.to/yigi/hibernate-vs-jdbc-vs-jpa-vs-spring-data-jpa-1421
- https://www.geeksforgeeks.org/hibernate-annotations/