Object-Relational Mapping

SCS 2209 | Database II

Hello!

I am Ravindu Sachintha

Senior Software Engineer,
CodeGen International

You can reach me at:
ravindusachintha53@gmail.com

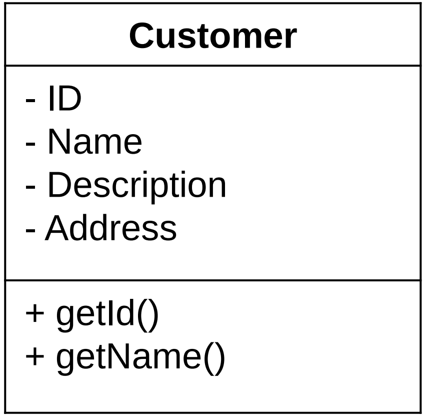
1.

Overview of ORM

OOP & Relational Databases

Problems in OOP & RDBMS

Business Classes & Objects

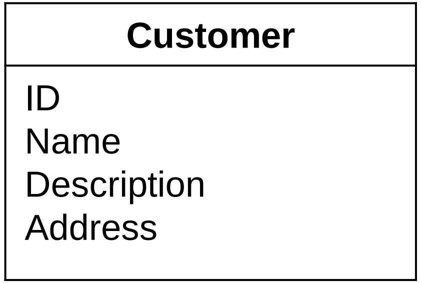


Customer

- ID
- Name
- Description
- Address

+ getId()
+ getName()

Database Tables & Records



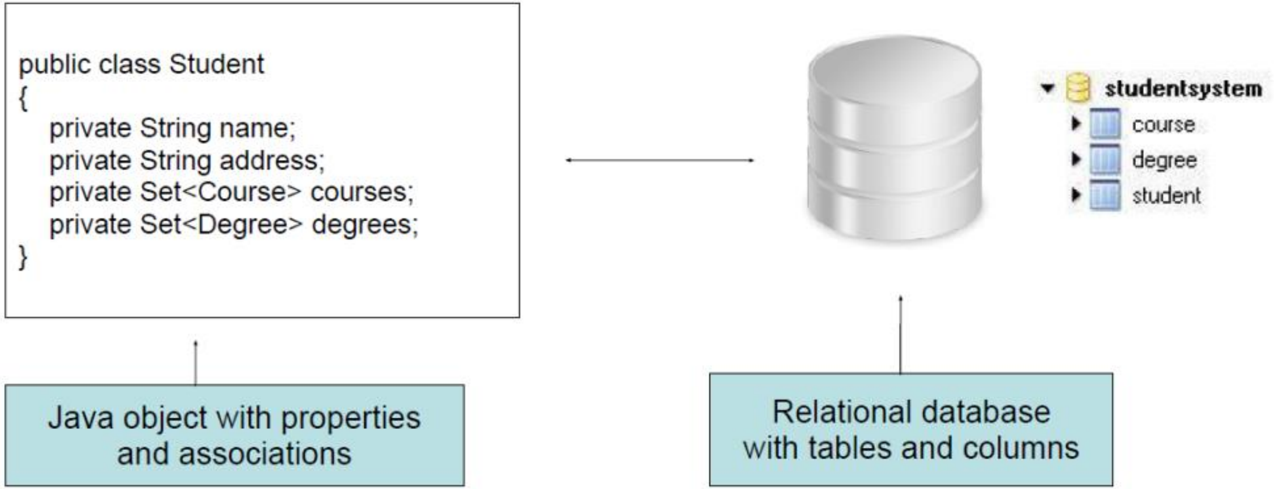
Customer

ID
Name
Description
Address

Relationships? Cardinality?

When working with object-oriented systems, there's a ***mismatch*** *between the* ***object model*** *and the* ***relational database.***

(Impedance Mismatch)



```
public class Student
{
   private String name;
   private String address;
   private Set<Course> courses;
   private Set<Degree> degrees;
}
```

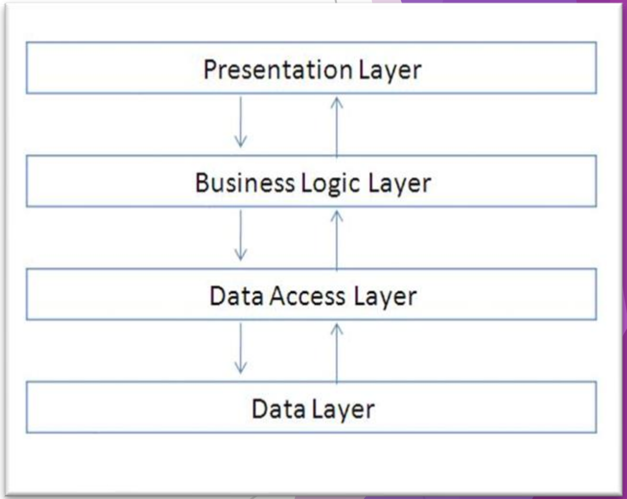
Java object with properties and associations

Relational database with tables and columns

A solution for data persistence with Layered (N-tier) Architecture

Needs to,

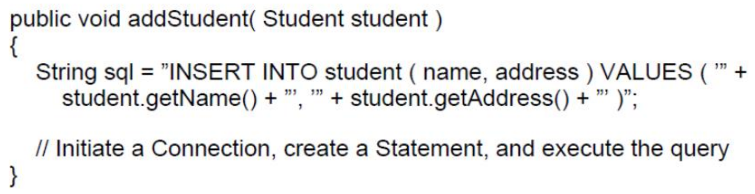
- **build SQL statements** for CRUD operations
- handle **data types** in objects for **data fields** in tables
- handle **data values** for special cases (ex:- empty strings, date formats, null values, etc.)
- handle **IDs, keys,..**



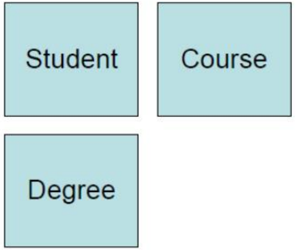
Problems with traditional approaches

Writing SQL conversion methods by hand (using DB Connection / JDBC),

- Tedious and requires **lots of code**
- Extremely **error-prone**
- Non-standard SQL **ties** the application to **specific databases**
- Vulnerable to **changes** in the **object model**
- Difficult to represent **associations between objects**



```
public void addStudent( Student student )
{
   String sql = ”INSERT INTO student ( name, address ) VALUES ( ‘” +
      student.getName() + ”’, ‘” + student.getAddress() + ”’ )”;

   // Initiate a Connection, create a Statement, and execute the query
}
```



Student

Course

Degree

Object-Relational Mapping (ORM)

A programming technique for converting **data between incompatible systems**

ORM

- Converts data between **relational databases** (e.g.:- Oracle, MySQL) and **object-oriented** programming languages (e.g.:- Java, TypeScript).

- It creates a model of the object-oriented program with a **high level of abstraction**. The mapping describes the relationship between an object and data without knowing how the data is structured.

- It **eliminates** the need to create a **data layer tier** (data layer is implicit).

Types of ORMs

Active Record Pattern

Maps data within the structure of objects in the code.

(e.g:- Ruby on Rails, Laravel's Eloquent)

Pros:

- Simple
- Easy to learn and understand

Cons:

- High database coupling (and testing)
- Performance bottlenecks

Data-mapper Pattern

Decouple the business logic in the objects from the database.

(e.g:- Java Hibernate, Doctrine-Symfony)

Pros:

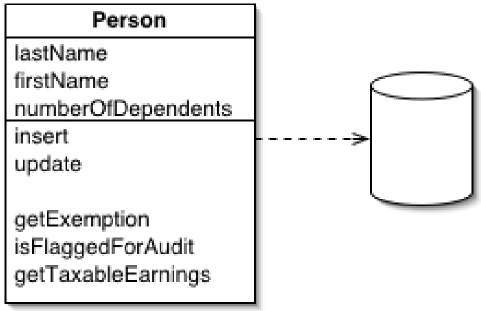
- Greater flexibility between domain and database
- More performant(compared to AR)

Cons:

- Hard to set up

Types of ORMs

Active Record Pattern



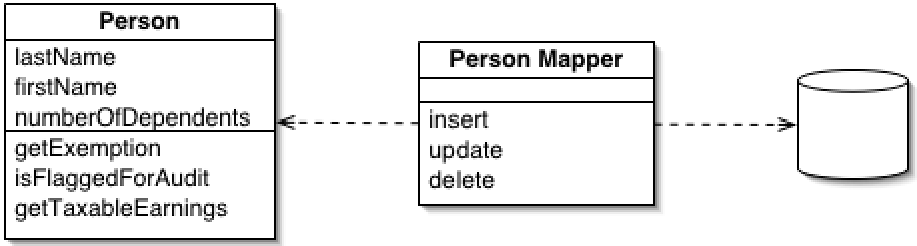
Person

lastName
firstName
numberOfDependents

insert
update

getExemption
isFlaggedForAudit
getTaxableEarnings

Data-mapper Pattern



Person

lastName
firstName
numberOfDependents

getExemption
isFlaggedForAudit
getTaxableEarnings

Person Mapper

insert
update
delete

ORM vs SQL

- Most RDs support SQL to build data interfaces and applications.
- Need a lot of work, but it is more flexible and detailed than an ORM abstraction.

Native Querying with SQL

- Developer highly responsible for safety and security

SQL Query Builders

- Add a layer of abstraction over the raw SQL without masking all of the underlying details
- Still developer needs to understand the database structure

Advantages of ORM

- Productivity
  - Eliminate repetitive code
  - Fast development of application
- Maintainability
  - Few lines of code
- Performance
  - Minimize row reads and joins
- Database **vendor independence**
- Transaction management

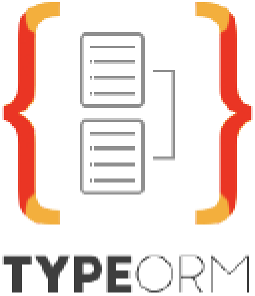
Advantages of ORM

- Less error prone
- Code reuse
- Reduced testing
- Lets business code to **access objects** rather than database tables
- Hides details of SQL queries from OO logic
- No need to deal with database implementation, only **deal with domain objects**

Disadvantages of ORM

- **Performance issues** due to extra-generated code
- Developer **needs to know SQL**

  High-level abstractions don’t always generate the best SQL code
- Sometimes create a **poor/incorrect data mapping**
- A poorly-written ORM layer affects on **schema** and database **migrations**


TYPEORM

Prisma

Sequelize

HIBERNATE

2.

ORM concepts

Entities, Value Objects & Relationships

ORM

Relation/Table	- Class
Record/Row/Tuple	- Object
Attribute/Column	- Member/Field
Relationship	- Composition/ Aggregation
Hierarchy(is-a)	- Inheritance

ORM Entities

- Model **collections of real-world objects** of interest to the app.
- Have **properties/attributes** of **database data types.**
- Participate in **relationships**.
- Have **unique IDs** consisting of **one or more properties**.
- Are **persistent objects** of **persistent classes.**
- Correspond to database **rows** of **matching unique id.**

Value Objects

- Persistent objects can be **entities or value objects**.
- Value objects can represent E/R **composite attributes and multivalued attributes**.

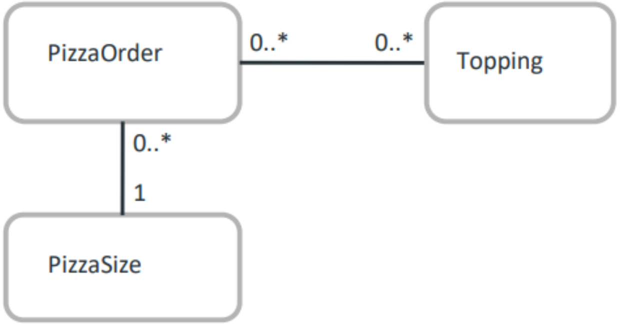
  e.g.:-
  - one address consisting of several address attributes for a customer
  - Programmers want an object for the whole address, hanging off the customer object
- Value objects **provide details** about **some entity**, have lifetime tied to their entity, and don’t need own unique id.

Creating Unique IDs

- A **new entity object needs a new ID**, and the database is holding all the old rows, so it is the proper agent to assign it.
- This can’t be done with a standard SQL insert, which needs predetermined values for all columns.
- Every production database has a SQL extension to do this.
  e.g.:-
  - Oracle’s sequences
  - SQL Server’s auto-increment data type
- The ORM system **coordinates with the database to assign the ID**, in effect standardizing an extension of SQL.

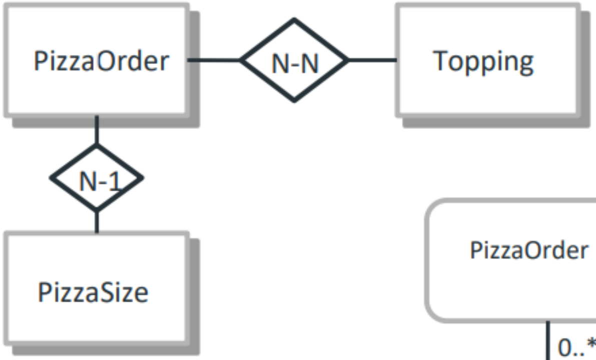
Entity Model

- Uses UML-like diagrams to express **object models** that can be handled by this ORM methodology.
- Currently handles only **binary relationships between entities**, and expects foreign keys for them in database schema.
- Supports **updates and transactions.**

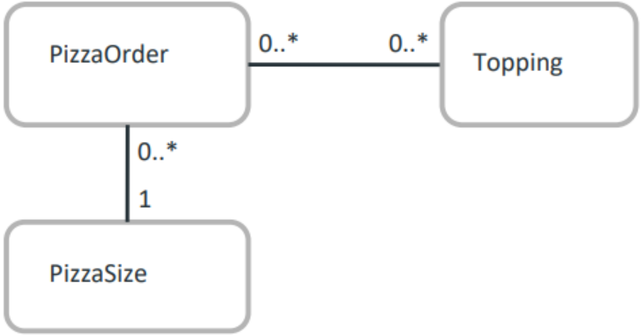


Classic Relationships

“A PizzaOrder has a PizzaSize and a set of Toppings”



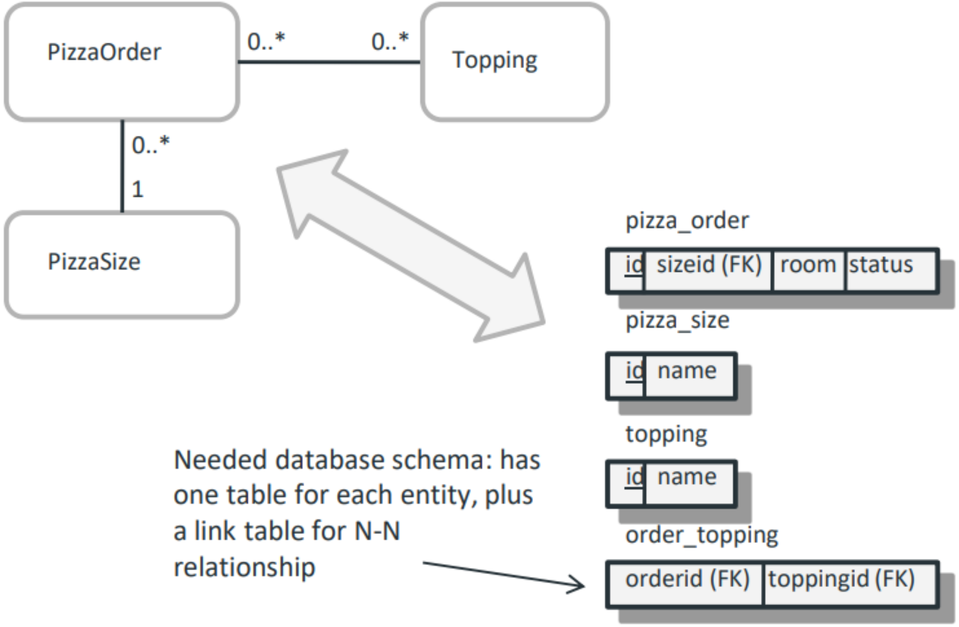
E-R diagram



UML class diagram or entity model: no big diamonds, type of relationship is inferred from cardinality markings

Classic Relationships

Schema mapping, entities to tables and vice versa



pizza_order

id	sizeid (FK)	room	status

pizza_size

id	name

topping

id	name

order_topping

orderid (FK)	toppingid (FK)

Needed database schema: has one table for each entity, plus a link table for N-N relationship

Inheritance

E.g.:- **generalize Topping to PizzaOption**, to allow other options in the future.

  - Topping ISA PizzaOption
  - Shape ISA PizzaOption, …

Then a PizzaOrder can have a collection of PizzaOptions.

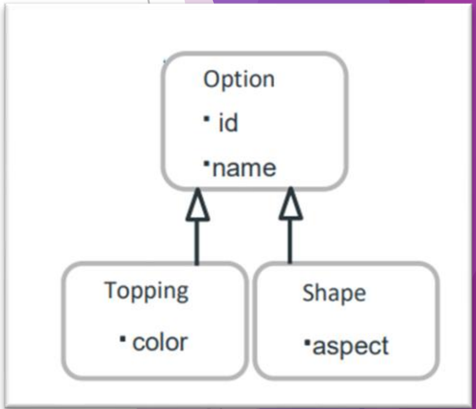
We can process the PizzaOptions **generically**, but when necessary, be sensitive to their **subtype**: Topping or Shape

- Inheritance is supported directly in Java, C#, etc., ISA "relationship"
- **Inheritance is not native to RDBs**, but part of EER, extended entity-relationship modeling, and long-known schema-mapping problem.

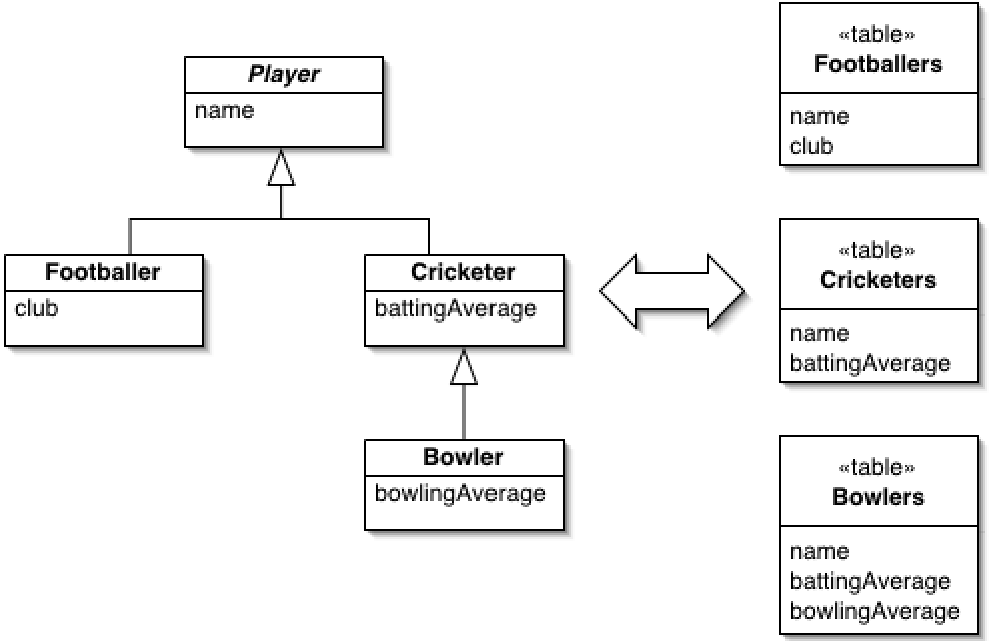
Inheritance Hierarchies

- TypeORM can handle **inheritance hierarchies** and **polymorphic associations** to them.
- TypeORM provides **concrete-table**, **single-table**, and **embeddeds** per hierarchy solutions.

  - Concrete-table: create a base class (abstract) for common properties
  - Single-table: multiple classes with their own properties, but in the database they are stored in the same table
  - Embeddeds: composition over inheritance by using embedded columns



Inheritance Mapping (concrete table)



Player
name

Footballer
club

Cricketer
battingAverage

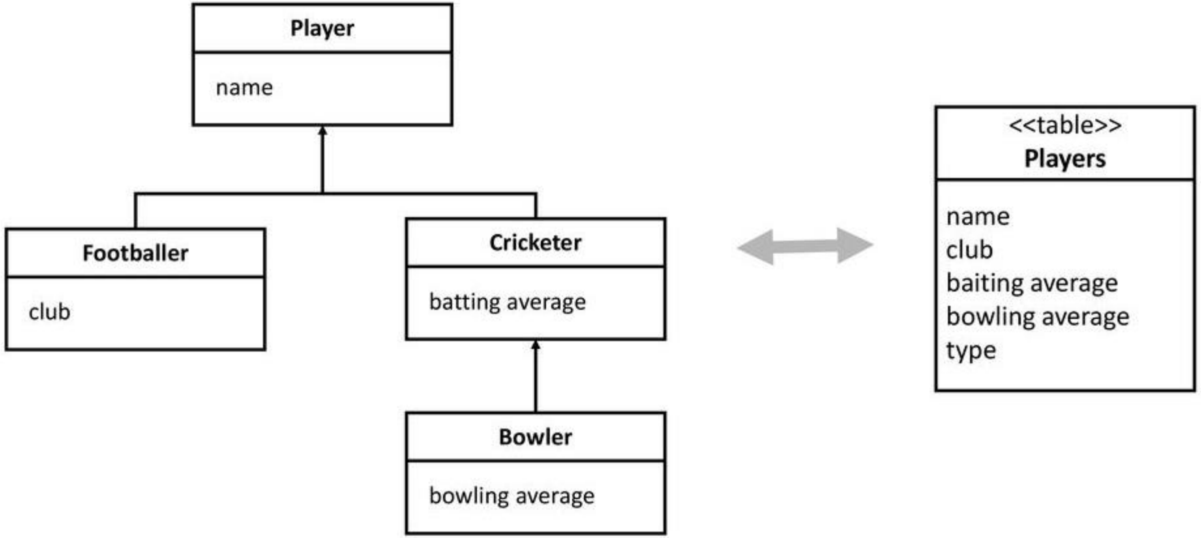
Bowler
bowlingAverage

«table»
Footballers
name
club

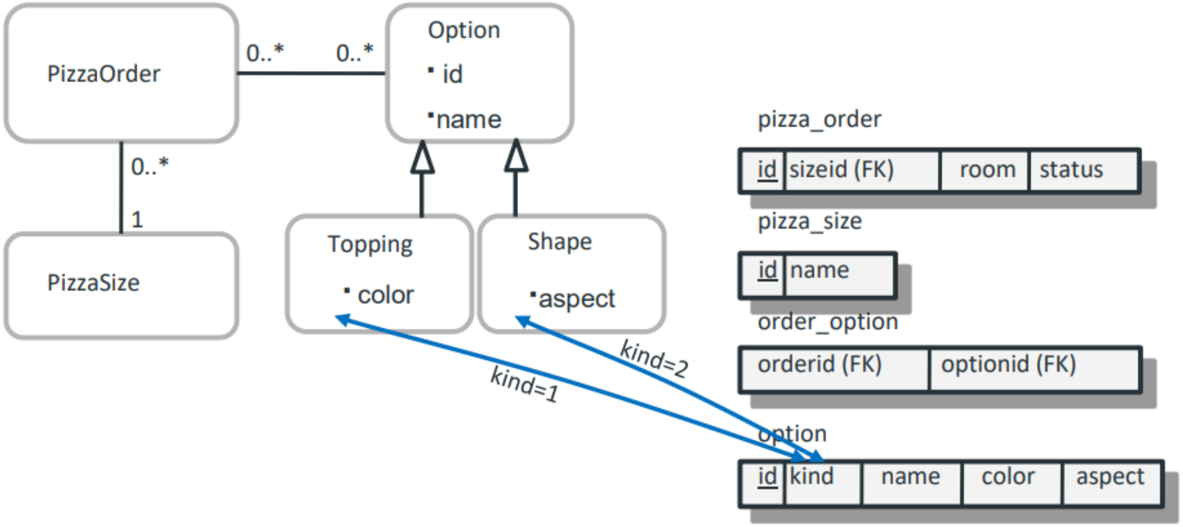
«table»
Cricketers
name
battingAverage

«table»
Bowlers
name
battingAverage
bowlingAverage

Inheritance Mapping (single table)



Inheritance Mapping (single table)



pizza_order

id	sizeid (FK)	room	status

pizza_size

id	name

order_option

orderid (FK)	optionid (FK)

option

id	kind	name	color	aspect

Discriminator column to specify subtype (not seen in object properties)

Inheritance using a single table

- The **discriminator column** (here “kind”) is handled by the O/R layer and does not show in the object properties.
- The hierarchy can have **multiple levels**.
- Single-table approach is usually the **best-performing** way.
- But we have to give up non-null DB constraints for subtype-specific properties.

3.

TypeORM

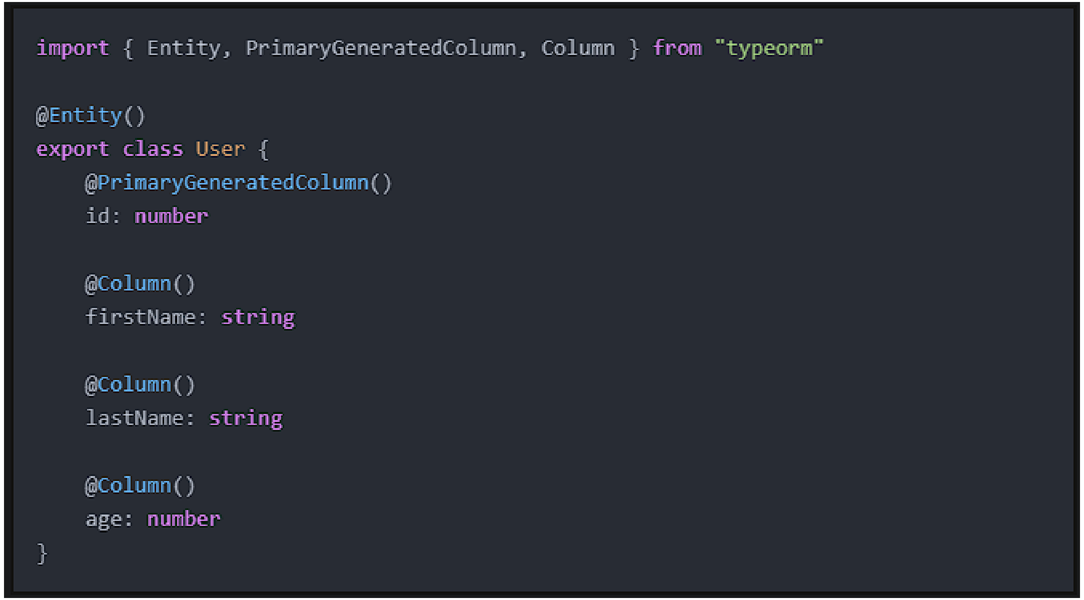
Fundamentals

Features

- Supports both DataMapper and ActiveRecord (your choice).
- Entities and columns.
- Entity manager.
- Repositories and custom repositories.
- Clean object-relational model.
- Associations (relations).
- Eager and lazy relations.
- Uni-directional, bi-directional, and self-referenced relations.
- Supports multiple inheritance patterns.
- Cascades, Transactions, …….

Simple Example

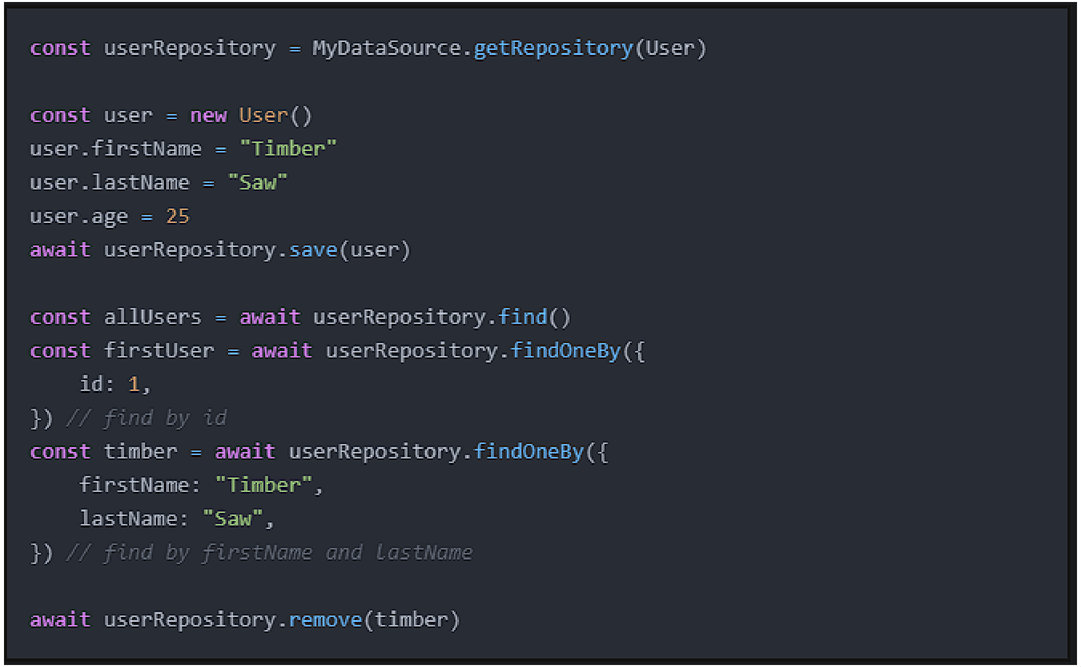
Model



```
import { Entity, PrimaryGeneratedColumn, Column } from "typeorm"

@Entity()
export class User {
    @PrimaryGeneratedColumn()
    id: number

    @Column()
    firstName: string

    @Column()
    lastName: string

    @Column()
    age: number
}
```

Simple Example

Domain logic with data mapper



```
const userRepository = MyDataSource.getRepository(User)

const user = new User()
user.firstName = "Timber"
user.lastName = "Saw"
user.age = 25
await userRepository.save(user)

const allUsers = await userRepository.find()
const firstUser = await userRepository.findOneBy({
    id: 1,
}) // find by id
const timber = await userRepository.findOneBy({
    firstName: "Timber",
    lastName: "Saw",
}) // find by firstName and lastName

await userRepository.remove(timber)
```

Activity 01

- Set up a development environment with the following components
    - TypeORM with TypeScript
    https://typeorm.io/#installation
    - Local MySQL/MariaDB
    - Local Mongo DB

- Complete the step-by-step guide on the official TypeORM website. (Try changing DB instances as well)

  https://typeorm.io/#step-by-step-guide

Activity 02

- Complete the entity-inheritance guide on the official TypeORM website. (Try changing DB instances as well)

  https://typeorm.io/entity-inheritance

Thank You