SCS 2209 Database II

Isuru Nanayakkara
University of Colombo School of Computing

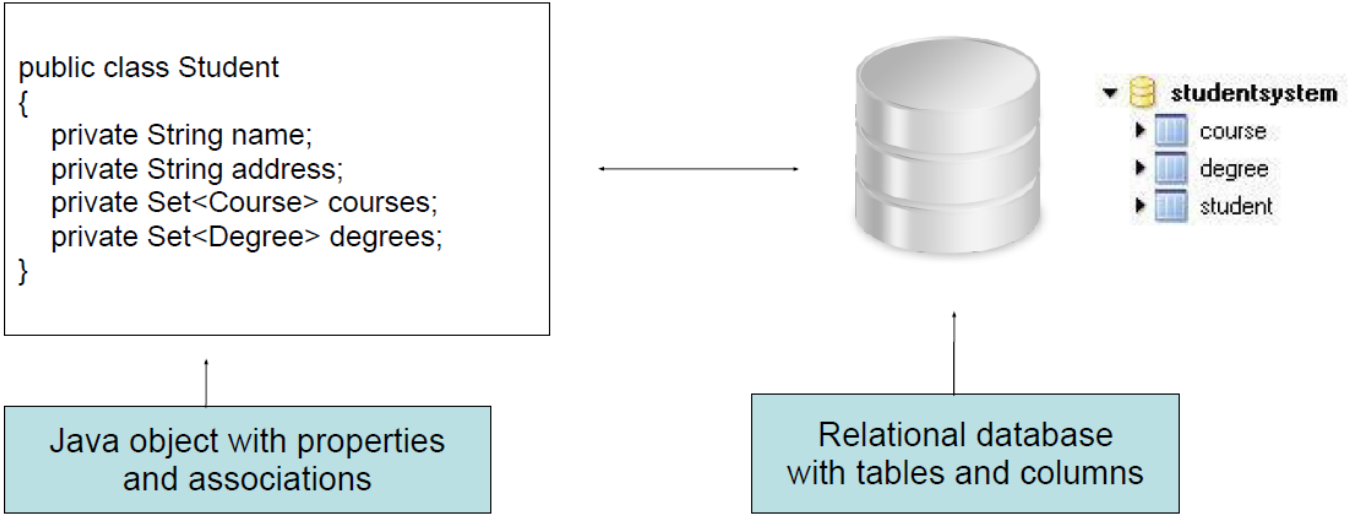
2. ORM & introduction to HQL

Topic Overview

- Problem - Object model and RDB mismatch
- Need for ORM
- What is ORM
- Advantages of ORM
- ORM entities and value objects
- Classic relationships
- Brief introduction to Hibernate

Problem

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



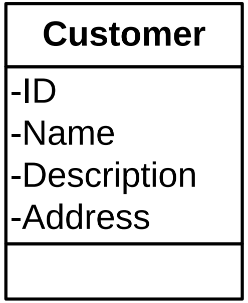
```
public class Student
{
   private String name;
   private String address;
   private Set<Course> courses;
   private Set<Degree> degrees;
}
```

Java object with properties and associations

Relational database with tables and columns

Usually we have

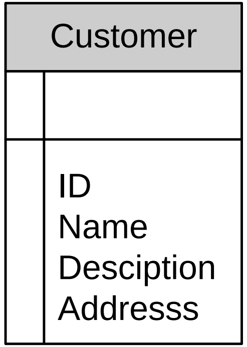
A business Object



Customer

-ID
-Name
-Description
-Address

A database Table



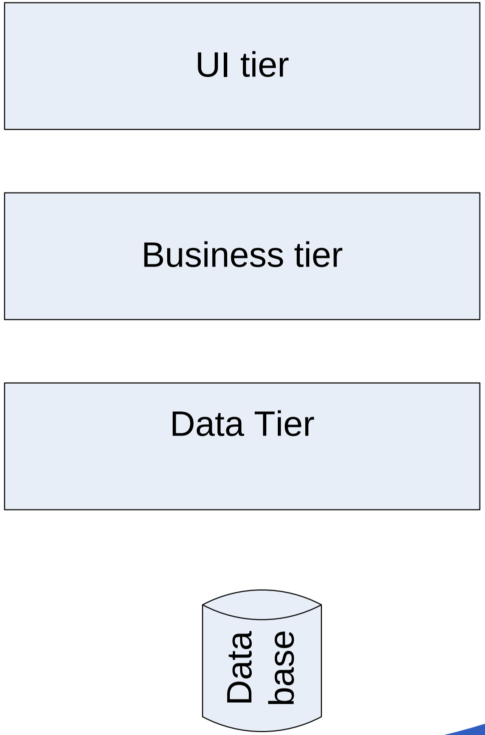
Customer

ID
Name
Desciption
Addresss

Traditional way to handle persistence

Using a N-Tier design

- UI Tier
- Business Tier
- Data Tier
- Database



UI tier

Business tier

Data Tier

This approach requires

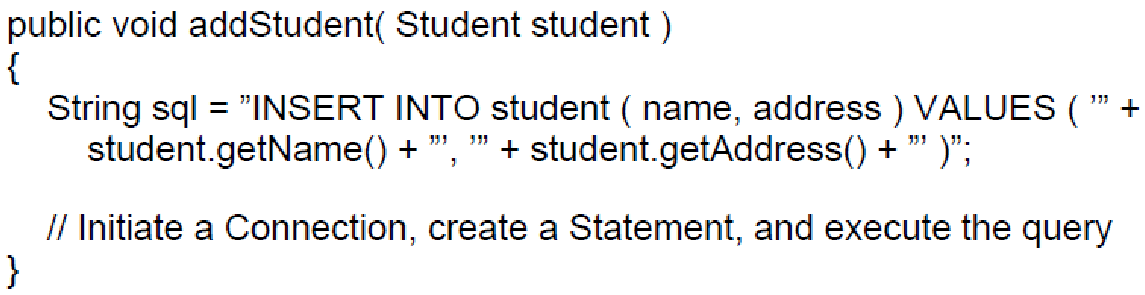
Long and tedious work to:

- **Build SQL statements** for Insert, update, Delete, select.
- Send the **Objects' properties** to the **data layer as parameters.**
- Account for **different types** of **databases/data fields** (ex: ' ' for strings and dates, format dates, handling Null values,..
- **Handle IDs, Keys**,..

Need for ORM

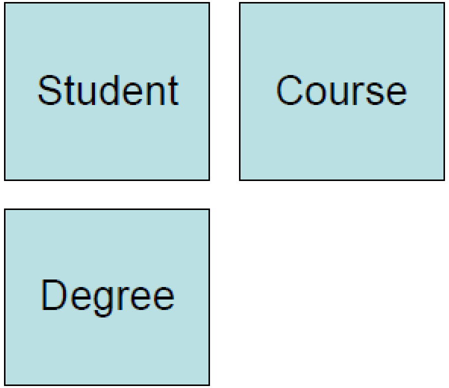
Write **SQL conversion** methods by hand using JDBC

- 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
}
```



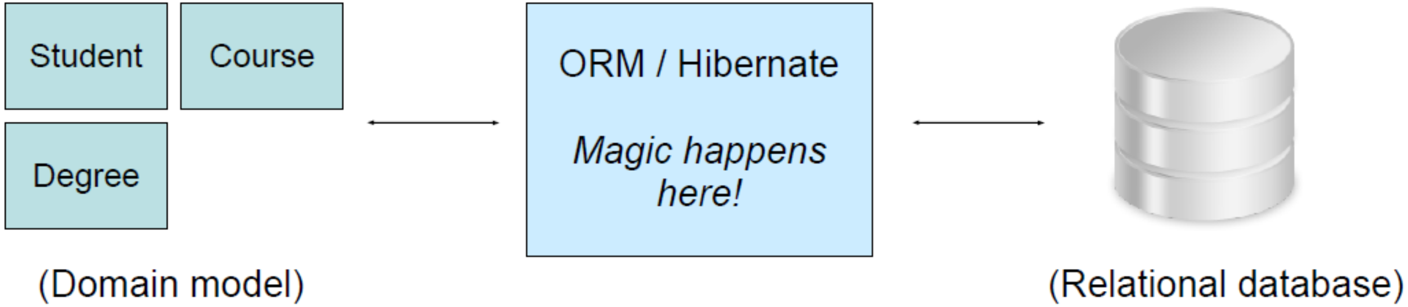
What is ORM?

What is ORM (Object Relational Mapping) ?

- ORM is a programming technique for **converting data between incompatible systems.**
- From **relational databases** (e.g. Oracle, MySQL) to **object oriented** (OO) programming languages (e.g. Java) and back.
- ORM can be viewed as the **automated** and **transparent persistence** of **objects** in a Java application to the database tables in an RDBMS using metadata that describes the **mapping between the objects and the database tables** (one or more).
- It **eliminates** the need to **create a data layer tier** (data layer is implicit).
- In short: ORM saves you from writing boring and **error prone code** thus **saving time** and getting **better quality**.

The preferred solution

- Use a **Object-Relational Mapping** System (e.g. Hibernate).
- Provides a simple **API** for **storing and retrieving** Java **objects** directly to and from the **database.**
- Non-intrusive: No need to follow specific rules or design patterns.
- Transparent: Your **object model is unaware.**



Main 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
- Less error prone
- Lets business code to **access objects** rather than database tables
- Hides details of SQL queries from OO logic
- ‘JDBC’ under the hood
- No need to deal with database implementation, only **deal with domain objects**

ORM

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

ORM Entities

- Like E/R entities, ORM entities model **collections of real-world objects** of interest to the app.
- Entities have **properties/attributes** of **database data types.**
- Entities **participate in relationships.**
- Entities have **unique ids** consisting of **one or more properties.**
- Entity instances (AKA entities) are **persistent objects** of **persistent classes.**
- Entity instances correspond to database **rows** of **matching unique id.**

Value Objects

- In fact, persistent objects can be **entities or value objects.**
- Value objects can represent E/R **composite attributes and multi-valued attributes**

  Example:

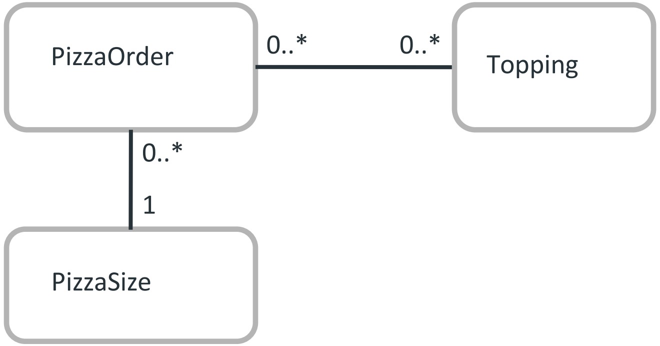
  - one address consisting of several address attributes for a customer.
  - Programmers want an object for the whole address, hanging off the customer object
- Value objects **provide details** about **some entity**, have lifetime tied to their entity, don't need own unique id.

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.
- Note this can't be done with standard SQL insert, which needs predetermined values for all columns.
- Every production database has a SQL extension to do this
  - 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.

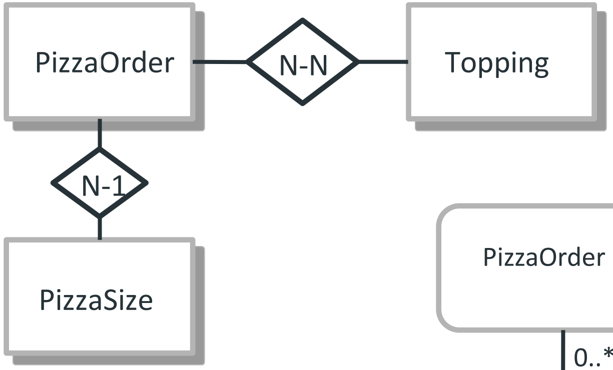
Entity Model

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

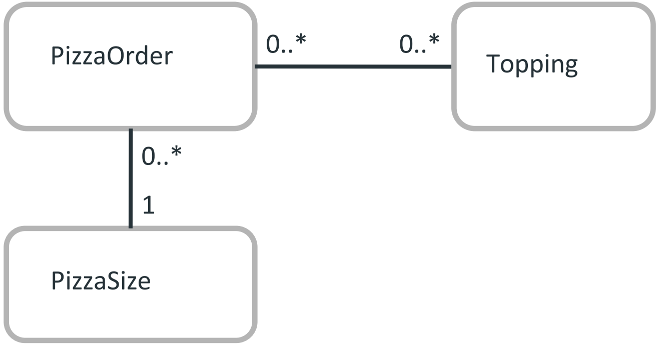


Classic Relationships

A PizzaOrder has a PizzaSize and a set of Toppings



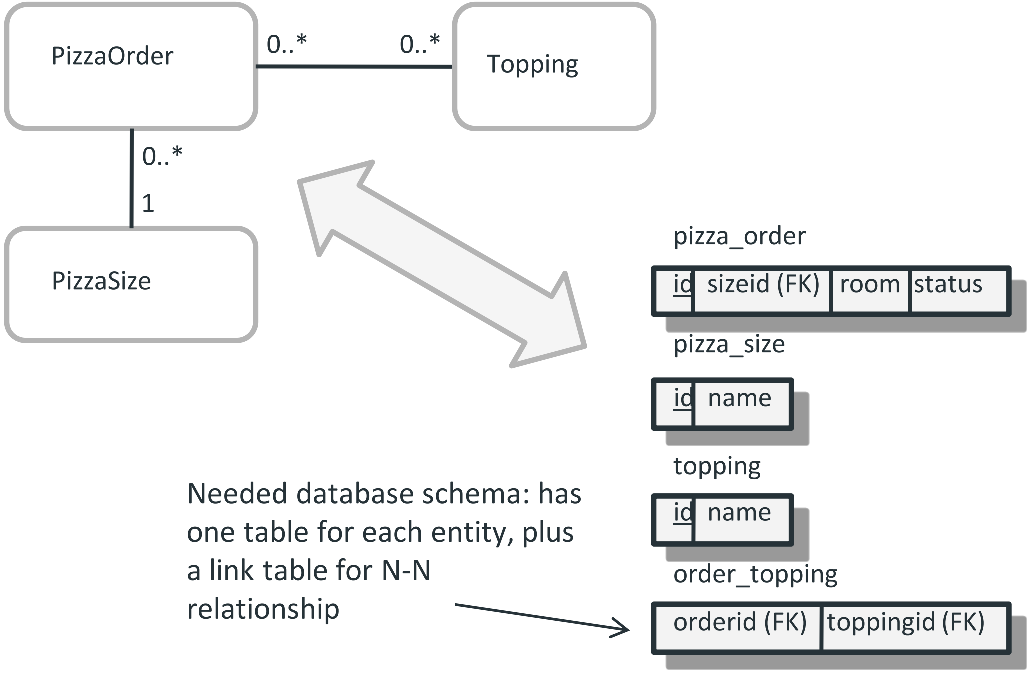
E-R diagram



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)

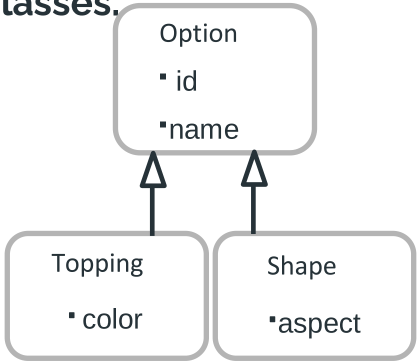
Needed database schema: has one table for each entity, plus a link table for N-N relationship

Inheritance

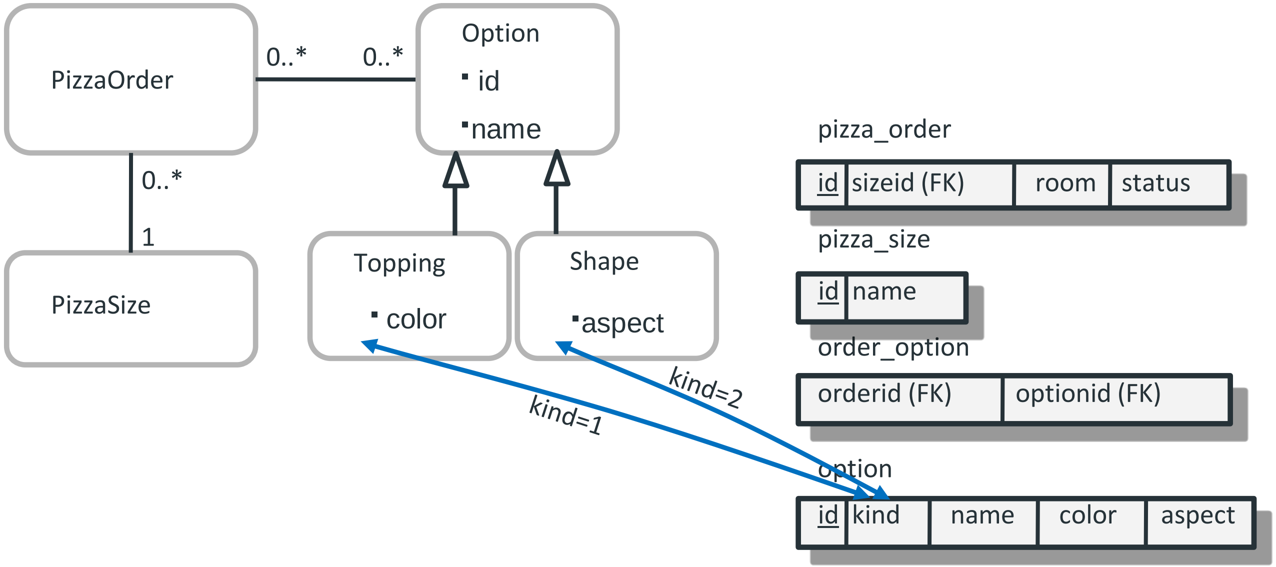
- Example: **generalize Topping to PizzaOption,** to allow **other options** in the future:
  - Topping ISA PizzaOption
  - Shape ISA PizzaOption, …
- Then a PizzaOrder can have a colle**ction of PizzaOptions**
  - We can process the **PizzaOptions generically**, but when necessary, be sensitive to their **subtype**: Topping or Shape
  - It is important to have "polymorphic associations", such as PizzaOrder to PizzaOption, that deliver the **right subtype object** when followed.
- Inheritance is supported directly in Java, C#, etc., ISA "relationship"
- **Inheritance is not native to RDBs,** but part of EER, extended entity-relationship modeling, long-known schema-mapping problem.

Inheritance Hierarchies

- Hibernate can handle **inheritance hierarchies** and **polymorphic associations** to them.
- Hibernate provide **single-table and multiple-tables** per hierarchy solutions.
  - Single-table: **columns for all subtypes**, null values if not appropriate to row's subtype.
  - Multiple-table: **table for common (superclass) properties**, table for each subclass for its specific properties, foreign key to top table.
  - Also hybrid: **common table** plus **separate tables for some subclasses.**



Inheritance Mapping (single table)



PizzaOrder

0..* 0..*

Option
- id
- name

0..*

1

PizzaSize

Topping
- color

Shape
- aspect

kind=1

kind=2

pizza_order

id	sizeid (FK)	room	status

pizza_size

id	name

order_option

orderid (FK)	optionid (FK)

option

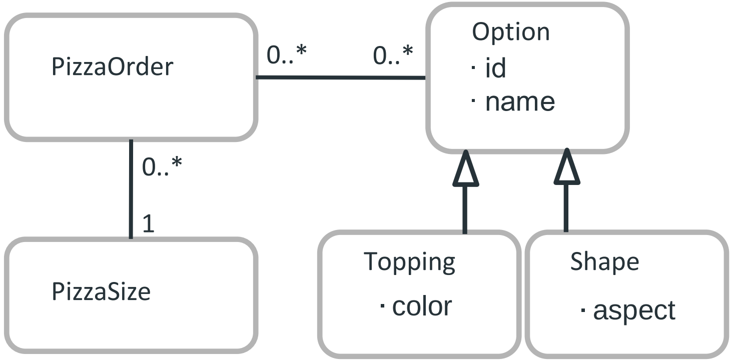
id	kind	name	color	aspect

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.

Inheritance Mapping (3 tables)



pizza_order

id	sizeid (FK)	room	status

pizza_size

id	name

order_option

orderid (FK)	optionid (FK)

option

id	name

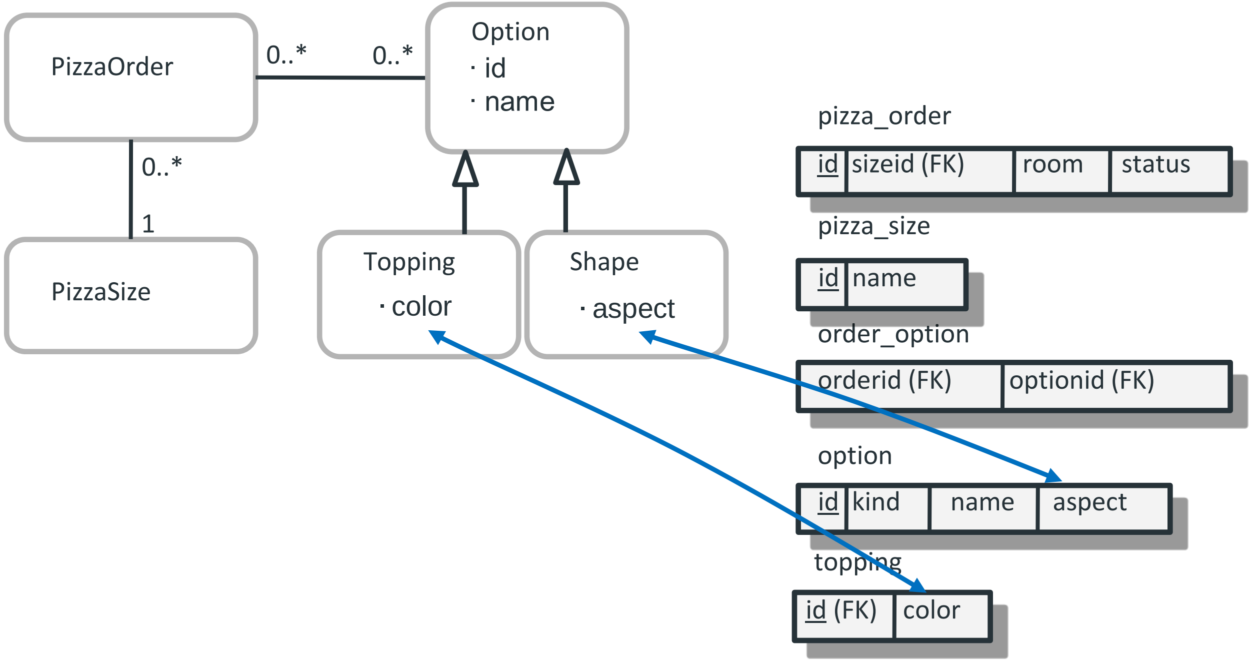
topping

id (FK)	color

shape

id(FK)	aspect

Inheritance Mapping (hybrid)



pizza_order

id	sizeid (FK)	room	status

pizza_size

id	name

order_option

orderid (FK)	optionid (FK)

option

id	kind	name	aspect

topping

id (FK)	color

Object and Object-relational table

Relational table

```
    CREATE TABLE people (
    name VARCHAR (30),
    NIC Varchar (10) primary key,
    phone VARCHAR (20) );
```

Object-relational table

```
    CREATE TYPE person AS OBJECT (
    NIC VARCHAR(10),
    name VARCHAR(30),
    phone VARCHAR(20) );
```

```
CREATE TABLE person_table OF person(
        NIC primary key);
```

Object-relational table

```
CREATE TABLE person_table OF person;
```

You can view this table in two ways:

- As a **single-column table**, in which **each row is a person object,** allowing you to perform **object-oriented operations**.
- As a **multi-column table**, in which **each attribute of the object type person** such as idno, first_name, last_name, and so on, **occupies a column**, allowing you to perform **relational operations**.

```
INSERT INTO person_table VALUES (person (101, 'John', 'Smith', 'jsmith@example.com',
'1-650-555-0135') );
```

Object-relational table

Method 1

- INSERT INTO person_table VALUES (“Sheela”, “123141”);

Method 2

- INSERT INTO person_table VALUES (
  person(“Sheela”, “123141”));

Activity 01

Consider the following schema:

employee (eno, ename, hireDate, salary)
project (projID, projName, budget)
emp_Proj (eno, projID, assignedDate)

1. Map the above schema to the tables using Object Relational Mapping.

2. Using the queries insert values to the created tables.

Activity 01

```
CREATE TYPE employee AS OBJECT  (
            eno VARCHAR(20),
            ename VARCHAR(30),
            hireDate Date,
            salary Float );

CREATE TABLE emp_table OF employee (
            eno primary key);

CREATE TYPE project AS OBJECT  (
             projID VARCHAR(20),
             projName VARCHAR(30),
             budget Float );

CREATE TABLE proj_table OF project (
              projID primary key);
```

```
CREATE TYPE emp_Proj AS OBJECT  (
            eno VARCHAR(20),
            projID VARCHAR(20),
            assignedDate Date);

CREATE TABLE emp_Proj_table OF emp_Proj (
             PRIMARY KEY (eno, projID),
             FOREIGN KEY (eno) REFERNCES emp_table,
             FOREIGN KEY (projID) REFERNCES
proj_table);
```

Activity 01

```
INSERT INTO emp_table VALUES (‘e001’, ‘A.B. Dias’,
‘05.06.1991’, 50000.00);
```

Or

```
INSERT INTO emp_table VALUES (employee (‘e001’, ‘A.B.
Dias’, ‘05.06.1991’, 50000.00));
```

```
INSERT INTO proj_table VALUES (‘P001’, ‘Highway’,
5000000.00);
```

Or

```
INSERT INTO proj_table VALUES (project (‘P001’,
‘Highway’, 5000000.00));
```

```
INSERT INTO empProj_table VALUES (‘e001’, ‘P001’,
‘05.06.2016’);
```

Or

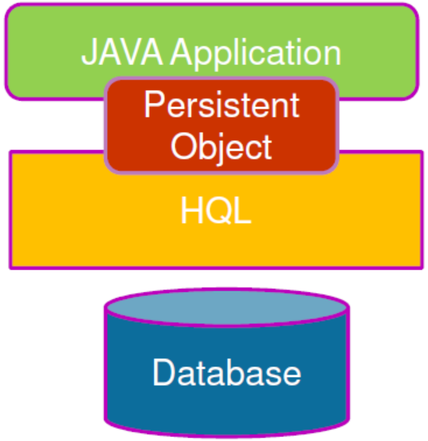
```
INSERT INTO empProj_table VALUES (emp_Proj (‘e001’,
‘P001’, ‘05.06.2016’));
```

HQL

- Hibernate Query Language.
- It is a **tool** used in object relational mapping for **Java environments.**
- Hibernate **supports many different relational databases.**
- Hibernate is an open source package.
- Uses objects and their properties.
- Keywords are not case sensitive but table, column names are case sensitive.

HQL

- Hibernate supports almost all the major RDBMS.
- Following is list of few of the database engines supported by Hibernate.
  - HSQL Database Engine
  - DB2
  - MySQL
  - PostgreSQL
  - FrontBase
  - Oracle
  - Microsoft SQL sever
  - Sybase SQL Server



High level view of the Hibernate Application Architecture

HQL pros and cons

Advantages	Disadvantages
Support Inheritance, associations, polymorphism	Generate many SQL statements in run time
Generate primary keys automatically	Same code need to be written in several files in the same application
Even the database changes HQL is independent	
If we try to insert data to non existing table, HQL will create a table and insert values	

Advantages of HQL

- Hibernate takes care of **mapping Java classes to database tables** using **XML files** and without writing any line of code.
- Provides simple APIs for **storing and retrieving Java objects** directly to and from the database.
- If there is change in Database or in any table then the **only need to change XML file properties.**
- Abstract away the unfamiliar SQL types and provide us to work around familiar Java Objects.
- Hibernate does not require an application server to operate.
- **Manipulates Complex associations** of objects of your database.
- Minimize database access with **smart fetching strategies.**
- Provides Simple querying of data.