Persistence & The Object Relational Model

GÖTEBORGS UNIVERSITET

DAT076/DIT126 Web Applications

Adam Waldenberg

Meet your student representatives

- TIDAL Albin Becevic <albbec@student.chalmers.se>
- TKDAT John Blåberg Kristoffersson <krjohn@student.chalmers.se>
- TIDAL Johan Ericsson <j_ericsson_85@hotmail.com>
- MPALG Madeleine Lexén <lexen@student.chalmers.se>
- TKITE Paulina Palmberg <paupal@student.chalmers.se>

The role of a student representative?

student.portal.chalmers.se/en/chalmersstudies/courseinformation/courseevaluation/Pages/Being-a-Student-representative.aspx

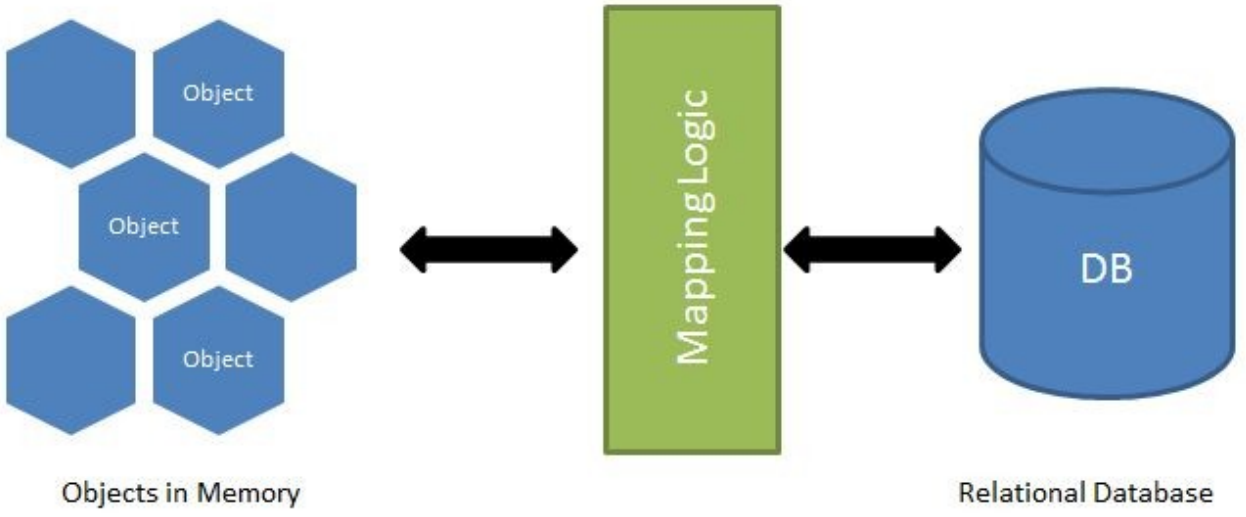
Object Relational Model

A combination of an object oriented database model and a relational database model

- Typically supports objects, classes and inheritance
- Can define data types and tabular structures like a relational data model
- Available for most languages to allow you to bridge the gap between the language and relational databases
- One of the first, or even the first ORM was developed for SmallTalk in the early 1990’s (one of the earliest popular object oriented languages)

Object Relational Mapping

Technique for converting data between incompatible type systems using object-oriented programming languages



Object Relational Mapping - Java

- Java is an object-oriented language
- Objects and classes arranged in a mixed hierarchy (graph) of scalar types and sub-classed relationships to other objects
- Objects contain methods for fetching various states
- We need to find an easy way to map this to a relational database
- Java supports annotations, so this is a perfect way to describe settings and definitions for the mappings

Object Relational Mapping - SQL

- SQL is a relational database language
- Rows (records), all with the same fields (columns) populate a specific table
- Links and dependencies to other tables via foreign keys
- Contain only scalar types

- An ORM framework will convert objects into groups of simpler (scalar) values upon storage and convert those back to objects upon retrieval

Object Relational Mapping – The Challenge

Two very different ways of modeling data

- This incompatibility problem and the difficulties that arise when trying to relate them to each other is called the ***object-relational impedance mismatch***
- How would you actually go about achieving this and implementing it?
- Different languages have different preconditions and possibilities

Java Persistence API

Java EE specifies a standard interface for performing ORM - *The Java Persistence API (JPA)*

- There are several implementations of this API:

- **EclipseLink** *(www.eclipse.org/eclipselink)*, reference implementation
- **Hibernate** *(hibernate.org), the most commonly used implementation at one point*
- **TopLink** *(www.oracle.com/middleware/technologies/top-link.html)*
- **ObjectDB** *(www.objectdb.com)*

- We call these implementations *persistence providers*

JPA Entities

A POJO class with an @*Entity* annotation defines a JPA entity

- An **entity class** is a class representing a table in the database. Each instance (**entity**) represents a record (row) in the table
- An ***entity class*** also holds meta-data describing how it's properties map to the database
- Entities ***persist*** data. If we create, update or delete an entity, the database is modified
- The class that handles this is called an ***entity manager***, given by the ***persistence provider*** and @***PersistenceContext*** or @***PersistenceUnit***

Persistence Unit

The persistence unit defines the name of the persistence unit and it's connection to the underlying data source

- The data source is what defines the connection to the underlying database in the application server

```
<persistence>
    <persistence-unit name="webshop" transaction-type="JTA">
        <jta-data-source>jdbc/webshopds</jta-data-source>
    </persistence-unit>
</persistence>
```

- The *transaction-type* attribute can be defined as either *JTA* (Java Transaction API) or *RESOURCE_LOCAL*

Persistence Unit continued

- The `resources/META-INF/persistence.xml` file consists of a persistence element and a number of child elements
- When using `RESOURCE_LOCAL`, the connection itself can be defined directly inside the `persistence.xml` file instead of using a JTA data source
- Can be defined with additional elements to configure the persistence unit
- Oracle offers extensive documentation on all the elements and attributes *(**docs.oracle.com/cd/E16439_01/doc.1013/e13981/cfgdepds005.htm**)*
- No need to worry about the full specification at the moment – you will get the chance to define the persistence unit in the assignment

JPA Entities continued

A number of requirements are enforced on entity classes

- The class must implement `Serializable` and have a default (no-arg) constructor
- The class must not be final and have no final methods
- Must be defined with the ***@Entity*** annotation
- Must have a primary key defined with the ***@Id*** annotation
- Use ***@GeneratedValue*** together with the ***@Id*** annotation to specify automatically generated primary keys
- The ***@Column*** and ***@Table*** annotations can be used to modify the default name for tables and columns in the relational database

JPA Entities continued

Persistent instance variables or Persistent properties - Two different ways to map the class to the database constructor

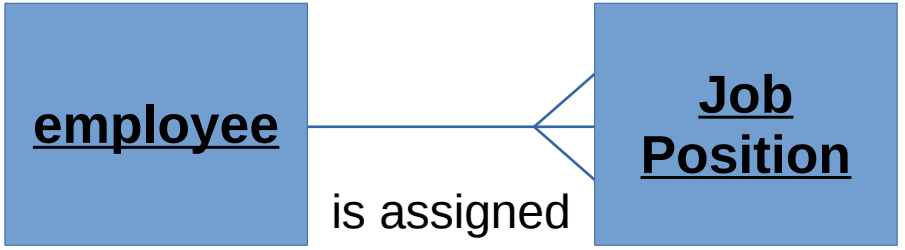
- **Persistent instance variable** — Associate instance variables with columns in the table
- **Persistent properties** — Associate properties (getters/setters) with columns in the table
- Exclusive – the two methods can not be mixed in one entity class
- *Generally we want to use persistent properties for the sake of flexibility*

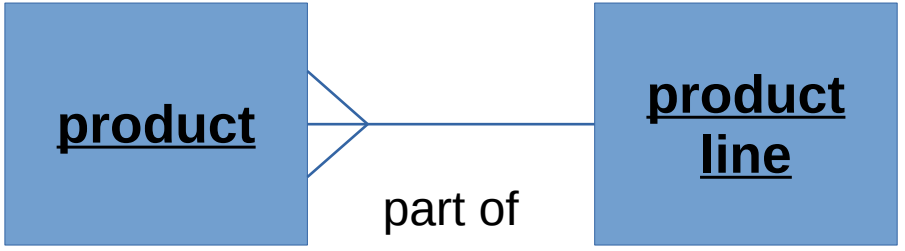
Relationship Mapping

Relationships can be mapped between entity classes to control how the model is connected in the relational database

- ***@OneToOne, @OneToMany, @ManyToOne*** and ***@ManyToMany*** define all the relational annotations we can use on properties
- We use the ***@JoinTable*** and ***@JoinColumn*** annotations plus the ***mappedBy*** attribute of relations to avoid the creation of an extra table in the database when creating the relationship between entities
- Generally we tend to only want that extra relationship table if we create a Many-To-Many relationship.

@OneToOne & @ManyToOne

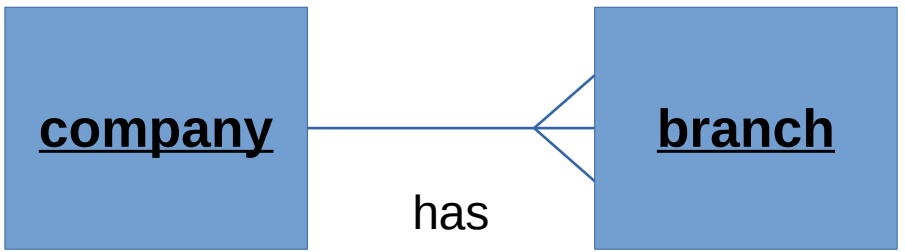




Most of the time there is no need for any attributes to these annotations / relationships

- Classical definition as defined by entity-relationship models
- If we choose ***unidirectional*** or ***bidirectional*** relationships comes down to the database design and query optimization

@OneToMany

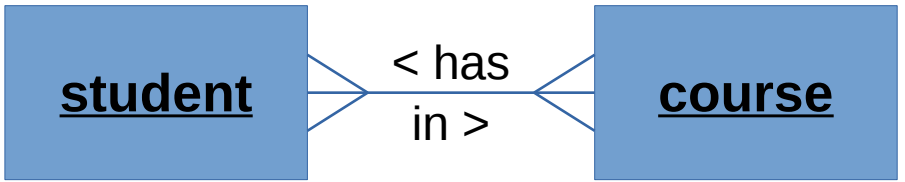


- Type of the instance variable must be a collection
- ***@JoinColumn*** specifies that this entity is the owner of the relationship (*or put differently*: the opposing table has a column with a foreign key back to us)

```
@Entity
public class Company {
    @OneToMany(mappedBy = “company”) private List<Branch> branches;
}

@Entity
public class Branch {
    @JoinColumn(name = “company_id”)
    @ManyToOne private Company company;
}
```

@ManyToMany



- ***@JoinTable*** is used to define the relationship table in the many-to-many relationship

```
@Entity
public class Student {
    @Id private int id;
    @JoinTable(name = “student_course”,
        joinColumns = @JoinColumn(name = “student_id”),
        inverseJoinColumn = @JoinColumn(name = “course_id”))
    @ManyToMany private List<Course> courses;
}

@Entity
public class Course {
    @Id private int id;
    @ManyToMany(mappedBy = “courses”) private List<Student> students;
}
```

Composite keys

```
public class CoursePK {
    private String code;
    private int year;
}

@Entity
@IdClass(CoursePK.class)
public class Course {
    @Id private String code;
    @Id private int year;
}
```

- To create a composite key in JPA, we create a separate class to define it
- *The* ***@IdClass*** annotation specifies that this entity is using the class CoursesPK for it's key definition
- At this point we can specify multiple ***@Id*** annotations in our entity class to define the composite key in the entity

A word of warning...

Keep an eye on your generated tables!

- When modeling your database and defining your entities – always keep track of what the persistence provider actually generates
- If you you make a mistake with ***@JoinTable / @JoinColumn / mappedBy,*** the generated tables and schema can become very inefficient. In some cases it won't even work correctly!
- To inspect the database, NetBeans has a very handy database explorer that you can use – just create a connection to your database from within NetBeans
- During testing, using the property ***<property name="eclipselink.ddl-generation" value="drop-and-create-tables" />*** in your `persistence.xml` to drop and re-create tables on each redeployment

Entity Manager

The class provided by the container to manage a specific entity

- When using a JTA-enabled persistence unit, you can inject it directly into an enterprise java bean

```
@PersistenceContext(unitName = “petshop”)
private EntityManager em;
```

- The entity manager contains methods for counting, finding, inserting, updating and deleting entities, plus methods for creating queries

```
em.persist(pet);        em.remove(pet);          em.createNamedQuery(...);
em.merge(pet);          em.refresh(pet);
```

JPA Query API's

Beyon the simple *find()* method available in the entity manager, JPA offers a number of methods and API's to query the database

- **Native Queries** (discouraged usage) — You can send a native queries. This would send the exact defined statement directly to the RDBMS *(Relational Database Management System)*
- **JPQL Queries** — An SQL derivate language for querying the relational database, bridging the differences
- **Named Queries** — A JPQL query associated to and annotated on an entity *(@NamedQuery).*
- **The Criteria API** — The last addition to the standard

JPQL explained

Kind of looks like SQL, but with subtle differences

```
SELECT c FROM Country c; /* returns all Country entities */
```

```
SELECT c.name FROM Country AS c; /* returns all Country names */
```

- JPQL also has support for parameter placeholders. It also rebuilds the query, and escapes all input parameters, protecting us from statement manipulation and injection attacks

```
SELECT e FROM Employee e WHERE e.id = :id; /* returns specific Employee
with primary key id */
```

```
query.setParameter("id", 42);
```

JPQL grammar

For an almost complete grammar of the JPQL language, visit:

dzone.com/refcardz/getting-started-with-jpa?chapter=7

- Also covers more on the entity manager, queries and configuration of the persistence unit

Named Queries

Named queries are just JPQL statements that have a functional name and are directly connected to a entity class

```
@NamedQuery(name = “Country.findAll”,
            query = “SELECT c FROM Country c”);
```

```
em.createNamedQuery(“Country.findAll”);
```

- Moves more of the model away from the database access classes and collects the query definitions in one place

The Criteria API

The newest addition to the standard

- Allows developers to programmatically define and control queries to the relational database
- Instead of a custom syntax, queries are built with pure Java
- Allows for the creation of type-safe queries
- Has been ***heavily criticized*** as being completely over-engineered , nonsensical and extremely difficult to use
- Has resulted in the creation of several convenience layers on top of the Criteria API

The Criteria API – Making it usable

Many convenience layers have been developed – this is a non-exhaustive list

- Blaze Persistence
  (github.com/Blazebit/blaze-persistence)
- Easy Criteria
  (github.com/sveryovka/easy-criteria)
- QueryDSL
  (www.querydsl.com/)

QueryDSL

QueryDSL and most of the other convenience layers allow us to write queries in a more natural flow

```
/* fetch the first customer with the name “Bob” */
Customer bob = query.select(customer)
  .from(customer)
  .where(customer.firstName.eq("Bob"))
  .fetchOne();
```

- Faster to write code and queries than with Criteria API
- More readable than Criteria API!

The DAO design pattern

Data access objects allow us s to isolate the application/business layer from the persistence layer

- Each entity has an access object
- Access objects use inheritance to implement common methods for all access objects *(such as **getAll()**, **find()**, **delete()**, **merge()** and so on)*