JPA Mappings

JPA Slides #2

Java and ORM

To bridge the OO-relational mismatch we map between classes/objects and tables/rows and more … i.e. ORM

We're using JPA 2.x as our ORM middleware
- Using annotations to define the mappings
- Also possible using XML mapping files, we don't …
- JPA defined as "framework" by Oracle (we call it middleware, gives no structure to application)

Mapping OO-model to Database

In general

- Package → Schema
- Class → Table
- Attribute → Column
- Associations → Relationships

Associations are tricky, more to come …

Note: The classes to map are normally in the core model

- Not all classes in model should be persisted (we have a persistence model, a submodel)

JPA Entity Class

An **entity class** is a Java class typically representing a database table. Specification;

- @Entity and @Id annotation
- Non-private default constructor
- No public attributes
- Serializable
- No final, whatsoever!
- Inheritance ok (mixed non-entity and entity classes is ok)
- Must be listed in a "persistence unit" more later...

@Entity annotation on...

- Abstract class, ok
- Interface or Enum, no!

Default Mapping Rules

If no annotations except @Entity and @Id default mapping rules applies

Basically

- Class mapped to single table. Table will have same name as class but uppercase
- Attributes mapped to column names, uppercase
- JDBC rules for mapping simple Java types to database types
  - int, Integer, ... byte[], Byte[], ...String, Data, Calendar, TimeStamp,any ENUM, any Serializable.
- Collections mapped to extra table
- Relationships creates columns for fk in some of the involved tables (possible extra (sometimes unnecessary) join tables)

Customize Mapping

If not satisfied with default mappings use class, field or method annotations. Annotations for;

- Table
- Columns
- Others

We give a few examples

Many more mapping examples

Customize Class ->Table

Use @Table class annotation

```
// Other name for table (CUST instead of CUSTOMER)
@Entity
@Table(name="CUST", schema="RECORDS")
public class Customer { ... }


// Unique constraint for full row (i.e. no duplicates in rows)
@Entity
@Table(name="ALLOCATION", uniqueConstraints={
   @UniqueConstraint(columnNames={"CONSULT_ID", "PROJECT_ID"})
})
```

Customize Attribute -> Column

```
// Other name and restrict length
@Column(name="Desc", length = 50)
private String description;

// Must use on Date and Calendar
@Temporal(DATE)
protected java.util.Date endDate;


// Transient specifies that an attribute should not be persisted
// (possible a calculated value).
@Entity
public class User {
     …
    @Transient ShoppingCart cart;  //Don't save cart

     …
}
```

Entity Class Identity

Entity class must have a id representing the database table primary key

- For simplicity we use an attribute id of type Long
- Specify id with @Id

```
public myClass {
    @Id
    private Long id;  // Will have pk-column ID in table
}
```

Generate the Primary key

Simplest is to let database generate the id value
- Specify using @GeneratedValue in conjunction with @Id
- If using generated id, never supply any id when creating entity (constructor or other …)

```
// Generate pk's 1, 2, 3, ….
@GeneratedValue(strategy=GenerationType.AUTO)
@Id
private long id;
```

The Id Problem

Entity classes should define equals-method (and hashCode)

- Only significant value used should be the id

If letting database generate id, the object have no id before really written to database

Can't depend on object id before persisted.

- Will cause problems if not observant … can't use some containers (Set)

Collections and Enums

If class has a Collection or Map of primitive types
- Annotate with @ElementCollection, @CollectionTable (possible FetchType.Lazy upcoming...)
- Will create extra table holding collection data
- If non-primitive... more to come...

If class has Enum
- Annotate with @Enumerated(EnumType.STRING)
- Will end up in same table

Embedded Objects

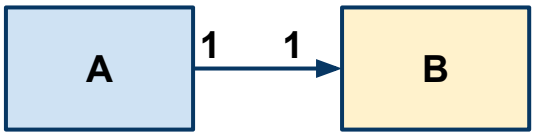
Embedded object depends on some entity class for it's identity (no own identity, i.e. a value object, identifying relationship)

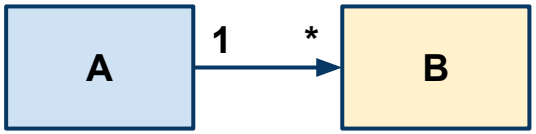
```
@Embeddable
public class Address { ... }

@Entity
public class Employee {
    @Embedded
    private Address address;
    ...
}
```

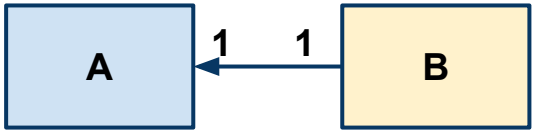
Ends up in same table (Employee)

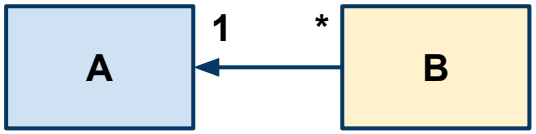
Associations

Classes A and B



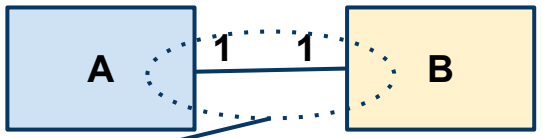


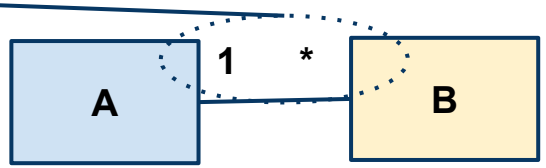


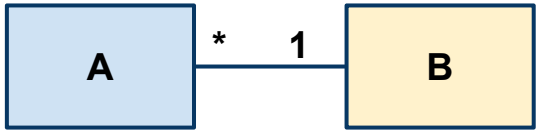


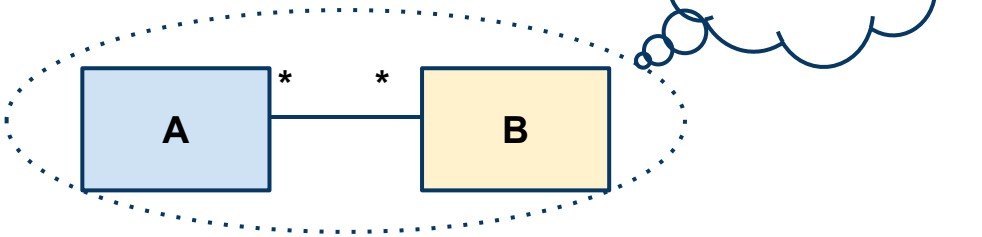
Unidirectional OK

Cardinality









Bidirectional i.e. mutual dependencies. Avoid!

Mapping Associations

Classes (objects) are connected with associations, database tables with relationships

Mapping an association will result in relationships between tables

- Not a perfect match … .associations have direction, relationships not
- Relationships only have 1:N cardinality (the 1:1 cardinality must be forced through UNIQUE constraint on foreign key)

RUNTIME: Associations = object references, relationships = matching row id's (key's). Have to be careful!

Associations: UML vs Database

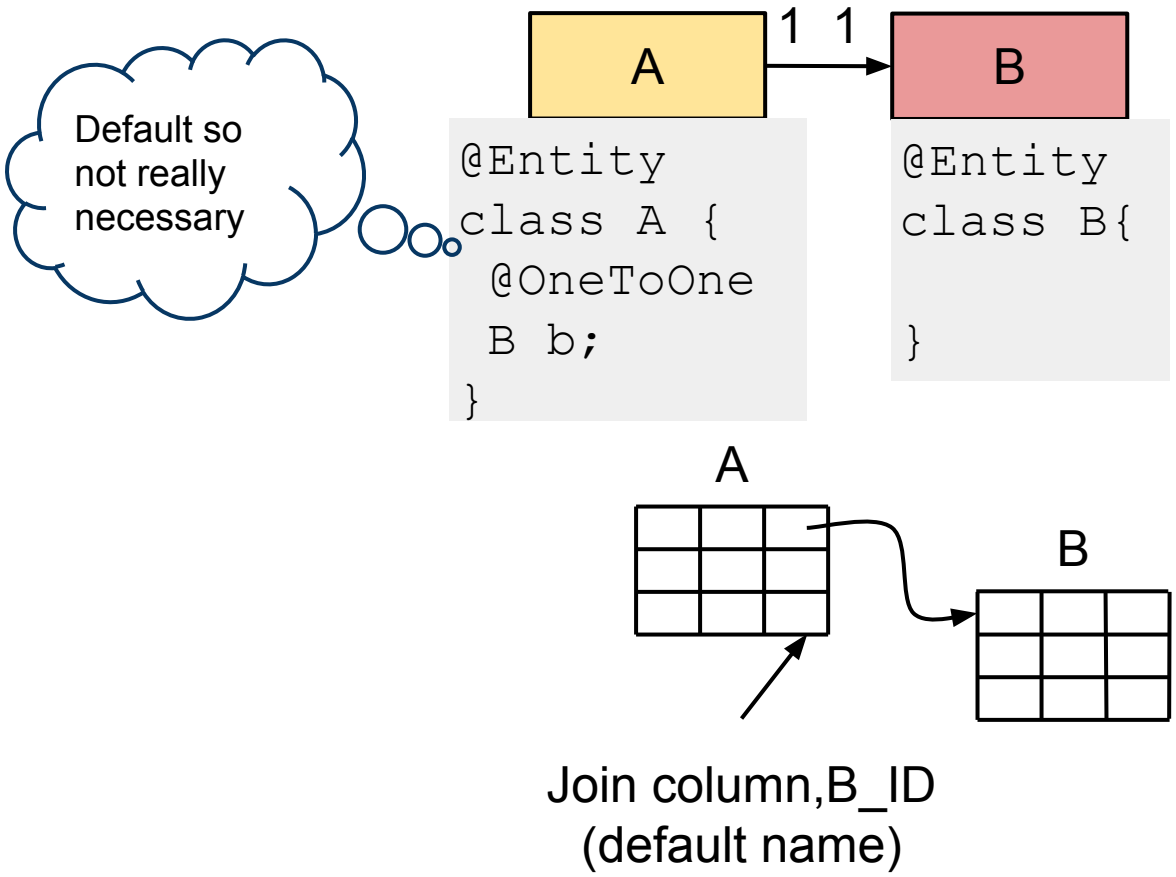
UML associations denotes a references in Java

- UML 1:1 says one object having a reference to another. But the id of the object isn't considered! It's just some objects associated

But when working with databases the id's are what's count

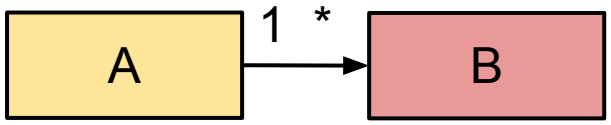
- Database (ER) 1:1 says one table row is related to one unique row from another table, not to any row (like splitting a table vertically).

Unidirectional 1:1 Mapping

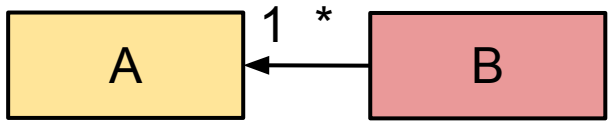


If the exact identity of B is important (database 1:1) need to use @Column(unique=true) for B_ID

Unidirectional 1:* Mapping



OR



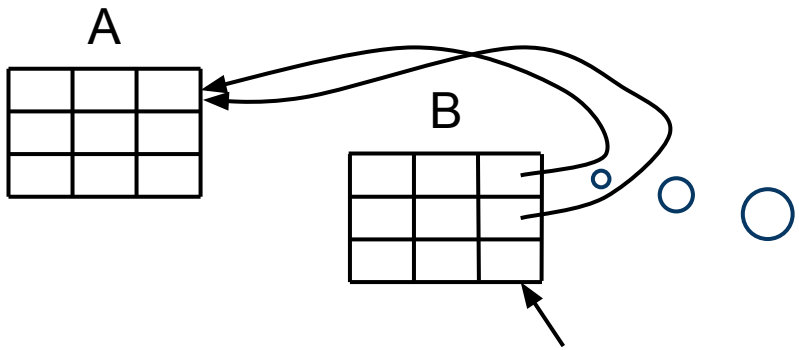

```
@Entity
class A {
 @OneToMany
 @JoinColumn(
name = B_FK)
 List<B> b;
}
```

```
@Entity
class B{

}
```

```
@Entity
class A {

}
```

```
@Entity
class B{
   @ManyToOne
   A a;
}
```

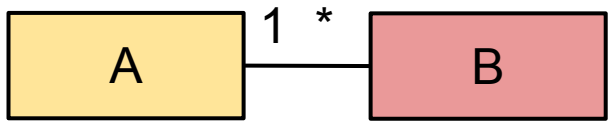
If not using JoinColumn extra table created



Join column, B_FK (will be added)

Will end up the same in database

Bidirectional 1:* Mapping

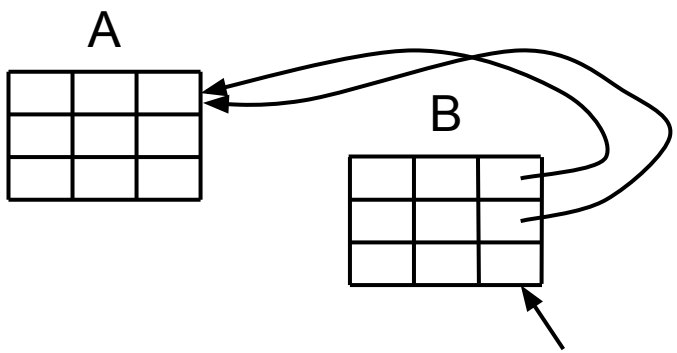



```
@Entity
class A {
@OneToMany(mappedBy
= "a") List<B> b;
}
```

```
@Entity
class B{
@ManyToOne
@JoinColumn(name =
"B_FK") A a ;
```

The owning side (table with foreign keys)

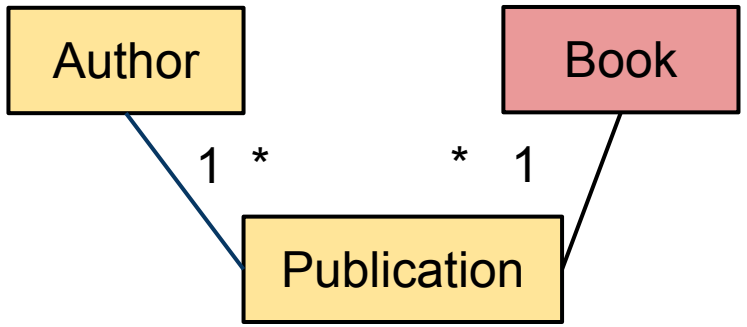
The inverse side



Join column, B_FK (will be added)

Mapping *:*

Many to many transformed to … this!




```
class Author {
 Collection<Book> bs;
}
```

```
class Book{
  Collection<Author> as;
}
```

```
@Entity
class Author {
}
```

```
@Entity
class Book {
}
```

Optional

```
@Entity
@Table(name="PUBLICATION", uniqueConstraints={
   @UniqueConstraint(columnNames={"AUTHOR_ID", "BOOK_ID"})
})
class Publication {
     @ManyToOne
     Author a;
     @ManyToOne
     Book b;
}
```

Will get extra table

Fetching Strategies

When to load associated objects

- EAGER, when owner loaded
- LAZY, when code executed

Default (otherwise annotate)

- @OneToOne, EAGER
- @ManyToOne, EAGER
- @OneToMany, LAZY
- @ManyToMany, LAZY

```
// Example
@OneToMany(fetch=FetchType.EAGER)
List<OrderItems> oi;
```

Summary Association Mapping

SE best practices
- Limit number of associations
- Prefer unidirectional, review use cases to decide direction

If need to navigate in “other” (non existing) direction in code have to search
- Probably best to let database search (i.e. use queries, upcoming)

Mapping Inheritance

Different strategies

- Single table for hierarchy (all super/sub-objects in same table)
- Joined strategy, many tables
- .. more...

```
//Superclass
@MappedSuperclass
public class Person … {
    // Common code
}

//Subclass, everything will end up in table Employee
@Entity
public class Employee extends Person {
}
```

Warnings

If mappings "wrong" you will get (sometimes not easy to understand) exceptions!

- If strange exceptions carefully review all mappings

Persistence Unit (PU) Revisited

As noted we have a config file persistence.xml containing "named" PUs

- All entity classes must be listed in PU, if not exception, "not a known entity type"
- All classes in PU must be collocated in same database
- Possible to specify table generation strategy
  - None, no tables created (should exist)
  - Create, will create when executing program
  - Drop and Create, delete and create when executing program

- Transactional type for EM, always, Java Transaction API, JTA more to come …

Persistence Unit Sample

NetBeans will generate!

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="
http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.
org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="managing_pu" transaction-type="JTA">
    <jta-data-source>jdbc/test</jta-data-source>
    <class>jpa.mgn.core.Author</class>
    <class>jpa.mgn.core.Book</class>
    <class>jpa.mgn.core.Publication</class>
    <class>jpa.mgn.core.Review</class>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>
    <properties>
      <property name="javax.persistence.schema-generation.database.action" value="
create"/>
    </properties>
  </persistence-unit>
</persistence>
```

Validation

As noted: All layers should validate incoming data!

"Bean Validation constraints may be applied to persistent entity classes, embeddable classes, and mapped superclasses."
- Constraints checked immediately after the PrePersist, PreUpdate, and PreRemove lifecycle events, more to come …

The JPA @Column may also impose constraints

Bean Validation vs JPA Constraints

@NotNull is a Bean Validation annotation. It has nothing to do with database constraints itself.

@Column(nullable = false) is the JPA way of declaring a column to not accept null values

When to use?

- Need both to be safe … NotNull for application (control layer), Column for database layer.

JPA and JAXB

Possible to have both @Entity and @XmlRootElement on same class

- Get database data as XML directly … possible for REST applications

Other way round

If you're a skilled database developer, start with database and let NetBeans generate the mapped entity classes but ...

... don't touch generated code! Separate out!