JPA Mappings

JPA Slides #2

JPA and ORM

Object relational mapping

- How to map between the primitive data in the database tuples (rows in database tables) and Java objects?

We're using JPA 2.0 as our ORM "framework" (not what I call framework, **middleware** better)

- Using annotations to define the mappings
- Also possible using XML mapping files, we don't …

Sadly Persistence JPA seems a little brittle , …
this has been the most troublesome part of the course, …
may the force be with you

Mapping OO-models to Database

In general

- Package → Schema (have only one package, the model)
- Class → Table
- Attribute → Column
- Associations → Relationships

Associations are tricky

- Relational model only has relationships and the 1:N cardinality (the 1:1 cardinality must be forced through UNIQUE constraint on foreign key)
- ..more to come...

JPA Entity Class

Class, possible to map to database table(s). A Java class with

- @Entity class annotation and @Id attribute annotation.
- Default constructor
- Serializable
- No final, whatsoever!
- Must be listed in a "persistence unit" (config file) more later...

@Entity annotation on...

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

Entity Class Identity

Entity classes should define equals-method (and hashCode) else possible problems...

- In our case we only use the id attribute in equals, more to come ...
- Also: Same type or mixed type equals?

Default Mapping Rules

If no annotations except @Entity and @Id default mapping rules applies (again: convention over customization)

- 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.
- Relationships creates columns for fk (possible extra/unnecessary join tables)

Upper- Lower-case confusion...

- Depends on database...? Most seems not to be case sensitive! Always check!!!

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 at http://www.objectdb.com/api/java/jpa/annotations/orm

Customize Table

Use @Table class annotation

```
// Other name for table
@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 Columns

@GeneratedValue
- Used with @Id to let database generate primary keys
- If using generated id, never supply any id when creating entity (constructor or other …)
- More to come …

```
// Generate pk's1, 2, 3, ….
@GeneratedValue(strategy=GenerationType.AUTO)
@Id
private Long id;
```

@Column used for name and restrictions

```
// Other name and restrict
@Column(name="DESC", nullable=false, length=512)
public String getDescription() { return description; }
```

Customize Columns, cont

@Temporal must be used for Date and Calendar

```
@Temporal(DATE)
protected java.util.Date endDate;
```

@Transient specifies that an attribute is not persistent (possible a calculated value)

```
// Don't save current user
@Entity
public class Employee {
    @Id int id;
    @Transient User currentUser;
        ...
}
```

The Id Problem

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

Can't depend on object id before. Will cause problems if not observant

- equals method
- .. as a consequence can't use in 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

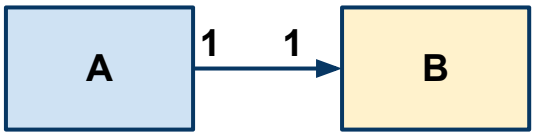
Embedded object dependents on some entity class for it's identity (no own identity, i.e. a value object, identifying relationship)

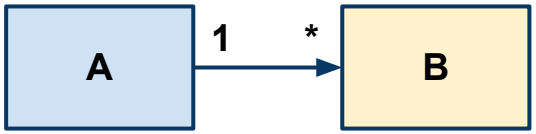
```
@Embeddable
public class Address { ... }

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

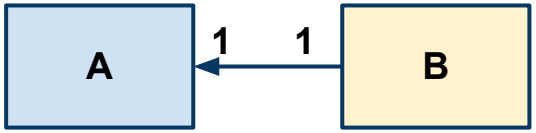
Ends up in same table (Employee)

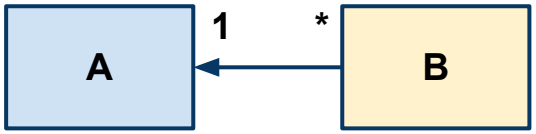
Associations

Classes A and B

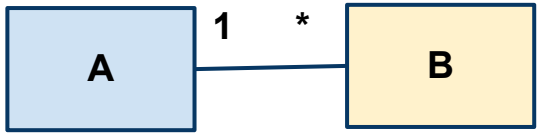


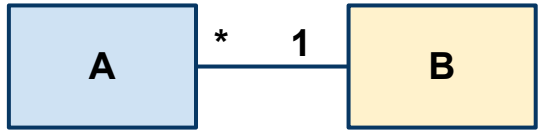


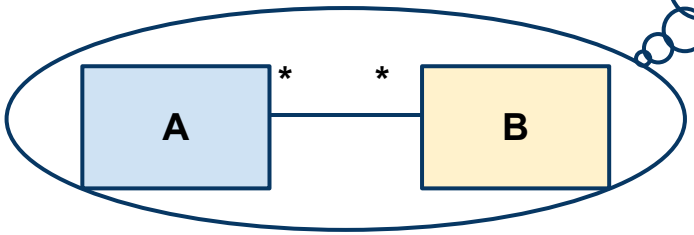


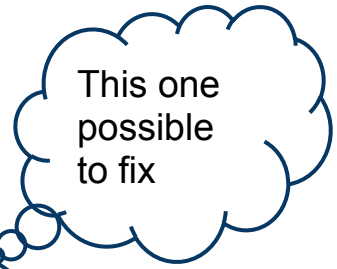


Unidirectional OK









Bidirectional BAD
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

RUNTIME: Associations = object references, relationships = matching row id's (key's)

Associations: UML vs Database

UML associations means a references in Java

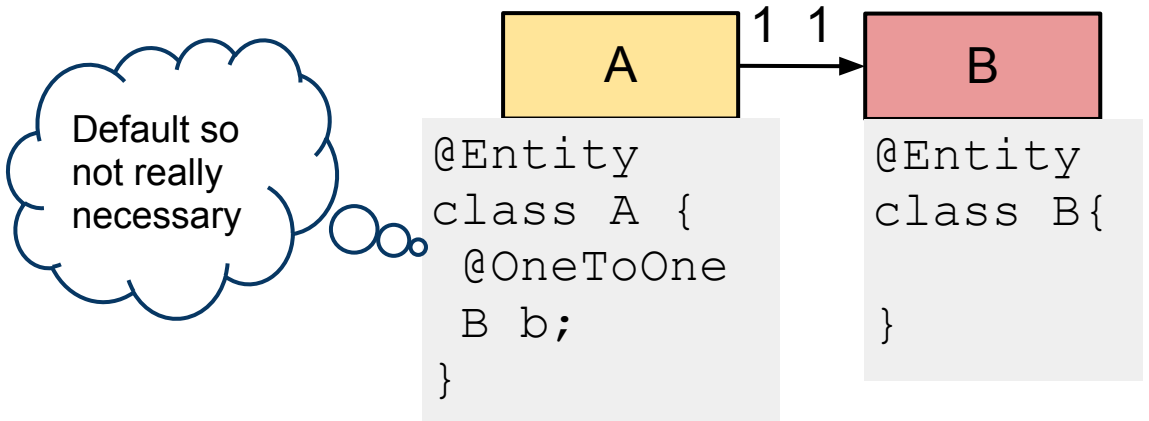
- UML 1:1 says one object having a reference to another. But the id of the objects aren't considered! It's just some objects associated

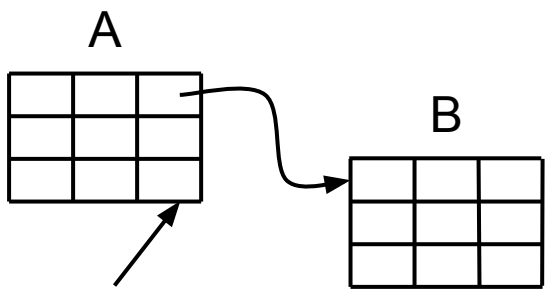
But when working with databases the id's are what's count

- Database (ER) 1:1 says one pk is related to one unique pk from other table (like splitting a table vertically). The id's are related!

Unidirectional 1:1 Mapping

@XToY , X is object having the annotation Y is associated object

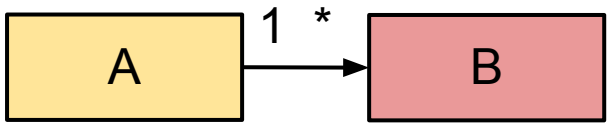




Join column,B_ID
(default name)

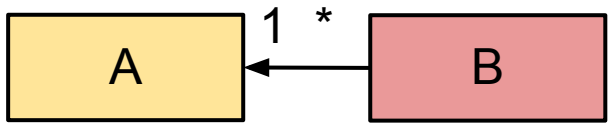
If the exact identity of B is important (database 1:1) need to use Column(unique=true) for B_ID

Unidirectional 1:* Mapping




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

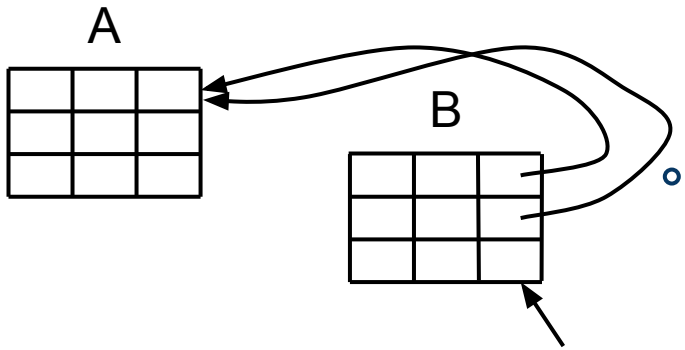
```
@Entity
class B{

}
```




```
@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

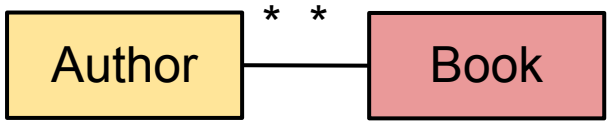
Mapping Bidirectional Association

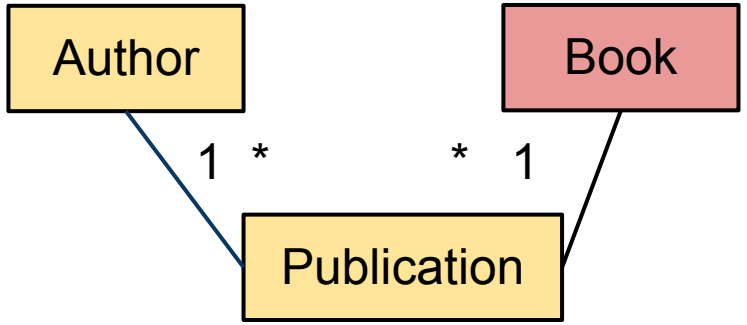
As noted; We'll avoid this so just a quick one to many bidirectional

```
// Class A
@OneToMany( mappedBy = "a")  // Must use mappedBy
private List<B> bs;

// Class B
@ManyToOne
@JoinColumn(name = "AUTHOR_FK")
private A a;


// Navigation
A a = ..get an A
List<B> bs = a.getBs();

B b = … get a B
A a = b.getA();
```

Mapping M:N

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

Summary Association Mapping

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

If need to navigate in “other” (non existing) direction 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 {
}
```

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;
```

Generation of Tables

We can specify that JPA should create the tables (using the annotated classes) when application runs, a **table generation strategy (DLL)**
- Possible to specify in NetBeans

Possible strategies
- None, no tables created
- Create, will create
- Drop and Create, delete and create

Strategy defined in persistence.xml, upcoming...

JPA Constraints and Bean Validation

@NotNull is a JSR 303 Bean Validation annotation. It has nothing to do with database constraints itself.

@Column(nullable = false) is the JPA way of declaring a column to be not-null. I.e.

When to use
@NotNull is on control layer
@Column is for entity classes in model layer

As noted: All layers should validate incoming data!

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 application but ...

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