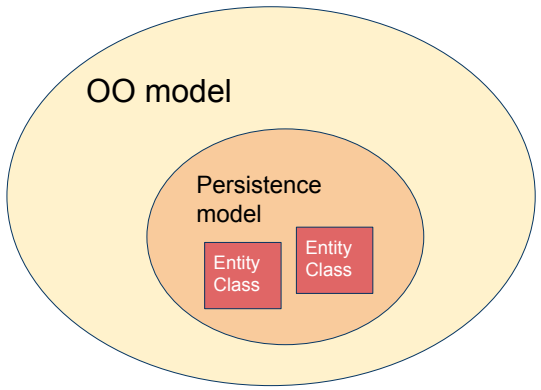
JPA Mappings

JPA Slides #2

Content

- Persistence model
- Entity Class
- Natural and surrogate keys
- Uni- and bidirectional mappings
- Validation
- Persistence unit

Persistence Model



The classes to persist are normally in the core model but …

- … not all classes in OO-model should be persisted …
- … we have a **persistence model**, a subset of the OO-model
- Classes (instances of) that should be persisted referred to as : **Entity classes**

Shop case: Which classes in persistence model?

Basic JPA Entity Class

```
@Entity
public class Book implements Serializable {

     @Id
     private Long id;
     private String title;
     private Float price;
     private String description;

     protected Book() {  // Non private must have
     }

     public Book(String title, Float price,
                                        String description) {
          this.title = title;
          this.price = price;
          this.description = description;
     }

     public Long getId() {
          return id;
     }
     ...
}
```

An **entity class** is a Java class typically representing a database table
- Mapping between class and table specified by annotations
- Instances of classes (attributes) will fill a row in the table

Specification (basics):
- Class fulfilling Bean specification (no args ctor, serializable, …)
- @Entity and @Id annotation mandatory
- No final, whatsoever!
- Inheritance OK
- Mixed non-entity and entity classes is also OK)
- Abstract class, OK
- Interface or Enum, NO!
- Must be listed in config file persistence.xml (in a "persistence unit" more later…)

Default mapping rules (basics)
- Class mapped to single table.
  - Table will have same name as class but uppercase
- Attributes mapped to column names, uppercase
- JDBC rules for mapping primitive Java types to database types
  - int, Integer, … byte[], Byte[], …String, Data, Calendar, TimeStamp,any ENUM, any Serializable.
- … more to come …

Customization

```
@Entity
@Table(
    name = "AvancedBook", // Default is BOOK
    uniqueConstraints={@UniqueConstraint(
        columnNames={"TITLE"})} // Just for demo,
                                // of course title may be same
)
public class Book implements Serializable {

    @Id
    private Long id;
    @Column(name = "book_title", nullable = false,
        updatable = false)
    private String title;
    private Float price;
    @Column(length = 50)
    private String description;
    ...
```

As usual JEE offers very many customization options

- Explanation of annotations
  - Have to look around a bit, it's there but sometimes not easy to find…
- Also see code samples

If attribute not should be persisted use **@Transient**

- Example: Customer shoppingcart shouldn't be persisted (content persisted in Order table or similar)

Collections and Enums

```java
@Entity
public class Book implements Serializable {

    // Will create extra table Tag
    @ElementCollection(fetch = FetchType.LAZY)
    @CollectionTable(name = "Tag" )
    private List<String> tags;

    public enum Genre {
        HORROR,
        MISANTHROPY,
        ROMANCE
    };
    // Will end up in same table
    @Enumerated(EnumType.STRING)
    private Genre genre;

    ...
```

Attributes as Collections or Enums no problems

- Collections in separate table (relationship created)
- Enums in same table

Embeddable

Any class

```
@Embeddable
public class Review implements Serializable {

    @Temporal(TemporalType.DATE) // Must have temporal for dates
    private Date reviewDate;
    private String reviewText = "No review yet";

    ...
}
```

In entity class

```
@Embedded
private Review review;
```

Embeddable will end up in same table

Natural vs Surrogate Keys

Natural keys. A natural key is one or more existing data attributes that are unique to the business concept.

Surrogate key. Introduce a new column, called a surrogate key, which is a key that has no business meaning.

Possible to use surrogate or natural keys with JPA

- Very heated topic and more debate
- You choose what you like …

Hmm … there seems to be quite a few natural surrogate keys?

- Person number (social security number, thou has some business meaning, gender, location …)
- Car registration number
- Order number
- Article number
- Shipping number
- …

Keys

Database generated surrogate key

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
```

Using inner PK-class for natural key

```
@EmbeddedId
private PK id;

@Embeddable  // Must have default ctor, equals, hashCode
public static class PK implements Serializable {
    @Column(name = "BOOK_TITLE")
    private String title;
    @Getter
    private String author;

    public PK(String title, String author) {
    this.title = title;
    this.author = author;
    }
}
```

Many possibilities for keys

- Surrogate keys may use @Generated value
  - Will create table SEQUENCE in JavaDB
  - NOTE: Before persisted object will have no id …
    - … will affect equals and hashCode …
    - … if objects in Collection possible not found, or other problems …
    - If using surrogate key persist object directly after creation
  - An article ….
  - Possible for application to obtain key from database, see link keys above.
- Natural keys may use @EmbeddedId or @IdClass and some key class

Inheritance and Generics

Inheritance

```
@MappedSuperclass
public class Person implements Serializable { … }

@Entity
public class Employee extends Person { … }
```

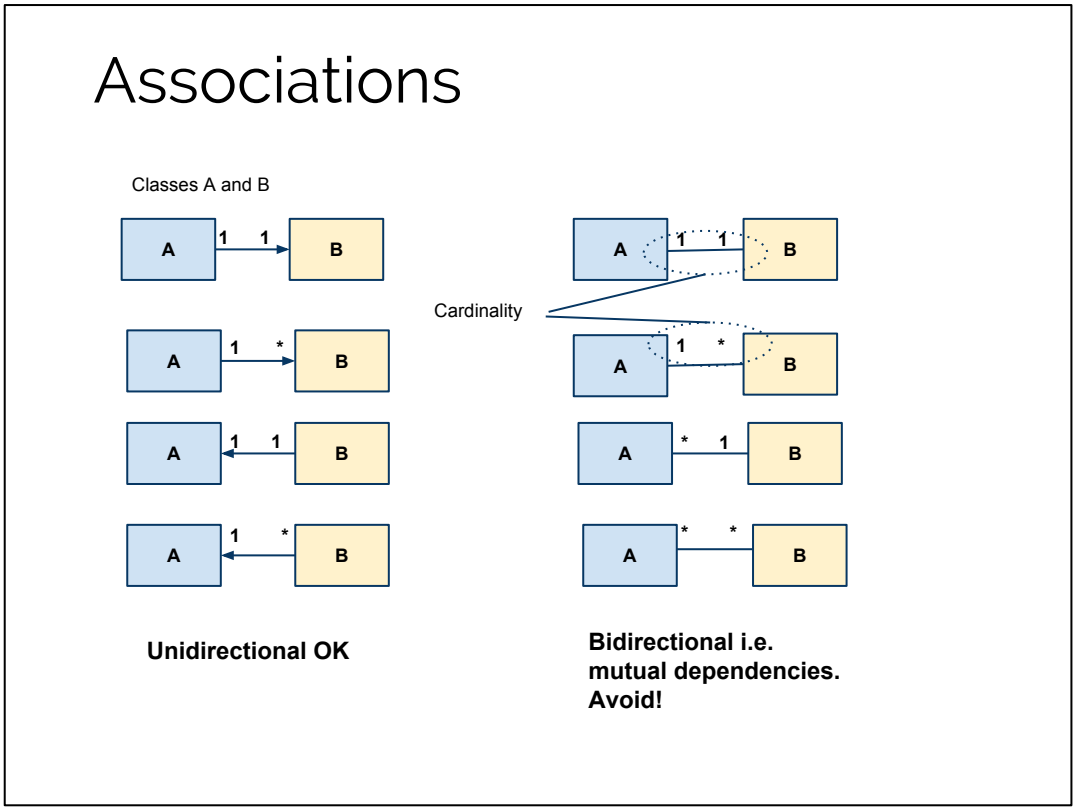
Generic Entity

```
@Entity
public class GenericEntity<T extends Serializable>
                                  implements Serializable {

    private T genericAttribute;

    …
}
```

JPA will handle generic classes and inheritance

Different strategies

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



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

- Mapping an association will result in relationships between tables
- Not a perfect match …
  - associations built on memory addresses, relationships built on keys
  - associations have direction, relationships not
  - In OO 1:1 cardinality must be forced through UNIQUE constraint on foreign key

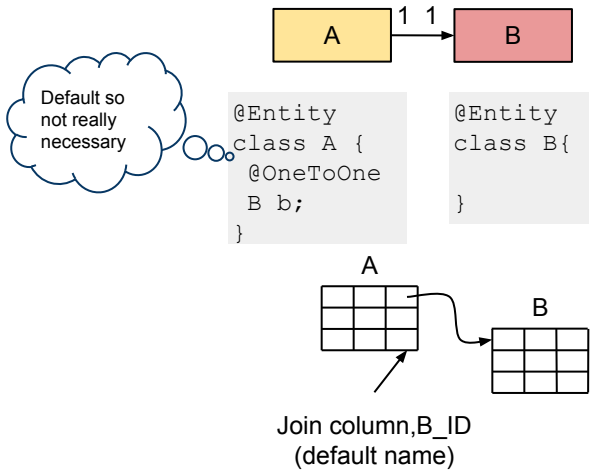
UML associations denotes a references in Java

- 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.

SE best practices for associations

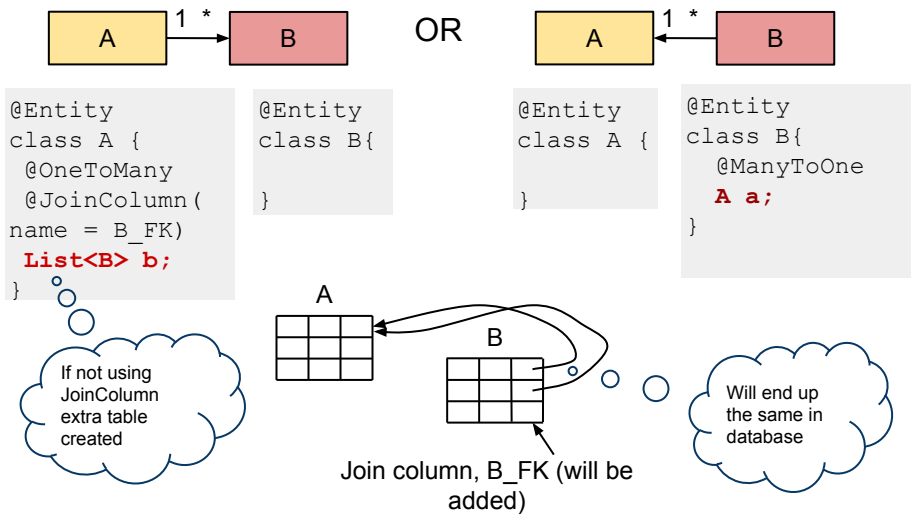
- Limit number of associations
- Prefer unidirectional, review use cases to decide direction
  - If need to navigate in "other" (non existing) direction in code have to search
  - Probably best to let database search (i.e. use queries, upcoming)

Unidirectional 1:1 Mapping



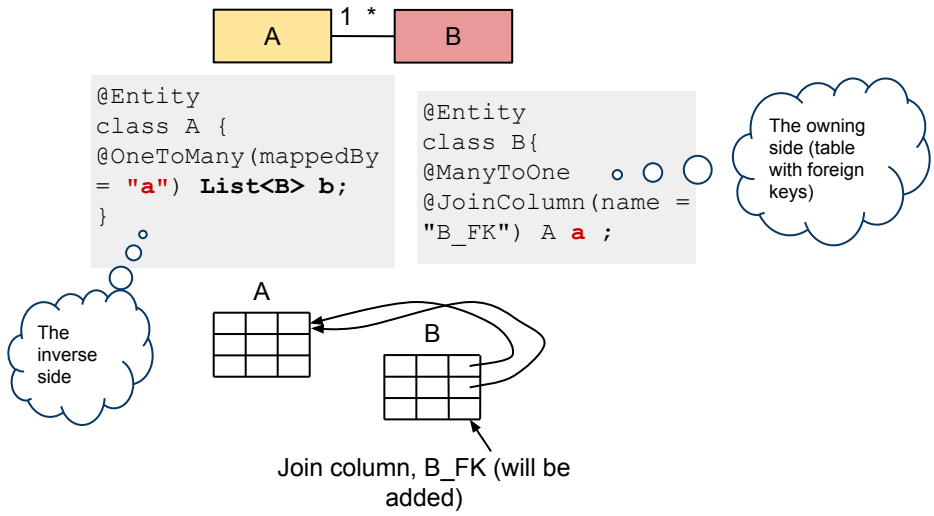
If the exact identity of B is important (database 1:1) need to use @Column(unique=true) for B_ID

Unidirectional 1:* Mapping

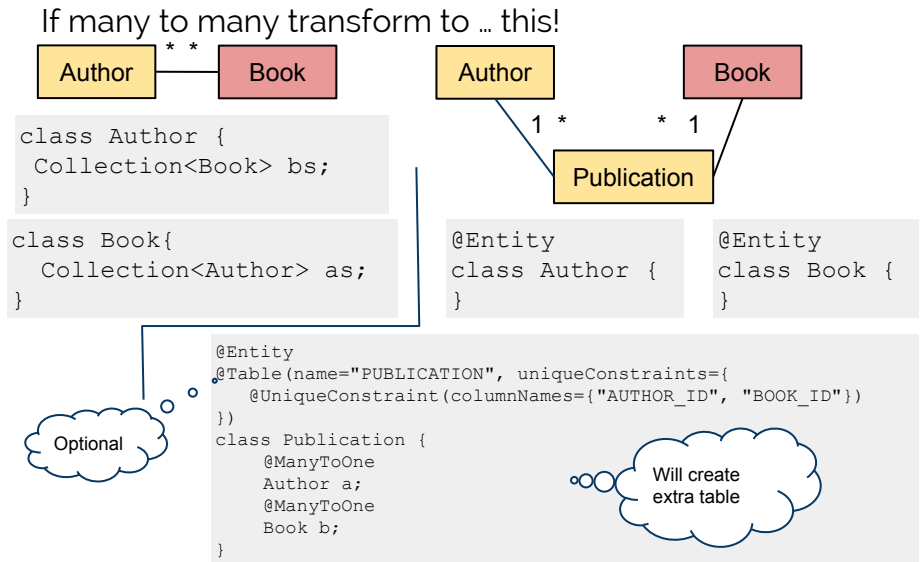


Note: @OntToMany more like “think as a programmer”, @ManyToOne mapping more like “think as a database modeller”

Bidirectional 1:* Mapping



Mapping *:*



Many to many associations will need association class

- In slide using Publication
- Will create table for association class

Entity Classes and Validation

```
@Entity
public class Author  {

    @Column(nullable = false,
                        updatable = false)
    private String name;

     ...
}
```

Possibly with many constraints on entity classes

- @NotNull is a Bean Validation annotation. It has nothing to do with database constraints itself.
  - Probably better use this in control layer
- @Column(nullable = false, updatable=false) is the JPA way of declaring a table column
  - not to accept null values
  - not be able to update
  - Use in persistence layer (entity classes)

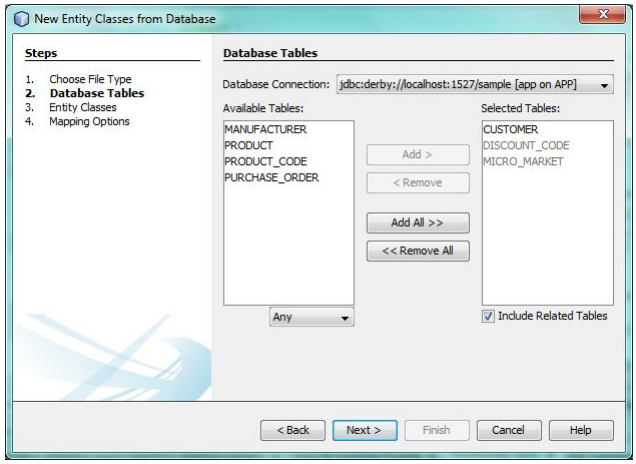
Persistence Unit Details

```
<?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>
```

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, always, Java Transaction API, JTA more to come …

Other Way Round



If skilled DB-designer

- Start with DB-design
- Auto Generate entity classes … (NetBean can do …)!
  - Will add associations (and more) all over … i.e. rather messy code
  - Also: Newer touch auto generated code …!