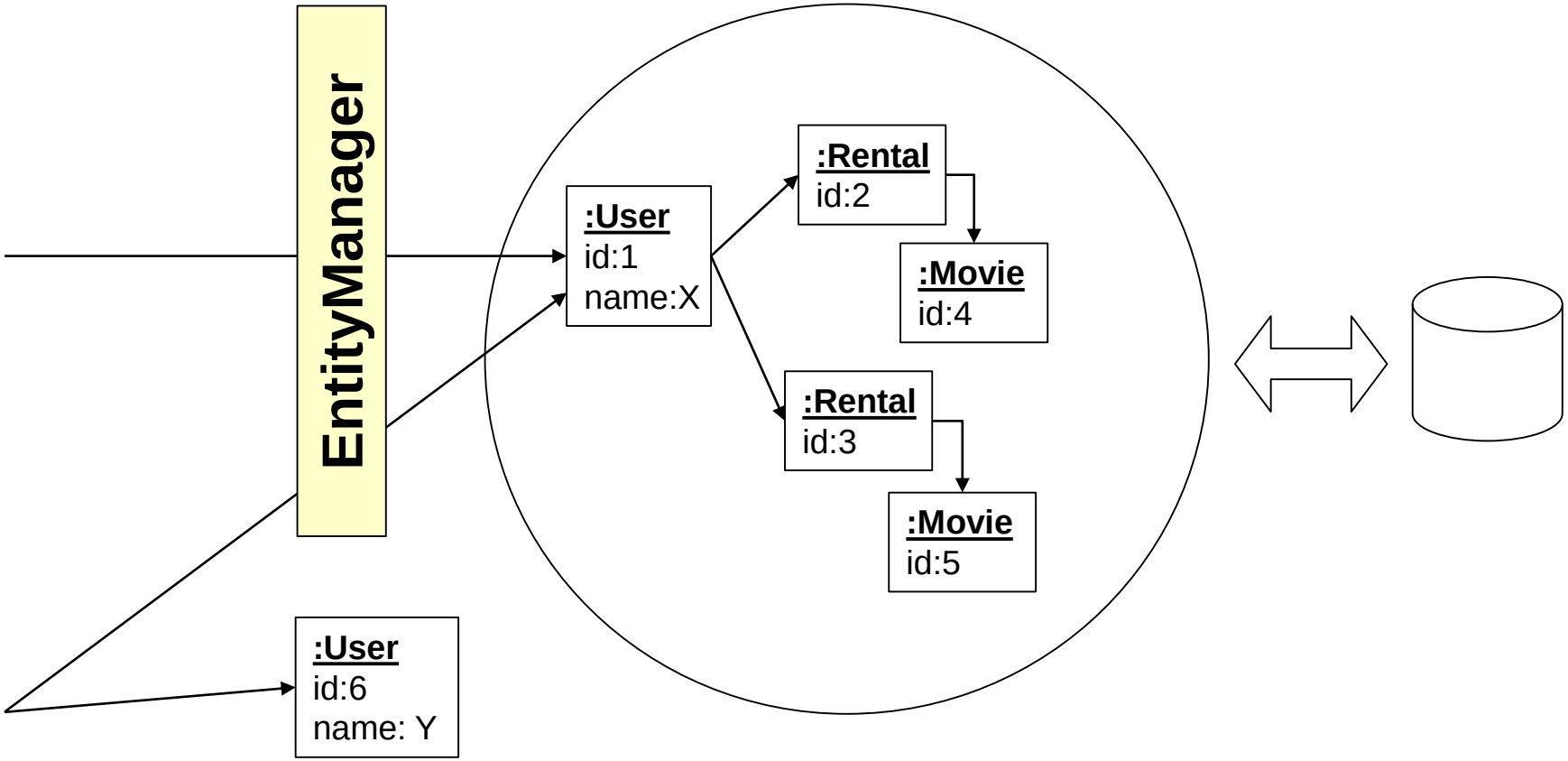
Java Persistence API

- **Entity Manager**
- **Relations**
- **Samples (with focus on owner / inverse)**
- **Summary**

Entity Manager

- **Persistence Context**
  - Managed set of entity instances (=> *managed objects*)
  - Persistence context is only accessible over the entity manager
  - Persistence context guarantees object uniqueness
    - Only one Java instance with the same persistent identity may exist in a Persistence Context
  - If persistence context participates in a transaction, then
    - Cache is associated with the transaction object
    - Changes performed on the entities are persisted upon COMMIT
    - Lazily referenced objects may be accessed (they are loaded on demand)
  - Open Session in View (OSIV)
    - Option to bind the persistence context to the thread processing of a web request (by default turned on)
    - Allows for lazy loading in web views (despite the original transactions already being completed) but no automatic persistence of changes

Entity Manager



Entity Manager

```
public interface EntityManager {
   public void persist(Object entity);
   public <T> T merge(T entity);
   public void remove(Object entity);
   public <T> T find(Class<T> entityClass, Object primaryKey);

   public void flush();
   public void setFlushMode(FlushModeType flushMode);
   public FlushModeType getFlushMode();

   public void refresh(Object entity);
   public void clear();
   public void detach(Object entity);
   public boolean contains(Object entity);

   public Query createQuery(String query);
   public <T> TypedQuery<T> createQuery(String query, Class<T> c);
}
```

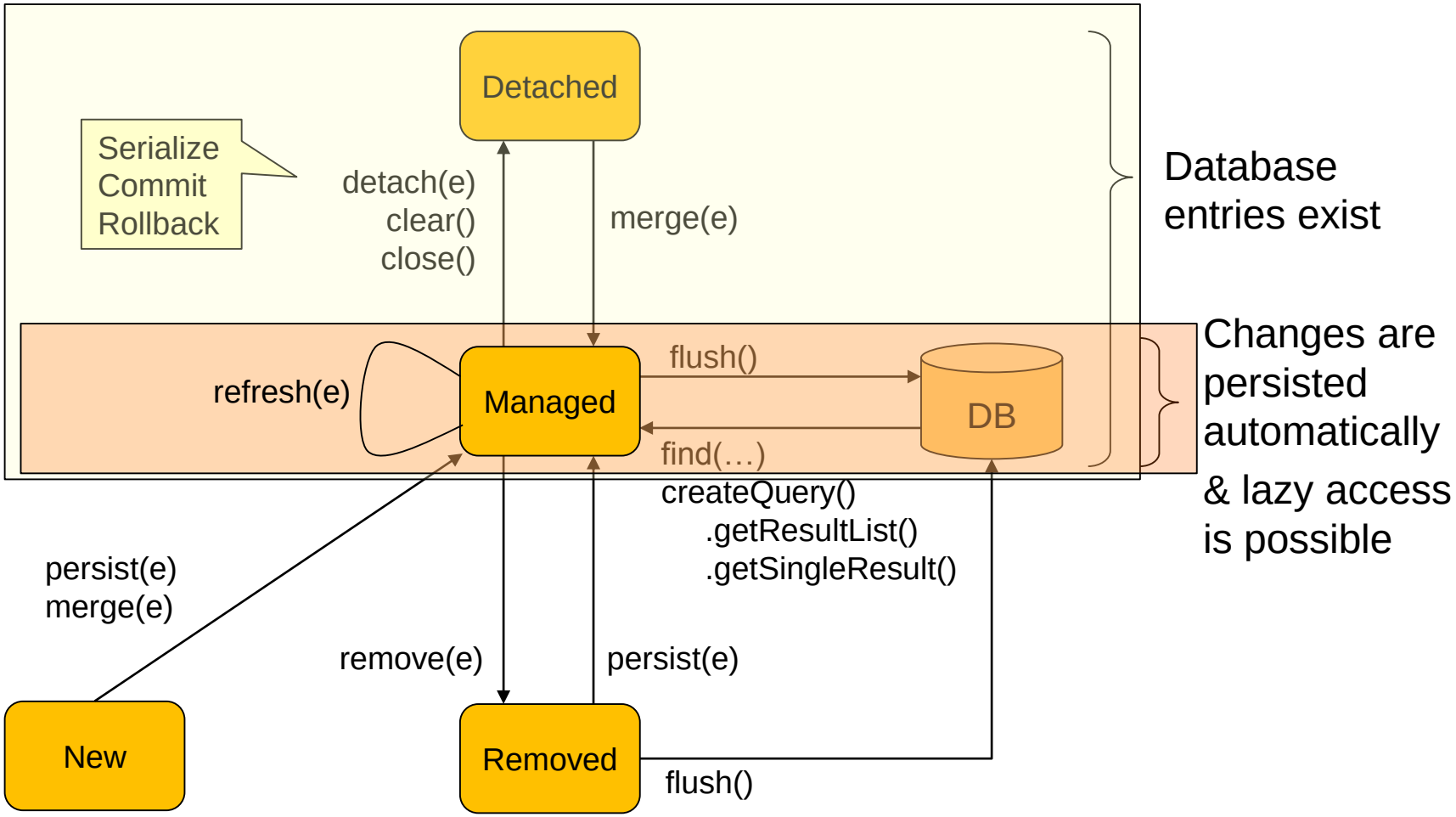
entity must be managed

entity must be managed

Entity Bean Lifecycle

- **New (Transient)**
  - Newly created entities which are not mapped to any DB row
  - Once they become managed, an insert statement is issued at flush time
- **Managed (Persistent)**
  - Persistent entities are associated with a DB row and are managed by the currently running persistence context
  - State changes are propagated to the DB at flush time (COMMIT)
  - Lazily referenced objects can be accessed
- **Detached**
  - Detached entities are associated with a DB row, but they are not managed by a persistence context
  - State changes are not propagated to the DB
  - Lazily referenced objects are NOT accessible
- **Removed**
  - Removed entities are only scheduled for deletion
  - Delete statement is executed at flush time

Entity Bean Lifecycle

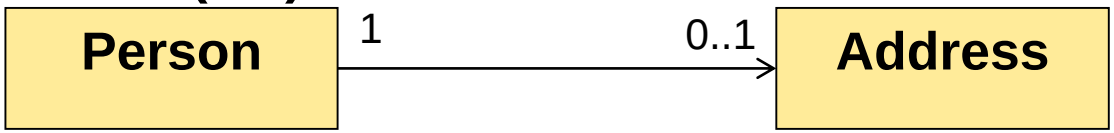


Java Persistence API

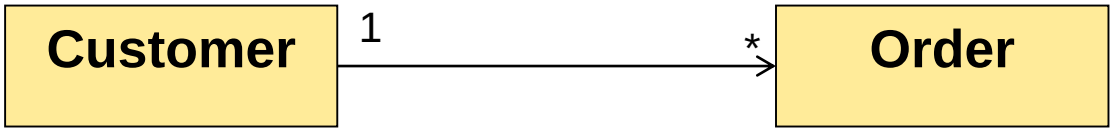
- **Entity Manager**
- **Relations**
  - Relation Types
  - Cascade Types
  - Fetch Types
- **Samples (with focus on owner / inverse)**
- **Summary**

Entity Relationships (unidirectional)

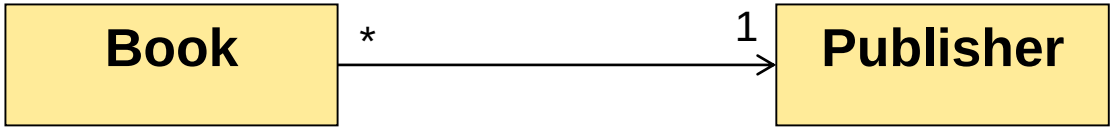
- **One-To-One (1:1)**



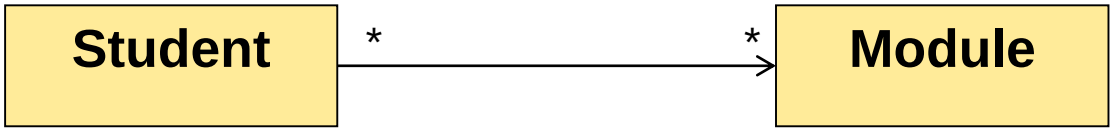
- **One-To-Many (1:*)**



- **Many-To-One (*:1)**

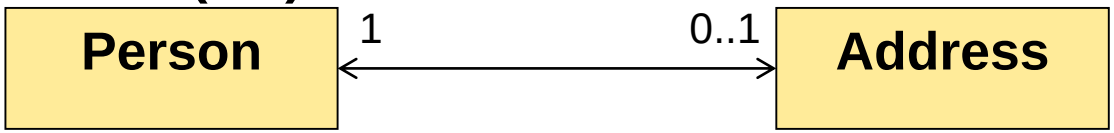


- **Many-To-Many (*:*)**

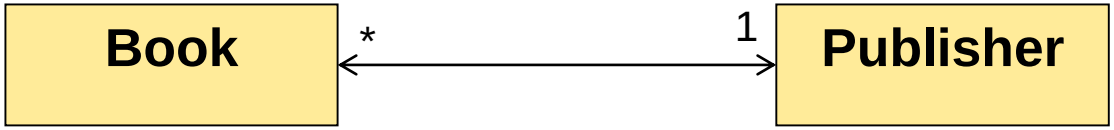


Entity Relationships (bidirectional)

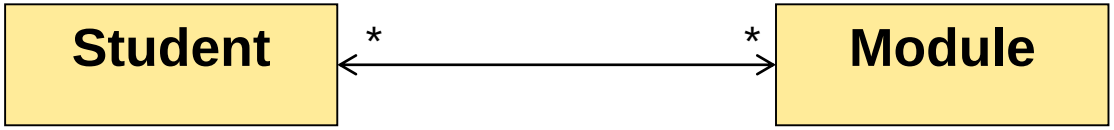
- **One-To-One (1:1)**



- **Many-To-One (*:1) / One-To-Many (1:*)**



- **Many-To-Many (*:*)**



@OneToOne

```
public @interface OneToOne {
   // The entity class that is the target of the association
   // Defaults to the type of the field of the association
   Class targetEntity() default void.class;

   // The field that owns the relationship (defined on non-owning
   String mappedBy() default ""; // side of the association)

   // The operations that must be cascaded to the target
   CascadeType[] cascade() default {};

   // Whether the association should be lazily or eagerly fetched
   FetchType fetch() default EAGER; // lazy is a hint

   // Whether the association is optional.
   boolean optional() default true;

   // Whether to apply the remove operation to entities removed
   boolean orphanRemoval() default false; // from the association
}
```

@OneToMany

```
public @interface OneToMany {
   // The entity class that is the target of the association.
   // Optional if the collection is defined using generics.
   Class targetEntity() default void.class;

   // The field that owns the relationship.
   // Required unless the relationship is unidirectional.
   String mappedBy() default "";

   // The operations that must be cascaded to the target
   CascadeType[] cascade() default {};

   // Whether the association should be lazily or eagerly fetched
   FetchType fetch() default LAZY;

   // Whether to apply the remove operation to entities removed
   boolean orphanRemoval() default false;
}
```

@ManyToOne

```
public @interface ManyToOne {
   // The entity class that is the target of the association
   // Defaults to the type of the field of the association
   Class targetEntity() default void.class;

   // The operations that must be cascaded to the target
   CascadeType[] cascade() default {};

   // Whether the association should be lazily or eagerly fetched
   FetchType fetch() default EAGER;

   // Whether the association is optional.
   boolean optional() default true;
}
```

@ManyToMany

```
public @interface ManyToMany {
   // The entity class that is the target of the association.
   // Optional if the collection is defined using generics.
   Class targetEntity() default void.class;

   // The field that owns the relationship.
   String mappedBy() default "";

   // The operations that must be cascaded to the target
   CascadeType[] cascade() default {};

   // Whether the association should be lazily or eagerly fetched
   FetchType fetch() default LAZY;
}
```

Attributes on Associations

	@OneToOne	@OneToMany	@ManyToOne	@ManyToMany
targetEntity	X	X	X	X
mappedBy	X	X		X
cascade	X	X	X	X
fetch	X	X	X	X
optional	X		X	
orphanRemoval	X	X		

Relationship Directions

- **Relationships can be**
  - Unidirectional
    - Entity in which the relationship is defined is the owner
  - Bidirectional
    - Has an owning side
    - Has an inverse side

- **Owning side**
  - For 1:1 relations: the owning side contains the foreign key
  - For n:1 relations: owner is the entity where the foreign key is stored
  - For n:n relations: either side may be the owning side
  - Owning side determines the updates to the relationships in the database

Relationship Directions

- **mappedBy**
  - The inverse side must refer to its owning side by use of `mappedBy` of the Relationship annotation
  - The many-side of a one-to-many / many-to-one bidirectional relationship must be the owning side, hence `mappedBy` cannot be specified on the `@ManyToOne` annotation

	unidirectional	bidirectional
1:1	✓	✓
1:n	✓	✗
n:1	✓	✓
n:n	✓	✓

Relationship Cascading

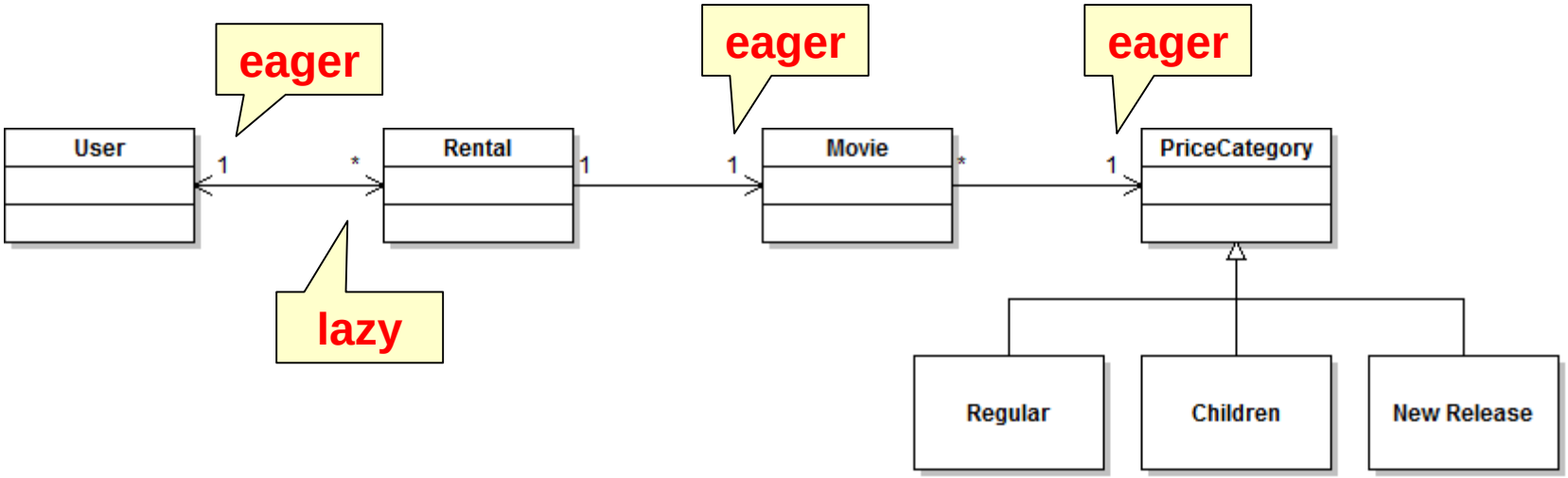
- **On associations, cascading types can be defined**
  - `CascadeType.PERSIST`
    - Cascades persist operation on associated entities
  - `CascadeType.REMOVE` ◆⟶
    - If entity is removed, then all associated entities are removed as well
    - Should only be declared on @**One**ToOne and @**One**ToMany associations
  - `CascadeType.REFRESH`
    - If entity is refreshed (`em.refresh`), associated entities are refreshed as well
  - `CascadeType.MERGE`
    - If entity is merged (`em.merge`) with the persistence context, then associated entities are merged as well
  - `CascadeType.DETACH` (*since JPA 2.0)*
    - If entity is detached (`em.detach`), then associated entities are detached as well
  - `CascadeType.ALL`: Combination of the above five attributes

Relationship Fetch Type

- **On associations, a fetch type can be defined**
  - Allows to specify when objects are loaded (eager and lazy loading)
  - Can be specified on
    - `@OneToMany` / `@OneToOne` / `@ManyToMany` / `@ManyToOne`
    - On regular fields using the `@Basic` annotation
  - `FetchType.EAGER`
    - Dependent objects are loaded with original object
    - Default for `@OneToOne` and `@ManyToOne`
    - Default for regular fields
  - `FetchType.LAZY`
    - Dependent objects are loaded on demand
    - Original object must not be detached upon loading associated entities!
    - Default for `@OneToMany` and `@ManyToMany`

Relationship Fetch Type

- **Model Classes of the Movie Rental Application**



Relationship Fetch Type

- **Automatically Loading of Entities**

```
// 1. Loading the user entity
select count(u.user_id) from users u
// findById returns null if entity does not exist
select u.user_id, u.user_email, u.user_firstname, u.user_name
from users u where u.user_id=?

// 2. Loading the associated rental objects with their movies
select r.user_id, r.rental_id, r.movie_id, r.rental_rentaldate,
r.rental_rentaldays, m.movie_id, m.pricecategory_fk,
m.movie_releasedate, m.movie_rented, m.movie_title,
pc.pricecategory_id, pc.pricecategory_type
from rentals r
left outer join movies m on r.movie_id=m.movie_id
left outer join pricecategories pc on
                      m.pricecategory_fk=pc.pricecategory_id
where r.user_id=?
```

Relationship Fetch Type

- **Automatically Loading of Entities (if all associations are lazy)**

```
// 1. Loading the user entity
select count(u.user_id) from users u
// findById returns null if entity does not exist
select u.user_id, u.user_email, u.user_firstname, u.user_name
from users u where u.user_id=?

// 2. Loading the associated rental objects                    1
select r.user_id, r.rental_id, r.movie_id, r.rental_rentaldate,
r.rental_rentaldays from rentals r where r.user_id=?

// 3. Loading the corresponding movie objects (2 rentals)      n
select m.movie_id, m.pricecategory_fk, m.movie_releasedate,
m.movie_rented, m.movie_title from movies m where m.movie_id=?

select m.movie_id, m.pricecategory_fk, m.movie_releasedate,
m.movie_rented, m.movie_title from movies m where m.movie_id=?
```

orphanRemoval Attribute

- **Orphan database entries**
  - Entries in the DB which are no longer accessible are removed

```
@Entity
public class Order {
   @Id @GeneratedValue
   Integer id;
   @OneToMany
   private List<OrderLine> lines;
   ...
}
```

```
@Entity
public class OrderLine {
   @Id @GeneratedValue
   Integer id;
   ...
}
```

    - `order.getOrderLines().remove(0); // => orderline is not removed`
      `// unless orphanRemoval = true`
  - Only defined in annotations `OneToXXX`
  - Comparable to a cascade delete (initiated by changing the relation)
  - Usually `orphanRemoval=true` if `REMOVE` ∈ cascade, actually `orphanRemoval=true` has same semantics (`REMOVE` needs not be added)

OrderBy

- **Order of elements in a xxxToMany Relation is not defined**
  - Can be defined with an `@OrderBy` annotation
  - Parameter is a field of the referenced entity which is to be used for sorting (default: ascending)
  - Examples
    - `@OrderBy` ordered by primary key
    - `@OrderBy("name")` ordered by name (ascending)
    - `@OrderBy("name DESC")` ordered by name (descending)
    - `@OrderBy("city ASC, name ASC")` phonebook order
  - Programmer is responsible to keep the order upon changes in the list

Summary: Relation Annotation Attributes

- **OneToOne / OneToMany / ManyToOne / ManyToMany - Attributes**
  - targetEntity — Class
    - e.g. if result type is Object or a raw collection, not used with generics
  - fetch — EAGER / LAZY
    - determines fetch type
  - cascade — MERGE / PERSIST / REFRESH / DETACH / REMOVE / ALL
    - determines cascade operation
  - mappedBy — String — *not for ManyToOne*
    - used for bidirectional associations (on the inverse side)
  - optional — boolean — *only for OneToOne/ManyToOne*
    - determines, whether null is possible (0..1)
  - orphanRemoval — boolean — *only for OneToOne/OneToMany*