ITMD 415/515
Advanced Software Development

Week 5 – ORM and JPA

Scott Spyrison

Administrative Stufff

- Goncalves Examples from Java EE 7 Book
  - https://github.com/agoncal/agoncal-book-javaee7
- Extra Reading
- Midterm

Thinking Ahead to your FP

- Think of a domain or model you would like to work with going forward
  - Must support multiple Entities
  - Must support the idea of relationships between Entities
  - Your overall idea should support 2 different types of users
  - Specs will follow so you have time to develop your thoughts and ideas along with material
- One of the questions on your midterm will be to document your design
- You will introduce your model over the next 2 labs, and continue working with it all semester

Relational Databases vs OOP

- Relational Databases
  - Store data in tables consisting of rows and columns
  - Primary keys
  - Relationships with other tables via foreign keys or join tables
- OOP
  - Objects inherit state and behavior from other objects
  - Can contain other objects as state
  - Have references to collections of other objects

Object Relational Mapping

- Join the worlds of RDBMS and OOP
- Delegate correspondence with RDBMS to an external tool or framework that:
  - Provides an object-oriented view of the relational database
  - Provides bi-directional correspondence
    - Reads from the database to the object model
    - Persists from the object model to the database

JPA Specification

- A specification not an implementation
  - EclipseLink is the current reference implementation
  - History
- Abstraction Layer over JDBC and SQL
- jakarta.persistence package
- JPA Provides
  - Object Relational Mapping (to map relational databases to an object model - entities)
  - Entity Manager (manage entities, i.e. CRUD)
  - JPQL
  - Transactions and Locking
  - Lifecycle for persistent objects
  - Database and/or script generation

JPA Entities

- Live short-term in memory
- Persistently in database
- POJOs with annotations:

```
@Entity
public class Book {

      @Id @GeneratedValue
      private Long id;
      private String title;
      private Float price;
      private String description;
      private String isbn;
      private Integer nbOfPage;
      private Boolean illustrations;

      public Book() {
      }

      // Getters, setters
}
```

JDBC vs JPA

- JPA is built on top of JDBC
- Compare and Constrast

Entity Manager

- Central piece of JPA
- API to find, query, create, remove and synchronize entities with databases
- Manages state and life cycle of entities
- EntityManager is an interface implemented by a persistence provider (like EclipseLink)
- **Each EntityManager instance is associated with a *persistence context***

Persistence Context

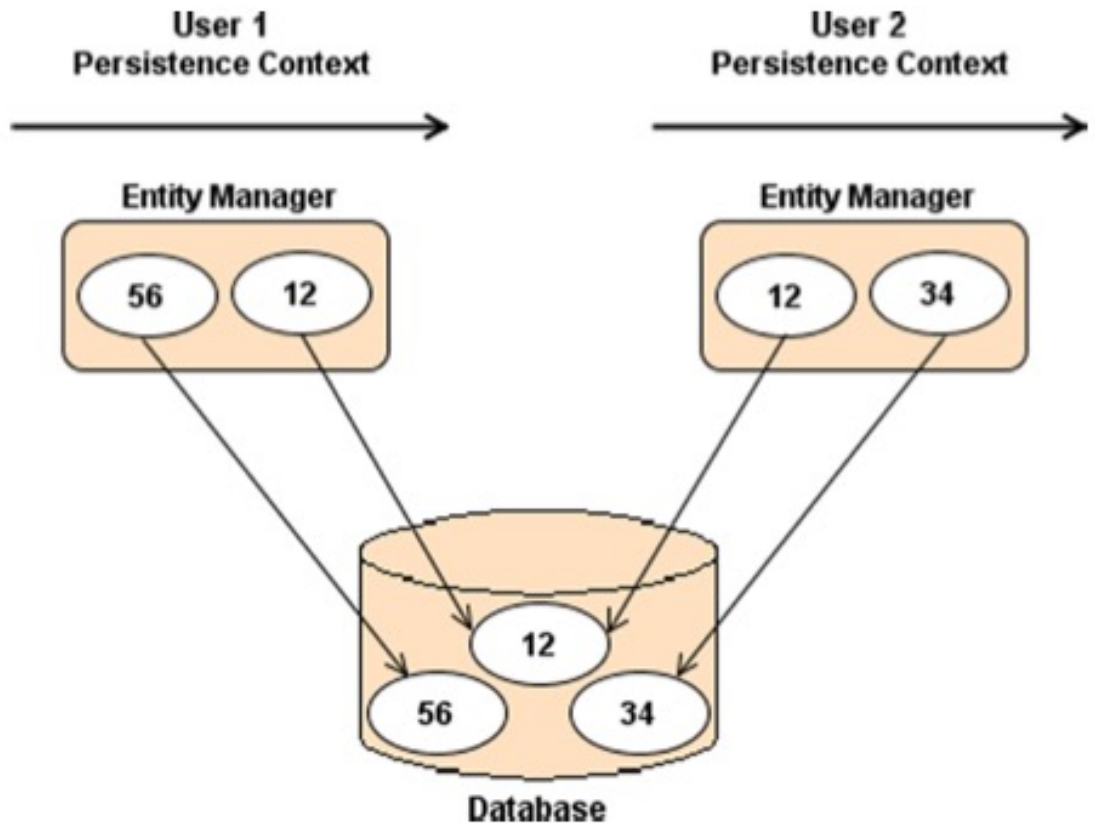


Figure 6-1. *Entities living in different users' persistence context*

Persistence Context

- Each EntityManager instance is associated with a *persistence context*
  - A persistence context is a set of managed entity instances at a given time for a given user's transaction
  - Persistence Identities are unique within context
  - A **first-level cache**. Objects live in the persistence context for the duration of the transaction, and are then flushed to the database

Obtaining an Entity Manager

- Application Managed (Lab 4)
  - Persistence.createEntityManagerFactory(“foo”)
  - Requires EntityTransaction
  - *Listing 6-2 in Goncalves*
- Container Managed (Lab 6)
  - @PersistenceContext(unitName=“foo”)
  - Uses JTA for transaction
  - *Listing 6-3 in Goncalves*

Manipulating Entities - CRUD

- C(reate)
  - EntityManager.persist(e) adds an entity to the current persistence context and inserts to the database if it doesn't already exist
- R(ead)
  - EntityManager.find(pk) finds an entity and adds it to the current persistence context
  - EntityManager.getReference(pk) find an entity and adds it to the current persistence context, but lazily fetches its data
- U(update)
  - EntityManager.merge(e) reattaches a detached entity to the current persistence context and updates the database
  - Updating a managed Entity through setters
- D(delete)
  - EntityManager.remove(e) deletes an entity from the database and detaches it from the current persistence context. Will be GC'd.

Manipulating Entities

- Synchronizing
  - EntityManager.flush(e) entity to database, no commit
  - EntityManager.refresh(e) entity from database (overwrite in memory state)
- Persistence Context
  - EntityManager.contains(e) an entity in the current persistence context
  - EntityManager.clear() empties the current persistence context
  - EntityManager.detach(e) removes an entity from the current persistence context
- *Chapter 6, Goncalves Ex 03*

Entity Life Cycle

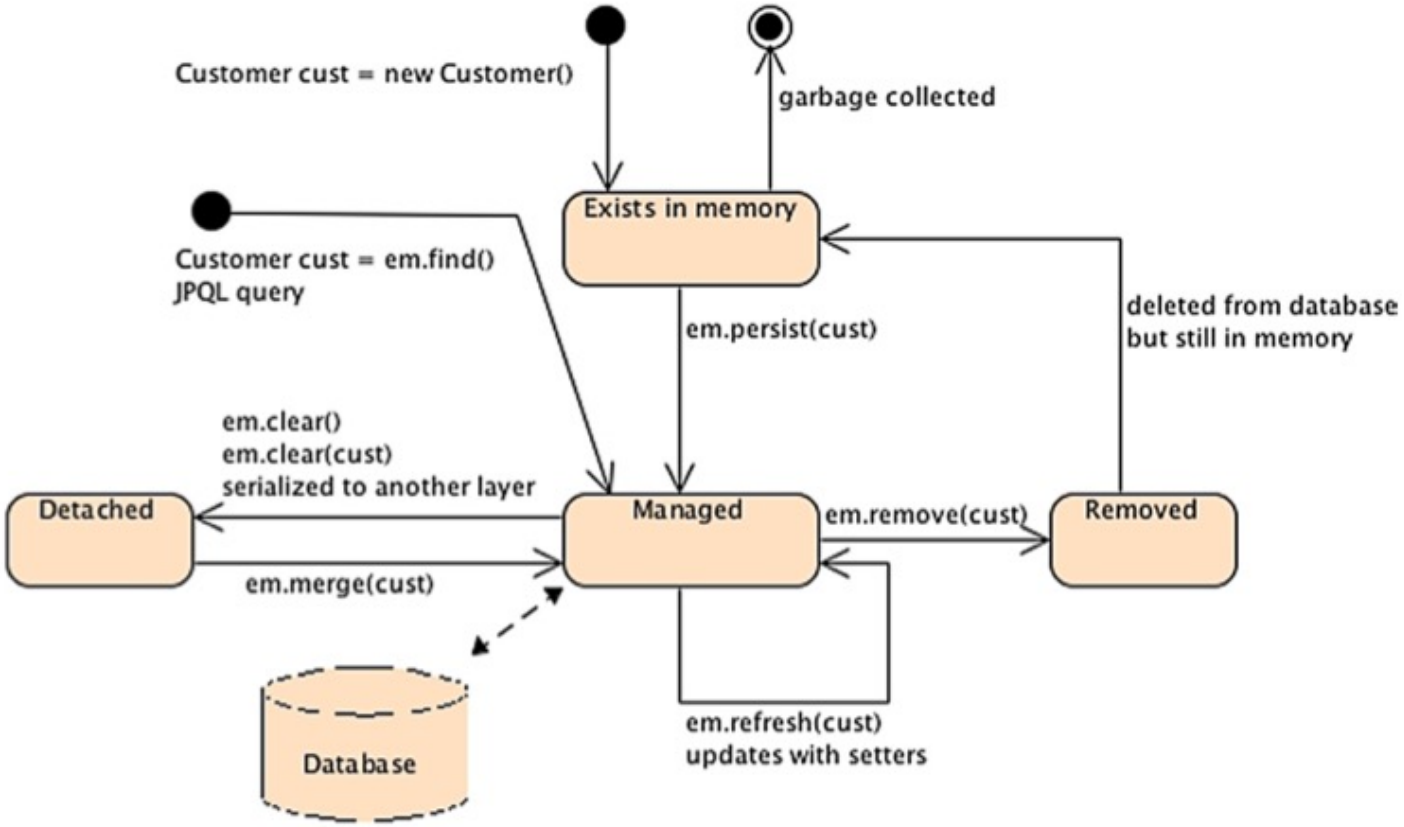
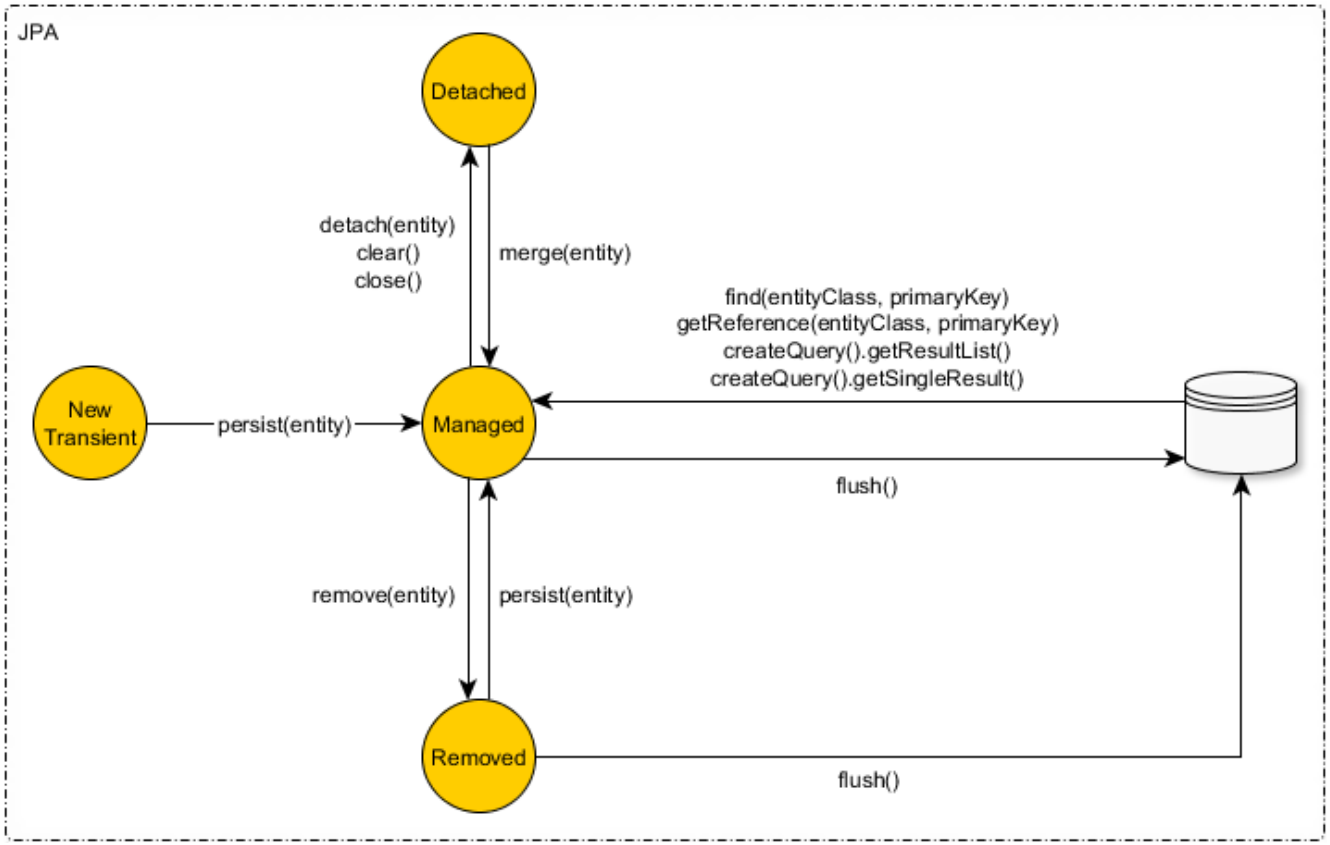


Figure 6-6. *Entity life cycle*

Vlad's Entity Life Cycle

- https://vladmihalcea.com/a-beginners-guide-to-jpa-hibernate-entity-state-transitions/



JPA Reference Slides

- Keep these handy for the next few weeks
- Links between Optional Reading, Reference Documentation and Goncalves Examples

Tables

- @Table (*Goncalves Ex01*)
  - http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Entities/Table
  - name
  - schema
  - uniqueConstraints{}

Tables

- @SecondaryTable (*Goncalves Ex02*)
  - http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Entities/SecondaryTable
  - Up until now, an Entity was a table. Sometimes, existing data models spread the data across multiple tables.

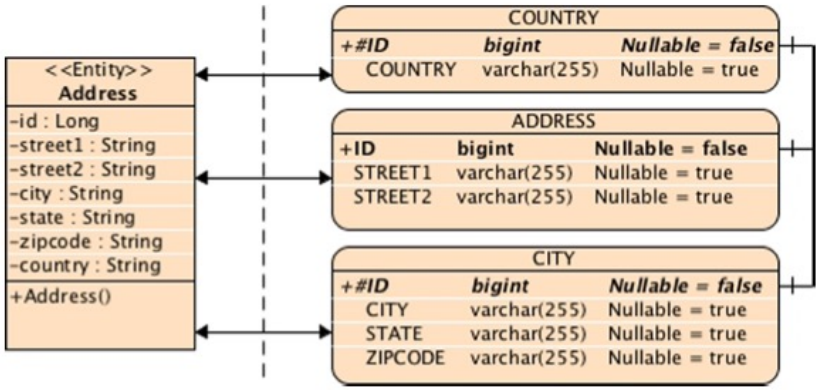


Figure 5-1. *The Address entity is mapped to three tables*

Primary Keys

- Uniquely identify each row in a table
- JPA **requires** entities to have an identifier mapped to a primary key
- Comprises either single or set of columns
  - Simple Primary Key
  - Composite Primary Key
- If you can, *stick to a simple key*
- Natural ID vs Generated (Surrogate) ID:
  - https://vladmihalcea.com/database-primary-key-flavors/

Simple Primary Keys

- @Id (*Goncalves Ex01 and Ex02*)
  - http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Mapping/Entities/Ids/Id
  - Must correspond to single attribute of entity
  - Must not change
  - Allowable types:
    - *Primitive Java types:* byte, int, short, long, char
    - *Wrapper classes of primitive Java types:* Byte, Integer, Short, Long, Character
    - *Arrays of primitive or wrapper types:* int[], Integer[]
    - *Strings, numbers, and dates:* String, BigInteger, Date

Id Generation for Simple PK

- @Id value can be created manually by application or automatically by the provider
- @GeneratedValue (*Goncalves Ex01*) has different strategies
  - http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Entities/Ids/GeneratedValue
  - SEQUENCE
  - IDENTITY
  - TABLE
  - AUTO

equals and hashCode

- Review Object as a Superclass in Java SE Tutorial
  - Also: IBM Hashing it Out
  - Also: Understanding Inheritance in Java
- If you override equals, you *must* override hashCode
- equals
  - compares two objects, returns true if equal
- hashCode
  - If two objects are equal, their hash codes must also be equal
  - By default, returns the memory address
  - Overriding equals invalidates default implementation

equals and hashCode and JPA

- Vlad
  - https://vladmihalcea.com/hibernate-facts-equals-and-hashcode/
  - https://vladmihalcea.com/how-to-implement-equals-and-hashcode-using-the-jpa-entity-identifier/
  - https://vladmihalcea.com/the-best-way-to-implement-equals-hashcode-and-tostring-with-jpa-and-hibernate/

Option One – Natural ID/Business Key

- So sayeth Vlad:
- “Those entity fields having the property of being unique in the whole entity object space are generally called a business key.”
- “The business key is also independent of any persistence technology employed in our project architecture, as opposed to a synthetic database auto incremented id.”
- “So, the business key must be set from the very moment we are creating the Entity and then never change it.”

Option One – Natural ID/Business Key

- Decide/Design what attribute(s) determine uniqueness in each of your entities
  - ISBN
  - UPC Code
  - Person ID Number
  - Drivers License Number
  - Library Card Number
  - Serial Number
  - First, Last, Middle Name, Sex at Birth, Date of Birth, and Instant of Birth
  - SSN
  - Some Made Up ID Number
  - Etc
- **These should be unique in your database – whether PK or not**
- **These unique identifiers are generated by your application**
- Use these attribute(s) for your equals and hashCode

Option Two – Generated ID

- You can use the database @GeneratedID in equals/hashCode provided that
  - You consider null in your equals method, and return false if one of the **entity identifiers** is null
  - hashCode always returns the same value
  - https://vladmihalcea.com/how-to-implement-equals-and-hashcode-using-the-jpa-entity-identifier/ (ref: **Fixing the entity identifier equals and hashCode)**

Option Three – UUID

- If you can't figure out a business key, and you don't want to use the database @GeneratedID, you can use UUID:
  - Database agnostic
  - Can be generated by application
  - https://vladmihalcea.com/uuid-identifier-jpa-hibernate/ (ref: **GenerationType.AUTO)**
  - Note – this strategy may need some adjustment for use with EclipseLink, but the fundamentals here are good. ***@NaturalID is Spring only – don't use unless***

Attributes

- @Basic (*Goncalves Ex09*)
  - http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Mapping/Basic_Mappings/Basic
  - optional (default true)
  - fetch (default FetchType.EAGER)

Attributes

- @Column (*Goncalves Ex11*)
  - Defines the properties for a column such as name, size, unique, nullable, insertable, updateable
  - http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Mapping/Basic_Mappings/Column
  - Name
  - Unique (default false)
  - Nullable (default true)
  - Length (default 255)

Attributes

- @Temporal (*Goncalves Ex14*)
  - http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Mapping/Basic_Mappings/Temporal
  - TemporalType.DATE
  - TemporalType.TIME
  - TemporalType.TIMESTAMP
- No longer required with JPA 2.2 when using the java.time API (LocalDate, LocalTime, LDT)

Attributes

- @Transient (*Goncalves Ex14*)
  - http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Mapping/Basic_Mappings/Transient
  - A way to not map an attribute to persistent table
    - For example, a calculated field

Attributes

- @Enumerated (*Goncalves Ex17*)
  - http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Mapping/Basic_Mappings/Enumerated
  - Can use EnumType.STRING to store data with String as opposed to ordinal
  - Adding a new constant to the top of the enum would re-arrange ordinals…

Embeddables

- @Embeddable (*Goncalves Ex29 and Ex32*)
- @Embedded
  - http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Entities/Embeddable
  - Used with composite primary keys
  - Embeddables do not have a persistent identity
  - They are embedded in owning entities
  - If the owning entity is removed, so is the embedded

Composite Primary Keys

- When it is not possible to use a single column or attribute, JPA supports compound or composite
- Two strategies, both require separate primary key class
  - Must implement equals and hashCode
  - public class
  - Public accessors and mutators
  - Default noarg constructor
  - Implement serializable if they need to cross architectural layers
- Same database output, but querying is different

Composite Primary Keys

- Embedded Composite Primary Key Class
- @Embeddable with @EmbeddedId (*Goncalves Ex04*)
  - http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Entities/Ids/EmbeddedId
  - Create @Embeddable class conforming to above requirements
  - Identify primary key class with @EmbeddedId on entity

Composite Primary Keys

- Nonembedded Composite Primary Key Class
- @IdClass (*Goncalves Ex06*)
  - http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Entities/Ids/IdClass
  - Each field needs to also be declared on entity and annotated with @Id
  - Primary key class does not require annotations, but must conform to above requirements
  - Identify primary key class as @IdClass on entity

Embedded vs NonEmbedded

- For Composite PK, which way is better?
  - https://vladmihalcea.com/the-best-way-to-map-a-composite-primary-key-with-jpa-and-hibernate/

Access Type

- Field Access (Annotate attributes)
  - Default used in the book
  - AccessType.FIELD
- Property Access (Annotate getters)
  - AccessType.PROPERTY
- My recommendation – be **consistent** with your choice. (*Goncalves Ex20)*

JPQL

- What about querying? Everything up until here as been EntityManager.find(pk)
- JPQL is an object-oriented query language, in which we are looking for entities or collections of entities
- Roots in SQL, but uses standard dot notation

Simplied JPQL Statement Syntax

```
SELECT <select clause>
FROM <from clause>
[WHERE <where clause>]
[ORDER BY <order by clause>]
[GROUP BY <group by clause>]
[HAVING <having clause>]
```

JPQL Tools and Syntax

- NetBeans JPQL Editor
- Right-click Persistence Unit and choose “Run JPQL Query”
- *Goncalves Chapter 6 Ex 21*

Dynamic Queries

- Created as needed by the application
- “On the fly”
- Can accept named or positional parameters
- Use EntityManager.createQuery() method, passing a String representing JPQL query
  - Query query = em.createQuery ()
  - TypedQuery<E> query = em.createQuery()
  - query.getResultList();

Named Queries

- Similar to PreparedStatement
- Static and Unchangeable
  - lose flexibility
- Translated to SQL when the application starts
  - gain efficiency and performance
- EntityManager.createNamedQuery()
- The names of NamedQueries must be unique within persistence context, even across entities

Other Query APIs

- Criteria API
  - For example: criteriaQuery.select(c).where(builder.equal(c.get("first Name"), "Vincent"));
- Native Queries
  - Allows execution of native SQL through JPA
- Stored Procedure Queries
  - Allows execution of existing database SP's through JPA
- Query By Example (QBE or Matcher)
- QueryDSL

Collections of Basic Types

- @CollectionTable
- @ElementCollection (*Goncalves Ex23*)
  - Useful for storing Set or List of basic types such as Strings or Integers

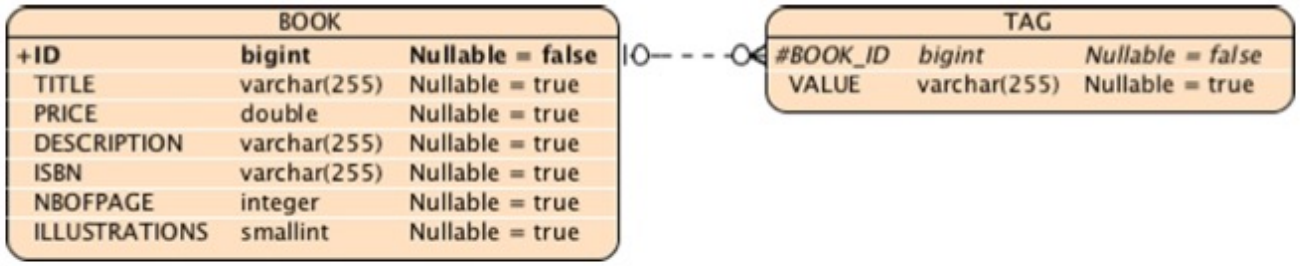


Figure 5-2. *Relationship between the BOOK and the TAG tables*

Collections of Basic Types

- @CollectionTable
- @MapKeyColumn (*Goncalves Ex24*)
  - Useful for storing Map of basic types such as <Integer,String>

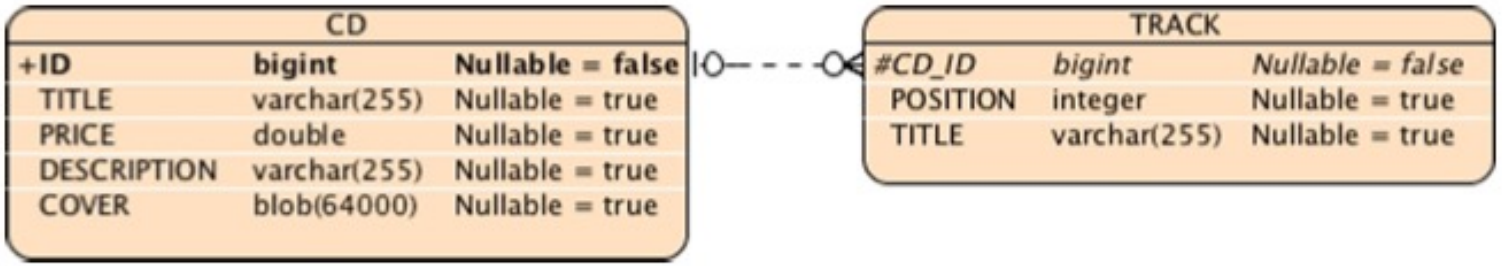


Figure 5-3. *Relationship between the CD and the TRACK tables*

Relationship Mapping

- Entities have relationships with other entities
- Direction or Navigation
  - Unidirectional
  - Bidirectional
- Multiplicity or Cardinality
  - One-to-one
  - One-to-many
  - Many-to-one
  - Many-to-many

Unidirectional

- Has only an owning side
- In this example, you can only access customer addresses via the Customer entity (the owner)

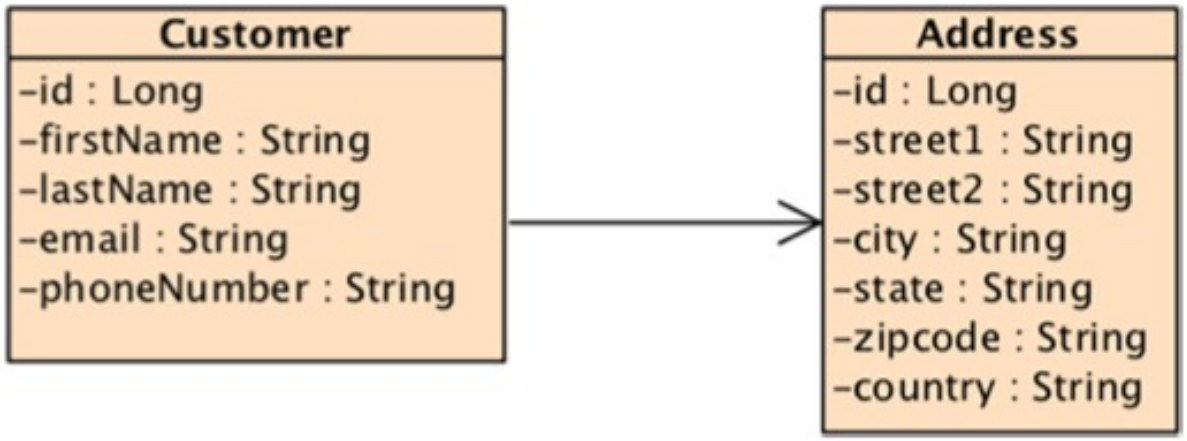


Figure 5-10. A unidirectional association between Customer and Address

Bidirectional

- Has both an owning and an inverse side
- In this example, you can access addresses from the customer entity, but you can also access customers from the address entity

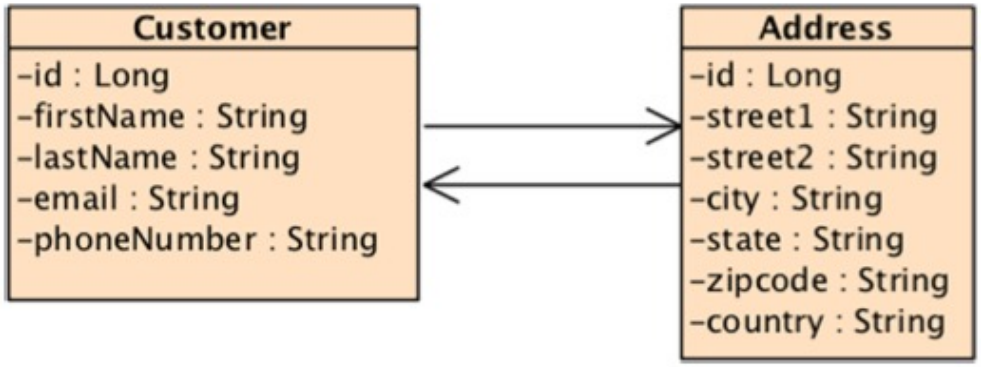


Figure 5-12. *A bidirectional association represented with two arrows*

One-to-One

- Each entity instance is related to a single instance of another entity.

One-to-One Unidirectional

- @OneToOne
  - Annotation goes on owning side
  - FK is on owning entity's table
  - Use @JoinColumn on owner to customize FK
    - *Goncalves Ex34 (CBE)*
    - *Goncalves Ex39 (with @JoinColumn)*

One-to-One Bidirectional

- @OneToOne
  - Annotation goes on both sides
  - **Inverse** side uses mappedBy to point to owner
  - FK is on **owning** entity's table
  - Use @JoinColumn on **owner** to customize FK
  - *Application must manage both sides of the relationship*
    - Can be done programmatically
    - Can be done through setter methods

One-to-Many

- An entity instance can be related to multiple instances of the other entities.

One-to-Many Unidirectional

- @JoinTable (*Goncalves Ex43)*
  - name
  - joinColumns
  - inverseJoinColumns

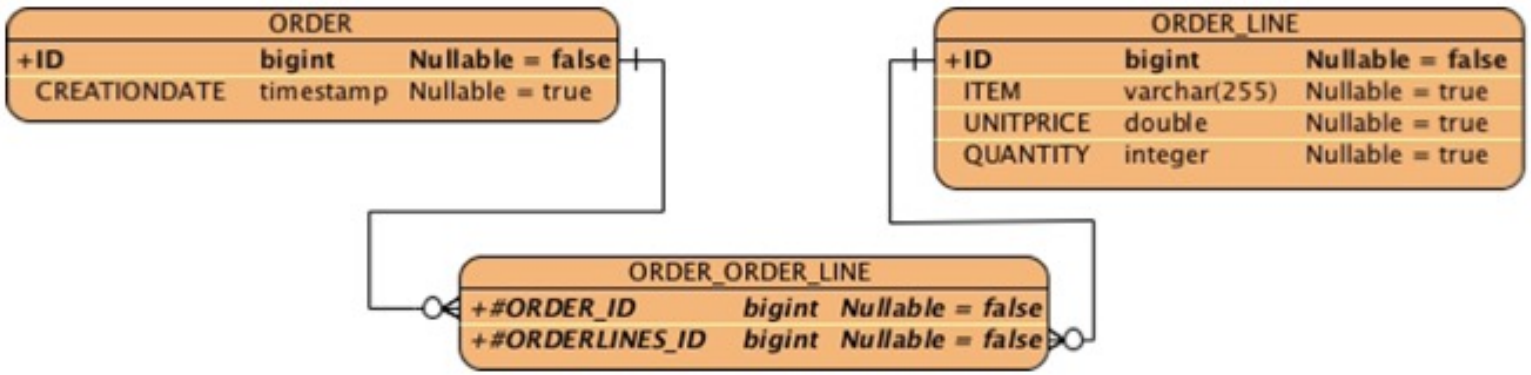


Figure 5-16. *Join table between ORDER and ORDER_LINE*

One-to-Many Unidirectional

- @OneToMany
  - Annotation goes on owning side
  - Defaults to join table
  - Use @JoinTable on owner to customize FKs in join table
    - joinColumns
    - inverseJoinColumns

Many-to-One

- Multiple instances of an entity can be related to a single instance of the other entity. This multiplicity is the opposite of a one-to-many relationship.

Many-to-One Unidirectional

- @JoinColumn

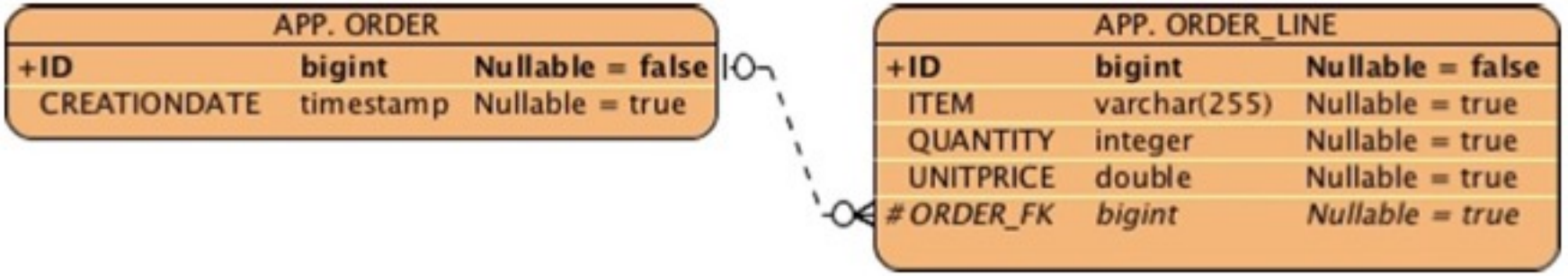


Figure 5-17. *Join column between Order and Order_Line*

Many-to-One Unidirectional

- @ManyToOne
  - Annotation goes on owning side (always the many side for ManyToOne)
  - Defaults to join column (FK on the many side)
  - Use @JoinColumn on owner to customize FK

Many-to-One Bidirectional

- @ManyToOne and @OneToMany
  - Annotation goes on both sides
  - **Inverse** (OneToMany) side uses mappedBy to point to owner
  - Defaults to join column (FK on many side)
  - Use @JoinColumn on owner to customize FK
  - *Application must manage both sides of the relationship*
    - Can be done through setter and add methods

Many-to-Many

- The entity instances can be related to multiple instances of each other.

Many-to-Many Bidirectional

- @ManyToMany
  - *Goncalves Ex46*
  - @JoinTable with joinColumns and inverseJoinColumns, and mappedBy on inverse

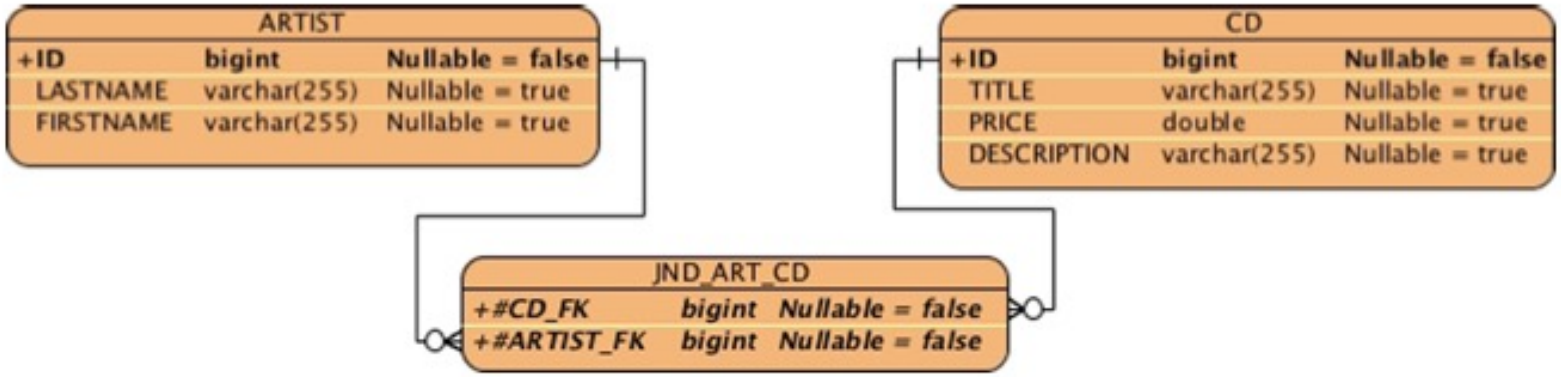


Figure 5-18. *Artist, CD, and the join table*

Many-to-Many Bidirectional

- @ManyToMany
  - Annotation goes on both sides
  - Use @JoinTable on owner to customize FKs in join table
    - joinColumns
    - inverseJoinColumns
  - Inverse side uses mappedBy to point to owner
  - *Application must manage both sides of the relationship*
    - Can be done through add methods – remember these are collections

Relationship Mapping Tips

- Navigating relationships on a detached entity will cause exceptions
- Application bears the responsibility of maintaining relationships between objects
  - We can use setters and add methods
- Don’t forget to initialize collections!
- Table/Entity with FK is the owner

Bidirectional Relationship Rules

- The inverse side of a bidirectional relationship must refer to its owning side by using the mappedBy element of the @OneToOne, @OneToMany, or @ManyToMany annotation. The mappedBy element designates the property or field in the entity that is the owner of the relationship.
- The many side of many-to-one bidirectional relationships must not define the mappedBy element. The many side is always the owning side of the relationship.
- For one-to-one bidirectional relationships, the owning side corresponds to the side that contains the corresponding foreign key.
- For many-to-many bidirectional relationships, either side may be the owning side.

Ordering Relationships

- @OrderBy (*Goncalves Ex49*)
  - No effect on database schema
- @OrderColumn (*Goncalves Ex51*)
  - Impacts database schema

Inheritance Mappings

- SINGLE_TABLE strategy
  - A single table per **class hierarchy**
  - @DiscriminatorValue (*Goncalves Ch5 Ex56*)
- JOINED strategy (*Goncalves Ch5 Ex59*)
- TABLE_PER_CLASS strategy (*Ch5 Ex60*)
  - *This strategy is not required by the JPA specification. Avoid for more portable code.*

Inheritance Hierarchy

- Entities can inherit from
  - Entity superclasses (as we have just seen with inheritance mapping)
  - Nonentity superclasses (*Goncalves Ch5 Ex63*)
- @MappedSuperclass (*Goncalves Ch5 Ex66*)
  - Contain persistent state and mappings, but are not entities

Cascading

Cascade Type	Description
PERSIST	Cascades persist operations to the target of the association
REMOVE	Cascades remove operations to the target of the association
MERGE	Cascades merge operations to the target of the association
REFRESH	Cascades refresh operations to the target of the association
DETACH	Cascades detach operations to the target of the association
ALL	Declares that all the previous operations should be cascaded

Concurrency

- Versioning
  - @Version read-only attribute
  - Not required, but recommended if the entity can be modified concurrently by more than one process or thread
  - Automatically enables optimistic locking
- Locking
  - Optimistic – Obtain lock before commit
  - Pessimistic – Very resource intensive
- *Goncalves Chapter 6 Example 36*

Optimistic Locking

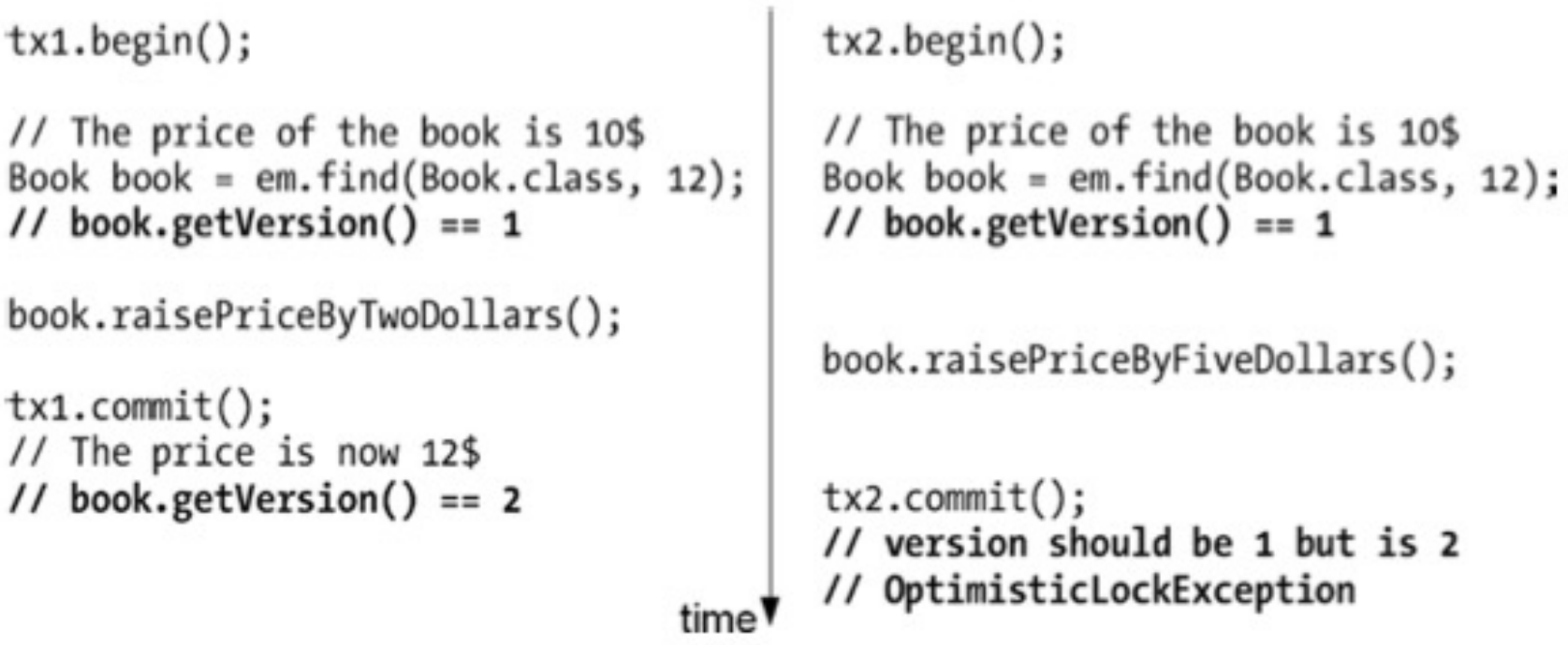


Figure 6-5. *OptimisticLockException thrown on transaction tx2*

Callbacks

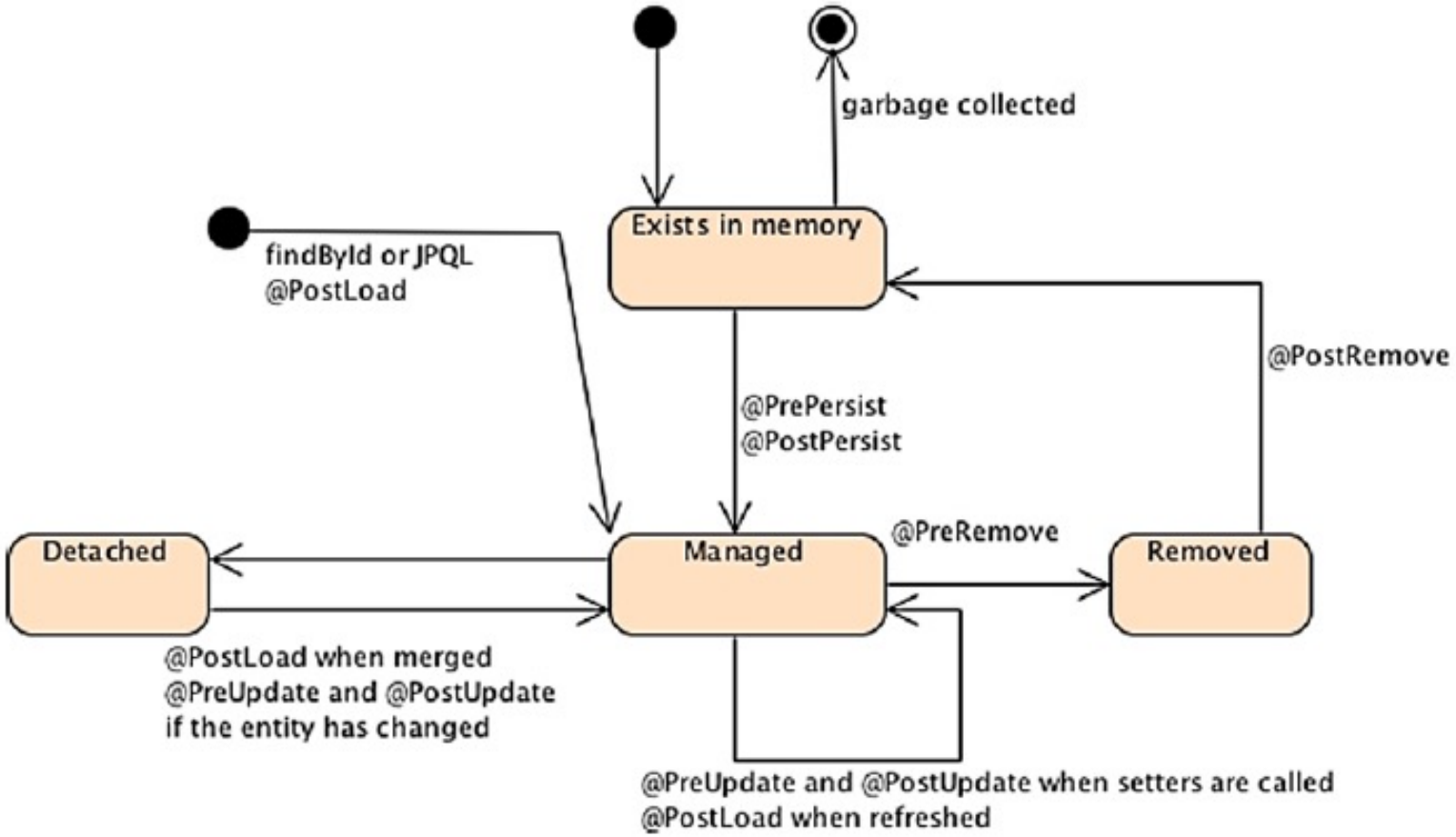


Figure 6-7. Entity life cycle with callback annotations

Callbacks

Annotation	Description
@PrePersist	Marks a method to be invoked before EntityManager.persist() is executed.
@PostPersist	Marks a method to be invoked after the entity has been persisted. If the entity autogenerates its primary key (with @GeneratedValue), the value is available in the method.
@PreUpdate	Marks a method to be invoked before a database update operation is performed (calling the entity setters or the EntityManager.merge() method).
@PostUpdate	Marks a method to be invoked after a database update operation is performed.
@PreRemove	Marks a method to be invoked before EntityManager.remove() is executed.
@PostRemove	Marks a method to be invoked after the entity has been removed.
@PostLoad	Marks a method to be invoked after an entity is loaded (with a JPQL query or an EntityManager.find()) or refreshed from the underlying database

Entity Listeners

- Can be used to extract the business logic to a class for re-use among multiple entities
- *Goncalves Chapter 6 Examples 39 and 42*

Callback Methods

- Work well when you have business logic only related to that entity.
- *Goncalves Chapter 6 Example 38*

Sources Used

- The Jakarta EE Tutorial. Retrieved Aug 24, 2020, from https://eclipse-ee4j.github.io/jakartaee-tutorial/toc.html
- Juneau, J. (2020). Jakarta EE 8 Recipes. New York, NY: Apress.
- Goncalves, A. (2013). Beginning Java EE 7. New York, NY: Apress.
- Some slides adapted with permission from Marty Hall (www.coreservlets.com – JSP and Servlets)