Entity Manager and Persistence Context

cphbusiness

- Entities are managed by the entity manager, which is represented by javax.persistence.EntityManager instances.
- Each EntityManager instance is associated with a **persistence context**: a set of managed entity instances that exist in a particular data store.
- A **persistence context** defines the scope under which particular entity instances are created, persisted, and removed.
- The EntityManager interface defines the methods that are used to interact with the persistence context.

The Entity Manager

cphbusiness

The entity manager is a (perhaps THE) central piece in JPA.

- It manages the state and life cycle of entities as well as querying entities within a persistence context.
- It is responsible for creating and removing persistent entity instances and finding entities by their primary key.
- It can lock entities for protecting against concurrent access by using optimistic or pessimistic locking.
- It can use JPQL queries to retrieve entities following certain criteria.

The Entity Manager

cphbusiness

```
Book2 book = new Book2();
book.setDescription("..");
 //...
EntityManagerFactory emf;
emf = Persistence.createEntityManagerFactory("pu-x");
EntityManager em = emf.createEntityManager();
Try{
  em.getTransaction().begin();
  em.persist(book);
  em.getTransaction().commit();
}
finally{
  em.close();
}

em.remove(book);
```

Just a POJO

Entities can be used as regular objects by different layers of an application

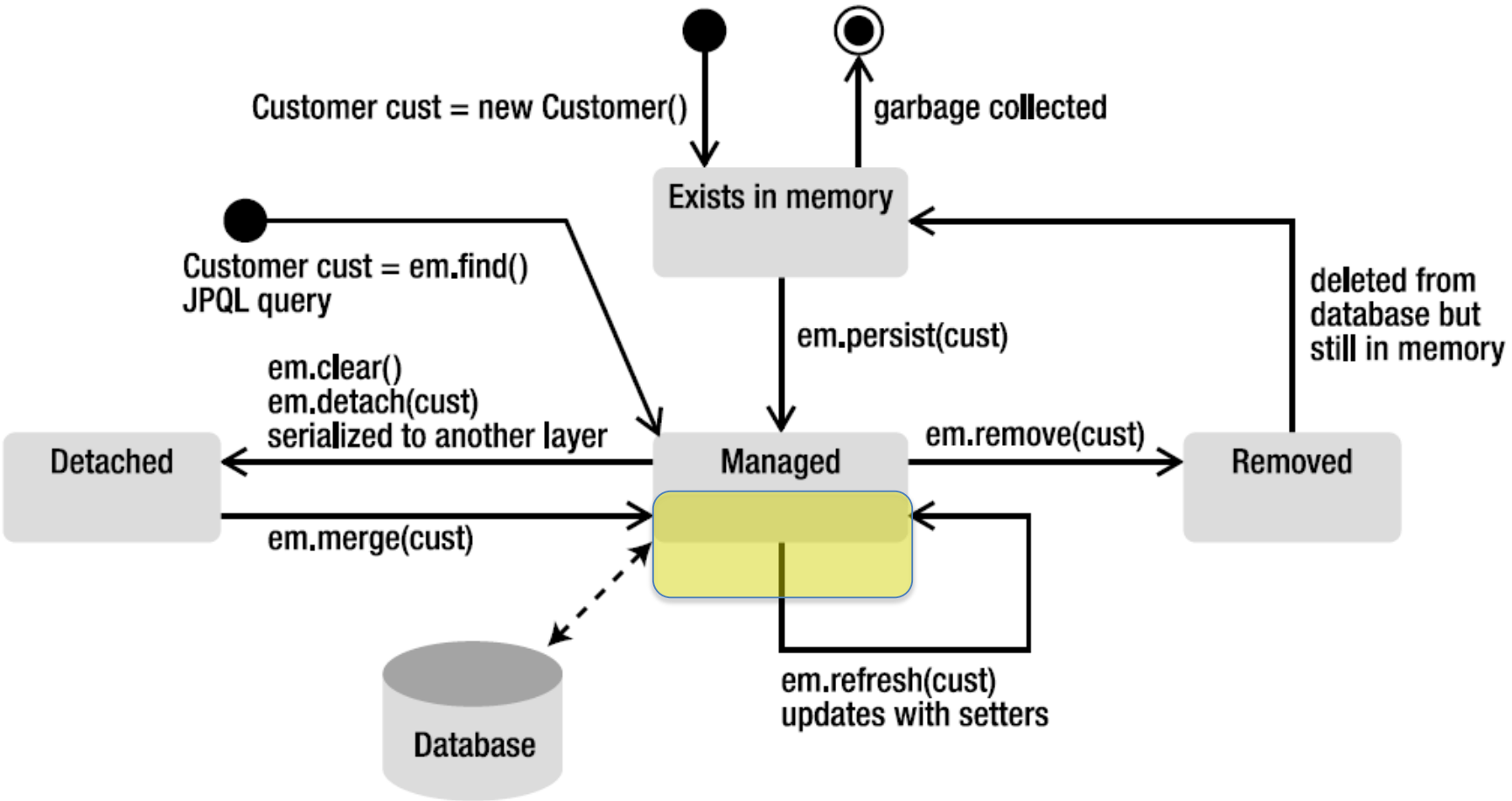
and become managed by the entity manager when we need to load or insert data into the database

Now the book is **Managed**

Again, just a POJO (**detached**)

Entity Life Cycle

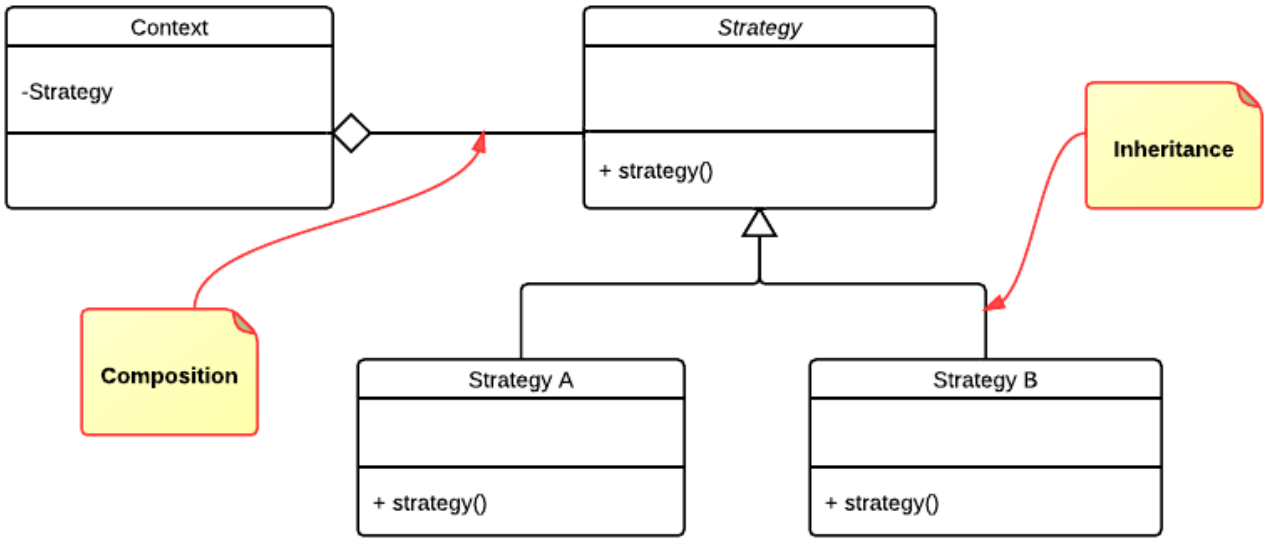
cphbusiness



Relationships

cphbusiness

In OO there are two ways to ”connect” classes. **Composition** (almost always the right choice) and **Inheritance**



In the following we will see how these two strategies can be mapped to a DataBase, using JPA

All Possible Cardinality-Direction Combinations

cphbusiness

Use the links below, for detailed information related to each type

Cardinality	Direction
One-to-one details	Unidirectional
One-to-one	Bidirectional
One-to-many details	Unidirectional
One-to-many	Bidirectional
Many-to-one details	Unidirectional
Many-to-one	Bidirectional
Many-to-many details	Unidirectional
Many-to-many	Bidirectional

We will investigate this in details in todays exercises

Bidirectional relationships

cphbusiness

Rules that applies to bidirectional relationships:

The inverse side of a bidirectional relationship must refer to its owning side by use of 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.

```
public class Customer .. {
…
  @OneToMany(mappedBy = "customer")
  private List<Address> addresses = new ArrayList();
```

```
@Entity
public class Address ..{
  private static final long serialVersionUID = 1L;
  ..
  @ManyToOne
  private Customer customer;
```

Side with the Foreign Key

Bidirectional relationships the *mappedBy* element

cphbusiness

The **mappedBy** element designates the property or field in the entity that is the owner of the relationship.

- The many side of one-to-many / many-to-one bidirectional relationships must be the owning side, hence the mappedBy element cannot be specified on the ManyToOne annotation.
- 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

Relationships - Lazy Fetching

cphbusiness

The cost of retrieving and building an object's relationships far exceeds the cost of selecting the object

The solution to this issue is **lazy fetching** (lazy loading). Lazy fetching allows the fetching of a relationship to be deferred until it is accessed

Lazy fetching involves some *magic* in the JPA provider to transparently fault in the relationships as they are accessed.

```
@OneToOne(fetch =FetchType.
@JoinColumn(name="ADDR_ID")
private Address address;
```

EAGER FetchType
LAZY FetchType

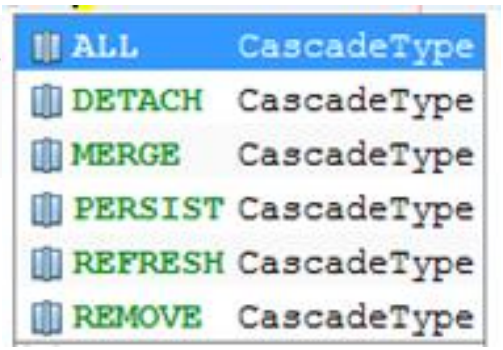
https://en.wikibooks.org/wiki/Java_Persistence/Relationships#Lazy_Fetching

Relationships - Cascading

cphbusiness

Relationship mappings have a cascade option that allows the relationship to be cascaded for common operations.
Cascade is normally used to model dependent relationships, such as Order -> OrderLine.
Cascading the orderLines relationship allows for the Order's -> OrderLines to be persisted, removed, merged along with their parent.

```
@OneToOne(cascade={CascadeType.
@JoinColumn(name="ADDR_ID")
private Address address;
```



https://en.wikibooks.org/wiki/Java_Persistence/Relationships#Cascading