Managing Persistent Object in JPA

JPA Slides #4

Container Managed Persistence

We use **container managed persistence** i.e. all JPA code in EJBs

- I.e. container will insert transactions where needed
- Very many customizations options for transactions in JPA, not covered

Other possibility

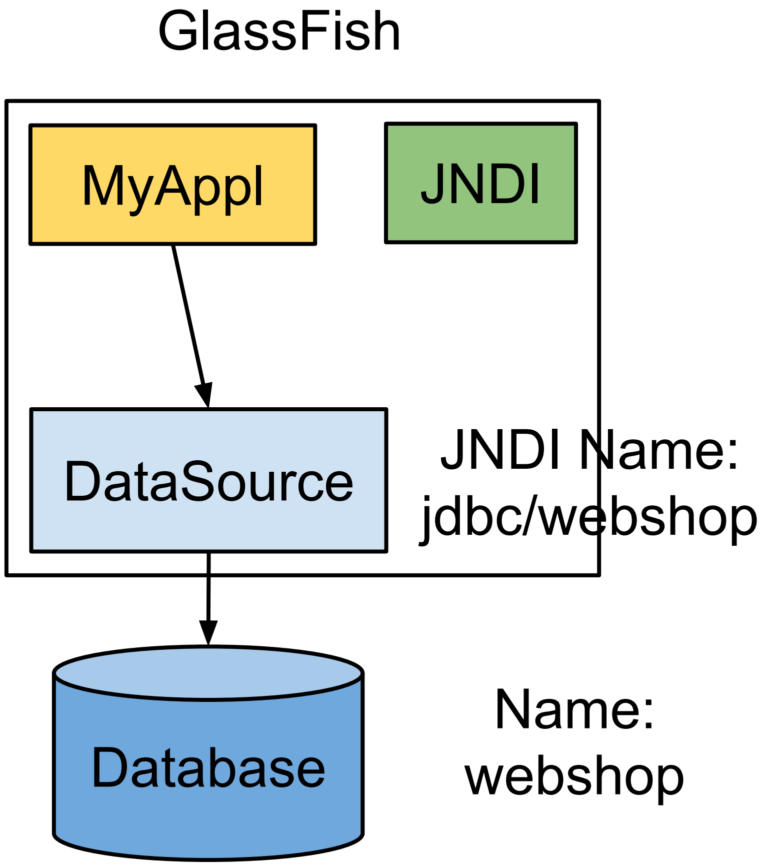
- **Application managed persistence**, application handles everything (must use if non container environment, JSE, specifically JUnit)
- Avoid

Container Managed Environment

Application in container

DataSource object (defined in glassfish-resources.xml). Created outside application. Has a JNDI name

JNDI, Java Naming and Directory Interface, key/value-store (imagine: global Tree<String, Object>). String is the name of the object.



Persistence Context

"A set of managed entity instances in which for any persistent entity identity [primary key] there is a unique entity instance. Within the persistence context, the entity instances and their life cycle are managed by the **EntityManager (EM)**"

- PC is a a first level cache of entities handled by the EM

I.e a table row (unique by primary key) will be represented as a unique in memory object in PC (identity shall use ==)

- Every PC is associated with a PU (the list of entity classes, …)
- Every EM handles a single PC

EntityManager API

Basic API to handle entity classes
- javax.persistence.EntityManager

The CRUD operations
- em.persist(object)   // Note: After this call object has id
- em.find(… primaryKey)
- em.merge(object) // update and more …
- em.remove(object)
- …
- Injection of EntityManager into EJBs

```
@PersistenceContext("PU name")   // Name optional if only one PU
private EntityManager em;
```

Types of Entity Managers

Two kinds of

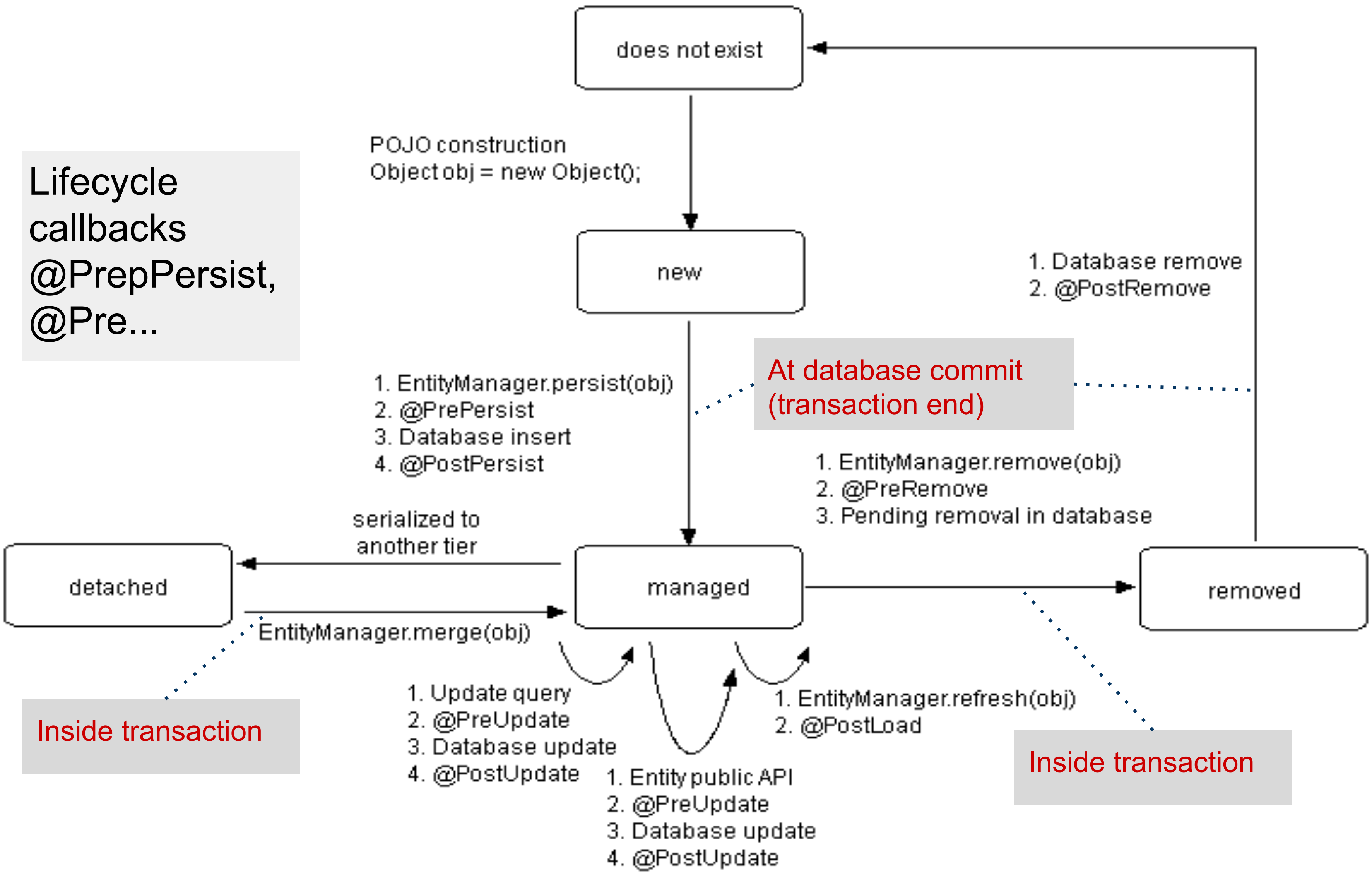
- **Container Managed Entity Managers** (container manages life cycle of EM), we use!
  - **Transaction-Scoped:** PC managed by EM follows the transaction, transaction committed EM gone (Use: **stateless** environment)
  - **Extended:** Single EM bound to life cycle of some **stateful** managed object
  - Both use JTA transaction (Java EE server transactions).
  - A somewhat old but very detailed presentation

- **Application Managed** Entity Managers not used
  - Retrieved by use of EntityManagerFactory not used

Entity Instance Life Cycle

Entity Instance (@Entity) may be
- **New**, no persistent identity
- **Managed**, has identity in PC, will be synchronized with real database at next commit
- **Detached** (not managed), has identity but not in PC will **not** be synchronized with database
- **Removed**, has identity in PC but will be removed from real database at next commit

Entity Instance Life Cycle cont.



Lifecycle callbacks @PrepPersist, @Pre...

does not exist

POJO construction
Object obj = new Object();

new

1. Database remove
2. @PostRemove

At database commit (transaction end)

1. EntityManager.persist(obj)
2. @PrePersist
3. Database insert
4. @PostPersist

1. EntityManager.remove(obj)
2. @PreRemove
3. Pending removal in database

serialized to another tier

detached

managed

removed

EntityManager.merge(obj)

1. Update query
2. @PreUpdate
3. Database update
4. @PostUpdate

1. Entity public API
2. @PreUpdate
3. Database update
4. @PostUpdate

1. EntityManager.refresh(obj)
2. @PostLoad

Inside transaction

Inside transaction

Code: Entity Lifecycle

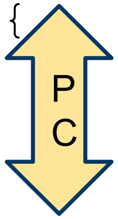
```
@Stateless
public class ShoppingCartBean
    @PersistenceContext
    EntityManager em

    public OrderLine createOrderLine( Product p, Order o ){
        OrderLine ol = new OrderLine(p,o);
        em.persist( ol) ;
        return ol;
    }
}
```

This is transaction scoped EM , more to come ...

New Entity

Managed Entity (has id)

Detached Entity



Code: Detached Entities

```
@Stateless
public class ShoppingCartBean
    @PersistenceContext
    EntityManager em

    public OrderLine createOrderLine( Product p, Order o ){
        OrderLine ol = new OrderLine(p,o);
        em.persist(ol) ;
        return ol;
    }

    public OrderLine updateOrderLine( OrderLine ol ){
        OrderLine ol2 = em.merge( ol) ;
        ...
        return ol2;
    }
}
```

P
C

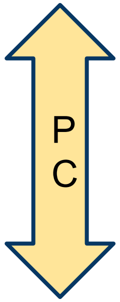
Detached

P
C

Managed

Code: Scope of Identity

```
@Stateless
public class ShoppingCartBean
    @PersistenceContext
    EntityManager em

    public OrderLine createOrderLine( Product p, Order o ){
        OrderLine ol = new OrderLine(p,o);
        em.persist( ol) ;
        OrderLine ol2 = em.find(OrderLine.class, ol.getId());
        // ol == ol2 is TRUE
        return ol;
    }
}
```



Retrieval returns same instance

More on Detach and Refresh

Entities detached at

- transaction commit (after last line in method as already seen)
- transaction rollback (exception)
- explicitly detaching the entity

```
em.flush()  // Must do before...
em.detach(p) // ...this
```

- Clearing the PC: em.clear()
- Detaching object: em.detach(...);
- Closing the EM: em.close()
- Passing by value (serializing)

Refresh

```
// State refreshed from database
em.refresh(p); // p managed (before and after)
```

More on Merging

Merge will create new managed copy or state copied

```
// p detached
p1 = em.merge(p)     // p1 != p !!! New object
p = em.merge(p)      // Could look like this, tricky...

// Also possible, skip returntype
em.merge(p)  // Any modification on p inside transaction persisted
```

Possible to check if managed

```
     /* True if
      *  p retrieved with em.find(), em.getRefrence()
      *  em.persist(p) called or
      *  persist cascaded
      */
    em.contains(p)
```

Gotcha's

Some possible

- Must keep track if detached or not
- LazyInitializationException. Combination: Lazy fetching/detached object. Object is detached, trying to access non fetched parts, Exception!

Surviving...

What's happening when...

- Persist an already managed instance?
- Merge a removed instance?
- Detach a new instance?
- Refresh a detached instance?
- ...

JPA typical behavior

- If possible harm: Exception
- If harmless: Nothing happens
- Have to find out ...

Automatic PC Propagation

<u>Automatic PC propagation</u>, all EJBs involved use the same PC (EJBs calling EJBs)

- Other benefit of using EJB/Container
- If no propagation have to pass EM around (extra method parameter (JSE)
- ... many, many, details...

Transaction Scoped EM/PC

PC follows transaction (normally you use this)

- Bean will check for propagated PC, if so use it

```
@Stateless
public class SomeClass {
    @PersistenceContext
    private EntityManager em;  // Injected

    // Method starts transaction (if needed) ...
    public void someMethod( ... ){
       // Transaction start, PC created (if no propagated)
       // PC follows transaction
       em.persist( object );
    }  // .. and commits, PC gone
}
```

Extended EM/PC

PC follow life cycle of session bean.

- A transaction may span multiple method calls. <u>This is special</u>

```
@Stateful
// PC life cycle tied to bean life
cycle                                   public class SomeClass {
   @PersistenceContext(unitName = "nameOfMyPU",
                         type=PersistenceContextType.EXTENDED)
   private EntityManager em;
   public void someMethod( ... ){
        em.persist( object );     // Same PC here as ...
    }
   public void someOtherMethod( ... ){
        em.persist( object );    // .. here
   }
}
```

Bean Managed Transaction

Possible need transactions outside EJBs (i.e. tests)
- No automatic transaction handling or context propagation

```
// Outside EJB object but container can inject using CDI
@Inject
UserTransaction utx; // Object representing the transaction

private void clearAll() throws Exception {
    utx.begin();
    // Get an entitymanager from somewhere
    EntityManager em = ....getEntityManager();
    em.joinTransaction();
    //Order matters
    em.createQuery("delete from Publication").executeUpdate();
    em.createQuery("delete from Author").executeUpdate();
    em.createQuery("delete from Book").executeUpdate();
    utx.commit();
}
```

Entity Life Cycle Callbacks

Can annotate methods to be called during life object cycle

- @PrePersist, @PostPersist,...
- Bean validation triggered just before @PrePersist, @PreUpdate
- Method (logic) only related to the annotated entity (single class)
- Signature for callback method; void anyName()
- Possible customization: Separate listener class

Cascading

Should storing, deleting, etc. apply to associated objects?

- If car deleted probably engine should be deleted (also in database)! A cascade will automate the process
- If no cascade: Have to persist each single objects in correct order
- Adding a cascade

```
@OneToOne( cascade = { CascadeType.REMOVE })
private Engine e;
```

- Cascade types: ALL, PERSIST, REMOVE, MERGE, REFRESH, CLEAR, ALL
- Cascade may require bidirectional mappings