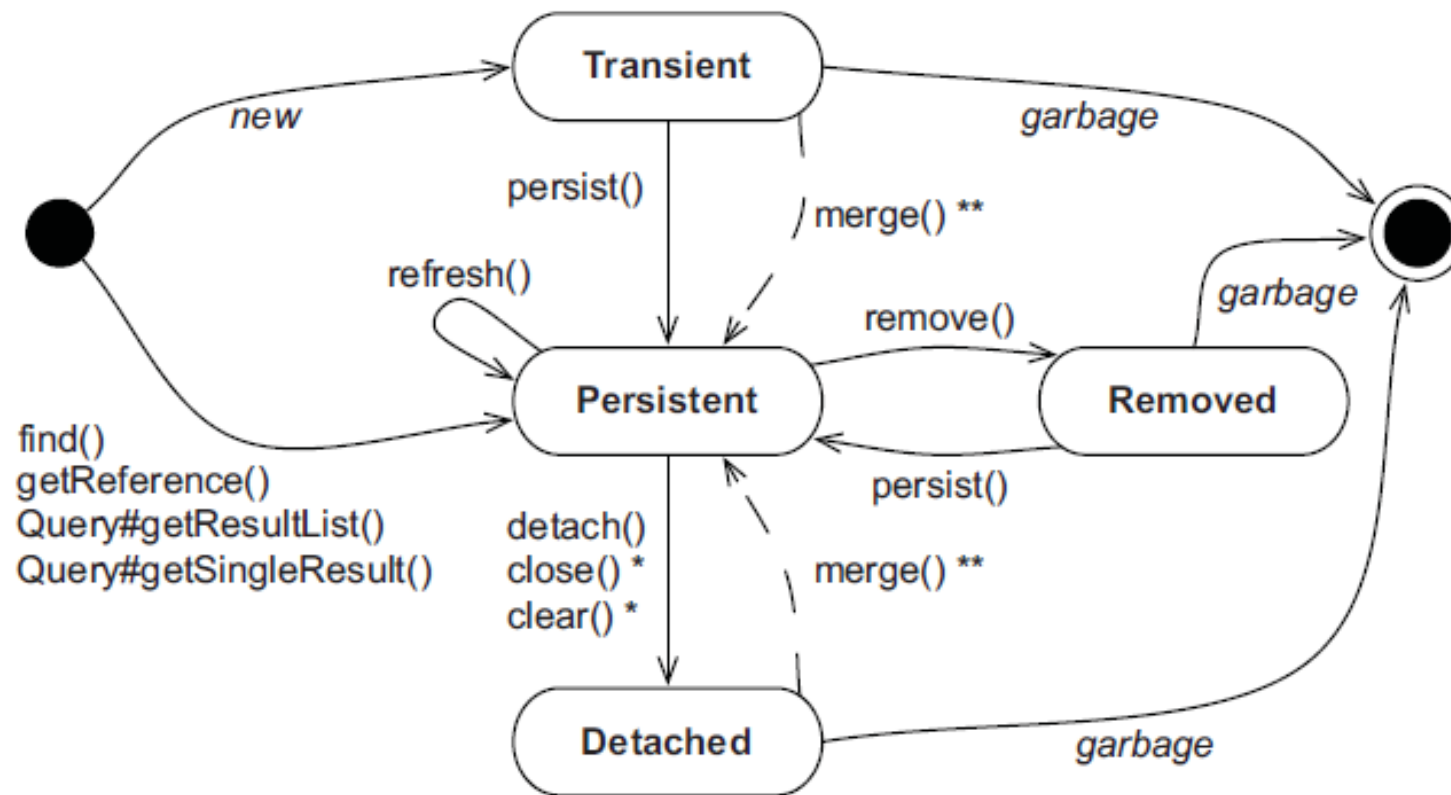


# Gestione dei dati

- Ciclo di vita della persistenza (*persistence life cycle*): gli stati attraversati da un'entità persistente
- *Unit of work*: insieme di operazioni che possono cambiare lo stato di un'entità considerate atomicamente come un unico gruppo
- Contesto di persistenza (*persistence context*): servizio che memorizza tutte le modifiche e i cambiamenti di stato effettuati in una particolare *unit of work*

# Ciclo di vita di un'istanza persistente



\* Affects all instances in the persistence context

\*\* Merging returns a persistent instance, original doesn't change state

## Persistence context

- Inizia con `EntityManagerFactory#createEntityManager()` e finisce con `EntityManagerFactory#close()`: l'applicazione definisce lo *scope* del *persistence context*, demarcando la *unit of work*
- Monitora e gestisce tutte le entità nello stato persistente
- Propaga al database lo stato delle istanze in memoria (*flushing*)
- Agisce come *cache* di primo livello
- Garantisce che un'unica istanza rappresenti un particolare record del db (*persistence scope identity*):  
$$\text{entityA} == \text{entityB} \iff \text{entityA.getId().equals(entityB.getId())}$$

# Unit of work

Singola unit of work in un singolo thread

```
EntityManager em = null;
UserTransaction tx = TM.getUserTransaction();
try {
    tx.begin();
    em = JPA.createEntityManager();           ← Application-managed

    // ...

    tx.commit();                             ← Synchronizes/flushes persistence context
} catch (Exception ex) {
    // Transaction rollback, exception handling
    // ...
} finally {
    if (em != null && em.isOpen())
        em.close();                         ← You create it, you close it!
}
```

# Rendere i dati persistenti

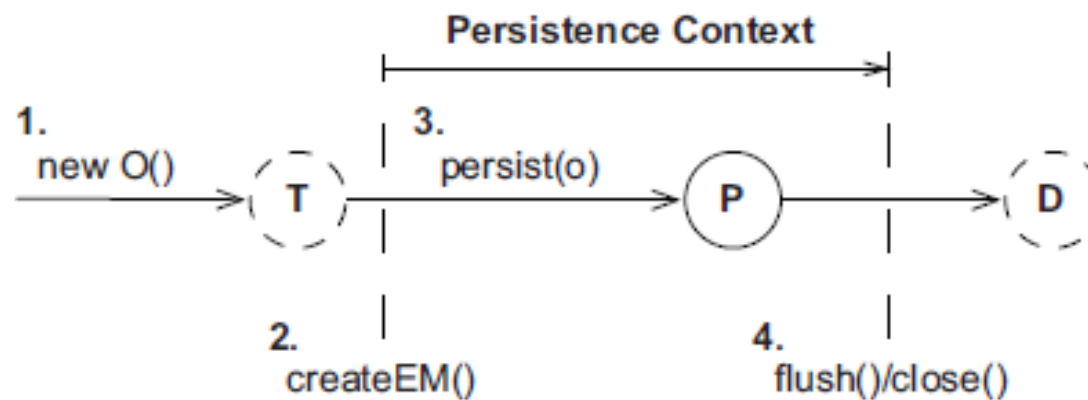
```
Item item = new Item();  
item.setName("Some Item");
```

← Item#name is NOT NULL.

```
em.persist(item);
```

```
Long ITEM_ID = item.getId();
```

← Has been assigned

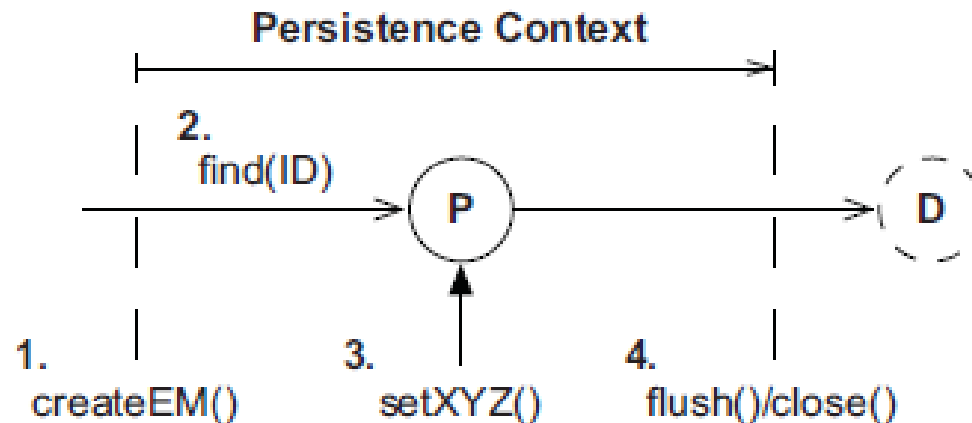


# Recuperare e modificare i dati persistenti

```
Item item = em.find(Item.class, ITEM_ID);  
if (item != null)  
    item.setName("New Name");
```

← Hits database if not already in persistence context

← Modify



# Recuperare e modificare i dati persistenti

```
Item itemA = em.find(Item.class, ITEM_ID);  
Item itemB = em.find(Item.class, ITEM_ID);  
  
assertTrue(itemA == itemB);  
assertTrue(itemA.equals(itemB));  
assertTrue(itemA.getId().equals(itemB.getId()));
```

← Repeatabl read

# Recuperare e modificare i dati persistenti

Con `getReference()` non si interroga il db, non si ottiene un'istanza totalmente inizializzata (solo l'id), ma un suo proxy (sempre che l'oggetto non si trovi già in cache...)

```
Item item = em.getReference(Item.class, ITEM_ID);  
  
PersistenceUnitUtil persistenceUtil =  
    JPA.getEntityManagerFactory().getPersistenceUnitUtil();  
assertFalse(persistenceUtil.isLoaded(item));  
  
// assertEquals(item.getName(), "Some Item");  
// Hibernate.initialize(item);  
  
tx.commit();  
em.close();  
  
assertEquals(item.getName(), "Some Item");
```

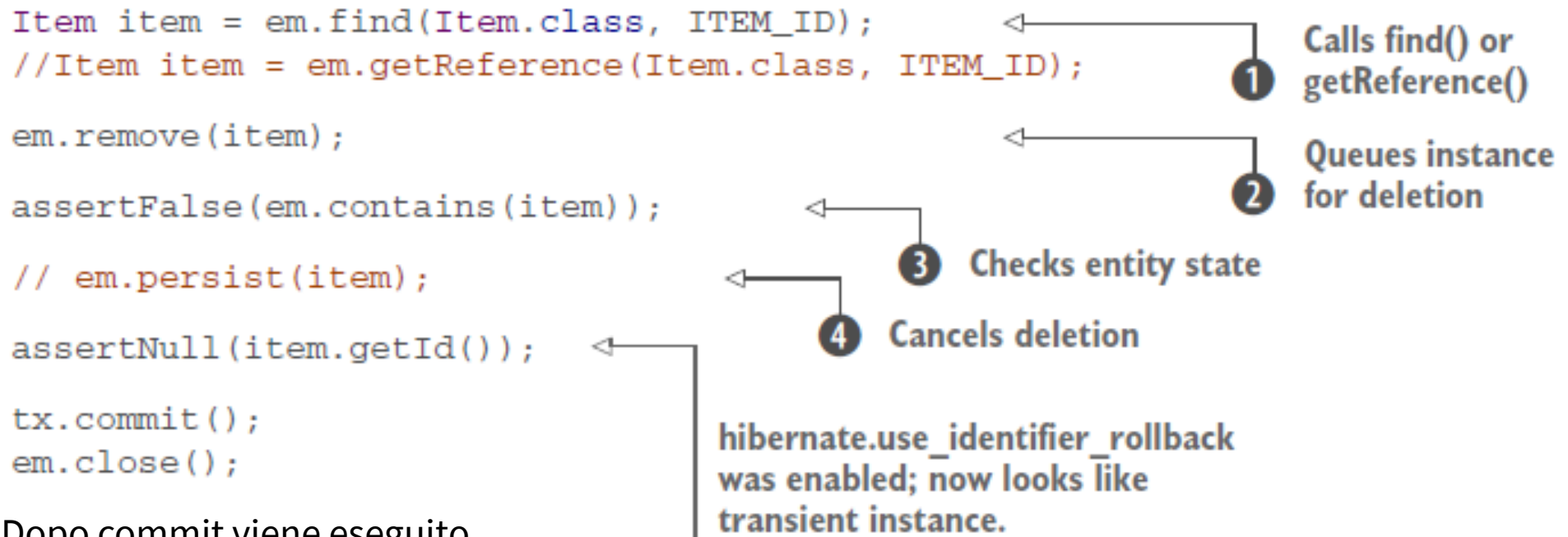
The diagram illustrates the sequence of operations for the provided code:

- 1** `getReference()`: Points to the first line of code.
- 2** **Helper method**: Points to the `JPA.getEntityManagerFactory().getPersistenceUnitUtil();` line.
- 3** **Initializes proxy**: Points to the `assertFalse(persistenceUtil.isLoaded(item));` line.
- 4** **Loads proxy data**: Points to the `tx.commit();` and `em.close();` lines.
- 5** **item in detached state**: Points to the final `assertEquals(item.getName(), "Some Item");` line.

Se quando il proxy viene inizializzato, il record non c'è più, viene lanciata una `EntityNotFoundException`



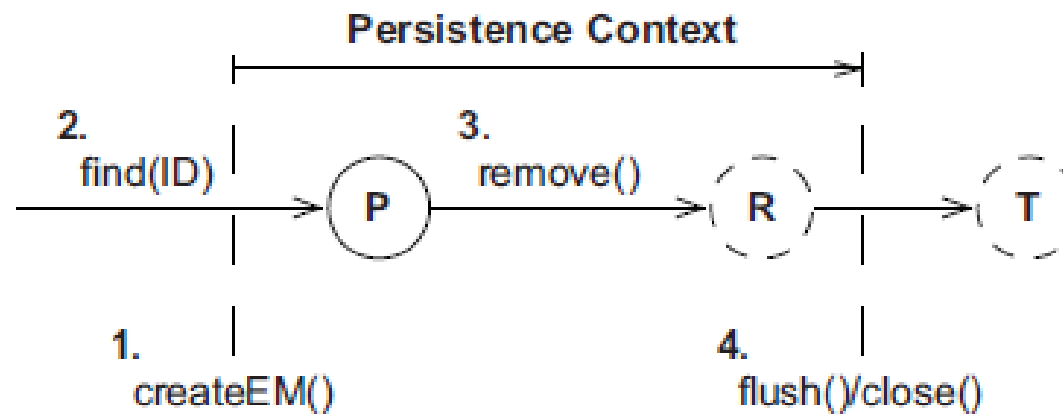
# Rendere il dato transitorio (cancellazione da db)



Dopo commit viene eseguito  
Il delete e il garbage  
collector cancella item

Property di persistence.xml per resettare id dopo remove e avere un  
oggetto in stato transitorio da salvare ancora in un nuovo contesto di  
persistenza

# Rendere il dato transitorio (cancellazione da db)



# Refresh dei dati

```
Item item = em.find(Item.class, ITEM_ID);  
item.setName("Some Name");  
  
// Someone updates this row in the database  
  
String oldName = item.getName();  
em.refresh(item);  
assertNotEquals(item.getName(), oldName);
```

Utile con contesti di persistenza estesi, che comprendono diversi cicli di request/response

# Replicazione dei dati

```
tx.begin();

EntityManager emA = getDatabaseA().createEntityManager();
Item item = emA.find(Item.class, ITEM_ID);

EntityManager emB = getDatabaseB().createEntityManager();
emB.unwrap(Session.class)
    .replicate(item, org.hibernate.ReplicationMode.LATEST_VERSION);

tx.commit();
emA.close();
emB.close();
```

# Caching nel persistent context

- Non caricare istanze di entità in una unit of work se non si ha necessità di modificarle: si può incorrere in una `OutOfMemoryException`!
- `EntityManager#detach(i)` per staccare una singola istanza
- `EntityManager#clear()` svuota l'intero persistent context

# Caching nel persistent context

Per rendere il persistence context read-only (disabilita il dirty checking)

```
em.unwrap(Session.class).setDefaultReadOnly(true);  
  
Item item = em.find(Item.class, ITEM_ID);  
item.setName("New Name");  
  
em.flush();           ← No UPDATE
```

# Caching nel persistent context

Per disabilitare il dirty checking per una singola istanza

```
Item item = em.find(Item.class, ITEM_ID);  
em.unwrap(Session.class).setReadOnly(item, true);  
item.setName("New Name");  
em.flush();
```

← No UPDATE

# Flushing (sincronizzazione) dei dati

- Al commit
- Prima dell'esecuzione di una query con `javax.persistence.Query` (o della corrispettiva API di Hibernate)
- Quando si invoca `flush()`

Si può controllare settando il `FlushModeType` di `EntityManager`



# Flushing (sincronizzazione) dei dati

```
tx.begin();
EntityManager em = JPA.createEntityManager();

Item item = em.find(Item.class, ITEM_ID);
item.setName("New Name");

em.setFlushMode(FlushModeType.COMMIT);

assertEquals(
    em.createQuery("select i.name from Item i where i.id = :id")
        .setParameter("id", ITEM_ID).getSingleResult(),
    "Original Name"
);

tx.commit();
em.close();
```

1 Loads Item instance

2 Changes instance name

Disables flushing before queries

Gets instance name 3

Flush!

La sincronizzazione avviene solo dopo il commit, all'interno del persistent Context vedo due nomi diversi salvati nella stessa istanza ``

# Lo stato *detached* (identità)

```
tx.begin();  
em = JPA.createEntityManager();
```

← 1 Creates persistent context

← 2 Loads entity instances

```
Item a = em.find(Item.class, ITEM_ID);  
Item b = em.find(Item.class, ITEM_ID);  
assertTrue(a == b);  
assertTrue(a.equals(b));  
assertEquals(a.getId(), b.getId());
```

3 a and b have same Java identity

← 4 a and b are equal

← 5 a and b have the same  
database identity

```
tx.commit();  
em.close();
```

← 7 Closes persistence  
context

```
tx.begin();  
em = JPA.createEntityManager();
```

```
Item c = em.find(Item.class, ITEM_ID);  
assertTrue(a != c);  
assertFalse(a.equals(c));  
assertEquals(a.getId(), c.getId());
```

8 a and c aren't identical

← 9 a and c aren't equal

```
tx.commit();  
em.close();
```

10 Identity test still true


## Lo stato *detached* (identità)

Le due istanze a e c non sono identiche! Sono state caricate in due contesti di persistenza diversi... (sono uguali per il db)

Questo può portare problemi se trattate come uguali istanze nello stato detached:

```
em.close();  
  
Set<Item> allItems = new HashSet<>();  
allItems.add(a);  
allItems.add(b);  
allItems.add(c);  
assertEquals(allItems.size(), 2);
```

That seems wrong  
and arbitrary.



Dovrebbe essere 1!

## Lo stato *detached* (identità)

Tutte le volte che si utilizzano entity nello stato detached e serve verificarne l'identità tra loro (di solito nelle collections basate sull'hash), bisogna fornirle dell'opportuna implementazione di `equals()` e di `hashCode()`

# Lo stato *detached* (identità)

```
@Entity
@Table(name = "USERS",
      uniqueConstraints =
        @UniqueConstraint(columnNames = "USERNAME"))
public class User {

    @Override
    public boolean equals(Object other) {
        if (this == other) return true;
        if (other == null) return false;
        if (!(other instanceof User)) return false;
        User that = (User) other;
        return
            this.getUsername().equals(that.getUsername());
    }

    @Override
    public int hashCode() {
        return getUsername().hashCode();
    }

    // ...
}
```

← Use instanceof.

← Use getters.

# Lo stato *detached*

Per rendere detached un'istanza (molte applicazioni leggono e fanno il rendering dei dati dopo la chiusura del persistent context):

```
User user = em.find(User.class, USER_ID);  
em.detach(user);  
assertFalse(em.contains(user));
```

## Lo stato *detached* (merge)

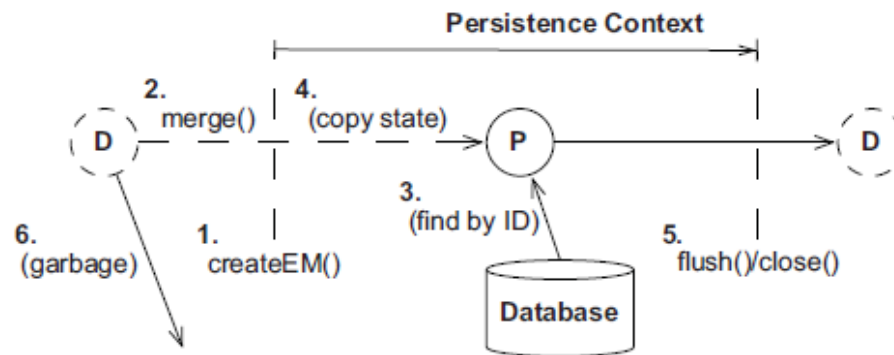
Abbiamo caricato da db un'istanza di user in un precedente persistent context, vogliamo modificarlo e salvare queste modifiche:

```
detachedUser.setUsername("johndoe");  
  
tx.begin();  
em = JPA.createEntityManager();  
User mergedUser = em.merge(detachedUser);  
mergedUser.setUsername("doejohn");  
  
tx.commit();  
em.close();
```

Discard detachedUser  
reference after merging.  
mergedUser is in  
persistent state.

← UPDATE in database

# Lo stato *detached* (merge)



`merge()` prima controlla se nel persistent context c'è una istanza persistente con lo stesso id se non la trova, la carica dal db e copia su questa l'istanza detached (*detachedUser* viene data al garbage collector e si può continuare a modificare *mergedUser*, verrà fatto un UPDATE)

se non la trova neanche nel db, istanzia un nuovo *User*, copia su questa l'istanza detached e al momento opportuno fa un INSERT. Lo stesso comportamento di `merge()` si ha passandogli un'istanza non detached, ma transient (priva quindi di un id): si può fare a meno di `persist()`, notare che adesso *detachedUser* non referencia più l'istanza iniziale!

Per cancellare un'istanza detached, prima `merge()`, poi `remove()` sull'istanza ritornata



# Transazioni

Le *unit of works* sono delimitate dalle transazioni, che servono a gestire la concorrenza

Le transazioni avvengono a livello di db e di sistema

Devono soddisfare le proprietà ACID:

- **Atomicity:** le operazioni in una transazioni agiscono atomicamente (se una fallisce si ha il rollback)
- **Consistency:** non devono violare le regole d'integrità del db (not null, unique...)
- **Isolation:** ciascuna transazione non è visibile alle altre
- **Durability:** i cambiamenti effettuati dalla transazione devono rimanere

Ovviamente l'applicazione deve poi garantire la correttezza della transazione

# Transazioni

La demarcazione delle transazioni può avvenire tramite:

- Le API di JDBC (setAutoCommit(false) sulla Connession, poi commit() e rollback()) → in ambito Java SE
- Le API di un Transaction Manager (sono standardizzate da JTA). La principale interfaccia da usare è *UserTransaction*

# Exception handling

```
EntityManager em = null;
UserTransaction tx = TM.getUserTransaction();
try {
    tx.begin();
    em = JPA.createEntityManager();
    // ...

    tx.commit();
} catch (Exception ex) {
    try {
        if (tx.getStatus() == Status.STATUS_ACTIVE
            || tx.getStatus() == Status.STATUS_MARKED_ROLLBACK)
            tx.rollback();
    } catch (Exception rbEx) {
        System.err.println("Rollback of transaction failed, trace follows!");
        rbEx.printStackTrace(System.err);
    }
    throw new RuntimeException(ex);
} finally {
    if (em != null && em.isOpen())
        em.close();
}
```

← Application-managed

← Synchronizes/flushes persistence context

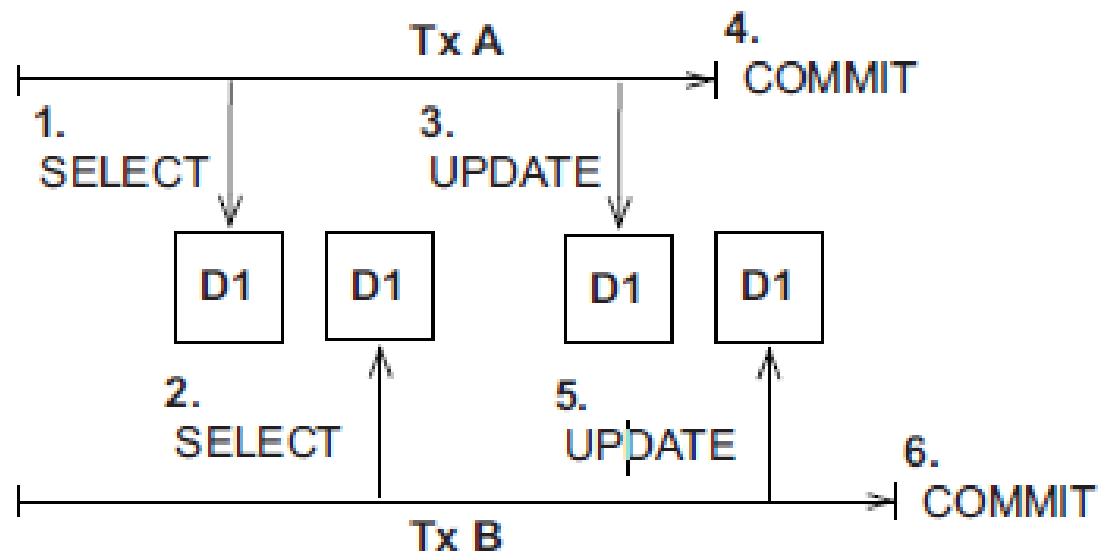
← Transaction rollback; exception handling

← Per poter loggare cmq l'eccezione che ha causato il rollback

← You create it, you close it!

# Concorrenza ottimistica

“L’ultimo commit vince”



# Concorrenza ottimistica

```
@Entity
public class Item implements Serializable {

    @Version
    protected long version;

    // ...
}

tx.begin();
em = JPA.createEntityManager();

Item item = em.find(Item.class, ITEM_ID);
// select * from ITEM where ID = ?

assertEquals(item.getVersion(), 0);

item.setName("New Name");

em.flush();
// update ITEM set NAME = ?, VERSION = 1 where ID = ? and VERSION = 0
```

← 1 Retrieves by identifier

← 2 Instance version: 0

← 3 Flushes persistence context

Se version non è più a 0... viene lanciata OptimisticLockException



# Dynamic data filters

Supponiamo che ogni utente possa fare offerte solo su oggetti venduti da altri utenti che abbiano un rank uguale o minore, vogliamo quindi che un utente loggato veda solo alcuni oggetti in vendita

```
@Entity
@Table(name = "USERS")
public class User {

    @NotNull
    protected int rank = 0;

    // ...
}
```

# Dynamic data filters

## Definizione di un filtro dinamico

```
@org.hibernate.annotations.FilterDef(  
    name = "limitByUserRank",  
    parameters = {  
        @org.hibernate.annotations.ParamDef(  
            name = "currentUserRank", type = "int"  
        )  
    }  
)
```

# Dynamic data filters

## Applicazione del filtro all'entity

```
@Entity
@org.hibernate.annotations.Filter(
    name = "limitByUserRank",
    condition =
        ":currentUserRank >= (" +
            "select u.RANK from USERS u " +
            "where u.ID = SELLER_ID" +
        ") "
)
public class Item {

    // ...
}
```



# Dynamic data filters

Il filtro va infine abilitato e parametrizzato per una particolare unit of work:

```
org.hibernate.Filter filter = em.unwrap(Session.class)
    .enableFilter("limitByUserRank");

filter.setParameter("currentUserRank", 0);
```

# Dynamic data filters

Si utilizza con le query JPQL o con le query create con *CriteriaQuery*

```
List<Item> items = em.createQuery("select i from Item i").getResultList();  
// select * from ITEM where 0 >=  
// (select u.RANK from USERS u where u.ID = SELLER_ID)
```

# Dynamic data filters

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery criteria = cb.createQuery();
criteria.select(criteria.from(Item.class));
List<Item> items = em.createQuery(criteria).getResultList();
// select * from ITEM where 0 >=
// (select u.RANK from USERS u where u.ID = SELLER_ID)
```

Non funzionano con `em.find(Item.class, ITEM_ID)`, perché non è detto che acceda al db con una query... potrebbe usare la cache del persistent context; non si possono filtrare le associazioni many-to-one o one-to-one (potrebbe cambiarne la natura), ma le collection sì (eseguono una query...)

# Dynamic data filters

Applicazione del filtro alla collection

```
@Entity
public class Category {

    @OneToMany(mappedBy = "category")

    @org.hibernate.annotations.Filter(
        name = "limitByUserRank",
        condition =
            ":currentUserRank >= (" +
                "select u.RANK from USERS u " +
                "where u.ID = SELLER_ID" +
            ") "
    )
    protected Set<Item> items = new HashSet<Item>();

    // ...
}
```

# Dynamic data filters

Utilizzo del filtro

```
filter.setParameter("currentUserRank", 0);  
Category category = em.find(Category.class, CATEGORY_ID);  
assertEquals(category.getItems().size(), 1);
```

# Query

Ci sono 3 fasi:

- Creazione della query
- Preparazione della query (binding degli argomenti, opzioni di paging)
- Esecuzione della query e lettura dei risultati

# Creazione delle query

- JPA rappresenta una query con *javax.persistence.Query* o *javax.persistence.TypedQuery*
- Le query si creano con *EntityManager#createQuery()* e le sue varianti
- Le query possono essere scritte con JPQL, costruite con CriteriaBuilder oppure in SQL

# Creazione delle query

JPQL:

```
Query query = em.createQuery("select i from Item i");
```

Se cambio nome alle classi...

Criteria Builder e CriteriaQuery API:

```
CriteriaBuilder cb = em.getCriteriaBuilder();  
// Also available on EntityManagerFactory:  
// CriteriaBuilder cb = entityManagerFactory.getCriteriaBuilder();  
  
CriteriaQuery criteria = cb.createQuery();  
criteria.select(criteria.from(Item.class));  
  
Query query = em.createQuery(criteria);
```

È type-safe, adatto a creazione dinamiche delle query



# Creazione delle query

SQL:

```
Query query = em.createNativeQuery(  
    "select * from ITEM", Item.class  
);
```

Tramite `java.sql.ResultSet` crea una List di Item

# Preparazione delle query

Per il binding dei parametri si utilizza `setParameter()`:

```
String searchString = // ...  
  
Query query = em.createQuery(  
    "select i from Item i where i.name = :itemName"  
).setParameter("itemName", searchString);
```

```
Item someItem = // ...  
  
Query query = em.createQuery(  
    "select b from Bid b where b.item = :item"  
).setParameter("item", someItem);
```

# Esecuzione delle query

getResultList() ritorna una java.util.List:

```
Query query = em.createQuery("select i from Item i");  
List<Item> items = query.getResultList();
```

```
Query query = em.createQuery("select i.name from Item i");  
List<String> itemNames = query.getResultList();
```

# Esecuzione delle query

Per ottenere un singolo risultato:

```
TypedQuery<Item> query = em.createQuery(  
    "select i from Item i where i.id = :id", Item.class  
)  
.setParameter("id", ITEM_ID);  
  
Item item = query.getSingleResult();
```

```
TypedQuery<String> query = em.createQuery(  
    "select i.name from Item i where i.id = :id", String.class  
)  
.setParameter("id", ITEM_ID);  
  
String itemName = query.getSingleResult();
```

Usando l'interfaccia TypedQuery non devi fare il cast del risultato di getSingleResult

# Esecuzione delle query

GetSingleResult() se non trova risultati lancia un'eccezione! Non un semplice null...

```
try {
    TypedQuery<Item> query = em.createQuery(
        "select i from Item i where i.id = :id", Item.class
    ).setParameter("id", 12341);

    Item item = query.getSingleResult();
    // ...
} catch (NoResultException ex) {
    // ...
}
```