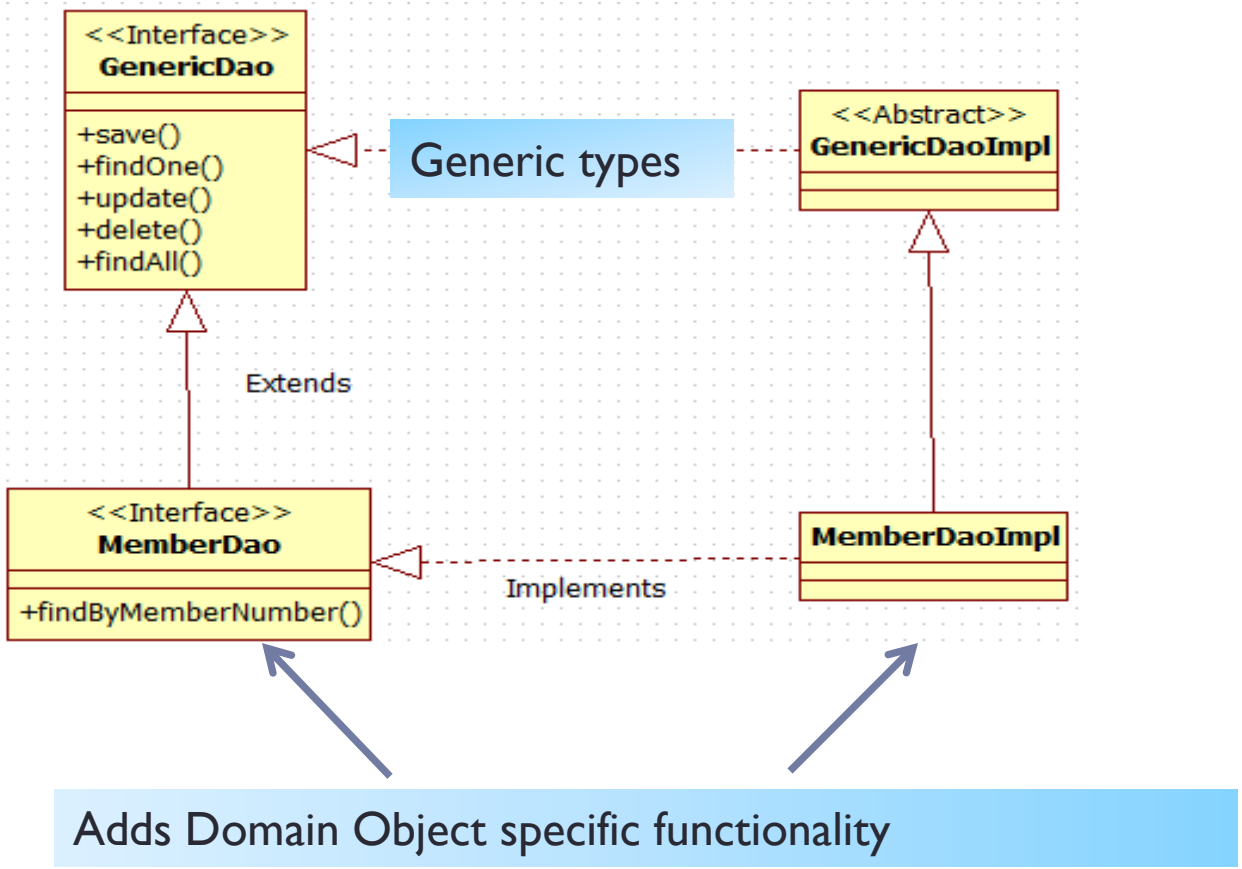
Persistence APIs

CRUD Services
Core J2EE DAO pattern

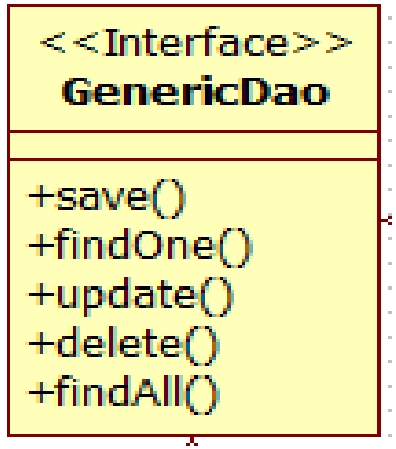
- **Data Access Object (DAO)**
  - Manage the connection with the data source
  - Abstract and encapsulate access to data source
  - Provides CRUD access:
    - Create
    - Read
    - Update
    - Delete
- Hides the data source implementation details
- Interface allows for different storage schemes
- Adapter between the client and the data source

“Classic” ORM GenericDAO



Generic DAO Interface

```
public interface GenericDao<T> {

        void save(T t);

        void delete(Long id);

        T findOne(Long id);

        T update(T t);

        List<T> findAll();
}
```



Generic DAO Implementation

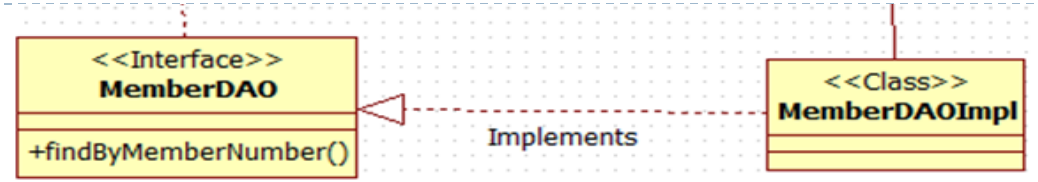
```
public abstract class GenericDaoImpl<T> implements GenericDao<T> {

        @PersistenceContext
        protected EntityManager entityManager;
        protected Class<T> daoType;
        public void setDaoType(Class<T> type) {
             daoType = type;
        }
         public void save( T entity ){
             entityManager.persist( entity );
          }
         public void delete( T entity ){
             entityManager.remove( entity );
          }
        public void delete(Long id) {
            T entity = findOne( id );
            delete( entity );
         }
        public T findOne( Long id ){
            return (T) entityManager.find( daoType, id );
         }
        public List<T> findAll(){
                return entityManager.createQuery( "from " + daoType.getName().getResultList();
            }
        public T update( T entity ){
             return entityManager.merge( entity );
          }
}
```

JPA implementation uses EntityManager

Hiberate implementation uses SessionFactory

```
@Autowired
    protected SessionFactory sessionFactory;
```

Domain Class specific DAO

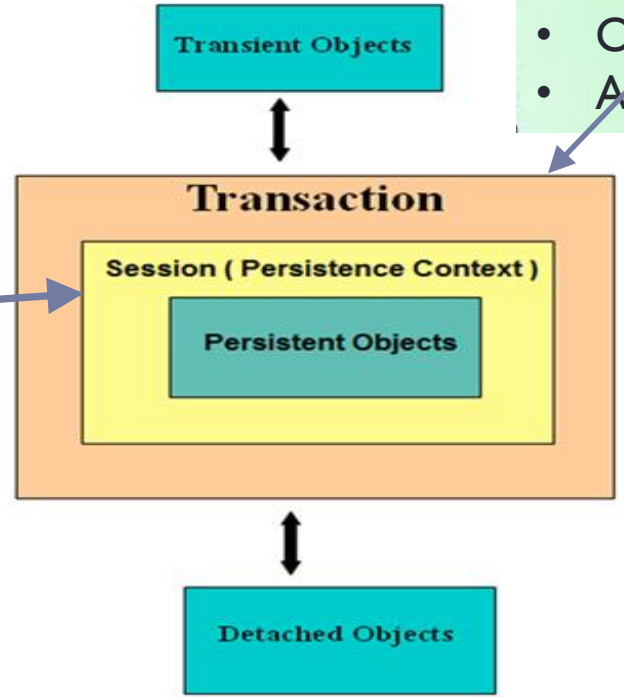



```
public interface MemberDao extends GenericDao<Member> {
        public Member findByMemberNumber(Integer number);
}
```

```
@Repository
public class MemberDaoImpl extends GenericDaoImpl<Member> implements MemberDao {

        public MemberDaoImpl() {
           super.setDaoType(Member.class );
           }

        public Member findByMemberNumber(Integer number) {

           Query query = entityManager.createQuery("select m from MEMBER m  where m.memberNumber
    =:number");
           return (Member) query.setParameter("number", number).getSingleResult();

        }
}
```

ORM – RDB Interactions

Transaction
- Logical unit of work
- One or more DB queries
- Atomic [All or None]



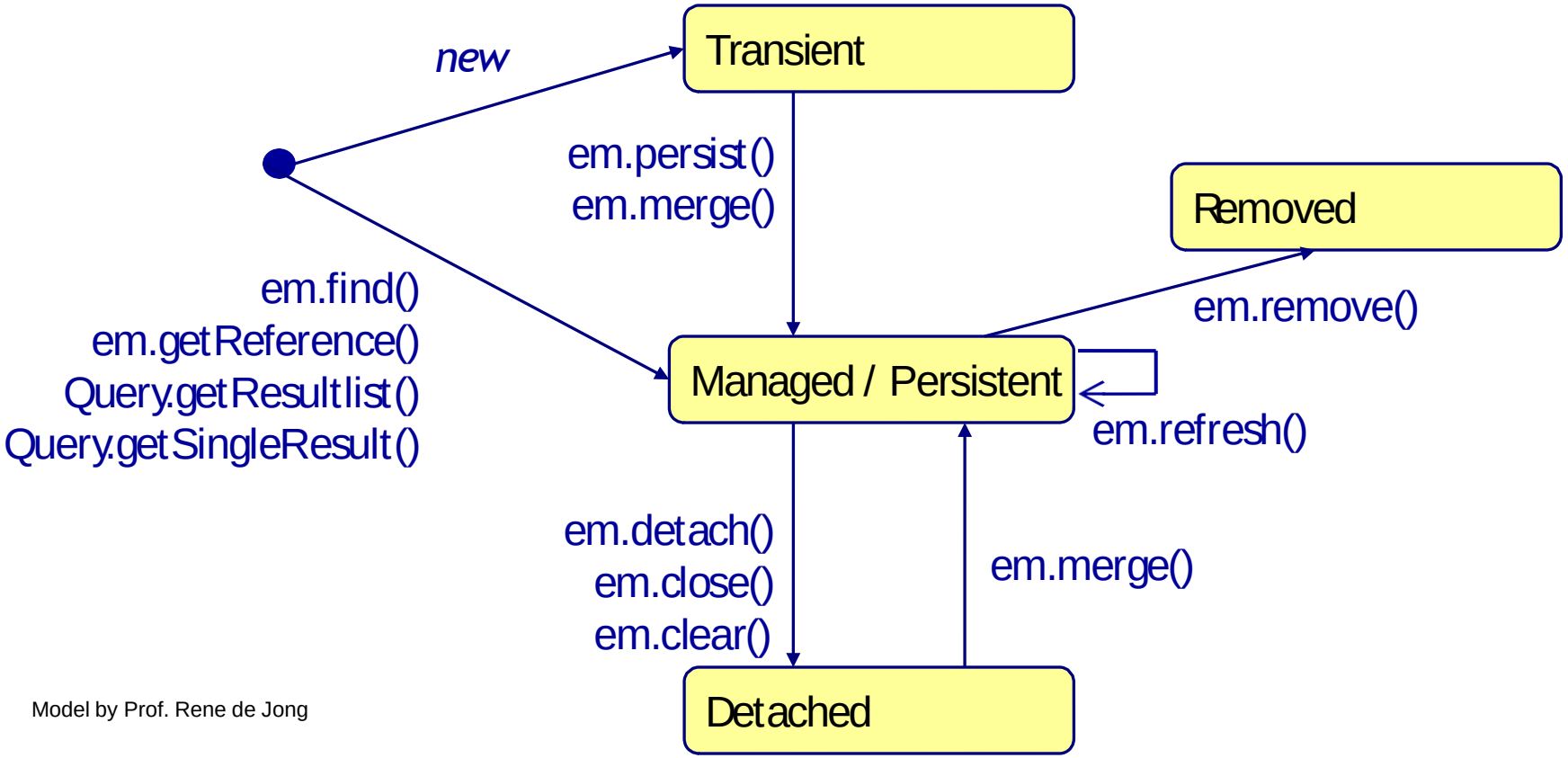
Persistence Context
- Establishes DB connection
- Holds DB-aware objects

Entity Life Cycle

- We've seen how to map entities
  - But haven't discussed the EntityManager
  - We first have to understand the Entity Life Cycle

- The Entity Life Cycle consists of the following 4 states:
  - Managed, Transient, Detached, Removed
  - Methods of the EntityManager change the state of Entities

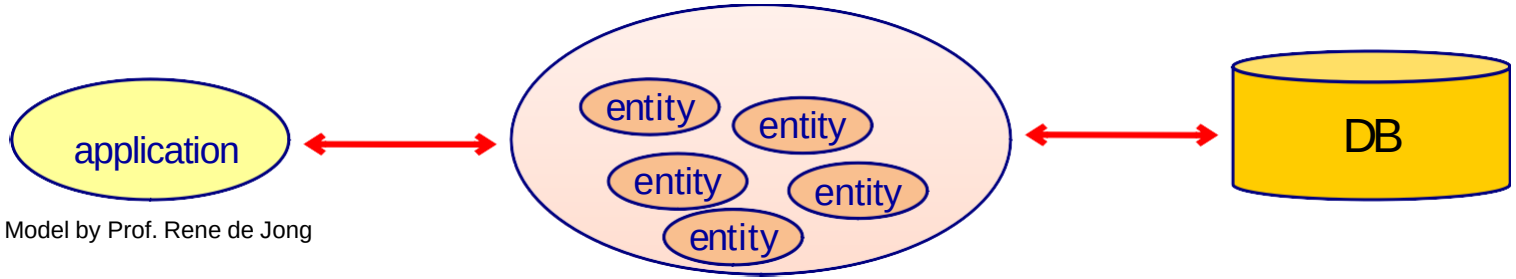
States and Methods



Model by Prof. Rene de Jong

Persistence Context

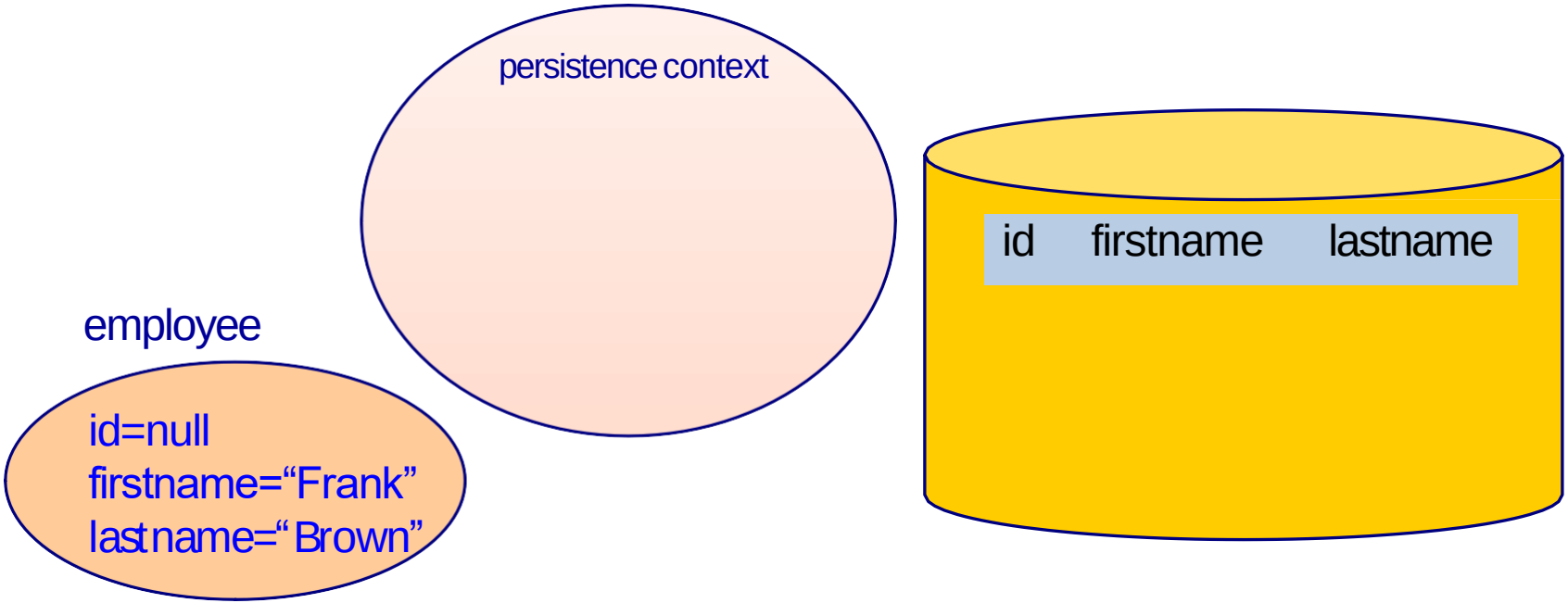
- PersistenceContext is another name for EntityManager
  - Uses same principles as Spring's ApplicationContext
- The PersistenceContext:
  - Guarantees managed entities are saved in DB
  - Tracks changes and pushes them to DB
  - Works like a cache



Model by Prof. Rene de Jong

Transient Entity

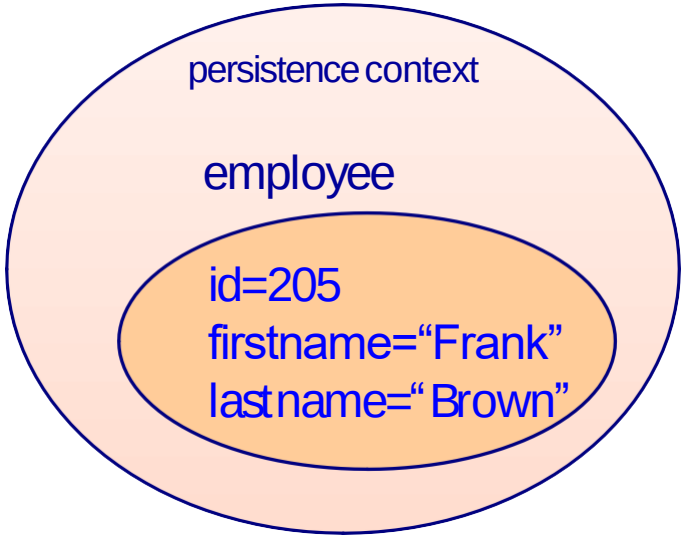
- A transient entity has no database identity

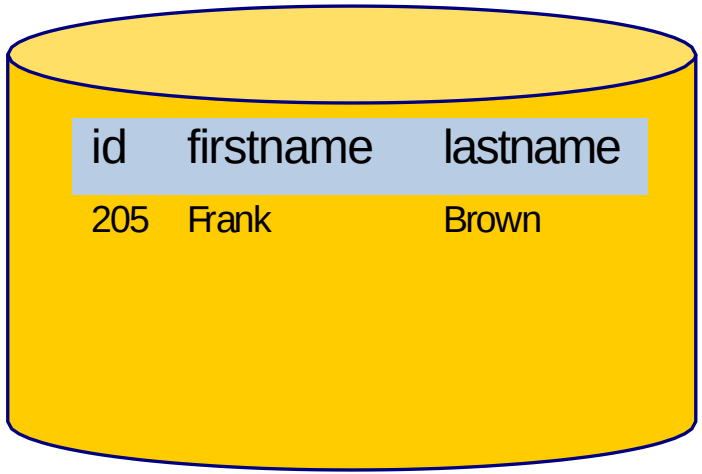


Model by Prof. Rene de Jong

Managed Entity

- A managed entity is managed by the persistence context (em) and has a DB identity

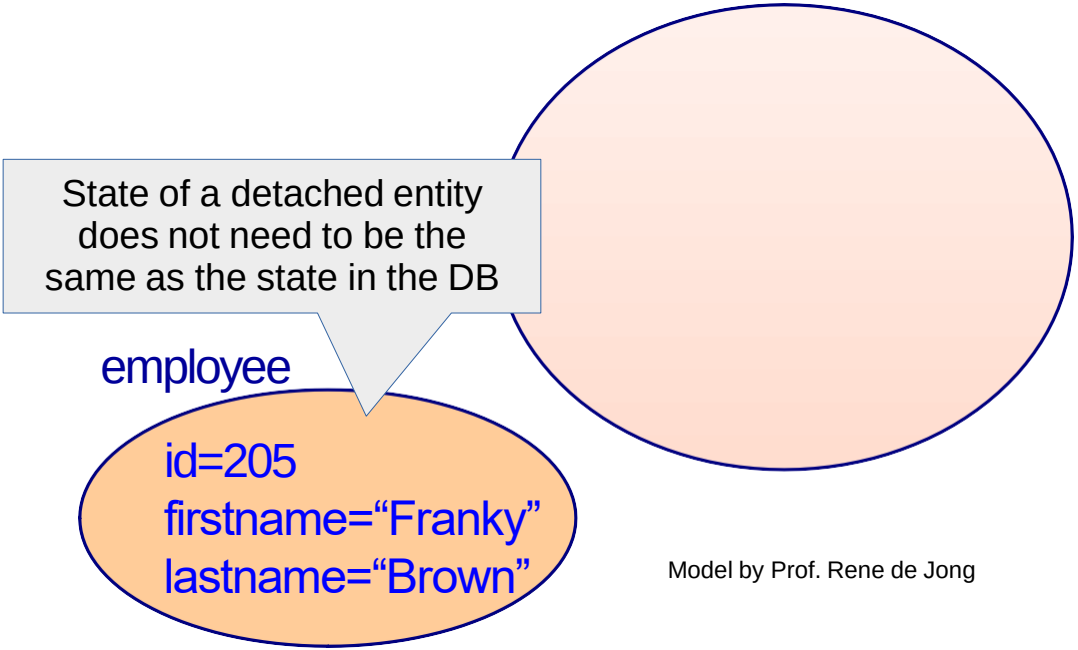




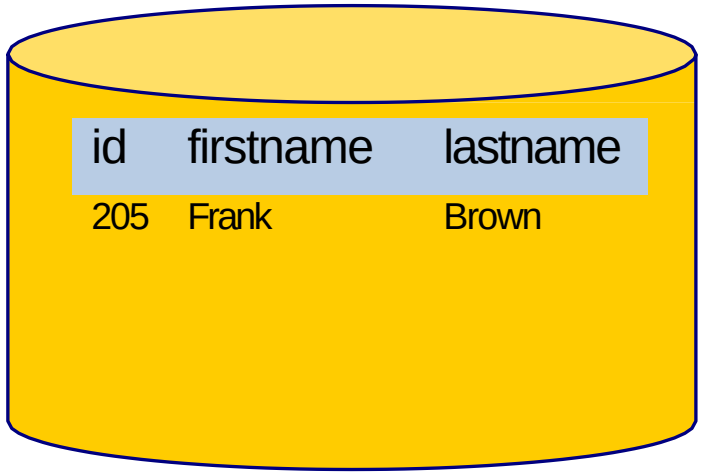
Model by Prof. Rene de Jong

Detached Entity

- A detached entity has a database identity
  - But is not managed by the current persistence ctx.

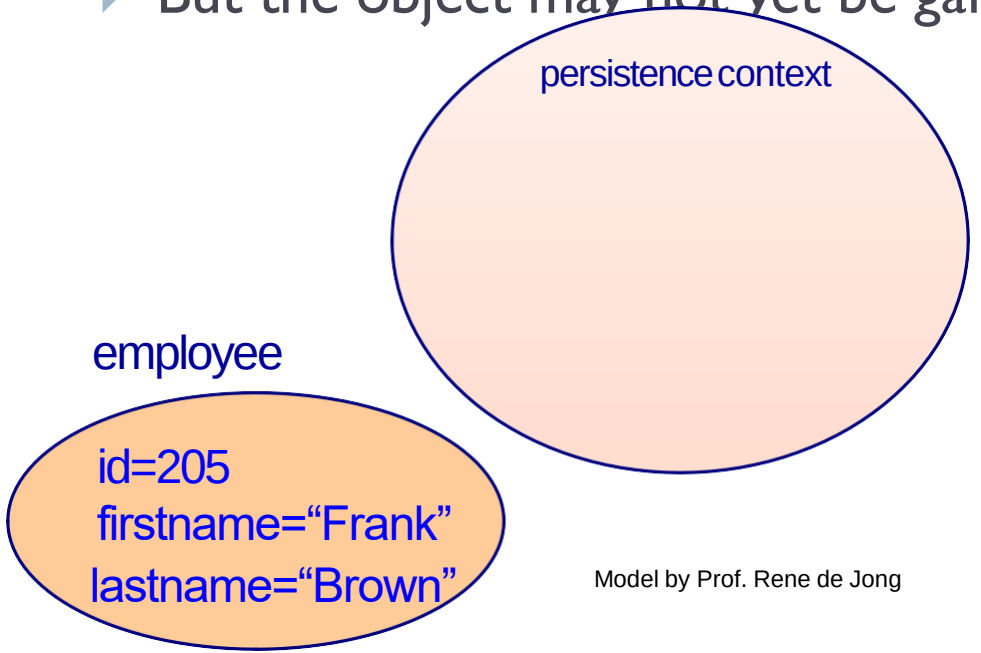


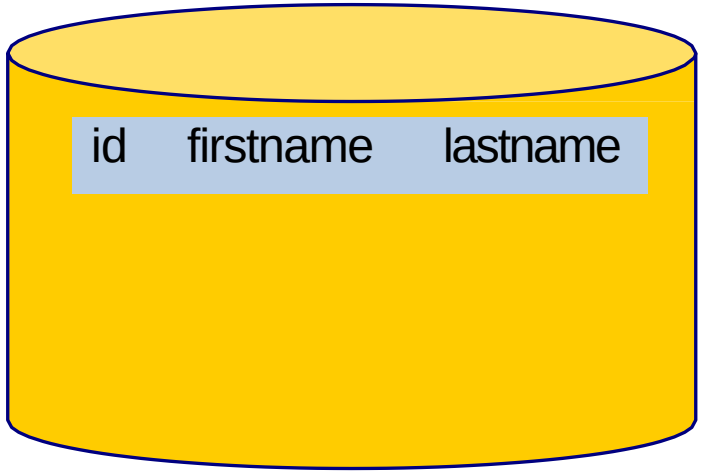
Model by Prof. Rene de Jong



Removed

- A removed Entity is removed from the DB and persistence ctx
  - But the object may not yet be garbage collected in Java





Model by Prof. Rene de Jong

EntityManager API

Method(s)	Data Base	Persistence Context
em.persist()	Uses SQL INSERT statements to create rows in the database.	Changes entity state from Transient to Managed
em.find() em.getReference()	Uses SQL SELECT statments to retrieve rows from the database	Creates managed entities
implicit update	Uses SQL UPDATE statements to update rows in the database	Only works on managed entities
em.merge()	Uses either INSERT or UPDATE depending on the state of the object	Changes transient or detached entities to managed
em.remove()	Uses SQL DELETE statements to delete rows from the database	Changes entity state from managed to removed
em.flush() em.refresh()	explicitly pushes changes to the db, explicitly gets changes from the db,	Only works on managed entities
em.contains() em.detach() em.clear()	No database activity	checks if entity is in cache, removes entity from cache, removes all entities from cache

Create == Persist

- The .persist() method takes the provided entity
  - Tries to check that it is Transient, makes it Managed

```
em.getTransaction().begin();
Customer c = new Customer("Jack", "Welsh");
em.persist(c);  em.getTransaction().commit();
```

- Hibernate implements:
  - For @GeneratedValue Checks that @Id == null
    - Else throws PersistentObjectException: detached object
  - For assigned IDs no check
    - DB Throws EntityExistsException if insert fails (same ID)

No Return Value

- Persist() returns void
  - Instead of returning the ID of the inserted row
  - JPA does not require immediate execution of insert

- Hibernate inserts entities with @GeneratedValue right away
  - Waits to insert entities with Assigned Ids

em.find() and em.getReference()

- find() and .getReference() are very similar
- Both retrieve an entity by its primary key value
- SELECT * FROM … WHERE id = …
- If an entity is already cached the DB is not hit
- What's the difference?

```
em.getTransaction().begin();
Person p1 = em.find(Person.class, 1L);
```

```
em.getTransaction().begin();
Person p1 = em.getReference(Person.class,1L);
```

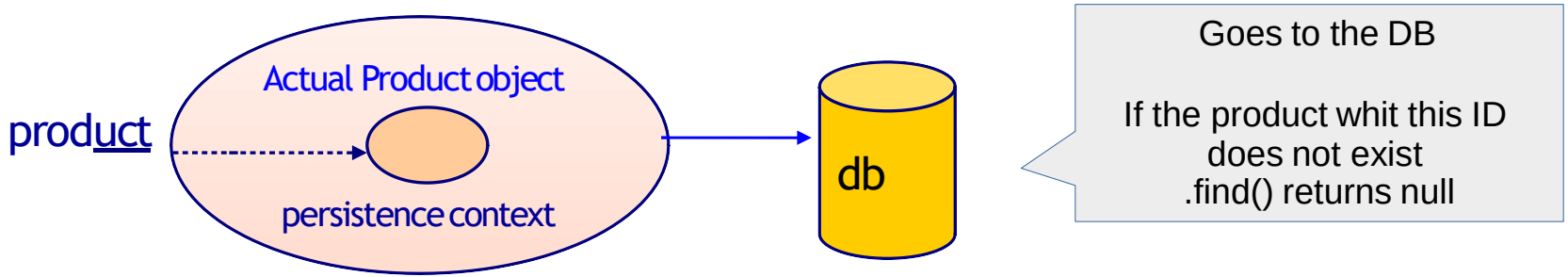
Difference

- .find() retrieves the object right away
  - Creating a DB hit right away

- .getReference() provides a proxy
  - Does not load the object's values until first needed

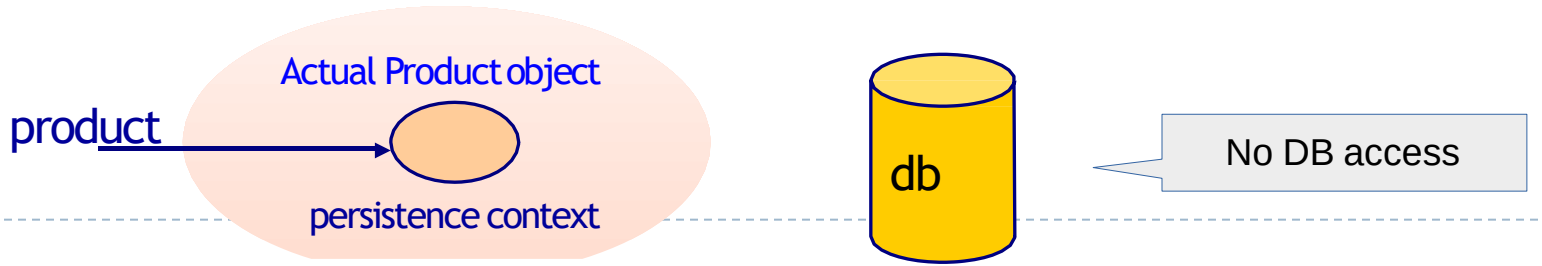
em.find()

Product product = em.find(Product.class, 1L);



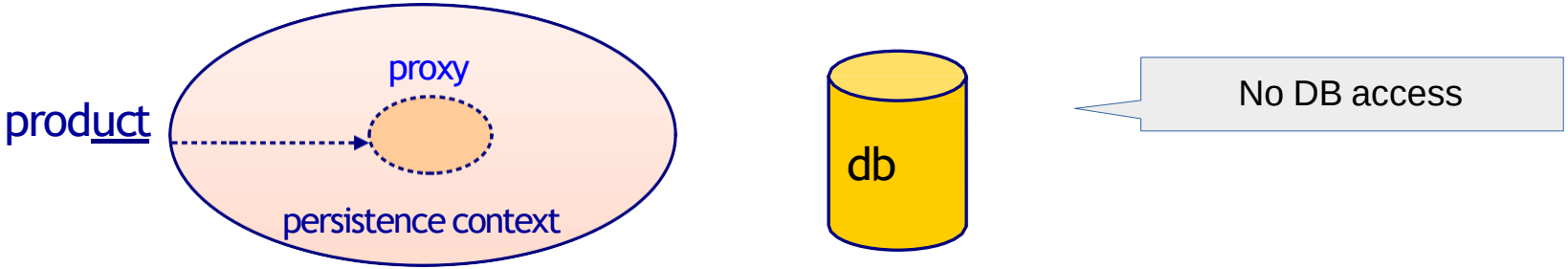
product.getPrice()

Model by Prof. Rene de Jong



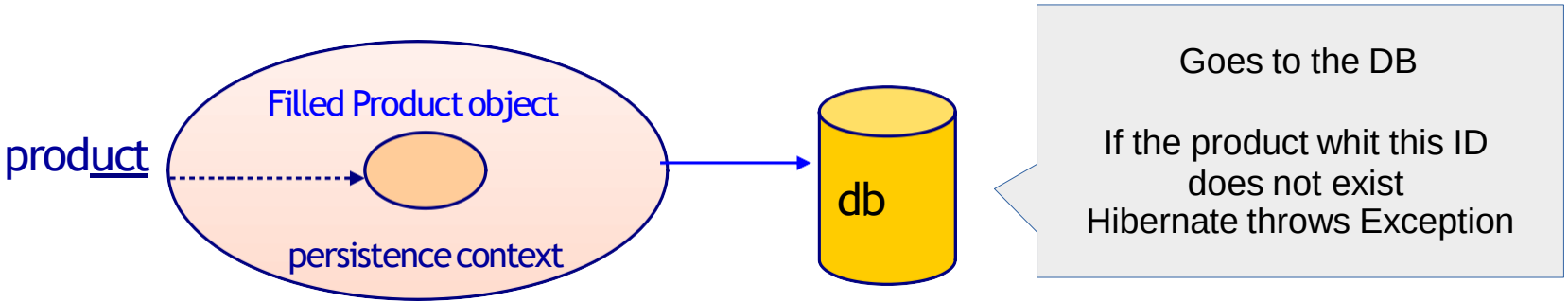
.getReference()

```
Product product = em.getReference(Product.class, 1L);
```




```
product.getPrice()
```

Model by Prof. Rene de Jong



When to use .getReference()

.getReference() is especially useful **combined** with **.remove()**

```
Person p1 = em.find(Person.class, 1L);
em.remove(p1);
```

```
Person p1 = em.getReference(Person.class, 1L);
em.remove(p1);
```

- 2 DB hits
  - One for .find()
  - One for .remove()
- 1 DB hit
  - 0 for .getReference()
  - 1 for .remove()

Implicit Update

- When a managed entity is changed
  - Changes are automatically pushed to the DB

```
em.getTransaction().begin();
Person p1 = em.find(Person.class, 1L);
p1.setName("Ben James");
em.getTransaction().commit();
```

- On tx.commit(), em.flush(), or before a query
  - Hibernate notices that an object is 'dirty'
  - Uses SQL Update to push changes to the DB

Detached Objects

- The Original hibernate API has an .update() method
  - Used to bring updates from detached objects to the database.
  - Changed state from Detached to Managed and marked object dirty

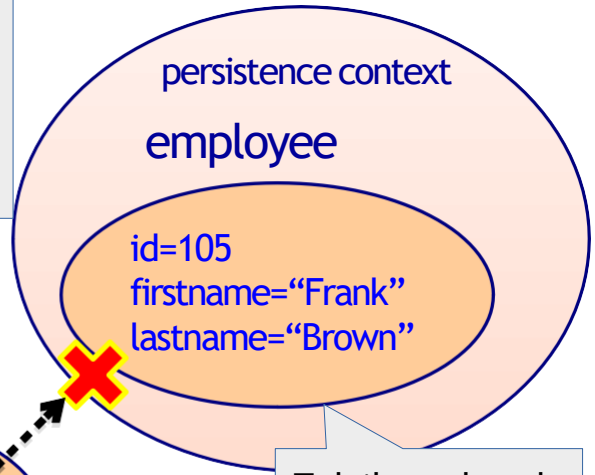
- Big Problem:
  - If the persistence context already has an object with the same type and primary key value a NonUniqueObjectException is created
  - Therefore JPA does not include this method

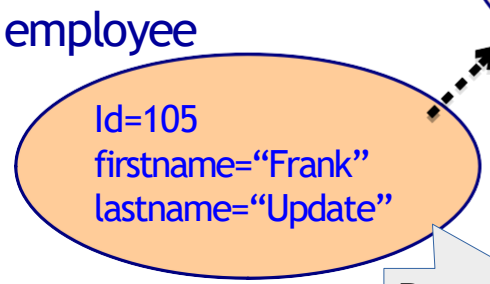
Problem

Not included in JPA

- **NonUniqueObjectException** with .update()

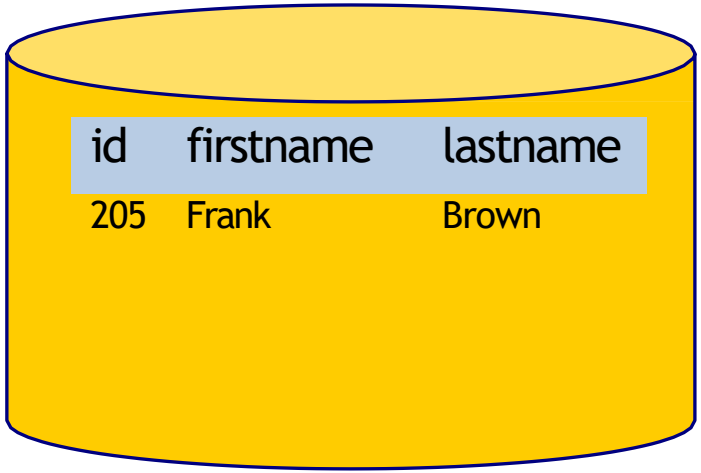
When .update() received a detached object for an entity that it was already managing Hibernate threw a NonUniqueObjectException





Existing, already managed object

Detached Object

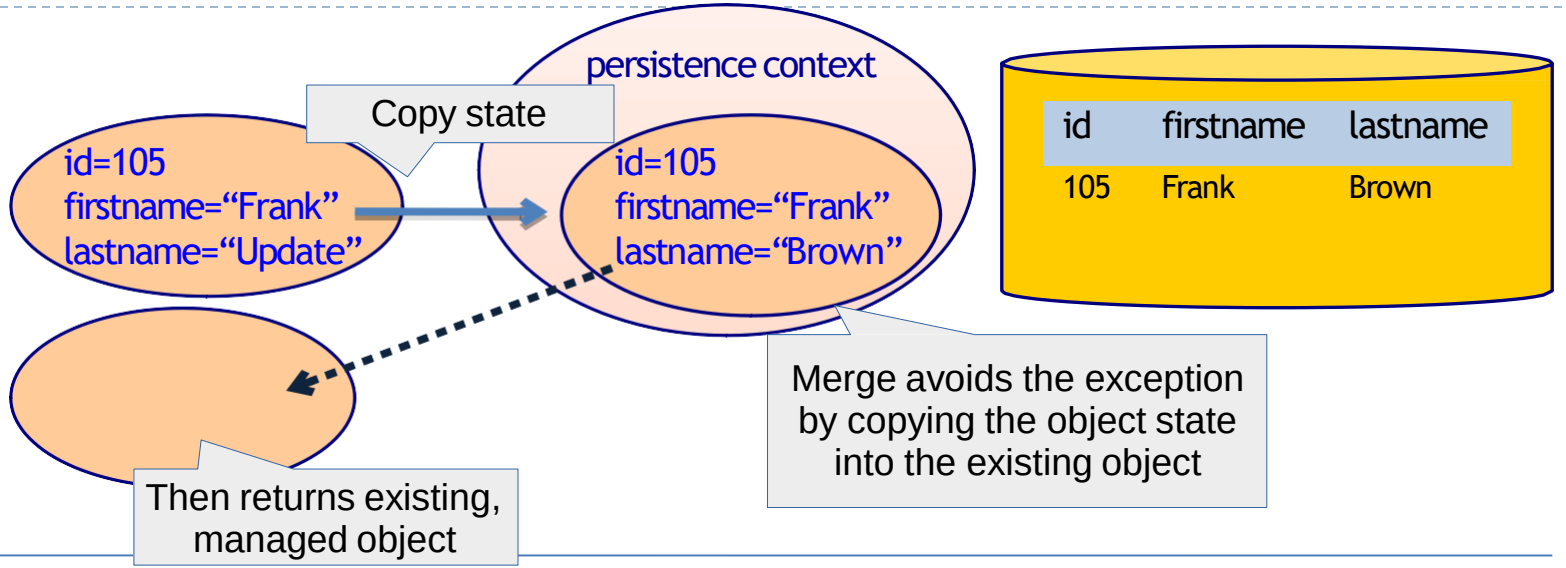


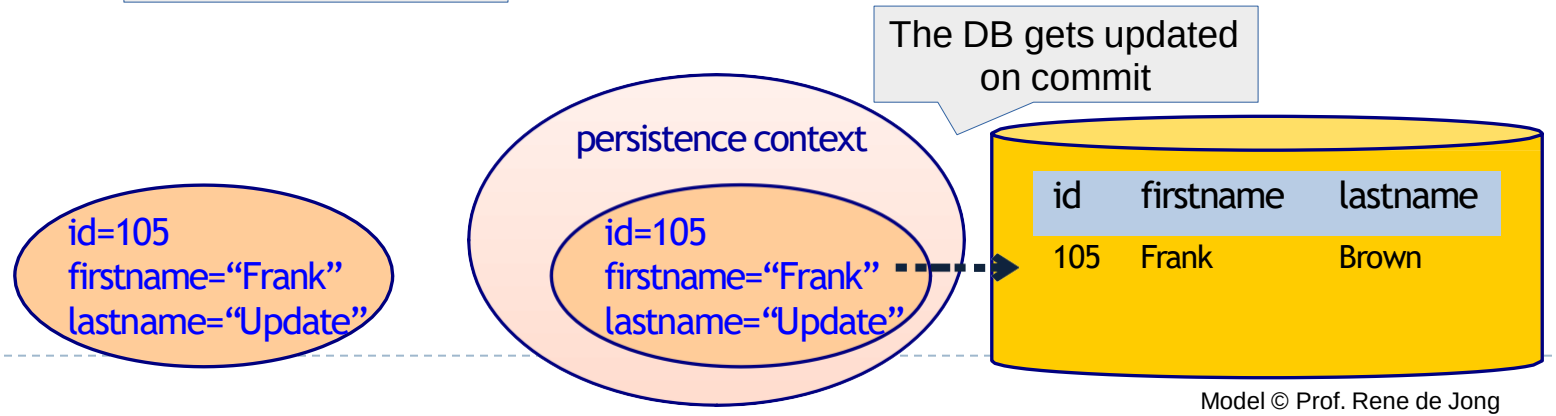
id	firstname	lastname
205	Frank	Brown

Model © Prof. Rene de Jong

Solution: use merge()

.merge() - Avoids NonUnique





Merge - Misunderstood

- Merge does not behave like .persist()
  - The object you pass to the method never becomes managed
  - Instead returns a different managed object (of the same type)

- If you continue working with the original object you can run into unexpected problems
  - Implicit updates are not persisted (object not managed)

Correct use of .merge()

Correct use:

- Always **use the return** value from .merge()

p is set to the return

```
em.getTransaction().begin();  p =
em.merge(p);  p.setName("Updated
name");  em.getTransaction().commit();
```

Update will be persisted

Incorrect use:

- Keep using object passed into .merge()

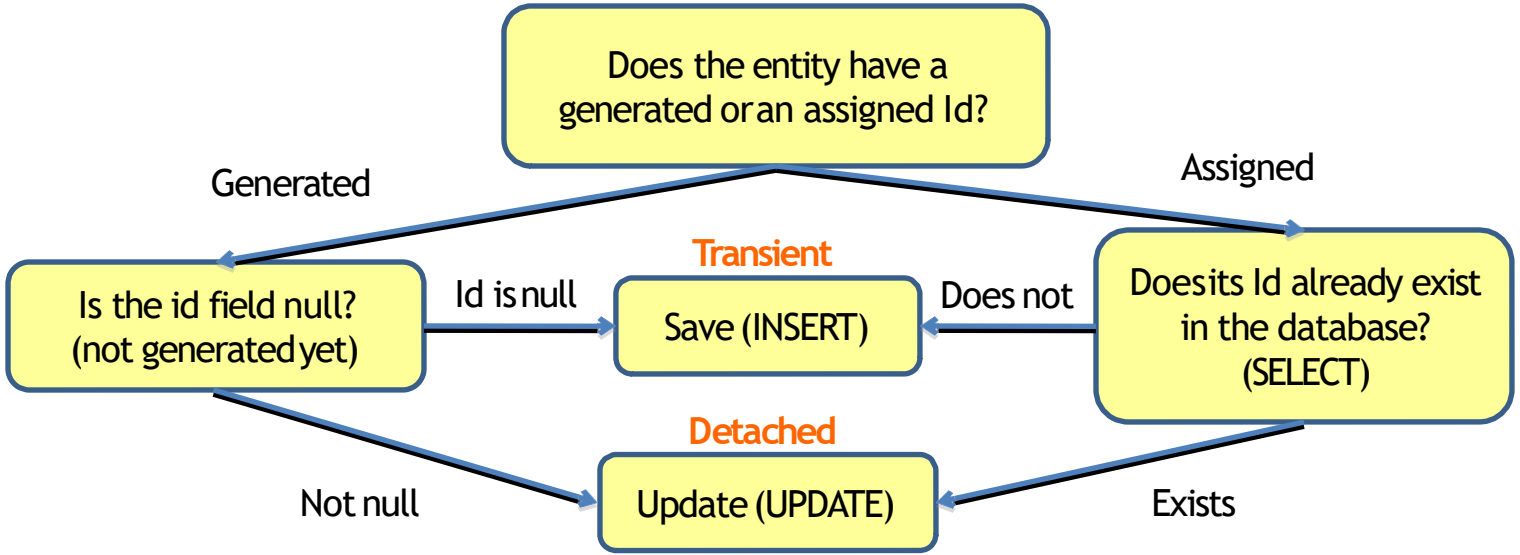
Return is lost p is still a detached object

```
em.getTransaction().begin();
em.merge(p);  p.setName("Updated
name");  em.getTransaction().commit();
```

Update will not reach the DB

Insert or Update

- .merge() has one more feature:
- It inserts Transient objects (returns a managed copy)
- Therefore can be used to save or update any object



.persist() or .merge()

- Is .persist() even needed?
  - You can do all inserts with .merge()

- .persist() makes your intent clearer
  - Different logic / implementation
  - Also checks id == null, but throws exception
  - Never accidentally updates

.remove()

- remove() deletes an entity from the DB
  - Only managed entities can be removed
- On tx.commit(), em.flush(), before query
  - SQL DELETE is used in the DB

```
em.getTransaction().begin();
Person p1 = em.getReference(Person.class, 1L);
em.remove(p1);
em.getTransaction().commit();
```

.getReference() gives managed entity object

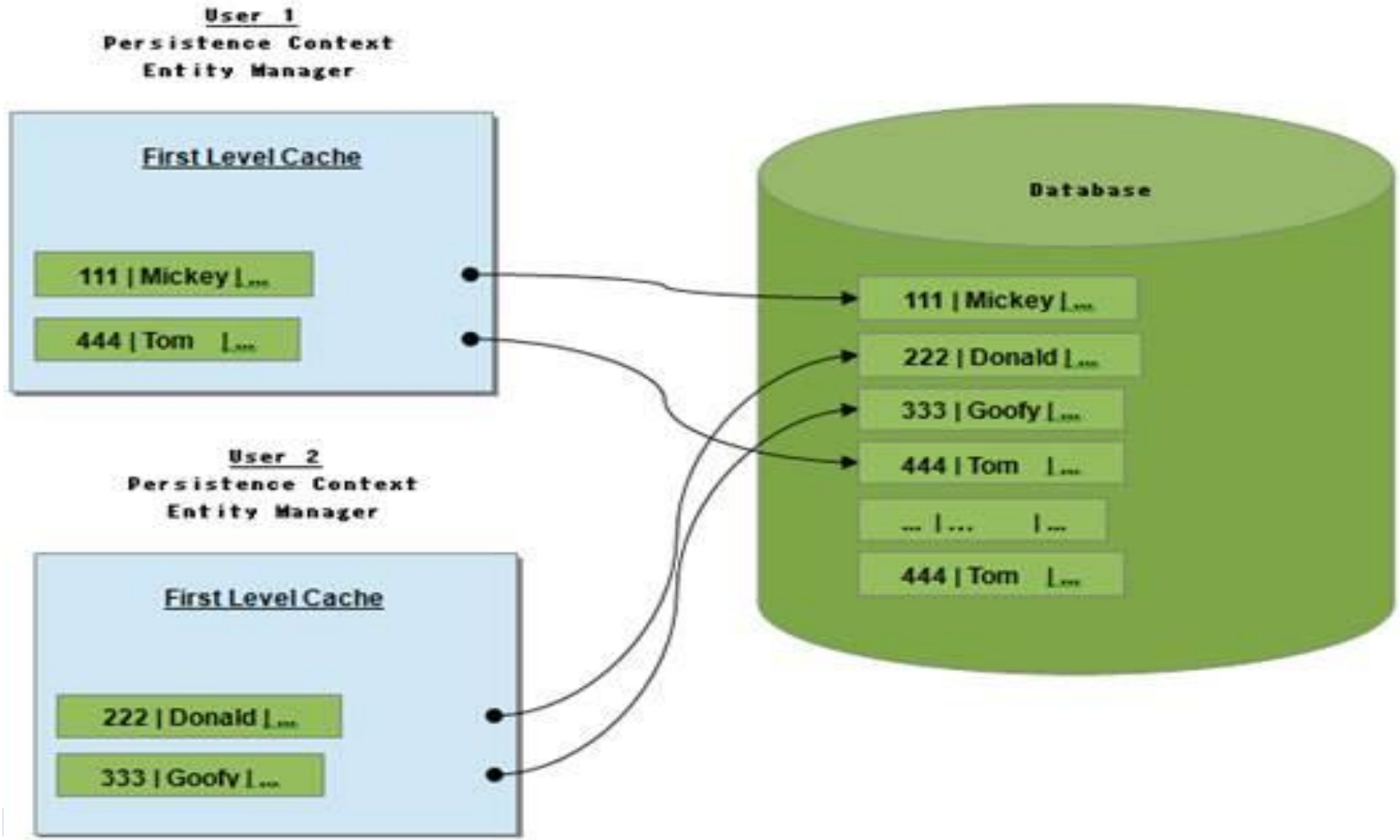
SQL DELETE at transaction commit

ORM caching mechanisms

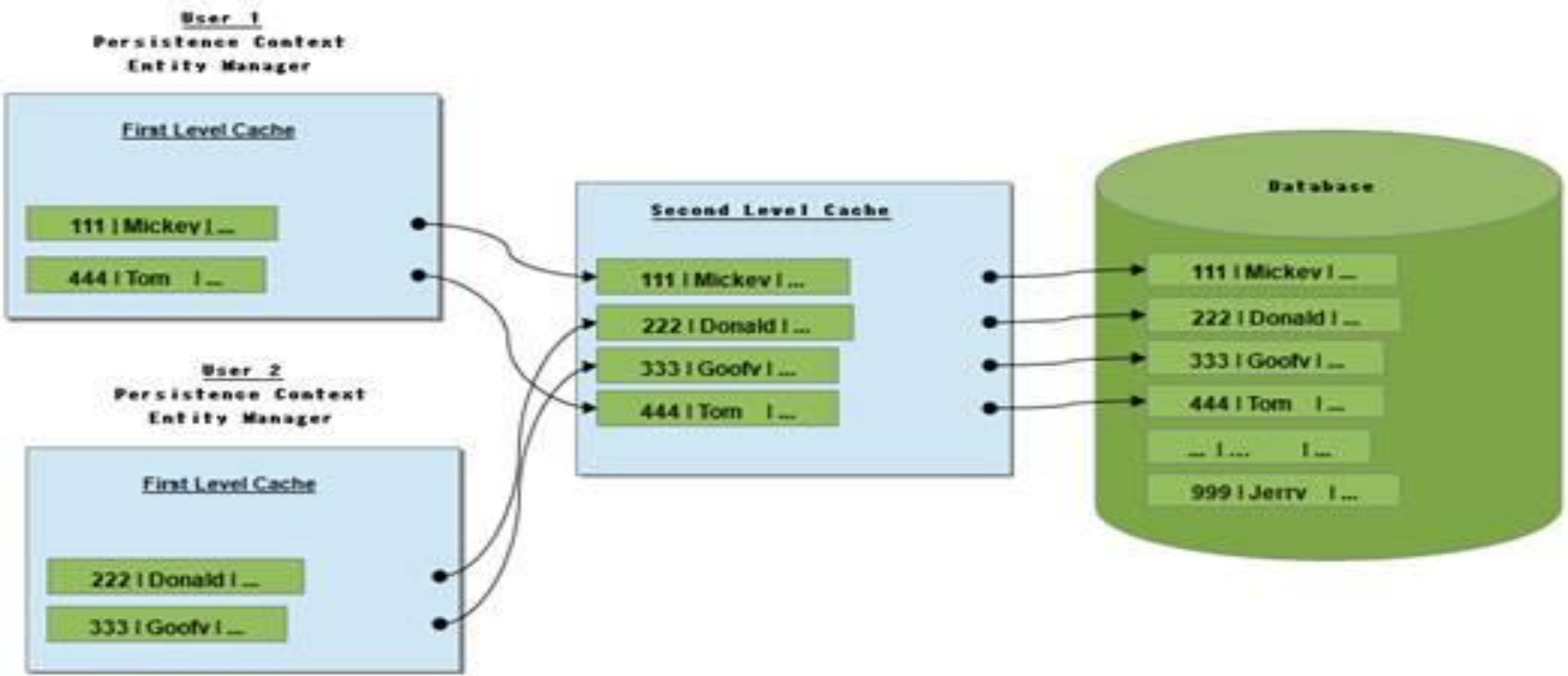
- Persistence provider manages local store of entity data
- Leverages performance by avoiding expensive database calls
- CRUD operation can be performed through normal entity manager functions
- Application can remain oblivious of the underlying cache and do its job without concern

- Level 1 Cache
  - Available within the same transaction [Persistence Context]
- Level 2 Cache
  - Available throughout the application.

Level 1 Cache



Level 2 Cache



Insert may be Held in Cache

- With .persist()
  - Hibernate pushes to the DB right away for @GeneratedValue entities
  - Hibernate holds it in cache until tx.commit() for assigned IDs

```
@Entity
public class Person {
        @Id
        @GeneratedValue
        private Long id;
        private String name;
```

Generated ID

```
em.getTransaction().begin();
Person p = new Person("Aaron James");
System.out.println("1");
em.persist(p);
System.out.println("2");
em.getTransaction().commit();
```

```
1
Hibernate: insert into Person (name) values (?)
2
```

```
@Entity
public class Person {
        @Id
        private Long id;
        private String name;
```

```
em.getTransaction().begin();
Person p = new Person("Aaron James");
p.setId(1L);
System.out.println("1");
em.persist(p);
System.out.println("2");
em.getTransaction().commit();
```

Assigned ID

Held in cache until .commit()

```
1
2
Hibernate: insert into Person (name, id) values (?, ?)
```

Retrievals use cache

- .find() and .getReference() do not hit the DB
  - If the object is already in cache

```
@Entity
public class Person {
        @Id  @GeneratedValue
        private Long id;
        private String name;
```

```
em.getTransaction().begin();
Person p = new Person("Aaron James");
System.out.println("1");
em.persist(p);
System.out.println("2");
long id = p.getId();
System.out.println("3");
em.find(Person.class, id);
System.out.println("4");
em.getReference(Person.class, id);
System.out.println("5");
em.getTransaction().commit();
System.out.println("6");
```

```
1
Hibernate: insert into Person (name) values (?)
2
3
4
5
6
```

Updates are held in cache

- Updates to managed objects are pushed on transaction commit

```
@Entity
public class Person {
        @Id
        @GeneratedValue
        private Long id;
        private String name;
```

Generated ID

```
em.getTransaction().begin();
Person p = new Person("Aaron James");
System.out.println("1");
em.persist(p);
System.out.println("2");
p.setName("Updated Name");
System.out.println("3");
em.getTransaction().commit();
System.out.println("4");
```

Update held until .commit()

```
1
Hibernate: insert into Person (name) values (?)
2
3
Hibernate: update Person set name=? where id=?
4
```

Removals 'held' in cache

- Removed objects are marked for deletion
  - No longer officially held in cache
  - But DELETE not executed until tx.commit()

.contains() returns false

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Person p = new Person("Aaron James");
System.out.println("1");
em.persist(p);
System.out.println("2");
em.remove(p);
System.out.println("3");
em.getTransaction().commit();
System.out.println("4");
```

Remove held until .commit()

```
1
Hibernate: insert into Person (name) values (?)
2
3
Hibernate: delete from Person where id=?
4
```

Changes Pushed Before Query

- All changes in cache are pushed before executing a query

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Person p = new Person("Aaron James");
System.out.println("1");
em.persist(p);
System.out.println("2");
p.setName("Updated Name");
System.out.println("3");
em.remove(p);
System.out.println("4");
TypedQuery<Person> q = em.createQuery("from Person", Person.class);
System.out.println("5");
List<Person> people = q.getResultList();
System.out.println("6");  em.getTransaction().commit();
System.out.println("7");
```

Changes can be: inserts, updates, deletes held in cache

This behavior can be changed by setting the **FlushMode**

Update not done because entity removed

```
1
Hibernate: insert into Person (name) values (?)
2
3
4
5
Hibernate: delete from Person where id=?
Hibernate: select person0_.id as id1_0_,  person0_.name as
name2_0_ from Person person0_
6
7
```

.flush()

- You can tell the entity manager to flush changes
  - Instead of waiting for .commit() or a query

```
em.getTransaction().begin();
Person p = new Person("Aaron James");
System.out.println("1");
em.persist(p);
System.out.println("2");
em.remove(p);
System.out.println("3");
em.flush();
System.out.println("4");

TypedQuery<Person> q = em.createQuery("from Person", Person.class);
  System.out.println("5");
  List<Person> people = q.getResultList();
  System.out.println("6");  em.getTransaction().commit();
  System.out.println("7");
```

Changes can be:
inserts, updates, deletes
held in cache

```
1
Hibernate: insert into Person (name) values (?)
2
3
Hibernate: delete from Person where id=?
4
5
Hibernate: select person0_.id as id1_0_,  person0_.name as
name2_0_ from Person person0_
6
7
```

.refresh()

- .refresh() 'refreshes' the data in the entity with the values found in the DB
  - Data in the DB may have changed
  - Can be used to undo updates

```
em.getTransaction().begin();
Person p = new Person("Aaron James");
System.out.println("1");  em.persist(p);
System.out.println("2");
Thread.sleep(5000); // sleep for 5 secs (other program changes db)
System.out.println("3");
// tries to 'get again' from db, but receives cached version  p =
em.find(Person.class, p.getId());
System.out.println(p.getName());
System.out.println("4");
em.refresh(p); // forced to go to db again
System.out.println(p.getName());
em.getTransaction().commit();
```

```
1
Hibernate: insert into Person (name) values (?)
2
3
Aaron James
4
Hibernate: select person0_.id as id1_0_0_,
person0_.name as name2_0_0_ from Person
person0_ where person0_.id=?
Updated Name
```

.contains()

- .contains() checks if the object is in the cache
  - Both assigned and generated are in cache right away
  - Assigned not in DB until commit

```
@Entity
public class Person {
        @Id
        @GeneratedValue
        private Long id;
        private String name;
```

Generated ID

```
em.getTransaction().begin();
Person p = new Person("Aaron James");
System.out.println(em.contains(p));
em.persist(p);
System.out.println(em.contains(p));
em.getTransaction().commit();
```

```
false
Hibernate: insert into Person (name) values (?)
true
```

```
@Entity
public class Person {
        @Id
        private Long id;
        private String name;
```

Assigned ID

```
em.getTransaction().begin();
Person p = new Person("Aaron James");
p.setId(1L);
System.out.println(em.contains(p));
em.persist(p);
System.out.println(em.contains(p));
em.getTransaction().commit();
```

```
false
true
Hibernate: insert into Person (name, id) values (?, ?)
```

.detach()

- .detach() detaches an entity from the cache
  - Entity state is then detached
  - .contain() no longer finds it

```
em.getTransaction().begin();
Person p1 = new Person("John");
Person p2 = new Person("Jane");
em.persist(p1);
em.persist(p2);
em.detach(p1);
System.out.println(em.contains(p1));
System.out.println(em.contains(p2));
em.getTransaction().commit();
```

```
Hibernate: insert into Person (name) values (?)
Hibernate: insert into Person (name) values (?)
false
true
```

.clear()

- .clear() removes all entities from the cache
  - All entity objects are detached
  - The cache is empty

```
em.getTransaction().begin();
Person p1 = new Person("John");
Person p2 = new Person("Jane");
em.persist(p1);
em.persist(p2);
em.clear();
System.out.println(em.contains(p1));
System.out.println(em.contains(p2));
em.getTransaction().commit();
```

```
Hibernate: insert into Person (name) values (?)
Hibernate: insert into Person (name) values (?)
false
false
```

.close()

- close() closes the EntityManager
  - All entities are automatically detached
  - Can no longer use the EntityManager

Main Point

1. The persistence framework has a set of simple common operations . We simply configure and use them improving flexibility, overall accuracy and performance of the system.
2. ***Science of Consciousness****: Research found that participants in the Transcendental Meditation program showed greater activation of the appropriate hemisphere of the brain. This means the brain responds more flexibly and dynamically.*