Persistence API

CS544: Enterprise Architecture

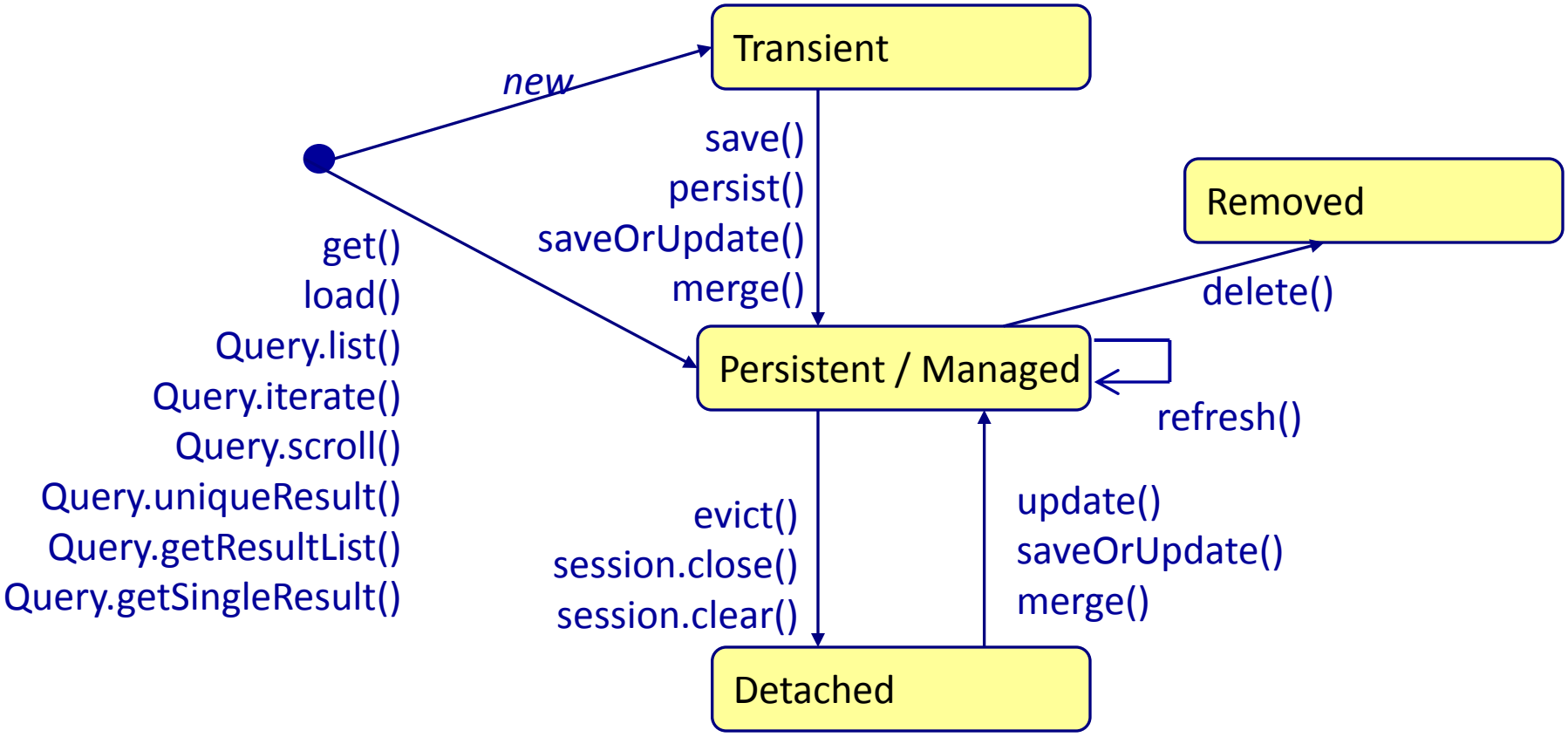
Object Persistence

- So far we've only discussed the basics of mapping classes to tables
- In this module we will discuss Hibernates object management
- We will look at the different states of the entity lifecycle, and the methods that are used to move objects between them
- We will also provide insight into the differences between some of Hibernates seemingly duplicate methods

Persistence API:

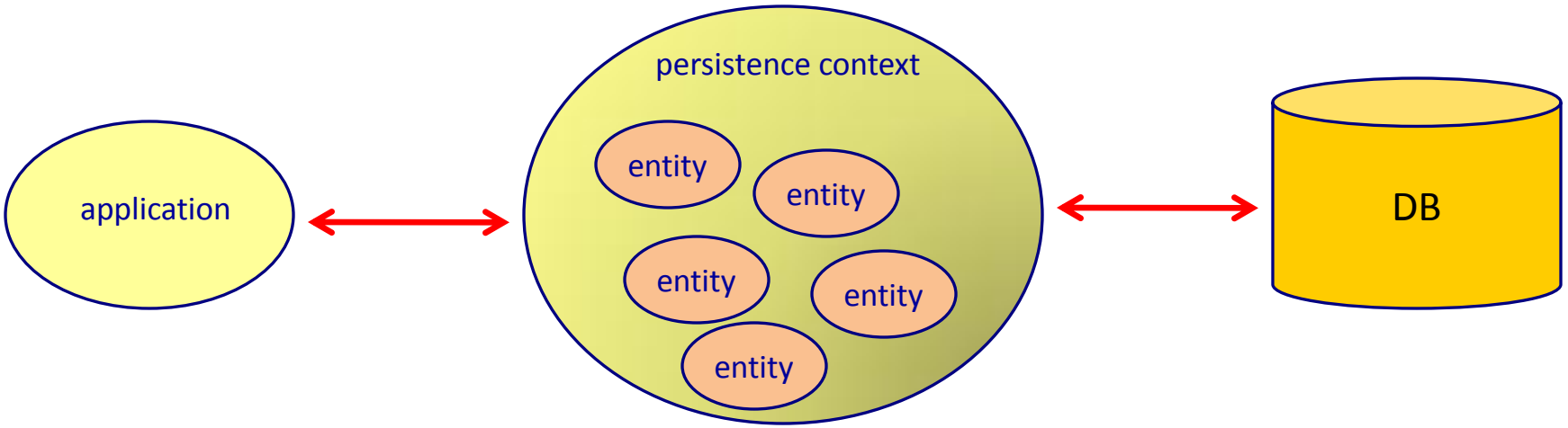
ENTITY OBJECT LIFECYCLE

Lifecycle of an entity



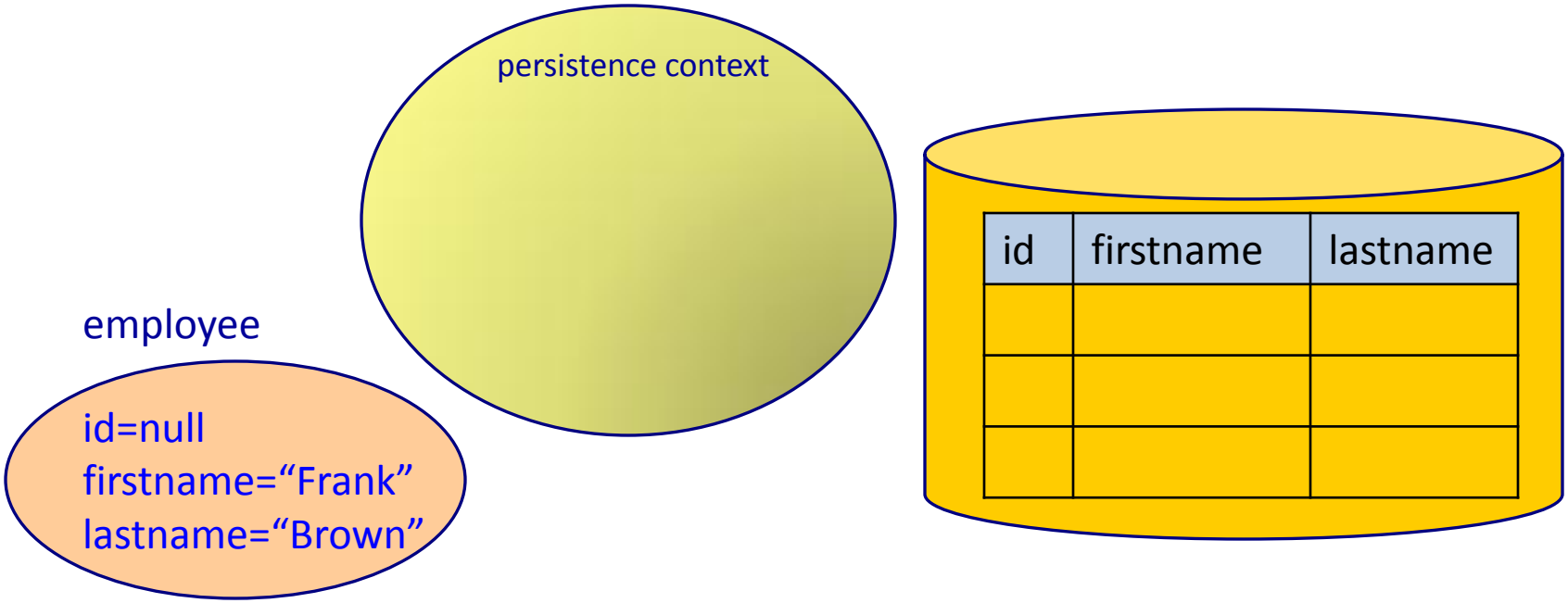
Persistence context

- Manages the entities
- Guarantees that managed enities will be saved in the database
- Tracks changes until they are pushed to the database
- Works like a cache



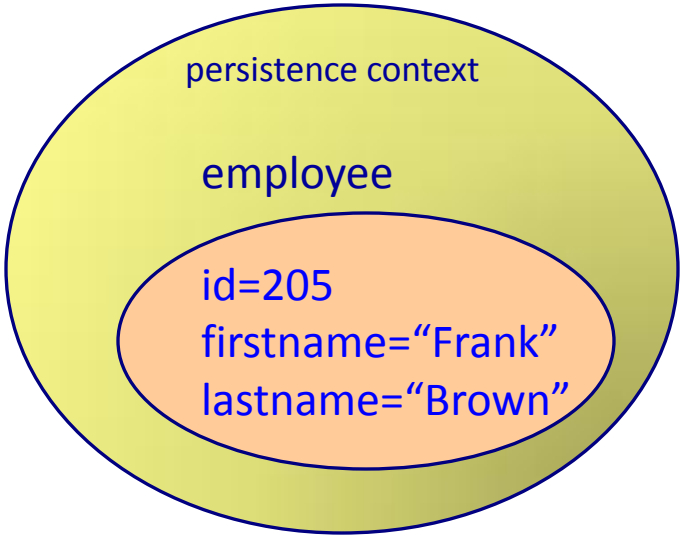
Transient entity

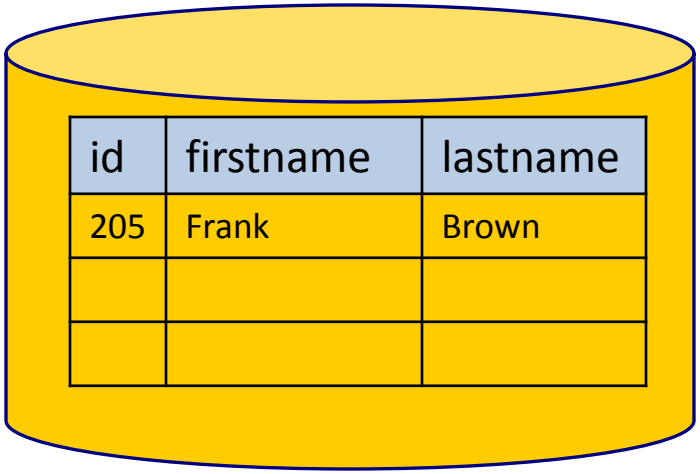
- A transient entity has no database identity



Managed entity

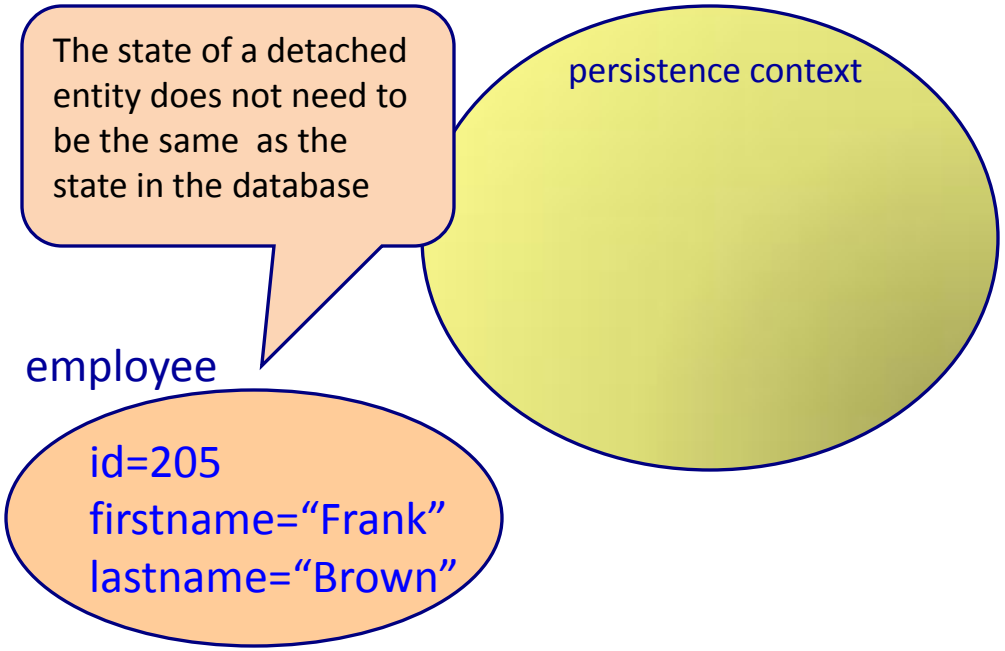
- A managed entity is managed by the persistence context and has a database identity

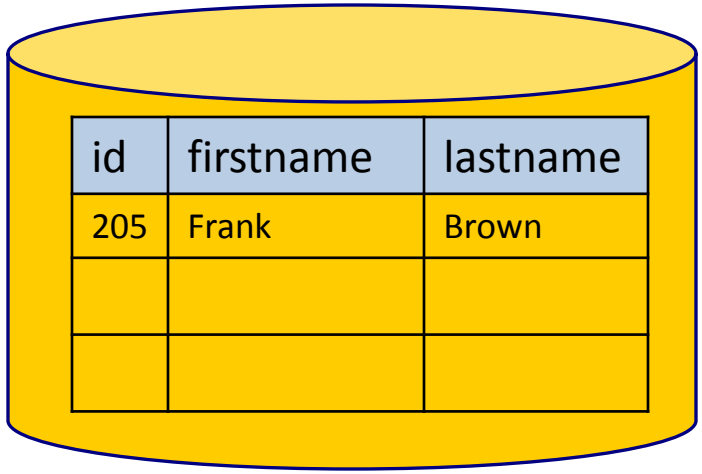




Detached entity

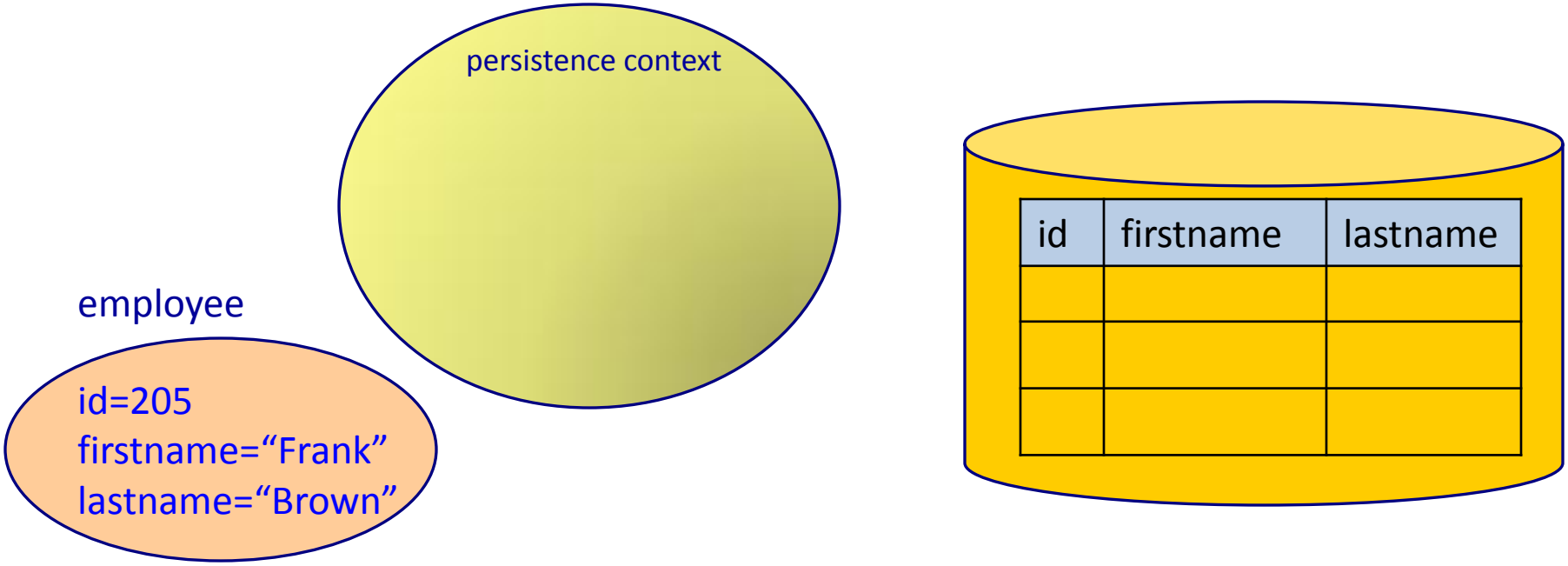
- A detached entity has a database identity, but is not managed by the current persistence context





Removed entity

- With a removed entity is the corresponding data removed from the database.



Session - Persistence Context

Type	Methods	Description
CREATE	session.persist() session.save()	Uses SQL INSERT statements to create rows in the database
RETRIEVE	session.load() session.get()	Uses SQL SELECT statments to retrieve rows from the database
UPDATE	*implicit update* session.update()	Uses SQL UPDATE statements to update rows in the database
CREATE / UPDATE	session.saveOrUpdate() session.merge()	Uses either INSERT or UPDATE depending on the state of the object
DELETE	session.delete()	Uses SQL DELETE statements to delete rows from the database
Cache related functions	session.refresh() session.flush()	Explicitly gets changes from the database, explicitly pushes changes to the database

Persistence API:

STORING ENTITY OBJECTS

About: save() and persist()

- Save() and persist() are very similar
  - Both will add the given entity to the persistence context, and INSERT it into the database

```
Person p1 = new Person("Frank Brown");
long id1 = (Long)session.save(p1);
```

Saves the entity and returns its generated id

```
Person p1 = new Person("Frank Brown");
session.persist(p1);
```

Persists the entity but does not return anything

- Save() will always insert right away (eager)
- Persist() will insert when it needs to (lazy)

Return value: save() / persist()

- Because save() executes right away it can guarantee to return the generated id
- Persist() on the other hand may wait until the end of the transaction to execute its INSERT
- Persist() therefore returns void

```
Person p1 = new Person("Frank Brown");
long id1 = (Long)session.save(p1);
```

Saves the entity and returns its generated id

```
Person p1 = new Person("Frank Brown");
session.persist(p1);
```

Will persists the entity but may not do so now, can therefore not guarantee a return id

Persistence API:

RETRIEVING ENTITY OBJECTS

About: get() and load()

- Get() and load() are very similar
  - Both retrieve an object by its primary key
    - SELECT * FROM … WHERE [*primairy key*] = …
    - If an entity is already cached the database is not hit

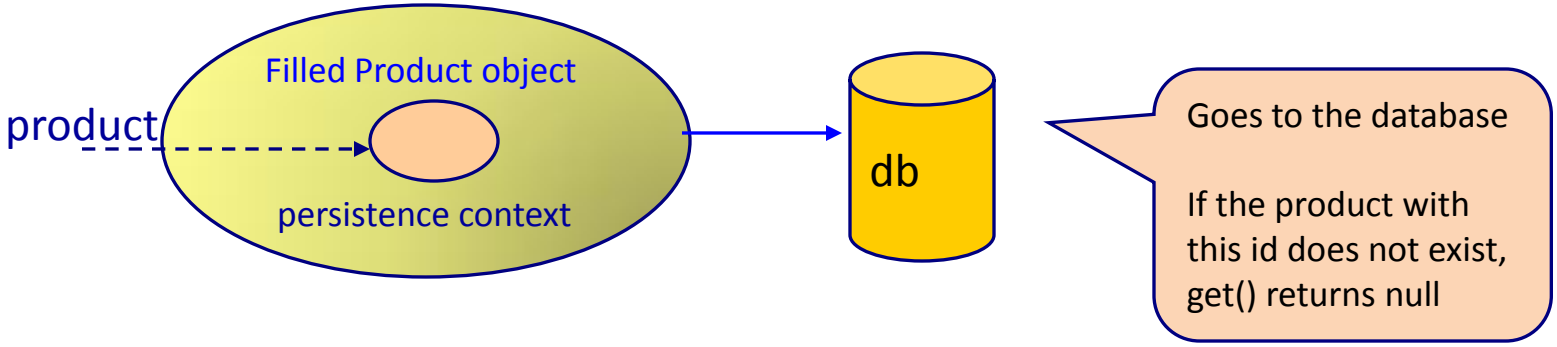
```
Person p1 = (Person) session.get(Person.class, 1L);
```

```
Person p1 = (Person) session.load(Person.class, 1L);
```

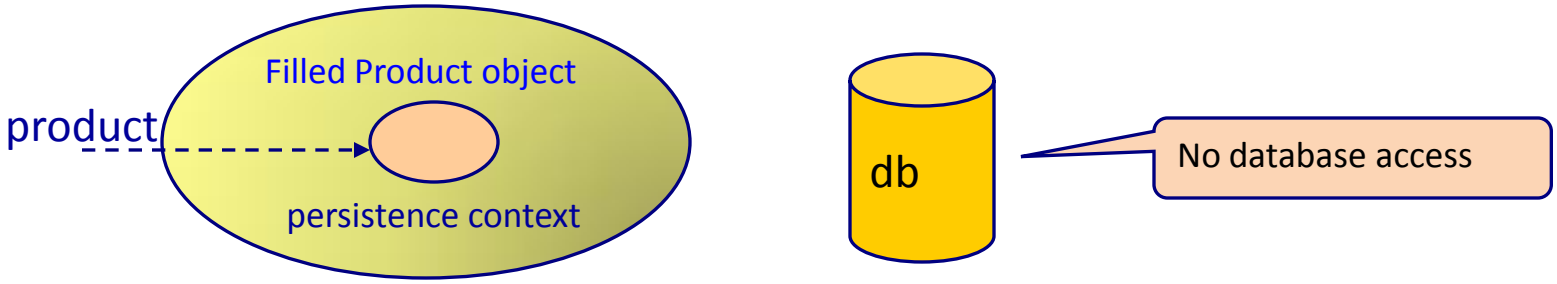
- Get() retrieves the object's values right away
- Load() provides a proxy and does not retrieve the object's values until they are first needed

get()

1. Product product = (Product)session.get(Product.class, new Long(productid));

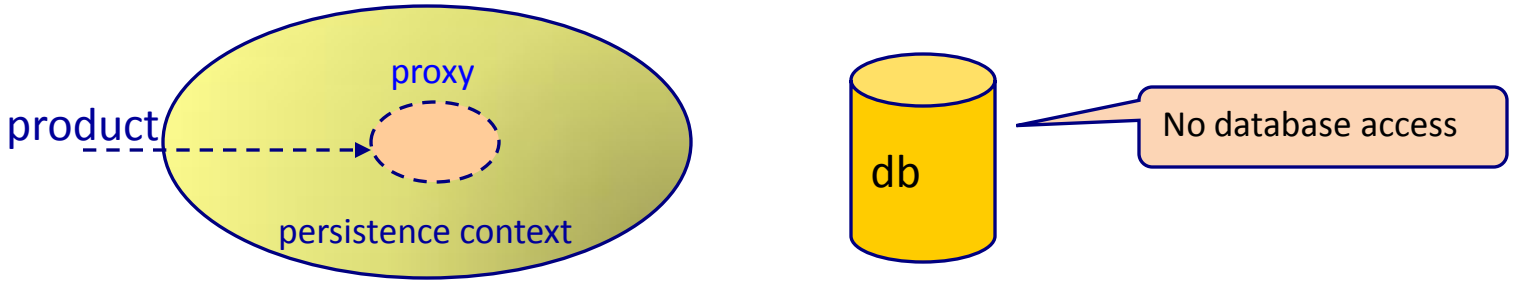


2. product.getPrice()

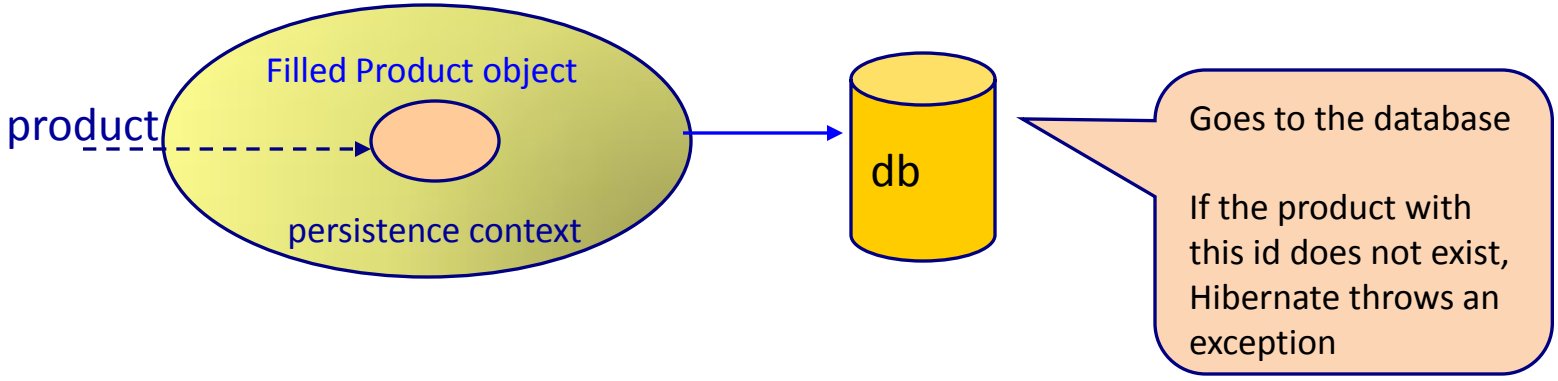


load()

1. Product product = (Product)session.load(Product.class, new Long(productid));



2. product.getPrice()



When to use load() instead of get()

```
Employee employee = (Employee) session.get(Employee.class, employeeid );
session.delete(employee);
```

- 2 database hits
  - 1 SELECT statement for the get() method
  - 1 DELETE statment for the delete() method

The load() method creates an Employee proxy object in the persistence context, set its id, but does not hit the database.

```
Employee employee = (Employee) session.load(Employee.class, employeeid );
session.delete(employee);
```

- 1 database hit
  - 1 DELETE statement for the delete() method

Persistence API:

UPDATING ENTITY OBJECTS

Implicit Update

- When a managed entity is changed inside a transaction, the changes are pushed to the database when the transaction commits

```
try {
  session = sessionFactory.openSession();
  tx = session.beginTransaction();
  Person p3 = (Person) session.get(Person.class, 1L);
  p3.setName("Implicitly Updated");
  tx.commit();
} catch (HibernateException e) {
  tx.rollback();
  e.printStackTrace();
} finally {
  if (session != null) {
    session.close();
  }
}
```

Get () puts the object into the persistence context (becomes managed)

Object is changed

Change is pushed to the database when the transaction commits

- Hibernate notices that an object is 'dirty' and uses a SQL UPDATE to push it to the database

About: update()

- You can also explicitly call the update() method

```
try {
  session = sessionFactory.openSession();
  tx = session.beginTransaction();
  Person p3 = (Person) session.get(Person.class, 1L);
  p3.setName("Explicitly Updated");
  session.update(p3);
  tx.commit();
} catch (HibernateException e) {
  tx.rollback();
  e.printStackTrace();
} finally {
  if (session != null) {
    session.close();
  }
}
```

Get () puts the object into the persistence context (becomes managed)

Object is changed

Explicit update()

Change is pushed to the database when the transaction commits

- Like the implicit update, changes are not pushed to the database until the transaction is committed

Implicit vs. Explicit update

- In the last example the explicit update was completely redundant – not needed
- Why even have an explicit update() method?

```
session.update(detached_object);
```

Update() can be used to re-attach

- The real power of the explicit update() is in re-attaching detached objects that have been updated outside of the persistence context
- Once attached their changes will also be pushed to the database on commit()

Re-Attach Example

```
session = sessionFactory.openSession();
Person p1 = new Person("Frank Brown");
session.save(p1);
session.close();

p1.setName("Detached Update");

try {
  session = sessionFactory.openSession();
  tx = session.beginTransaction();
  session.update(p1);
  tx.commit();
} catch (HibernateException e) {
  tx.rollback();
  e.printStackTrace();
} finally {
  if (session != null) {
    session.close();
  }
}

session = sessionFactory.openSession();
Person p2 =(Person)session.get(Person.class, 1L);
System.out.println("Name: " + p2.getName());
session.close();
```

Session.close() makes p1 detached

Detached update

Update re-attaches

Commit pushes updates of attached object to the db

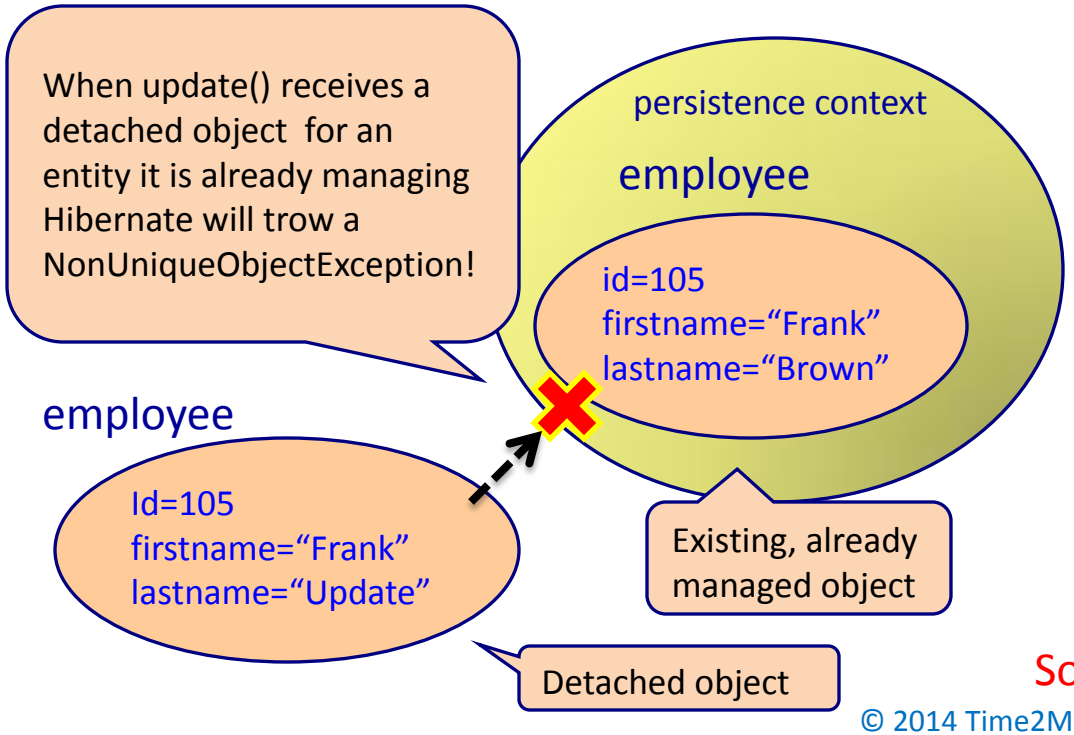
Load object to check if update was successful

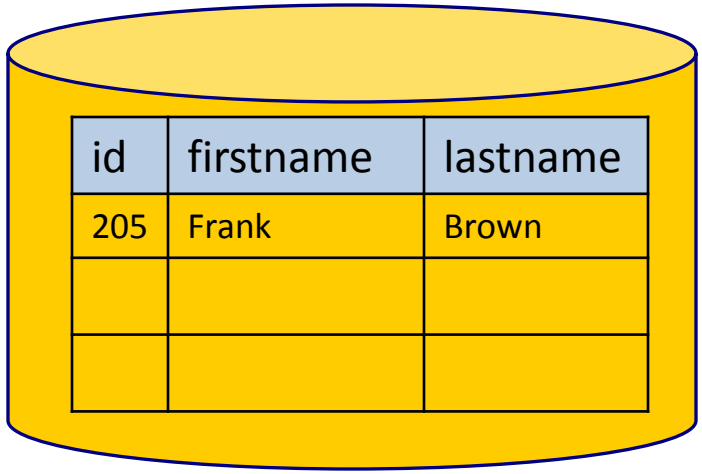
Person was updated successfully

```
Name: Detached Update
```

The Problem with Re-Attaching

- Update() will throw a NonUniqueObjectException
  - when re-attaching an object with the same id as an already managed object





Solution: use merge()

Exception Example

```
session = sessionFactory.openSession();
Person p1 = new Person("Frank Brown");
long id = (Long)session.save(p1);
System.out.println("Frank Brown Id: " +id);
session.close();

try {
  session = sessionFactory.openSession();
  tx = session.beginTransaction();

  Person p2 = (Person)session.get(Person.class, 1L);
  session.update(p1);

  tx.commit();
} catch (HibernateException e) {
  tx.rollback();
  e.printStackTrace();
} finally {
  if (session != null) {
    session.close();
  }
```

Session.close() makes p1 detached

Output shows id=1

New session, new persistence context

Loads person with id=1 (Frank Brown) into the persistence context

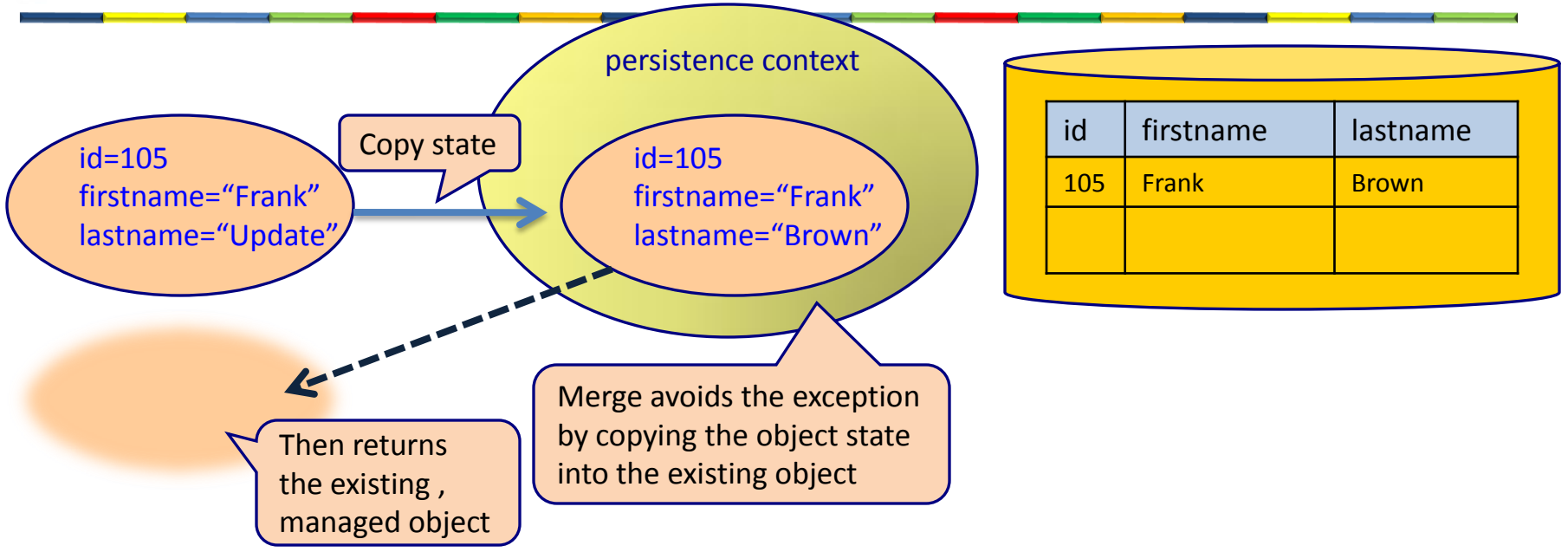
Tries to re-attach p1, also id=1 (Also Frank Brown)

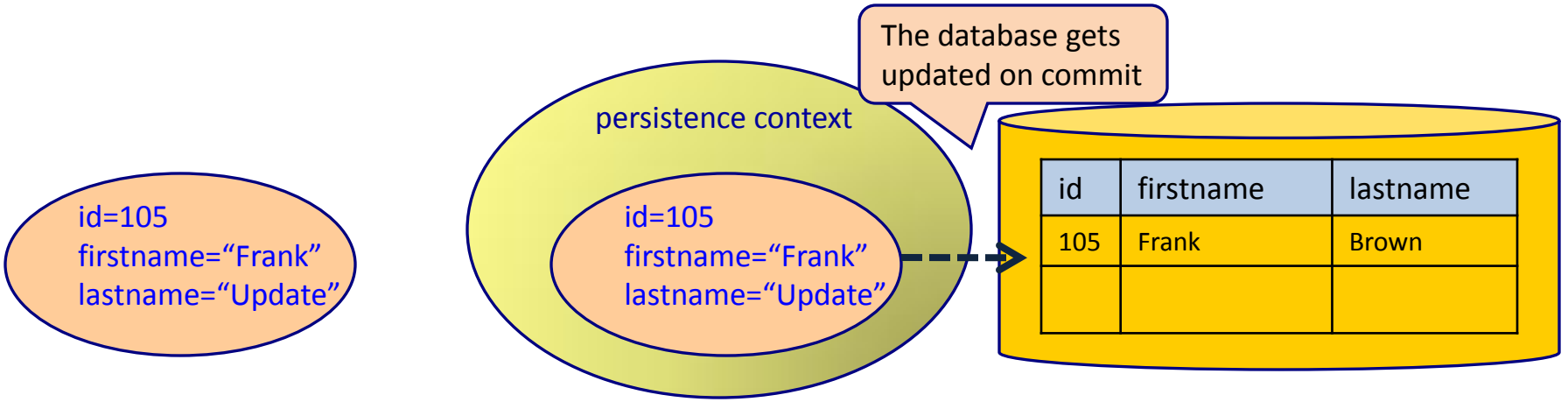
Output:

```
Hibernate: /* insert entity.Person */ insert into Person (id, name) values (null, ?)
Hibernate: call identity()
Frank Brown Id: 1
Hibernate: /* load entity.Person */ select person0_.id as id0_0_, person0_.name as name0_0_
from Person person0_ where person0_.id=?
org.hibernate.NonUniqueObjectException: a different object with the same identifier value was
already associated with the session: [entity.Person#1]
```

Exception

Merge – Avoids the Exception

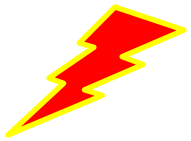




Merge - Misunderstood

- Merge does not behave like save(), update(), and saveOrUpdate()
  - The object you pass in to merge never becomes part of the persistence context (is not managed)
  - Instead merge returns an different object representing the same entity (which *is* managed)

- If you continue working with the original object you can run into unexpected problems, implicit updates to it are not persisted



Correct use of Merge

Correct Use of merge() – Always use the return value

```
Person p = new Person("Frank Brown");

try {
  session = sessionFactory.openSession();
  tx = session.beginTransaction();

  p = (Person)session.merge(p);
  p.setName("Upd. Frank Brown");

  tx.commit();
} catch (HibernateExc
  tx.rollback();
  e.printStackTrace();
} finally {
  if (session != null) {
    session.close();
  }
}
```

P is set to the return of merge()

Update will be commited

Incorrect Use of merge()

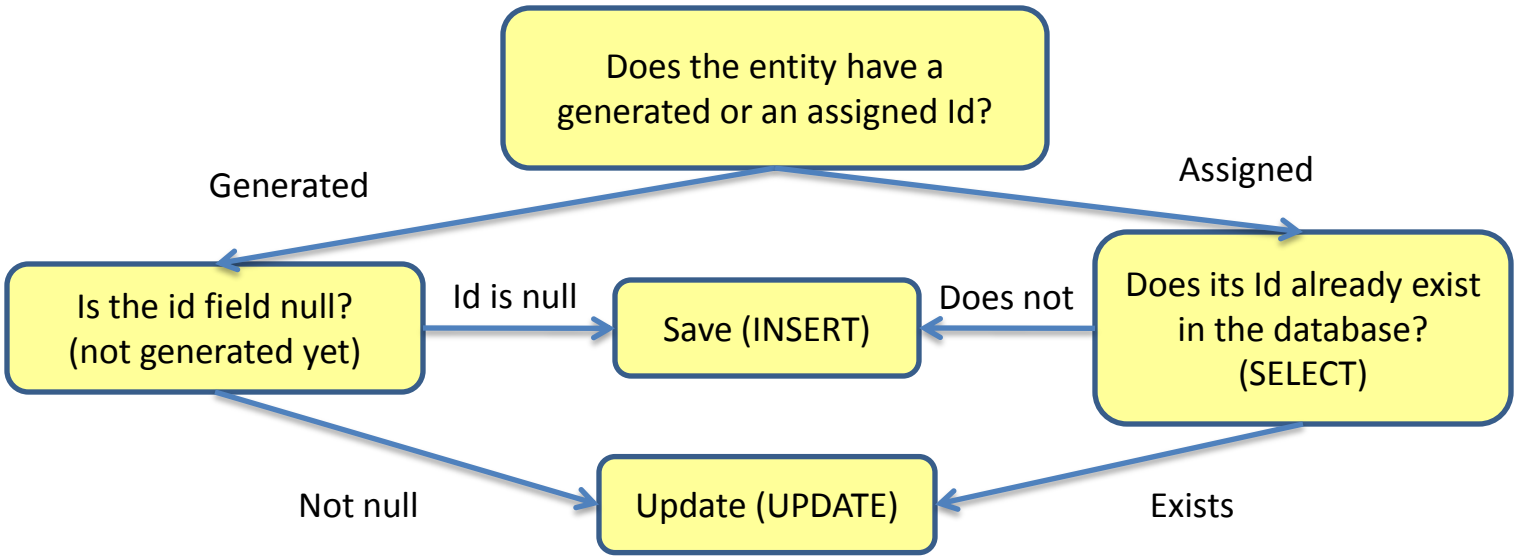
```
Person p = new Person("Frank Brown");

try {
  session = sessionFactory.openSession();
  tx = session.beginTransaction();

  session.merge(p);
  p.setName("Upd. Frank Brown");

  tx.commit();
} catch (HibernateException e) {
  tx.rollback();
  e.printStackTrace();
} finally {
  if (session != null) {
    session.close();
  }
}
```

Return is lost, P is still the detached object

Update will never get commited

saveOrUpdate()

- Hibernate checks if it needs to INSERT or UPDATE
  - If needed using a SELECT to find if it's already in the db



Persistence API:

REMOVING ENTITY OBJECTS

About: delete()

- Can be used to remove an entity from the db
  - Only managed entities can be removed
- Will only work inside a transaction
- On transaction.commit() a SQL DELETE is used to delete the entity from the database

```
try {
  session = sessionFactory.openSession();
  tx = session.beginTransaction();

  Person p1 = (Person) session.load(Person.class, 1L);
  session.delete(p1);

  tx.commit();
} catch (HibernateException e) {
  tx.rollback();
  e.printStackTrace();
}
```

Load entity so that is managed in the persistence context

Delete entity

SQL DELETE is not executed untill the transaction commits

Persistence API:

PERSISTENCE CACHE

When does Hibernate access the DB?

```
session = sessionFactory.openSession();
tx = session.beginTransaction();

Employee fmemployee = new Employee("Frank", "Miller");
System.out.println("1");
session.persist(fmemployee);
System.out.println("2");

tx.commit();
```

Hibernate will access the database during the persist() to retrieve the unique key

```
@Entity
public class Employee {

   @Id
   @GeneratedValue
   private int id;
   private String firstname;
   private String lastname;
   …
}
```

@GeneratedValue

```
1
Hibernate: /* insert Employee */ insert into Employee (id, lastname) values (null, ?)
Hibernate: call identity()
2
```

When does Hibernate access the DB?

```
session = sessionFactory.openSession();
tx = session.beginTransaction();

Employee fmemployee = new Employee("Frank", "Miller");
System.out.println("1");
session.persist(fmemployee);
System.out.println("2");

tx.commit();
System.out.println("3");
```

Hibernate will access the database during the commit()

```
@Entity
public class Employee {

  @Id
  private int id;
  private String firstname;
  private String lastname;
  …
}
```

Application generates the id

```
1
2
Hibernate: /* insert Employee */ insert into Employee (firstname, lastname, id) values
(?, ?, ?)
3
```

When does Hibernate access the DB?

```
session = sessionFactory.openSession();
tx = session.beginTransaction();

Employee fmemployee = new Employee("Frank", "Miller");
System.out.println("1");
session.persist(fmemployee);
System.out.println("2");
fmemployee.setFirstname("John");
System.out.println("3");
fmemployee.setLastname("Doe");
System.out.println("4");

tx.commit();
System.out.println("5");
```

Hibernate will access the database during the persist() to retrieve the unique key

Hibernate will access the database during the commit() to update the state of the changed employee

```
@Entity
public class Employee {

  @Id
  @GeneratedValue
  private int id;
  private String firstname;
  private String lastname;
  …
}
```

```
1
Hibernate: /* insert Employee */ insert into Employee (id, firstname, lastname) values
(null, ?, ?)
Hibernate: call identity()
2
3
4
Hibernate: /* update Employee */ update Employee set firstname=?, lastname=? where id=?
5
```

When does Hibernate access the DB?

```
session = sessionFactory.openSession();
tx = session.beginTransaction();

Employee fmemployee = new Employee("Frank", "Miller");
System.out.println("1");
session.persist(fmemployee);
System.out.println("2");
fmemployee.setFirstname("John");
System.out.println("3");
fmemployee.setLastname("Doe");
System.out.println("4");
Employee jdemployee= (Employee)session.get(Employee.class, fmemployee.getId());
System.out.println("5");

tx.commit();
System.out.println("6");
```

No database access, object is already in the persistence context

```
1
Hibernate: /* insert Employee */ insert into Employee (id, firstname, lastname) values
(null, ?, ?)
Hibernate: call identity()
2
3
4
5
Hibernate: /* update Employee */ update Employee set firstname=?, lastname=? where id=?
6
```

When does Hibernate access the DB?

```
session = sessionFactory.openSession();
tx = session.beginTransaction();

Employee fmemployee = new Employee("Frank", "Miller");
System.out.println("1");
session.persist(fmemployee);
System.out.println("2");
fmemployee.setFirstname("John");
System.out.println("3");
fmemployee.setLastname("Doe");
System.out.println("4");
Query query= session.createQuery("from Employee");
Collection<Employee> employeelist= query.list();
System.out.println("5");

tx.commit();
System.out.println("6");
```

```
@Entity
public class Employee {

   @Id
   @GeneratedValue
   private int id;
   private String firstname;
   private String lastname;
   …
}
```

Query on Employee, and we have a dirty Employee in the persistence context. First update the Employee in the database, and then execute the query

```
1
Hibernate: /* insert Employee */ insert into Employee (id, firstname, lastname) values
(null, ?, ?)
Hibernate: call identity()
2
3
4
Hibernate: /* update Employee */ update Employee set firstname=?, lastname=? where id=?
Hibernate: /* from Employee */ select employee0_.id as id0_, employee0_.firstname as
firstname0_, employee0_.lastname as lastname0_ from Employee employee0_
5
6
```

flush()

- Flushes any changes held in the session persistence context to the database
- Any changes made by an implicit or explicit update(), objects marked for deletion with delete() or for persistence with persist()
- All these changes can be committed to the database immediately without having to wait for the transaction to commit

```
session.flush();
```

flush()

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

APerson p1 = new APerson("Frank Brown", 1);
APerson p2 = new APerson("John Doe", 2);
APerson p3 = new APerson("Mary Smith", 3);
session.persist(p1);
session.persist(p2);
session.persist(p3);

p1.setName("Frank Updated");
p2.setName("John Updated");
session.delete(p3);

System.out.println("About to flush()");
session.flush();
System.out.println("Changes flushed");

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

Persist() does not insert entities with assigned ids right away

Creates changes to the objects in the persistence cache

flush() everything

No changes left to commit()

Person with assigned identifier

```
@Entity
public class APerson {
  @Id
  private long id;
  private String name;

  ...
```

DB Result:

```
mysql> select * from person;
+----+---------------+
| id | name          |
+----+---------------+
|  1 | Frank Updated |
|  2 | John Updated  |
+----+---------------+
```

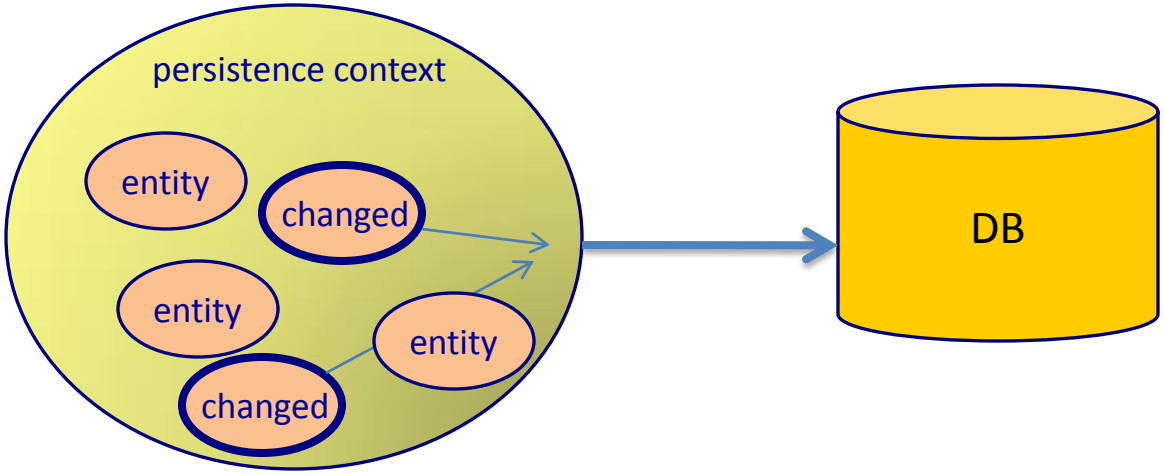
Hibernate Output:

Flush() commits all changes held in cache

```
About to flush()
Hibernate: /* insert entity.APerson */ insert into APerson (name, id) values (?, ?)
Hibernate: /* insert entity.APerson */ insert into APerson (name, id) values (?, ?)
Hibernate: /* insert entity.APerson */ insert into APerson (name, id) values (?, ?)
Hibernate: /* update entity.APerson */ update APerson set name=? where id=?
Hibernate: /* update entity.APerson */ update APerson set name=? where id=?
Hibernate: /* delete entity.APerson */ delete from APerson where id=?
Changes flushed
```

When are changes Flushed?

- There are actually 3 different scenarios under which Hibernate will commit all changes
  1. When an explicit session.flush() call is made
  2. When a transaction is committed
  3. Inside a transaction, before a query to the database is executed



refresh()

- The entity object will immediately be refreshed with the state currently held in the database
- Can be used to undo updates

```
session = sessionFactory.openSession();
tx = session.beginTransaction();

session.refresh(p1);

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

Although a transaction is not required, it is recommended to alway use a transaction

Immediately refreshes with state from the database, re-attaches if detached

Stale Cache – a Reason to Refresh

```
try {
   session = sessionFactory.openSession();
   tx = session.beginTransaction();

   Person p1 = (Person) session.get(Person.class, 1L);
   System.out.println(p1.getName());

   Thread.sleep(1000 * 60 * 5);
   System.out.println("Slept for 5 minutes");

   p1 = (Person) session.get(Person.class, 1L);
   System.out.println(p1.getName());

   session.refresh(p1);
   System.out.println(p1.getName());

   tx.commit();
} catch (Exception e) {
   ...
```

Retrieve a person entity and print its name (also puts it in the persistence cache)

Try to retrieve the same entity again 5 minutes later

Because of cache get() does not check the database!

Specifically refresh() to check the database for updates

Hibernate Output:

```
Hibernate: /* load entity.Person */ select person0_.id as id0_0_,
person0_.name as name0_0_ from Person person0_ where person0_.id=?
Frank Brown
Slept for 5 minutes
Frank Brown
Hibernate: /* load entity.Person */ select person0_.id as id0_0_,
person0_.name as name0_0_ from Person person0_ where person0_.id=?
Frank Upd.
```

No db check, just cache

refresh() checks the database and finds that it has been updated!

Entity Manager API

Method	Description
clear()	Empties persistence context, managed entities become detached
close()	Closes the entity manager, managed entities become detached
contains()	Checks if the instance is managed by this entity manager
detach()	Detaches the given entity
find()	Retrieves and entity from the db by primary key
flush()	Flushes all changes to the db
getReference()	Get and instance by Id, state may be lazily fetched
merge()	Merge the state of the given entity into the persistence context
persist()	Make the instance managed and persistent
refresh()	Update the state of the given entity from the db
remove()	Delete the given entity from the db

http://docs.oracle.com/javaee/6/api/javax/persistence/EntityManager.html

Active Learning

- What are the four different states of the entity lifecycle?

- What are the similarities and the differences between saveOrUpdate() and merge()?

Module Summary

- In this module we discussed the object persistence lifecycle and how to manage it
- We presented the different methods Hibernate provides on the Session object
- Although several methods provide similar functionality there are subtle differences
- These differences are important for enterprise applications and should be noted with care
- We also demonstrated Hibernates caching behavior and how it can be manipulated

Main Point

- Objects have a persistence lifecycle within Hibernate (different states that they can be in). The Hibernate Session API (and JPA EntityManager API) provide means to move objects from one state to another.
- *Science of Consciousness:* The TM Technique is a tool to move our mind into the transcendental state, the most fundamental and restful state of consciousness.