Queries

Queries

- So far we've used .find() or .getReference()
  - And then follow references to related objects
  - But what if you don't know an entity's ID?

- JPA offers **several ways** to query the DB
  - JPQL: Java Persistence Query Language (SQL like)
  - Criteria API: Create queries with Java objects
  - Stored Procedure Queries: executed stored procedures
  - Native Queries: Execute SQL and get objects

Our focus will be on JPQL

JQPL

- The Java Persistence Query Language (JPQL) is a standardization of the Hibernate Query language (HQL).
  - JPQL is a subset of HQL (HQL has a few extensions)

- JQPL syntax is **similar to SQL**, but OO:
  - Understands objects and attributes
  - Understands associations between objects
  - Understands inheritance and polymorphism

```
List<Account> accounts = em.createQuery("from Account a "
                        + "where a.class <> CheckingAccount "
                        + "and a.owner.firstName = 'Frank'", Account.class)
                        .getResultList();
```

CS544 EA

Hibernate

JPQL: Query Object

Creating and Executing

- You can **create** a typed or un-typed query:

```
// type safe version
TypedQuery<Person> query1 = em.createQuery("from Person", Person.class);
// Not type safe (original version)
Query query2 = em.createQuery("from Person");
```

Un-typed works same but IDE will complain about type safety

- You can **execute** it with .getResultList()

```
em.getTransaction().begin();

TypedQuery<Person> query = em.createQuery("from Person", Person.class)        List<Person> ppl =
query.getResultList();

em.getTransaction().commit();
```

Queries should always be inside a transaction

Named Queries

- You can create named queries
  - **Stored in meta-data** (annotations or XML)

```
@Entity
@NamedQueries({
        @NamedQuery(name = "Person.Everybody", query = "from Person")
})
public class Person {
        @Id
        @GeneratedValue
        private Long id;
        ...
```

  - **Retrieved** with .createNamedQuery()

```
TypedQuery<Person> q = em.createNamedQuery("Person.Everybody", Person.class);
List<Person> ppl = q.getResultList();
```

CS544 EA

Hibernate

JPQL: FROM clause

FROM Clause

- Simplest query only contains a FROM clause
  - The SELECT clause is optional (unlike SQL)

```
TypedQuery<Person> query1 = em.createQuery("from Person", Person.class);
```

- JPQL **keywords** are not case sensitive
  - FROM, from, FrOm, fRoM are all the same
- **Class and property** names are case sensitive
  - Person and person are two different classes!

Entity names, Property names

- JPQL Queries are based on Java:
  - Always use the **Entity** (class) name
  - Always use the **property** name

- JPQL never uses table or column names 

Reminder: @Entity and @Table

- @Entity(name="OtherName")
  - Changes table name
  - Changes **what the entity is called** (in a query)

- @Table(name="othername")
  - Only changes the table name
  - Inside a query the class name is used

Polymorphic Queries

- JPQL has **excellent polymorphism** support
  - Returns all Accounts regardless of sub-type

```
TypedQuery<Account> query = em.createQuery("from Account", Account.class);
List<Account> accounts = query.getResultList();
```

  - Returns all objects that implement the interface

```
TypedQuery<Serializable> query2 = em.createQuery("from Serializable", Serializable.class);
List<Serializable> serializables = query2.getResultList();
```

  - Returns every (entity) object in the database!

```
TypedQuery<Object> query3 = em.createQuery("from Object", Object.class);
List<Object> objects = query3.getResultList();
```

Aliases

- Alias entity names for ease of reference
  - Just like SQL aliases for table names

```
TypedQuery<Person> query = em.createQuery("from Person as p", Person.class);
```

  - Just like SQL the 'as' keyword is optional

```
TypedQuery<Person> query = em.createQuery("from Person p", Person.class);
```

Pagination

- JQPL has built-in pagination support
  - Selects a **part of a bigger result set**
  - Does not necessarily speed things up

```
em.getTransaction().begin();
TypedQuery<Person> pplQuery = em.createQuery("from Person", Person.class);

pplQuery.setFirstResult(0);
pplQuery.setMaxResults(50);

List<Person> ppl = pplQuery.getResultList();
for (Person p : ppl) {
        System.out.println(p);
}
em.getTransaction().commit();
```

Zero based index - 0 is first item

Page size

Returns first 50 people

Order By

- The 'order by' clause **sorts the result** by that

```
TypedQuery<Person> query1 =
      em.createQuery("from Person p order by p.lastName", Person.class);
```

By default sorted ascending (ASC)

- The **ASC or DESC keywords** can be added to sort in ascending or descending order

```
TypedQuery<Person> query1
      = em.createQuery("from Person p order by p.lastName DESC", Person.class);
```

Sorted Descending

CS544 EA

Hibernate

JPQL: WHERE clause

WHERE Clause

- WHERE lets you add constraints to the result
  - Refining which rows end up in the list

```
TypedQuery<Person> query
      = em.createQuery("from Person p where p.lastName = 'Johnson'", Person.class);
```

Selects all the people whose last name is Johnson

  - JPQL supports the same expressions as SQL
    - As well as some OO specific expressions

People whose first account has a balance is > 100

```
TypedQuery<Person> query
      = em.createQuery("from Person p where p.accounts[0].balance > 100",
Person.class);
```

JPQL Expressions

Type	Operators
Literals	‘string’, 128, 4.5E+3, ‘yyyy-mm-dd hh:mm:ss’
Arithmetic	+, -, *, /
Comparison	=, <>, >=, <=, !=, like
Logical	and, or, not
Grouping	(,)
Concatenation	||
Values	in, not in, between, is null, is not null, is empty, is not empty
Case	case … when … then … else … end, case when … then … else … end

JPQL Functions

- JPQL also provides several built-in functions
  - These work regardless of underlying DB

Type	Functions
Temporal	current_date(), current_time(), current_timestamp(), second(…), minute(…), hour(…), day(…), month(…), year(…)
String	concat(… , …), substring(), trim(), lower(), upper(), length(), str()
Collection	Index(), size(), minindex(), maxindex()

Indexed Collection Expressions

- [] can be used to access indexed collections
  - Only: **Map** and **@OrderColumn List**

```
TypedQuery<Person> query
      = em.createQuery("from Person p where p.accounts[0].balance > 100", Person.class);
```

Account list has to have @OrderColumn

```
TypedQuery<Person> query
      = em.createQuery("from Person p where p.pets['mimi'].species = 'Cat'", Person.class);
```

Map with String key

Query Parameters

- **Never concatenate** JPQL Strings!
  - Opens the door for **JPQL (SQL) injection**
  - Also makes your query messy

```
TypedQuery<Person> pplQuery
      = em.createQuery("from Person p where p.firstName = '" + firstName + "'", Person.class);
```

- Use named parameters instead:

```
TypedQuery<Person> pplQuery
      = em.createQuery("from Person p where p.firstName = :first", Person.class);
pplQuery.setParameter("first", firstName);
```

Separates instruction and data

Placeholder

Safely replace placeholder

Temporal Parameters

- Specify the **exact type** for temporal types
  - Using either java.util.Calendar or java.util.Date
  - Java 8 LocalDate is supported

```
TypedQuery<Person> q
      = em.createQuery("from Person p where p.birthDate < :date", Person.class);
Calendar cal = Calendar.getInstance();
cal.set(2000, 0, 1); // 2000-01-01
q.setParameter("date", cal, TemporalType.DATE);
```

Overloaded to receive java.util.Date or java.util.Calendar

Specify the temporal type

Positional Parameters

- Possible but **not recommended**
  - Uses ? as placeholder instead of unique names
  - Easily breaks if you add more parameters later
  - A lot less self documenting!

```
TypedQuery<Person> q
        = em.createQuery("from Person where firstName = ? and lastName = ?", Person.class);
q.setParameter(0, "Jackson");
q.setParameter(1, "Jarvis");

List<Person> ppl = q.getResultList();
```

What gets set?

.singleResult()

- Returns a single object instead of a List
  - Make sure there is **exactly one** result!
  - NoResultException, NonUniqueResultException

```
TypedQuery<Person> q = em.createQuery("from Person where id = 1", Person.class);
Person p = q.getSingleResult();
```

Guaranteed to be single result

```
TypedQuery<Person> q2 = em.createQuery("from Person", Person.class);
q2.setMaxResults(1);
Person p2 = q2.getSingleResult();
```

Guaranteed to be single result

Special Attribute: .id

- Your @Id property can be referred to as .id
  - **Even if it's called something else**
  - Except if another property (not @Id) is called id

```
TypedQuery<Employee> q =
      em.createQuery("from Employee where id = 1", Employee.class);
Employee e = q.getSingleResult();
```

```
@Entity
public class Employee {
      @Id
      @GeneratedValue
      private Long employeeId;
      private String firstName;
      private String lastName;
```

Special Attribute: .class

- You can **compare the class name** with .class
  - To restrict to a certain class with =
  - Or remove a certain class with != / <>

```
List<Account> accounts = em.createQuery("from Account a "
            + "where a.class <> CheckingAccount "
            + "and a.owner.firstName = 'Frank'", Account.class)
            .getResultList();
```

- The type() function does the same

```
List<Account> accounts = em.createQuery("from Account a "
            + "where type(a) = CheckingAccount "
            + "and a.owner.firstName = 'Frank'", Account.class)
            .getResultList();
```

CS544 EA

Hibernate

JPQL: Joins

Joins

- There are 2 types of joins:
  - Explicit joins that use the JOIN keyword

```
TypedQuery<Person> q = em.createQuery("select p from Person as p  "
                        + " JOIN p.address as a where a.city = 'Fairfield'", Person.class);
```

  - Implicit joins that don't use JOIN
    - Instead follow references by using the . operator

```
TypedQuery<Person> q = em.createQuery("from Person as p "
                        + " where p.address.city = 'Fairfield'", Person.class);
```

Explicit Joins

- Syntax for explicit join is:
  - JOIN table.property [as] alias
  - Alias can then be used inside WHERE clause

```
TypedQuery<Person> q = em.createQuery("select p from Person as p  "
                        + " JOIN p.address as a where a.city = 'Fairfield'", Person.class);
```

- Explicit join expands the result set
  - Have to **use a SELECT** clause to bring it back to one entity

Implicit Joins

- Implicit follows reference
  - **Only works for references**
  - Does not expand result set (no need for select)

@OneToOne
@ManyToOne

```
TypedQuery<Person> q = em.createQuery("from Person as p "
                        + " where p.address.city = 'Fairfield'", Person.class);
```

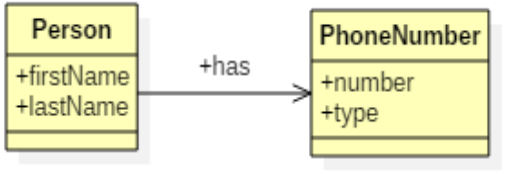
- You cannot implicit join a collection
  - Using [] with an indexed collection turns the collection into a reference

@OneToMany
@ManyToMany

Joining a Collection

- Joining a collection requires:
  - **Explicit join**, therefore also a **Select clause**
  - And the **Distinct keyword**

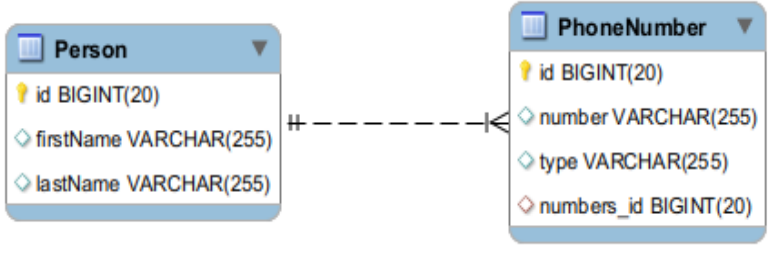
- First an example with just the explicit join
  - No Select
  - No Distinct

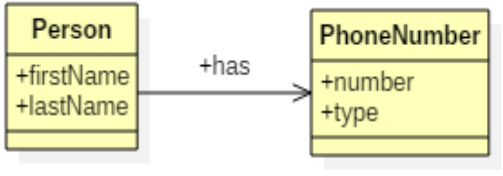


Mapped Domain

```java
@Entity
public class Person {
    @Id
    @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    @OneToMany
    @JoinColumn
    private List<PhoneNumber> numbers
        = new ArrayList<>();
```

```java
@Entity
public class PhoneNumber {
    @Id
    @GeneratedValue
    private Long id;
    private String number;
    private String type;
```





id	firstName	lastName
1	Edward	Towers
2	John	Brown

id	number	type	numbers_id
1	641-472-1234	Home	1
2	641-919-5432	Mobile	1
3	641-233-9876	Mobile	2
4	641-469-4567	Home	2

Joining a Collection Without Select or Distinct

- **Without Select** result contains **2 entities**

```
TypedQuery<Object[]> q = em.createQuery("from Person p "
        + "join p.numbers as n "
        + "where n.number like '641%'", Object[].class);
```

```
TypedQuery<Object[]> q = em.createQuery("from Person p "
        + "join p.numbers as n "
        + "where n.number like '641%'", Person.class);
```

Person	PhoneNumber
Edward, Towers	home, 641-472-1234
Edward, Towers	mobile, 641-919-5432
John, Brown	mobile, 641-233-9876
John, Brown	home, 641 469-4567

Joining Collection Without Distinct

- The select gives us a single entity
  - **Still have duplicates** because of join!

```
TypedQuery<Person> q = em.createQuery("select p from Person p "
            + "join p.numbers as n "
            + "where n.number like '641%'", Person.class);
```

Person
Edward, Towers
Edward, Towers
John, Brown
John, Brown

Distinct

- Distinct **removes duplicate** rows

```
TypedQuery<Person> q = em.createQuery("select distinct p from Person p "
            + "join p.numbers as n "
            + "where n.number like '641%'", Person.class);
```

- Joining a collection therefore requires:
  - Explicit **Join** with a **Select**
  - And the **Distinct** keyword

Person
Edward, Towers
John, Brown

Inner Joins

- All joins so far have been inner joins
  - If **one side is null there is no join**, no result row
- Outer Joins are also possible
  - Allow one of the sides to be null
  - Includes data that could not join

Person	Address
Edward, Towers	New York, New York
John, Brown	
Alice, Doe	Los Angeles, California

John Brown included even though no Address

Left Outer Join

- Left Outer Join means:
  - Data on the left (where we start) has to be there
  - Data that is joined (on the right) can be null

Right Outer Join
Not supported by JPA

```
TypedQuery<Person> q = em.createQuery("select distinct p from Person p "
        + "left outer join p.address a ", Person.class);
```

Do not need both
"left" and "outer"

Can say: left join
Can say: outer join

Person	Address
Edward, Towers	New York, New York
John, Brown	
Alice, Doe	Los Angeles, California

Join Fetch

- Join Fetch lets you Join so that:
  - Related entities are added to the EM Cache
  - Without adding them to the resultset

- Useful for avoiding the N+1 problem
  - We will talk about this more during optimization

Join Fetch

No SELECT clause needed
joined entities not added to ResultSet

```
TypedQuery<Customer> query = em.createQuery(
                "from Customer c "
                + "JOIN FETCH c.address a "
                + "JOIN FETCH c.books b "
                + "JOIN FETCH b.author "
                + "WHERE c.firstName like :name",
Customer.class);
query.setParameter("name", "J%");
List<Customer> customers = query.getResultList();
System.out.println(customers.size());
```

Important: don't join (fetch)
multiple collections!

This creates a Cartesian Product
(see optimization)

```
Hibernate:
   select
      customer0_.id as id1_3_0_,
      address1_.id as id1_0_1_,
      books2_.id as id1_2_2_,
      author3_.id as id1_1_3_,
      customer0_.address_id as address_4_3_0_,
      customer0_.firstName as firstNam2_3_0_,
      customer0_.lastName as lastName3_3_0_,
      address1_.city as city2_0_1_,
      address1_.state as state3_0_1_,
      books2_.author_id as author_i3_2_2_,
      books2_.name as name2_2_2_,
      books2_.books_id as books_id4_2_0__,
      books2_.id as id1_2_0__,
      author3_.name as name2_1_3_
   from
      Customer customer0_
   inner join
      Address address1_
         on customer0_.address_id=address1_.id
   inner join
      Book books2_
         on customer0_.id=books2_.books_id
   inner join
      Author author3_
         on books2_.author_id=author3_.id
   where
      customer0_.firstName like ?
1
```

CS544 EA

Hibernate

JPQL: SELECT clause

Select Clause

A result can contain:
Entities
Properties
A mix of the two

- SELECT specifies **which entities or properties** the query should return
  - We've already seen selecting an entity

```
TypedQuery<Person> q = em.createQuery("select distinct p from Person p "
        + "left outer join p.address a ", Person.class);
```

  - It's easy to select a single property

```
TypedQuery<String> q =
    em.createQuery("select p.firstName from Person p ", String.class);
```

Selects the firstName of all Person Objects

Multiple Items

- You can specify more than one entity / property
  - By default these are **returned as an Object[]**

```
TypedQuery<Object[]> q = em.createQuery(
                "select person, pet.species, adr.city "
                + "from Pet pet join pet.owner person, "
                + "join person.address adr", Object[].class);
List<Object[]> result = q.getResultList();

Person p = null; String petType = null; String city = null;
for (Object[] item : result) {
  p = (Person) item[0]; petType = (String) item[1]; city = (String) item[2];

  System.out.println(p.getFirstName() + " " + p.getLastName()
       + " owns a " + petType + " in " + city);
}
```

List

- Use **new list()** in JPQL to select as List

```
Query q = em.createQuery(
                "select new list(person, pet.species, adr.city) "
                + "from Pet pet join pet.owner person, "
                + "join person.address adr");
List<List<Object>> result = q.getResultList();


Person p = null; String petType = null; String city = null;
for (List<Object> item : result) {
        p = (Person) item.get(0);
        petType = (String) item.get(1);
        city = (String) item.get(2);

        System.out.println(p.getFirstName() + " " + p.getLastName()
      + " owns a " + petType + " in " + city);
}
```

I have not been able to make this work with a TypedQuery

Map

- **new Map()** selects as Map<String, Object>
  - Requires you to give aliases to each element
  - Alias will be used as Key in the map

```
Query query = em.createQuery(
  "select new map(p as person, sum(a.balance) as liquid) "
  + "from Person p join p.accounts a group by p.id ");

List<Map<String,Object>> items = query.getResultList();

Person p = null; Double liquid = null;
for (Map<String,Object> item : items) {
  p = (Person)item.get("person");
  liquid = (Double)item.get("liquid");

  System.out.println(p.getFirstName() + " " + p.getLastName()
     + "'s liquid assets: " + liquid);
}
```

New Object

- Results can be of **any object**
  - Do need constructor for what you provide
  - Ideal for constructing DTOs

```
TypedQuery<Home> query = em.createQuery(
        "select new hibernate06.Home(p, a) "
        + "from Person p " + "join p.address a ", Home.class);
List<Home> homes = query.getResultList();

Person p = null;
Address a = null;
for (Home home : homes) {
        p = home.getPerson();
        a = home.getAddress();

        System.out.println(p.getFirstName()
        + " " + p.getLastName()
        + " has a home in " + a.getCity());
}
```

Needs fully-qualified class name

No need for Casting!

```
public class Home {
        private Person person;
        private Address address;

        public Home(Person p, Address a) {
                this.person = p;
                this.address = a;
        }
```

Not an Entity just some class

Aggregates

- JPQL also has the typical **aggregate** functions
  - avg(…), sum(…), min(…), max(…)
  - count(*), count(…), count(distinct …), count(all …)

```
Query query = em.createQuery(
        "select new map(p as person, sum(a.balance) as liquid) "
        + "from Person p join p.accounts a group by p.id "
        + "having liquid > 100 ");
```

Sum of the balance of all acounts related to one Person and having at least 100

- **Group By** clause specifies groups to aggregate
- The **having** clause can filter groups

CS544 EA

Hibernate

JPQL: Bulk Operations

Bulk Update and Delete

- JQPL also supports bulk update and delete
  - Similar to SQL DML features

Bulk update:

```
Query query = em.createQuery("update Account set balance = balance - :fee");
query.setParameter("fee", 5.0);
int updated = query.executeUpdate();
```

Bulk delete:

```
Query query = em.createQuery("delete Book where publish_date < :date");
DateFormat df = DateFormat.getDateInstance();
query.setParameter("date", df.parse("01/01/2010"), TemporalType.DATE);
int deleted = query.executeUpdate();
```

Import.sql

- Hibernate will **automatically execute** import.sql
  - Placed on the classpath (src/main/resources)
  - Typically contains INSERT statements

- You can also specify other files to execute
  - Inside persistence.xml

```
<property name="hibernate.hbm2ddl.import_files" value="test.sql" />
```

CS544 EA

Hibernate

Alternate Query Types

Constraints

- Uses Java Objects to create a query
  - Use this **instead of concatenating JPQL strings**!
  - Has all the same features as JPQL

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Person> cq = cb.createQuery(Person.class);
Root<Person> person = cq.from(Person.class);
Join<Person, Address> address = person.join("address");
cq.where(
                cb.and(
                                cb.equal(person.get("firstName"), "John"),
                                cb.equal(address.get("city"), "Fairfield")
                        )
        );

TypedQuery<Person> query = em.createQuery(cq);
List<Person> ppl = query.getResultList();
```

Same as JPQL:
from Person p where
p.firstName = 'John' and
p.address.city = 'Fairfield'

For more info see: https://docs.oracle.com/javaee/6/tutorial/doc/gjrij.html

Stored Procedure Queries

- To execute a MySQL **stored procedure**:

```
create procedure calculate(
        IN x int, IN y int, OUT sum int, OUT prod int)
begin
        select x + y into sum;
        select x * y into prod;
end
```

```
StoredProcedureQuery query = em.createStoredProcedureQuery("calculate");

query.registerStoredProcedureParameter("x", Integer.class, ParameterMode.IN);
query.registerStoredProcedureParameter("y", Integer.class, ParameterMode.IN);
query.registerStoredProcedureParameter("sum", Integer.class, ParameterMode.OUT);
query.registerStoredProcedureParameter("prod", Integer.class, ParameterMode.OUT);

query.setParameter("x", 2);
query.setParameter("y", 3);
query.execute();

int sum = (int) query.getOutputParameterValue("sum");
int prod = (int) query.getOutputParameterValue("prod");
System.out.println("sum: " + sum + " prod: " + prod);
```

Native Queries

- Write SQL and receive Objects

```
Query query = em.createNativeQuery("SELECT * FROM Person", Person.class);
List<Person> ppl = query.getResultList();
ppl.forEach(x -> System.out.println(x.getFirstName()));
```

- Without the second parameter
  - native query returns Object[]

```
Query query = em.createNativeQuery("SELECT * FROM Person");
List<Object[]> ppl = query.getResultList();
ppl.forEach(x -> System.out.println(x[1]));
```

Summary

- JPQL is very similar to SQL
  - Uses class and property names
  - From clause is minimal requirement
  - Where clause can be used to refine result
  - Joins can be used to connect to other entities
  - Select can be used to certain entities / properties
  - It's possible to do bulk updates and deletes

- Criteria, StoredProcedure, and Native queries exist