

CS544

LESSON 6

JPA QUERIES

Query techniques

- Query creation from method names
- Using @Query
- Using named queries
- Using native queries
- Using specifications

METHOD BASED QUERY

Query creation from method names

```
public interface UserRepository extends JpaRepository<User, Long> {  
    User findByEmailAddress(String emailAddress);  
    List<User> findByLastname(String lastname);  
    List<User> findByEmailAddressAndLastname(String emailAddress, String lastname);  
}
```

Query methods rules

- The name of our query method must start with one of the following prefixes:
 - find...By, read...By, query...By, count...By, and get...By.
- If we want to specify the selected property, we must add the name of the property before the first By word.
 - findTitleBy
- If we want to limit the number of returned query results, we can add the First or the Top keyword before the first By word.
 - If we want to get more than one result, we have to append the optional numeric value to the First and the Top keywords.
 - findTopBy, findTop1By, findFirstBy, findFirst2By
- If we want to select unique results, we have to add the Distinct keyword before the first By word.
 - findTitleDistinctBy, findDistinctTitleBy
- We must add the search criteria of our query method after the first By word.
 - findByEmailAddressAndLastname
- If our query method specifies x search conditions, we must add x method parameters to it.
 - The number of method parameters must be equal than the number of search conditions.
 - The method parameters must be given in the same order than the search conditions.

Query method examples

```
Dog findById(Long id);  
Dog readById(Long id);  
Dog getId(Long id);  
Dog queryById(Long id);
```

These are all
the same

```
Integer countByName(String name);
```

```
List<Dog> findByAgeAndHeight(Integer age, double height);  
List<Dog> findByAgeAndNameAndColor(Integer age, String name, String color);  
List<Dog> findByNameOrAge(String name, Integer age);  
List<Dog> findByNameIgnoreCaseAndColor(String name, String color);
```

```
Dog findFirstByName(String name);  
Dog findTopByName(String name);
```

```
List<Dog> findTop10ByColor(String color);
```

Query method examples

```
Dog findFirstByName(String name);
```

```
Dog findTopByName(String name);
```

These 2 are
the same

```
List<Dog> findTop10ByColor(String color);
```

```
List<Dog> findByNameContaining(String subName);
```

```
List<Dog> findByNameStartingWith(String subName);
```

```
List<Dog> findByHeightLessThan(double height);
```

```
List<Dog> findByAgeLessThanOrHeightGreaterThan(Integer age, double height);
```

```
List<Dog> findByAgeGreaterThanAndAgeLessThan(Integer ageStart, Integer ageEnd);
```

```
List<Dog> findByAgeGreaterThanEqual(Integer age);
```

```
List<Dog> findByDateOfBirthBetween(Date start, Date end);
```

Supported keywords

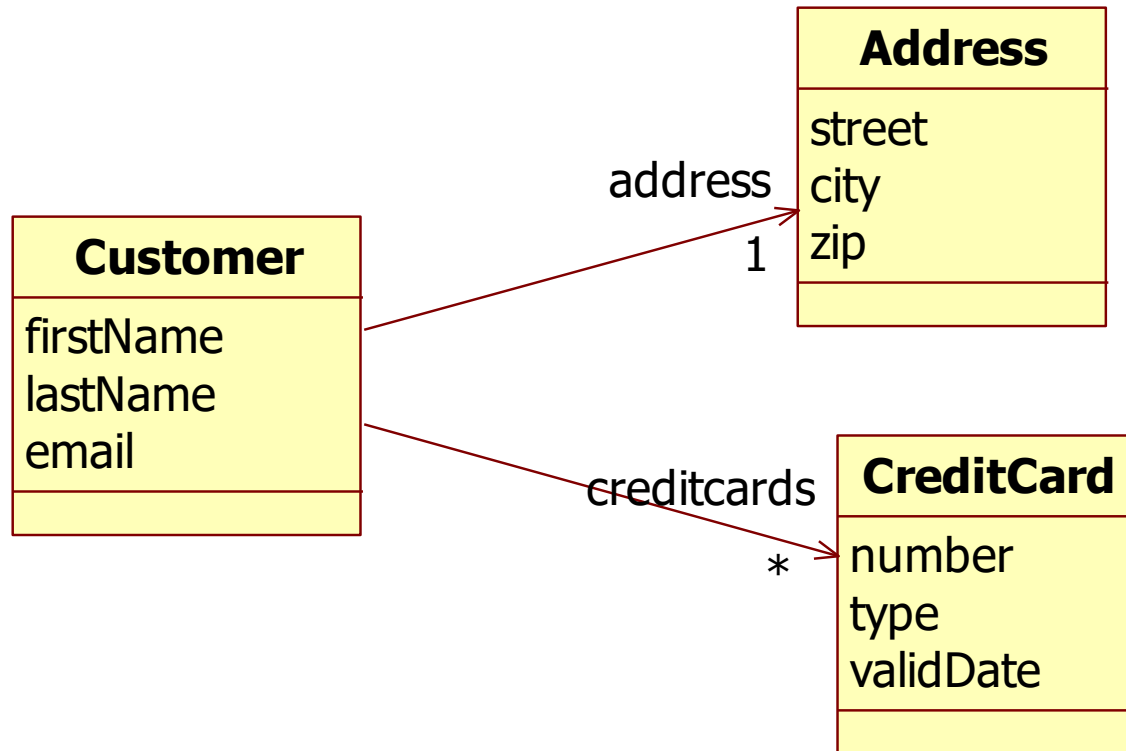
| Keyword | Sample | JSQL snippet |
|-------------------|--|---|
| And | <code>findByLastnameAndFirstname</code> | <code>... where x.lastname = ?1 and x.firstname = ?2</code> |
| Or | <code>findByLastnameOrFirstname</code> | <code>... where x.lastname = ?1 or x.firstname = ?2</code> |
| Between | <code>findByStartDateBetween</code> | <code>... where x.startDate between 1? and ?2</code> |
| LessThan | <code>findByAgeLessThan</code> | <code>... where x.age < ?1</code> |
| GreaterThan | <code>findByAgeGreaterThan</code> | <code>... where x.age > ?1</code> |
| After | <code>findByStartDateAfter</code> | <code>... where x.startDate > ?1</code> |
| Before | <code>findByStartDateBefore</code> | <code>... where x.startDate < ?1</code> |
| IsNull | <code>findByAgeIsNull</code> | <code>... where x.age is null</code> |
| IsNotNull,NotNull | <code>findByAge(Is)NotNull</code> | <code>... where x.age not null</code> |
| Like | <code>findByFirstnameLike</code> | <code>... where x.firstname like ?1</code> |
| NotLike | <code>findByFirstnameNotLike</code> | <code>... where x.firstname not like ?1</code> |
| StartingWith | <code>findByFirstnameStartingWith</code> | <code>... where x.firstname like ?1 (parameter bound with appended %)</code> |
| EndingWith | <code>findByFirstnameEndingWith</code> | <code>... where x.firstname like ?1 (parameter bound with prepended %)</code> |
| Containing | <code>findByFirstnameContaining</code> | <code>... where x.firstname like ?1 (parameter bound wrapped in %)</code> |
| OrderBy | <code>findByAgeOrderByLastnameDesc</code> | <code>... where x.age = ?1 order by x.lastname desc</code> |
| Not | <code>findByLastnameNot</code> | <code>... where x.lastname <> ?1</code> |
| In | <code>findByAgeIn(Collection<Age> ages)</code> | <code>... where x.age in ?1</code> |
| NotIn | <code>findByAgeNotIn(Collection<Age> age)</code> | <code>... where x.age not in ?1</code> |
| True | <code>findByActiveTrue()</code> | <code>... where x.active = true</code> |
| False | <code>findByActiveFalse()</code> | <code>... where x.active = false</code> |

Query method examples

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {  
  
    List<Customer> findByLastName(String lastName);  
    Optional<Customer> findByEmail(String email);  
    Customer findByFirstNameAndLastName(String firstName, String lastName);  
    List<Customer> findFirst2By();  
}
```

```
Optional<Customer> custopt = customerrepository.findByEmail("dpalmer@gmail.com");  
if (custopt.isPresent()) {  
    Customer thecustomer = custopt.get();  
    System.out.println(thecustomer);  
}  
  
Customer cust = customerrepository.findByFirstNameAndLastName("Chloe", "O'Brian");  
System.out.println(cust);  
  
for (Customer cust2 : customerrepository.findFirst2By()) {  
    System.out.println(cust2);  
}
```

Query methods example



Query methods example

```
@Entity
public class Customer {
    @Id
    @GeneratedValue
    private long id;
    private String firstname;
    private String lastname;
```

```
@OneToMany (cascade={CascadeType.PERSIST})
private Collection<CreditCard> creditcards=new ArrayList<CreditCard>();
```

```
@ManyToOne(cascade={CascadeType.PERSIST})
private Address address;
```

```
@Entity
public class CreditCard {
    @Id
    @GeneratedValue
    private int id;
    private String number;
    private String name;
    private Date expiration;
```

```
@Entity
public class Address {
    @Id
    @GeneratedValue
    private long id;
    private String street;
    private String city;
    private String zip;
```

Query method example

```
public interface CreditCardRepository extends JpaRepository<CreditCard, Long> {  
    CreditCard findByNumber(String number);  
    CreditCard findByNumberAndName(String number, String name);  
    List<CreditCard> findByName(String name);  
}
```

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {  
    List<Customer> findByFirstname(String name);  
    List<Customer> findByAddressZip(String zipcode);  
    List<Customer> findByCreditcardsNumber(String zipcode);  
}
```

Pagination

```
public interface ProductRepository extends JpaRepository<Product, Long>, PagingAndSortingRepository<Product, Long>
{
    List<Product> findAllByCategory(String category, Pageable pageable);
}
```

@Entity

```
public class Product {
    @Id
    @GeneratedValue
    private long id;
    private String name;
    private double price;
    private String category;
    private String supplier;
```

This interface adds the method **findAll(Pageable ...)**

Pagination and sorting

```
System.out.println("Get the first 10 products");
Page<Product> productPage= productRepository.findAll(PageRequest.of(0, 9));
productPage.forEach(p -> System.out.println(p));
System.out.println("Total number of elements = "+productPage.getTotalElements());
System.out.println("Total number of pages = "+productPage.getTotalPages());
System.out.println("-----");
```

```
System.out.println("Get the first 5 products from category phones");
List<Product> products= (List<Product>) productRepository.findAllByCategory("phones", PageRequest.of(0, 5));
products.stream().forEach(p -> System.out.println(p));
System.out.println("-----");
```

```
System.out.println("Get the first 10 products but sort by name");
productPage= productRepository.findAll(PageRequest.of(0, 9, Sort.by("name")));
productPage.forEach(p -> System.out.println(p));
System.out.println("-----");
```

Add sorting to the query

Main point

- One can create queries in the repository by defining methods according to a certain convention in the repository interface.

Science of Consciousness: Through the daily practice of transcending one's thoughts get more powerful which leads to more fulfillment. Thoughts leads to Action, Action leads to Achievement, Achievement leads to Fulfilment

@QUERY

Using @Query

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {  
  
    List<Customer> findByLastName(String lastName);  
  
    @Query("select c from Customer c where c.email = ?1")  
    Customer findByEmail(String email);  
}
```

The method name does not have any significance

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {  
  
    List<Customer> findByLastName(String lastName);  
  
    @Query("select c from Customer c where c.email = :email ")  
    Customer findByEmail(@Param("email") String email);  
}
```

Named parameter

JPQL examples



```
select b from Book b where b.price > 15
```

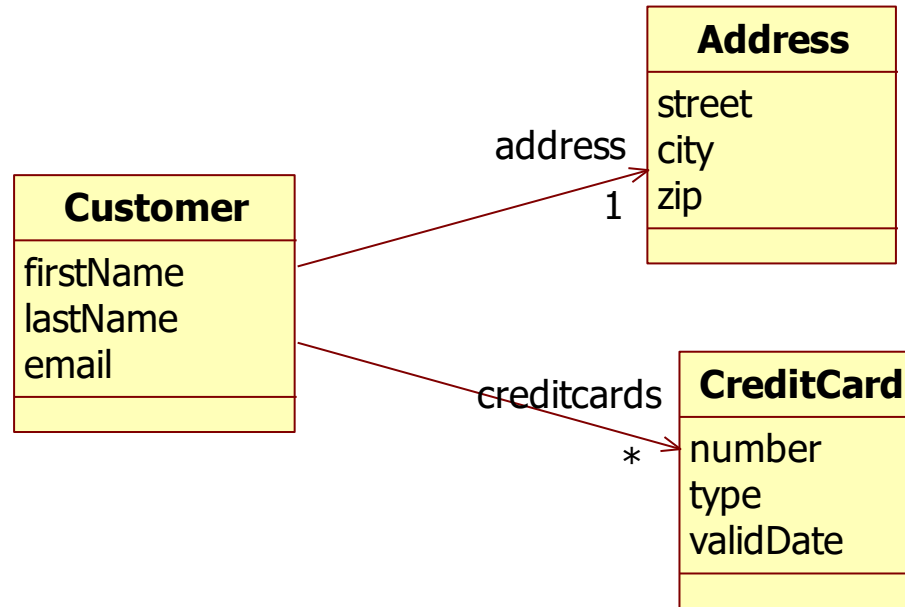
```
select b.title from Book b
```

```
select b from Book b where b.price between 10 and 15
```

```
select b from Book b where b.title like '%love%'
```

```
select b from Book b order by b.price asc
```

JPQL examples



```
select c from Customer c where c.address.city = 'Boston'
```

```
select c from Customer c JOIN c.creditcards cr where cr.number= '127865439867'
```

@Query: finding a property

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {  
  
    @Query("select c.lastName from Customer c where c.firstName= :firstName")  
    String findLastNameByFirstName(@Param("firstName") String firstName);  
  
}
```

NAMED QUERY

Named query

Named query

```
@Entity
@NamedQuery(name="Employee.findByFirstName", query="select e from Employee e where
e.firstname = :name")
public class Employee {

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

Must be a unique
name

Multiple named queries



```
@Entity
@NamedQueries({
    @NamedQuery(name="Employee.findByFirstName", query=" select e from Employee e where
e.firstname = :name"),
    @NamedQuery(name="Employee.findByLastName", query=" select e from Employee e where
e.lastname = :name")
})
public class Employee {

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

Using named queries

```
@Entity
@NamedQuery(name = "Customer.findByEmail", query = "select c from Customer c
where c.email = ?1")
public class Customer {
    ...
}
```

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {

    Customer findByEmail(String email);
}
```


Named queries with named parameters

```
@Entity
@NamedQuery(name = "Customer.findByEmail", query = "select c from Customer c
where c.email = :email")
public class Customer {
```

Named parameter

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {

    Customer findByEmail(@Param("email") String email);
}
```

@Param

NATIVE QUERY

Using native queries

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {  
  
    List<Customer> findByLastName(String lastName);  
  
    @Query(value = "SELECT * FROM customer WHERE EMAIL = ?1", nativeQuery = true)  
    Customer findByEmail(String email);  
}
```

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {  
  
    List<Customer> findByLastName(String lastName);  
  
    @Query(value = "SELECT * FROM customer WHERE EMAIL = :email", nativeQuery = true)  
    Customer findByEmail(@Param("email") String email);  
}
```

Named parameter

SPECIFICATION

Specification

```
public interface BookRepository extends JpaRepository<Book, Integer>, JpaSpecificationExecutor<Book> {  
}
```

This interface adds the method
findAll(Specification ...)

```
public class BookSpecifications {  
    public static Specification<Book> hasInTitle(String title) {  
        return (root, query, criteriaBuilder) -> criteriaBuilder.like(root.get("title"), "%" + title + "%");  
    }  
}
```

Specification

```
Specification<Book> titlespec = BookSpecifications.hasInTitle("Harry");  
List<Book> harryBooks = bookrepository.findAll(titlespec);  
System.out.println("----- All books with Harry in the title:");  
harryBooks.forEach(b-> System.out.println(b));
```

Code using the
specification

Multiple Specifications

```
public class BookSpecifications {  
    public static Specification<Book> hasInTitle(String title) {  
        return (root, query, criteriaBuilder) -> criteriaBuilder.like(root.get("title"), "%" + title + "%");  
  
        public static Specification<Book> hasAuthorWithFirstName(String authorFirstName) {  
            return (root, query, criteriaBuilder) -> criteriaBuilder.equal(root.get("authors").get("firstname"),  
authorFirstName);  
        }  
  
        public static Specification<Book> isInCategory(String category) {  
            return (root, query, criteriaBuilder) -> criteriaBuilder.equal(root.get("category"), category);  
        }  
    }  
}
```

Using the Specifications

```
Specification<Book> categoryspec = BookSpecifications.isInCategory("Programming");  
List<Book> programmingBooks = bookrepository.findAll(categoryspec);  
System.out.println("----- All programming books:");  
programmingBooks.forEach(b-> System.out.println(b));
```

```
Specification<Book> titlespec = BookSpecifications.hasInTitle("Harry");  
List<Book> harryBooks = bookrepository.findAll(titlespec);  
System.out.println("----- All books with Harry in the title:");  
harryBooks.forEach(b-> System.out.println(b));
```

```
Specification<Book> authorWithFirstNamespec = BookSpecifications.hasAuthorWithFirstName("John");  
List<Book> johnBooks = bookrepository.findAll(authorWithFirstNamespec);  
System.out.println("----- All books from Author with first name John :");  
johnBooks.forEach(b-> System.out.println(b));
```

Multiple Specifications



```
public static Specification<Book> hasAuthorWithFirstName(String authorFirstName) {  
    return (root, query, criteriaBuilder) -> criteriaBuilder.equal(root.get("authors").get("firstname"), authorFirstName);  
}
```

```
public static Specification<Book> withPriceGreatherThan(double price) {  
    return (root, query, criteriaBuilder) -> criteriaBuilder.greaterThan(root.get("price"), price);  
}
```


Combining Specifications

```
Specification<Book> authorWithFirstNamespec = BookSpecifications.hasAuthorWithFirstName("John");  
List<Book> johnBooks = bookrepository.findAll(authorWithFirstNamespec);  
System.out.println("----- All books from Author with first name John :");  
johnBooks.forEach(b-> System.out.println(b));
```

```
Specification<Book> priceGreatherThanspec = BookSpecifications.withPriceGreatherThan(15.0);  
List<Book> pricyBooks = bookrepository.findAll(priceGreatherThanspec);  
System.out.println("----- All books with price > 15.0 :");  
pricyBooks.forEach(b-> System.out.println(b));
```

```
List<Book> pricyBooksFromAuthor =  
bookrepository.findAll(Specification.where(authorWithFirstNamespec).and(priceGreatherThanspec));  
System.out.println("----- All books from John with price > 15.0 :");  
pricyBooksFromAuthor.forEach(b-> System.out.println(b));
```

Combining 2 specifications

Allows you to create a dynamic query

OPTIMIZATION

Bulk update and delete



```
update Customer c set c.status = 'Gold' where c.orders > :numberoforders
```

```
delete Customer c where c.status = :status
```

Modifying statements

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {
```

```
    List<Customer> findByLastName(String lastName);
```

@Modifying for update, delete statements

```
@Modifying  
@Transactional
```

Modifying method must be transactional

```
@Query("update Customer cust set cust.firstName = :firstname where  
        cust.lastName = :lastname")
```

```
int setFixedFirstnameFor(@Param("firstname") String firstName,  
                          @Param("lastname") String lastName);
```

```
}
```

Entities

```
@Entity
public class Customer {
    @Id
    @GeneratedValue
    private long id;
    private String firstname;
    private String lastname;

    @ManyToOne(cascade={CascadeType.PERSIST})
    private Address address;
```

```
@Entity
public class Address {
    @Id
    @GeneratedValue
    private long id;
    private String street;
    private String city;
    private String zip;
```

ManyToOne

```
public void run(String... args) throws Exception {  
    Address a1 = new Address("mainstreet 1", "Chicago", "58902");  
    Customer c1 = new Customer("Frank", "Brown");  
    c1.setAddress(a1);  
    customerRepository.save(c1);  
  
    Address a2 = new Address("mainstreet 4", "New York", "21345");  
    Customer c2 = new Customer("Frank", "Johnson");  
    c2.setAddress(a2);  
    customerRepository.save(c2);  
  
    List<Customer> customerList = customerRepository.findByFirstname("Frank");  
    customerList.stream().forEach(c -> System.out.println(c));  
}
```

```
List<Customer> findByFirstname(String name);
```

ManyToOne

```
Hibernate: select customer0_.id as id1_2_, customer0_.address_id as address_4_2_,
customer0_.firstname as firstnam2_2_, customer0_.lastname as lastname3_2_ from
customer customer0_ where customer0_.firstname=?
Hibernate: select address0_.id as id1_0_0_, address0_.city as city2_0_0_,
address0_.street as street3_0_0_, address0_.zip as zip4_0_0_ from address address0_
where address0_.id=?
Hibernate: select address0_.id as id1_0_0_, address0_.city as city2_0_0_,
address0_.street as street3_0_0_, address0_.zip as zip4_0_0_ from address address0_
where address0_.id=?
Customer{id=1, firstname='Frank', lastname='Brown', address=Address{id=2,
street='mainstreet 1', city='Chicago', zip='58902'}}
Customer{id=3, firstname='Frank', lastname='Johnson', address=Address{id=4,
street='mainstreet 4', city='New York', zip='21345'}}
```

Load all
customers

Load address
for customer1

Load address
for customer2

addresses are
loaded eagerly

ManyToOne

- Always make ManyToOne relations lazy.

```
@Entity
public class Customer {
    @Id
    @GeneratedValue
    private long id;
    private String firstname;
    private String lastname;

    @OneToMany(cascade={CascadeType.PERSIST})
    private Collection<CreditCard> creditcard=new ArrayList<CreditCard>();

    @ManyToOne(cascade={CascadeType.PERSIST}, fetch=FetchType.LAZY)
    private Address address;
```

Lazy

ManyToOne

- Always make ManyToOne relations lazy.

```
@Entity
public class Customer {
    @Id
    @GeneratedValue
    private long id;
    private String firstname;
    private String lastname;

    @ManyToOne(cascade={CascadeType.PERSIST}, fetch=FetchType.LAZY)
    private Address address;
```

```
@Query("select c from Customer c")
List<Customer> findByFirstnameLazy(String name);
```

```
Hibernate: select customer0_.id as id1_2_, customer0_.address_id as address_4_2_,
customer0_.firstname as firstnam2_2_, customer0_.lastname as lastname3_2_ from
customer customer0_
Customer{id=1, firstname='Frank', lastname='Brown'}
Customer{id=3, firstname='Frank', lastname='Johnson'}
```

ManyToOne with join fetch

- Always make ManyToOne relations lazy and use join fetch to load the related object.

@Entity

```
public class Customer {
```

@Id

@GeneratedValue

```
private long id;
```

```
private String firstname;
```

```
private String lastname;
```

@ManyToOne(cascade={CascadeType.PERSIST}, fetch=FetchType.LAZY)

```
private Address address;
```

@Query("select c from Customer c join fetch c.address")

```
List<Customer> findByFirstnameEager(String name);
```

1 query to get all customers and their addresses

```
Hibernate: select customer0_.id as id1_2_0_, address1_.id as id1_0_1_,
customer0_.address_id as address_4_2_0_, customer0_.firstname as firstnam2_2_0_,
customer0_.lastname as lastname3_2_0_, address1_.city as city2_0_1_, address1_.street
as street3_0_1_, address1_.zip as zip4_0_1_ from customer customer0_ inner join
address address1_ on customer0_.address_id=address1_.id
Customer{id=1, firstname='Frank', lastname='Brown'}
Customer{id=3, firstname='Frank', lastname='Johnson'}
```

OneToMany

@Override

```
public void run(String... args) throws Exception {  
    CreditCard creditCard1 = new CreditCard("123", "Frank Brown", new Date());  
    CreditCard creditCard2 = new CreditCard("345", "Frank Brown", new Date());  
    Customer c1 = new Customer("Frank", "Brown");  
    c1.getCreditcard().add(creditCard1);  
    c1.getCreditcard().add(creditCard2);  
    customerRepository.save(c1);  
  
    CreditCard creditCard11 = new CreditCard("123", "Frank Johnson", new Date());  
    CreditCard creditCard22 = new CreditCard("345", "Frank Johnson", new Date());  
    Customer c2 = new Customer("Frank", "Johnson");  
    c2.getCreditcard().add(creditCard11);  
    c2.getCreditcard().add(creditCard22);  
    customerRepository.save(c2);  
  
    List<Customer> customerList2 = customerRepository.findByFirstnameLazy("Frank");  
    customerList2.stream().forEach(c -> System.out.println(c));  
}
```

@Query("select c from Customer c")

List<Customer> findByFirstnameLazy(String name);

OneToMany

Load all
customers

```
Hibernate: select customer0_.id as id1_2_, customer0_.firstname as firstnam2_2_,
customer0_.lastname as lastname3_2_ from customer customer0_
Hibernate: select creditcard0_.customer_id as customer1_3_0_,
creditcard0_.creditcards_id as creditca2_3_0_, creditcard1_.id as id1_1_1_,
creditcard1_.expiration as expirati2_1_1_, creditcard1_.name as name3_1_1_,
creditcard1_.number as number4_1_1_ from customer_creditcards creditcard0_ inner
credit_card creditcard1_ on creditcard0_.creditcards_id=creditcard1_.id where
creditcard0_.customer_id=?
Hibernate: select creditcard0_.customer_id as customer1_3_0_,
creditcard0_.creditcards_id as creditca2_3_0_, creditcard1_.id as id1_1_1_,
creditcard1_.expiration as expirati2_1_1_, creditcard1_.name as name3_1_1_,
creditcard1_.number as number4_1_1_ from customer_creditcards creditcard0_ inner join
credit_card creditcard1_ on creditcard0_.creditcards_id=creditcard1_.id where
creditcard0_.customer_id=?
Customer{id=1, firstname='Frank', lastname='Brown', creditcards=[CreditCard{id=2,
number='123', name='Frank Brown', expiration=2022-04-01 21:42:57.818},
CreditCard{id=3, number='345', name='Frank Brown', expiration=2022-04-01
21:42:57.818}]}
Customer{id=4, firstname='Frank', lastname='Johnson', creditcards=[CreditCard{id=5,
number='123', name='Frank Johnson', expiration=2022-04-01 21:42:57.947},
CreditCard{id=6, number='345', name='Frank Johnson', expiration=2022-04-01
21:42:57.947}]}
```

Load all
creditcards
for customer1

Load all
creditcards
for customer2

OneToMany with join fetch

@Override

```
public void run(String... args) throws Exception {  
    CreditCard creditCard1 = new CreditCard("123", "Frank Brown", new Date());  
    CreditCard creditCard2 = new CreditCard("345", "Frank Brown", new Date());  
    Customer c1 = new Customer("Frank", "Brown");  
    c1.getCreditcard().add(creditCard1);  
    c1.getCreditcard().add(creditCard2);  
    customerRepository.save(c1);  
  
    CreditCard creditCard11 = new CreditCard("123", "Frank Johnson", new Date());  
    CreditCard creditCard22 = new CreditCard("345", "Frank Johnson", new Date());  
    Customer c2 = new Customer("Frank", "Johnson");  
    c2.getCreditcard().add(creditCard11);  
    c2.getCreditcard().add(creditCard22);  
    customerRepository.save(c2);  
  
    List<Customer> customerList2 = customerRepository.findByFirstnameEager("Frank");  
    customerList2.stream().forEach(c -> System.out.println(c));  
}
```

```
@Query("select c from Customer c join fetch c.creditcards")  
List<Customer> findByFirstnameEager(String name);
```



OneToMany with distinct join fetch

@Override

```
public void run(String... args) throws Exception {  
    CreditCard creditCard1 = new CreditCard("123", "Frank Brown", new Date());  
    CreditCard creditCard2 = new CreditCard("345", "Frank Brown", new Date());  
    Customer c1 = new Customer("Frank", "Brown");  
    c1.getCreditcard().add(creditCard1);  
    c1.getCreditcard().add(creditCard2);  
    customerRepository.save(c1);  
  
    CreditCard creditCard11 = new CreditCard("123", "Frank Johnson", new Date());  
    CreditCard creditCard22 = new CreditCard("345", "Frank Johnson", new Date());  
    Customer c2 = new Customer("Frank", "Johnson");  
    c2.getCreditcard().add(creditCard11);  
    c2.getCreditcard().add(creditCard22);  
    customerRepository.save(c2);  
  
    List<Customer> customerList2 = customerRepository.findByFirstnameEager("Frank");  
    customerList2.stream().forEach(c -> System.out.println(c));  
}
```

```
@Query("select distinct c from Customer c join fetch c.creditcards")  
List<Customer> findByFirstnameEager(String name);
```

OneToMany with distinct join fetch

```
Hibernate: select distinct customer0_.id as id1_2_0_, creditcard2_.id as id1_1_1_,
customer0_.firstname as firstnam2_2_0_, customer0_.lastname as lastname3_2_0_,
creditcard2_.expiration as expirati2_1_1_, creditcard2_.name as name3_1_1_,
creditcard2_.number as number4_1_1_, creditcard1_.customer_id as customer1_3_0_,
creditcard1_.creditcards_id as creditca2_3_0_ from customer customer0_ inner join
customer_creditcards creditcard1_ on customer0_.id=creditcard1_.customer_id inner join
credit_card creditcard2_ on creditcard1_.creditcards_id=creditcard2_.id
Customer{id=1, firstname='Frank', lastname='Brown', creditcards=[CreditCard{id=2,
number='123', name='Frank Brown', expiration=2022-04-01 21:50:43.899},
CreditCard{id=3, number='345', name='Frank Brown', expiration=2022-04-01
21:50:43.899}]}
Customer{id=4, firstname='Frank', lastname='Johnson', creditcards=[CreditCard{id=5,
number='123', name='Frank Johnson', expiration=2022-04-01 21:50:44.015},
CreditCard{id=6, number='345', name='Frank Johnson', expiration=2022-04-01
21:50:44.015}]}
```


Summary

- Always make ManyToOne relations lazy
 - Use join fetch to fetch them eagerly
- OneToMany relations are already lazy
 - Use distinct join fetch to fetch them eagerly
- Always check how often the ORM goes to the database

Main point

- When using JPA it is important to optimize the mapping and queries to get the most optimal database access performance.

Science of Consciousness: Nature always takes the most optimal path of least action.

Connecting the parts of knowledge with the wholeness of knowledge

1. Spring provides different ways to add queries to an enterprise application.
2. JPA optimization helps to get better performance

-
3. **Transcendental consciousness** is the field of all possibilities.
 4. **Wholeness moving within itself:** In Unity Consciousness, we experience the unity within all diversity in creation.

