JPA Queries

Part 2

Overview

Topics Covered:

- Navigating across relationships
- Using ‘member of’ operator
- Using ‘join’ operator
- Chaining methods

Navigating across relationships

HQL enables navigation across entity relationships more intuitively than SQL.

Example: Find All Students for a Specific Tutor:

```
Tutor tutor = em.find(Tutor.class, 1);
Query<Student> q = em.createQuery("from Student as student where student.tutor =:tutor");
q.setParameter("tutor", tutor);
List<Student>queryResult =q.getResultList();
for(Student s:queryResult) {
  System.out.println(s);}
```

Here, Student has a @ManyToOne relationship with Tutor.

Alternative Approach

```
Query  q = em.createQuery("select tutor.teachingGroup from Tutor as
tutor where tutor.name='Johan Smith'");

List<Student> studentsForJohan = q2.getResultList();

for(Student s:studentsForJohan) {

  System.out.println(s);

}
```

In this case, Tutor has a @OneToMany relationship with Student.

Working with collections- member of operator

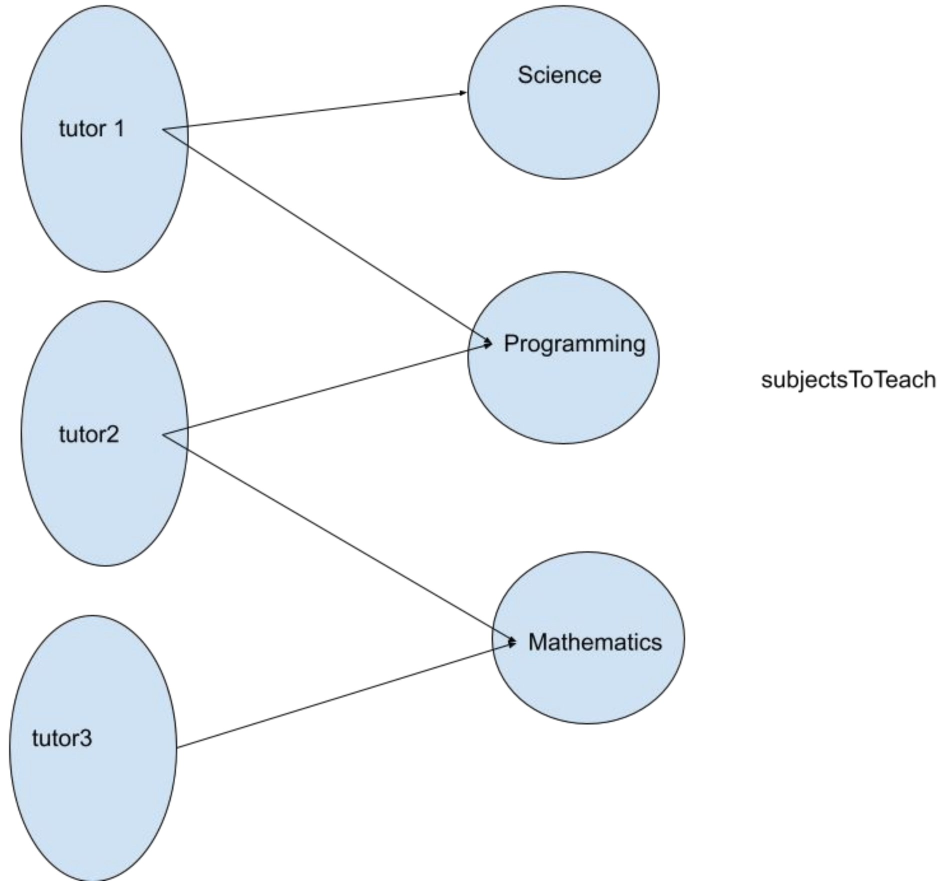
member of allows retrieving entities based on their collection relationships.

Example: Find All Tutors Who Teach a Specific Subject:

```
Subject programming = em.find(Subject.class, 3);
Query query= em.createQuery("from Tutor tutor where :subject member of tutor.subjectsToTeach");
query.setParameter("subject", programming);
List<Tutor>tutorsForProgramming = query.getResultList();
for(Tutor tutor : tutorsForProgramming) {
   System.out.println(tutor);}
```

Here, we find a Subject first, then retrieve all Tutor entities that have this subject in their subjectsToTeach collection.

Working with collections- member of



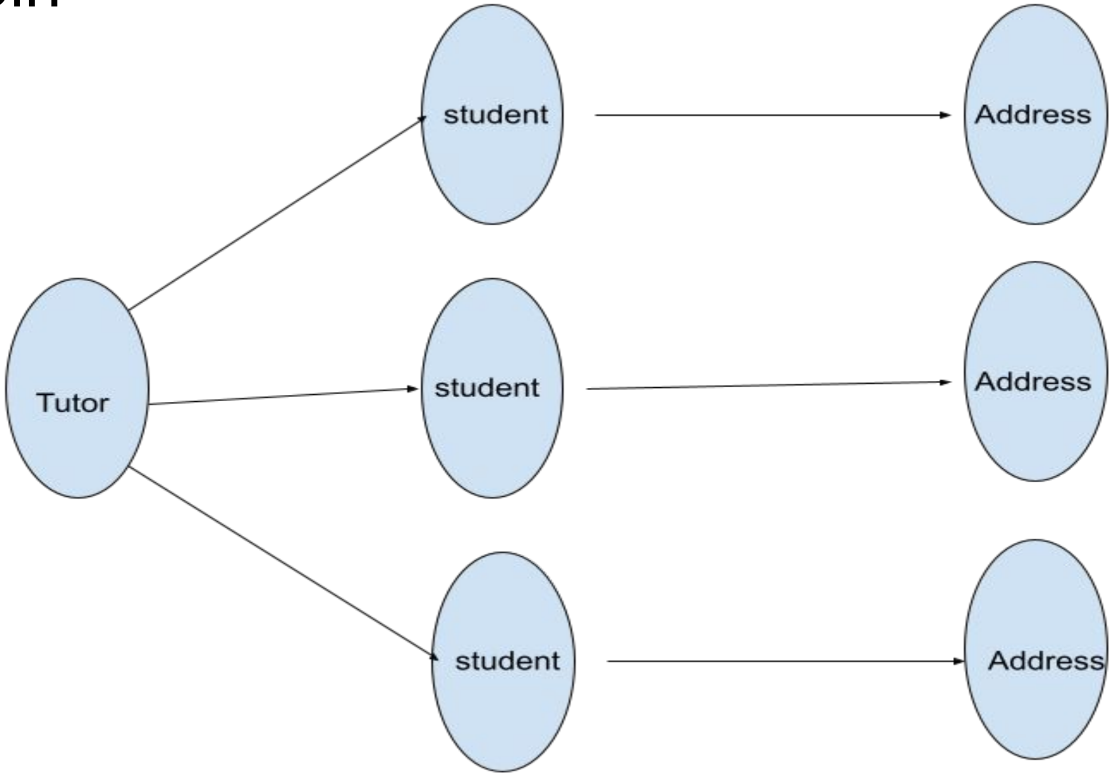
Using join for Complex Relationships

HQL joins are useful for navigating @OneToMany or @ManyToMany relationships.

Example: Find Tutors Who Have a Student Living in 'City 2'

```
Query query = em.createQuery("from Tutor as tutor join tutor.teachingGroup
as student where student.address.city = 'city 2'");

List<Object[]> results = query.getResultList();

for (Object[] item : results) {

  System.out.println(item[0] + "-------------------- "+ item[1]);

}
```

Using join



Using join

This query returns a list of pairs (List<Object[]>), where:

item[0] = Tutor

item[1] = Student

Using join, Selecting Only the Tutor

To return only Tutors, we modify the query:

```
Query query = em.createQuery("select distinct tutor from Tutor as tutor
join tutor.teachingGroup as student where student.address.city = 'city
2'");


List<Tutor> results = query.getResultList();
for (Tutor t : results) {
    System.out.println(t);
}
```

Now, we retrieve only tutors who have students from 'City 2'.

Chaining methods

Instead of writing multiple statements, we can chain them into a single line for cleaner code.

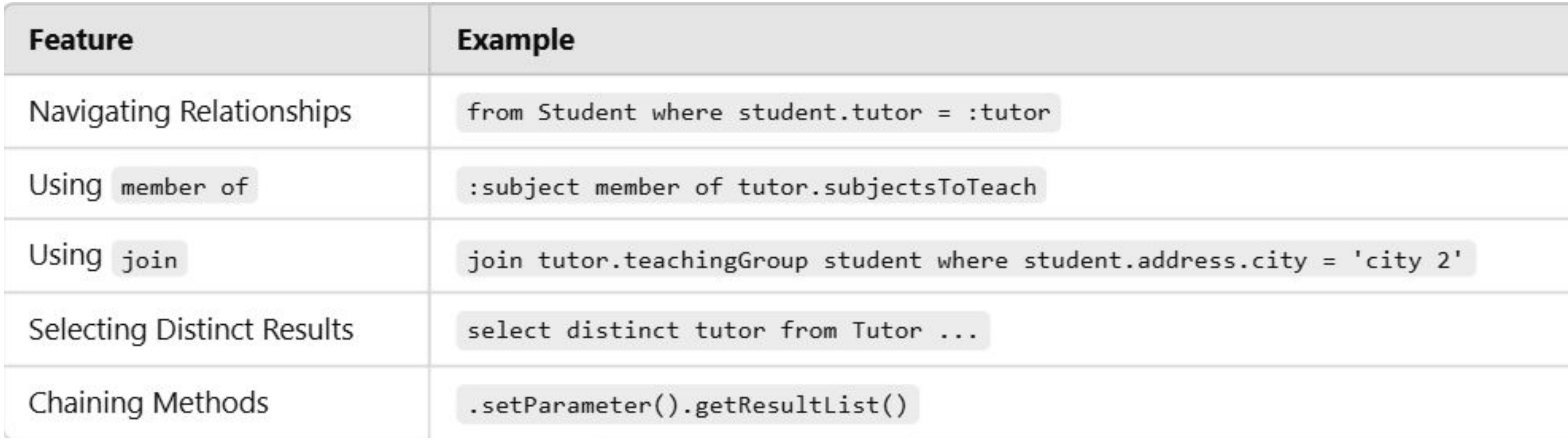
```
String city = "city 2";

List<Tutor>results = em.createQuery("select distinct tutor from Tutor tutor
join tutor.teachingGroup student where student.address.city = :city")

       .setParameter("city", city)

        .getResultList();

for(Tutor tutor:results) {

   System.out.println(tutor);

}
```

Chaining methods

Benefits of Method Chaining:

- Makes the code cleaner and more readable.
- Reduces redundant variable declarations.

Summary



Feature	Example
Navigating Relationships	`from Student where student.tutor = :tutor`
Using `member of`	`:subject member of tutor.subjectsToTeach`
Using `join`	`join tutor.teachingGroup student where student.address.city = 'city 2'`
Selecting Distinct Results	`select distinct tutor from Tutor ...`
Chaining Methods	`.setParameter().getResultList()`