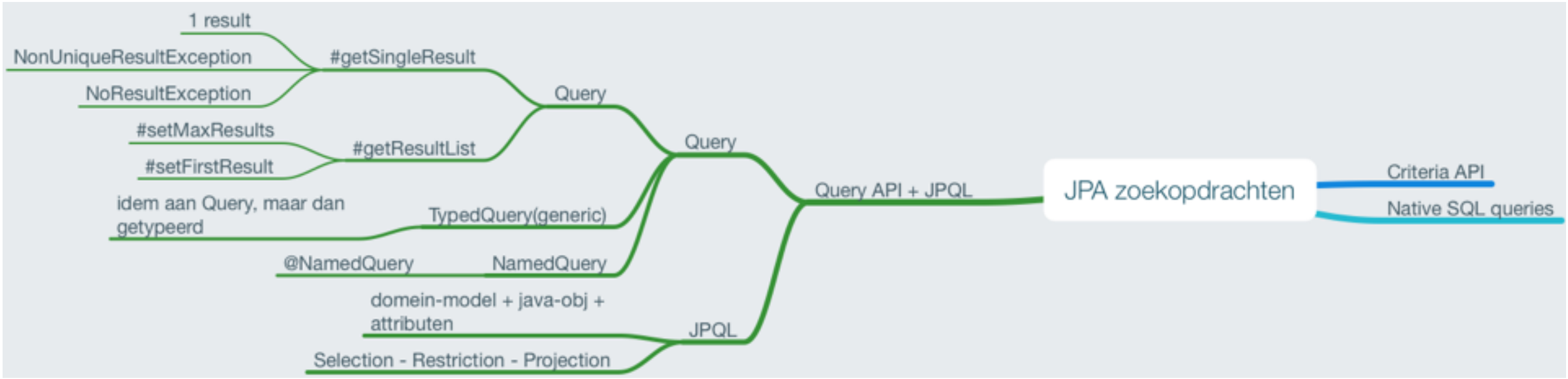
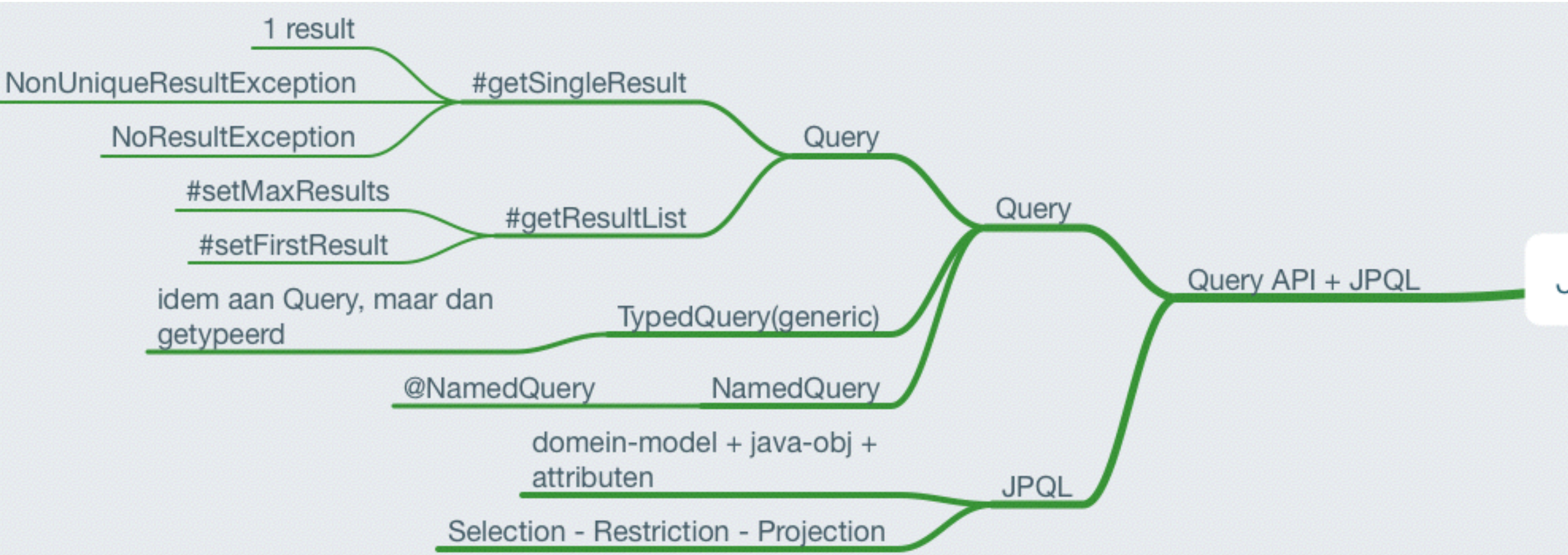
JPA - ZOEKOPDRACHTEN


JPA zoekopdrachten
Criteria API
Native SQL queries
Query API + JPQL
Query
Query
#getSingleResult
1 result
NonUniqueResultException
NoResultException
#getResultList
#setMaxResults
#setFirstResult
TypedQuery(generic)
idem aan Query, maar dan getypeerd
NamedQuery
@NamedQuery
JPQL
domein-model + java-obj + attributen
Selection - Restriction - Projection


1 result
NonUniqueResultException
NoResultException
#getSingleResult
#setMaxResults
#setFirstResult
#getResultList
Query
idem aan Query, maar dan getypeerd
TypedQuery(generic)
@NamedQuery
NamedQuery
Query
domein-model + java-obj + attributen
Selection - Restriction - Projection
JPQL
Query API + JPQL

```
Query query = em.createQuery("select p from Person as p");
List<Person> persons = (List<Person>) query.getResultList();
```

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

Verder kunnen we met de methode `setHint()` nog extra gegevens doorgeven aan het onderliggende persistentiemechanisme. Sommige hints zijn hierbij gestandaardiseerd, andere zijn specifiek voor de JPA-*provider*.

Zo kunnen we een *timeout* instellen op het uitvoeren van de *query* met de gestandaardiseerde hint `javax.persistence.query.timeout`

```
query.setHint("javax.persistence.query.timeout",5000);
```

FLUSH MODE

Tenslotte kunnen we de algemene *flushmode* van de *entity manager* instellen met de methode `setFlushMode()`. We hebben hier de keuze tussen *AUTO* en *COMMIT*. Bij *AUTO* kan de *entity manager* beslissen om de gegevens eerst naar de databank te schrijven alvorens een zoekoperatie uit te voeren zodat gewijzigde gegevens mee in rekening gebracht worden. Bij *COMMIT* worden de gegevens enkel naar de databank geschreven bij het beëindigen van de transactie en zullen tussentijdse wijzigingen niet verschijnen in het zoekresultaat.

Los van deze algemene instelling van de *flushmode* van de *entity manager*, kunnen we tevens de *flushmode* ook specifiek instellen voor een bepaalde zoekopdracht met de methode `setFlushMode()`. Ook hier hebben we de keuze tussen twee mogelijkheden:

`FlushModeType.COMMIT:` De gegevens worden voor de zoekoperatie niet weggeschreven maar enkel bij het voltooien van de transactie. Dit voorkomt onnodige communicatie met de databank, maar houdt tevens het risico in dat het zoekresultaat onvolledig is.

`FlushModeType.AUTO:` De gegevens worden voor de zoekoperatie wel weggeschreven. De wijzigingen worden dus in rekening gebracht in het zoekresultaat.

```
query.setFlushMode(FlushModeType.COMMIT);
```

NAMED QUERY

```
@NamedQuery(name="findAll", query="select p from Person as p")
@Entity
public class Person {
    ...
}
```

```
public class FindPerson {
    ...
        Query query = em.createNamedQuery("findAll");
        List<Person> persons =
                            (List<Person>)query.getResultList();
        return persons;
    ...
}
```

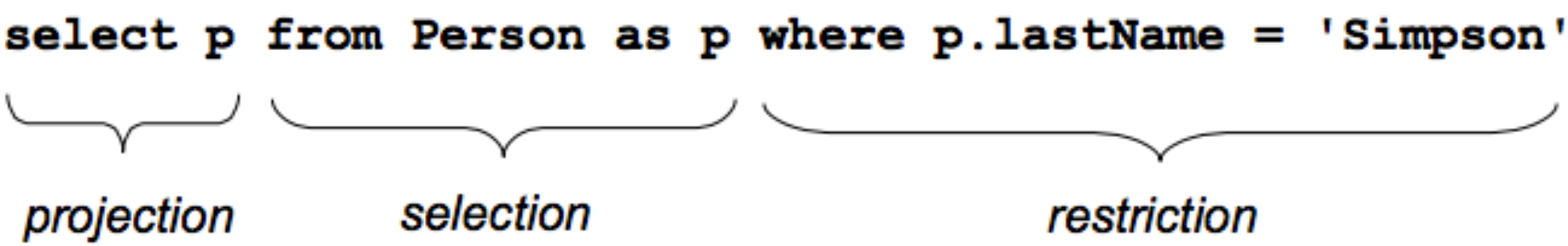
Sinds JPA 2.1 is het ook mogelijk *named queries* dynamisch toe te voegen tijdens de uitvoering van het programma. De *entity manager factory* beschikt hiertoe over de methode `addNamedQuery()`:

```
    EntityManagerFactory emf = Persistence
              .createEntityManagerFactory("course");
    EntityManager em = emf.createEntityManager();
    Query query = em.createQuery("select p from Person p");
    emf.addNamedQuery("findAll",query);
```

PROJECTION - SELECTION - RESTRICTION

Een zoekopdracht in JPQL is een tekstregel die bestaat uit drie verschillende onderdelen:

1. **Selection**: Hiermee wordt het geheel van entiteiten geselecteerd waarin men wil zoeken. Men noemt dit ook wel de *FROM*-clausule.
2. **Restriction**: Hiermee worden criteria opgelegd aan de entiteiten die men zoekt. Men noemt dit ook wel de *WHERE*-clausule.
3. **Projection**: Hiermee wordt aangegeven welke gegevens men wil hebben van de gevonden entiteiten en in welke vorm. Men noemt dit ook wel de *SELECT*-clausule.



5.7.2.3.3.4. Selectie van gerelateerde objecten

Indien een object een relatie heeft met een ander object kan dit gerelateerde object geselecteerd worden door te navigeren naar het gerelateerde object.

```
select p.medicalFile from Patient p
```

Vermits dit gerelateerde object een echt *entity*-object is, zal ook dit toegevoegd worden aan de *persistence context*.
Men kan hier eventueel ook verder navigeren naar attributen van het gerelateerde object:

```
select p.medicalFile.weight from Patient p
```

Hier gaat het dan terug om een scalaire waarde.
Het selecteren van gerelateerde objecten kan ook indien het gaat om een *to many*-relatie.

```
select s.students from School s
```

Het resultaat is hier een lijst van alle studenten van de school.

```
String q = "select p.firstName, p.lastName from Person p";
Query query = entityManager.createQuery(q);
List<String[]> results = (List<String[]>) query.getResultList();
for(String[] r : results) {
   System.out.println(r[0] + " " + r[1]);
}
```

5.7.2.3.3.3. Selectie van ingesloten objecten

Indien een *entity*-object een ingesloten (*embedded*) heeft, kunnen we dit expliciet opvragen in de selectie:

```
select p.address from Person p
```

Vermits een ingesloten object zelf geen *entity* object is, zal dit ook niet toegevoegd worden aan de *persistence context*.

Tenslotte kunnen in de projectie volgende aggregatie-functies gebruikt worden:

Aggregatie-functie	***Omschrijving***
`count(expr)`	Geeft het aantal objecten.
`max(expr)`	Geeft de hoogste waarde.
`min(expr)`	Geeft de laagste waarde.

Aggregatie-functie	***Omschrijving***
`avg(expr)`	Geeft het gemiddelde.
`sum(expr)`	Geeft de som.

In volgend voorbeeld zoeken we het aantal personen:

```
select count(p) from Person p
```

Ook hier kan men duplicaten vermijden:

```
select count(distinct p) from Person p
```

5.7.2.3.4.1. Parameters in de zoekopdracht

We kunnen in de zoekopdracht tevens parameters gebruiken. Deze kunnen ofwel met een index ofwel met een naam aangegeven worden:

```
Query query = entityManager.createQuery
              ("select p from Person as p where p.lastName=:last");
query.setParameter("last", "Simpson");
```

De naam van een parameter laten we voorafgaan door een dubbele punt. Met de methode `setParameter()` kunnen we vervolgens de waarde invullen.

```
Query query = entityManager.createQuery
              ("select p from Person as p where p.lastName=?1");
query.setParameter(1, "Simpson");
```

Bij een geïndexeerde parameter laten we de index voorafgaan door een vraagteken en vullen we de waarde achteraf in a.d.h.v. zijn index.

Voor tijden en datums moeten we afzonderlijke methoden gebruiken om de parameters in te vullen waarbij we tevens het type kunnen opgeven (`DATE`, `TIME`, `TIMESTAMP`).

```
Calendar today = new GregorianCalendar();
Query query = entityManager.createQuery
        ("select p from Person as p where p.birthDay=?1");
```

```
query.setParameter(1, today, TemporalType.DATE );
```

```
select p from Person p
            where p.gender = persons.GenderType.FEMALE
```

VOORBEELDEN

Operator	***Omschrijving***
`.`	Navigatie-operator. Hiermee kan men verder navigeren in gerelateerde objecten en hun attributen.
`+ - / *`	Rekenkundige operatoren die toegepast kunnen worden op velden en letterlijke waarden van een numeriek datatype.
`>,<,=,<=,>=,<>`	Vergelijkingsoperatoren die resulteren in `true` of `false`.
`between and` `not between and`	Gaat na of een waarde in een bepaald interval ligt, inclusief de grenzen. Deze operator kan gebruikt worden voor numerieke waarden, datums en strings.
`like`	Gaat na of een waarde aan een bepaald patroon beantwoordt. In dergelijke patronen worden volgende specifieke karakters gebruikt: `%`: Eender welke opeenvolging van karakters. `_`: Een enkel karakter. Indien men deze tekens zelf bedoelt, moeten ze voorafgegaan worden door een `\` (*escape*).
`is null` `is not null`	Gaat na of de inhoud van een bepaald veld leeg (`null`) is.
`is empty` `is not empty`	Gaat na of een verzameling gerelateerde objecten leeg is.
`in (x,y,...)` `not in (x,y,..)`	Gaat na of het veld voorkomt in een lijst van waarden. Deze waarden kunnen *literals* of parameters zijn.
`member of` `not member of`	Gaat na of het veld deel uitmaakt van een verzameling.

`not`
`and`
`or`
	Logische operatoren. Dit zijn **geen** *shortcut* operatoren zoals `&&` en `||` in Java.

VOORBEELDEN

Indien we bijvoorbeeld alle scholen willen hebben die geen studenten hebben, gebruiken we volgende zoekopdracht:

```
select s from School s where s.students is empty
```

```
select p from Patient p where p.medicalFile.weight > 100
```

5.7.2.3.4.5. Sortering van resultaten

Het resultaat kan gesorteerd worden volgens een bepaald veld met de clausule `order by [asc|desc]`

Bijvoorbeeld:

```
select p from Person p order by p.lastName desc
```

Eventueel kan de sortering op basis van meerdere velden gebeuren:

```
select p from Person p order by p.lastName desc, p.firstName asc
```

De sortering kan ook gebeuren op basis van een berekend veld in de *SELECT*-clausule. W definiëren hierbij een variabele die we nadien gebruiken voor de sortering.

Stel dat we de patienten willen ordenen volgens hun BMI:

```
select p, (p.medicalFile.weight/(p.medicalFile.height *
p.medicalFile.height)) as bmi from Patient p order by bmi
```

In de *SELECT*-clausule berekenen we de BMI en kennen deze toe aan de variabele `bmi`. We gebruiken deze variabele vervolgens in de `order by` om het resultaat te sorteren.

5.7.2.3.5.1. INNER JOIN

We kunnen ook gebruik maken van een INNER JOIN:

```
select p from Patient p inner join p.medicalFile mf where mf.weight
> 50
```

Het woord `inner` mag hier weggelaten worden.

De *inner join* selecteert objecten en hun gerelateerde objecten enkel indien deze relatie bestaat.

Indien we te maken hebben met *to-many* relaties moeten we dergelijke *inner join* gebruiken om restricties op basis van gerelateerde objecten op te nemen.

```
select s from School s join s.students st where st.name='Bart'
```