Query Criteria

Introduzione alle query criteria

- JPA 2 introduce nuove API per la realizzazione di query criteria. L'obiettivo è standardizzare le tecniche O.O. di costruzione delle queries offerte dai singoli providers
- Esiste una corrispondenza evidente tra le query JPQL e le query criteria. Ad esempio, la seguente query JPLQ:

```
SELECT e
FROM Employee e
WHERE e.name = 'John Smith`
```

corrisponde alla seguente query criteria:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Employee> c = cb.createQuery(Employee.class);

Root<Employee> emp = c.from(Employee.class);
c.select(emp).where(cb.equal(emp.get("name"), "John Smith"));
```

NOTA: le queries si possono scrivere anche senza l'uso dei generics

Elementi di una query criteria

- Gli elementi di una query criteria si possono distinguere in:
  - **CrtieriaBuilder** → Principale punto di accesso alle query criteria. Implementa il pattern builder la per costruzione e il collegamento dei vari pezzi che compongono la query

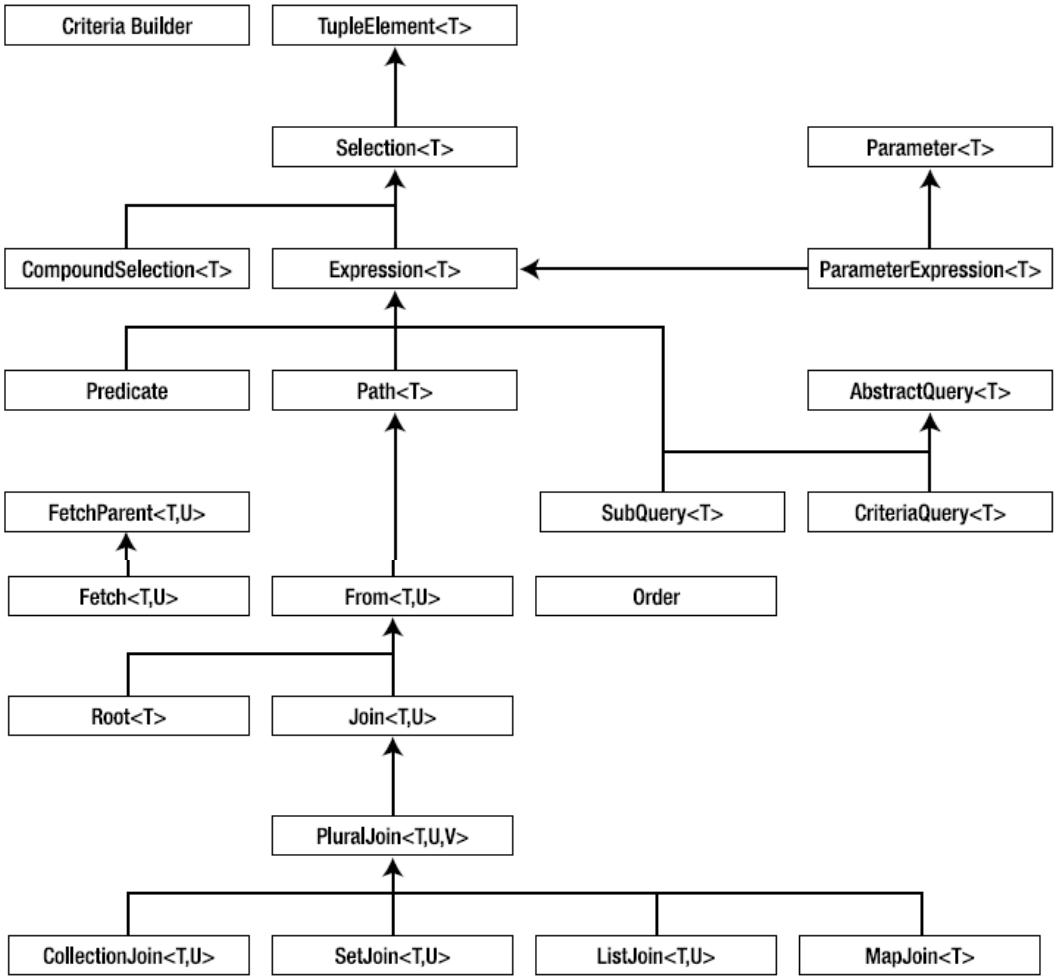
    Altro scopo di CrtieriaBuilder è costruire le condizioni (ad esempio da usare con il metodo where()) ed altri componenti della query
  - **CriteriaQuery<V>** → Possiede le operazioni che compongono una query. Volendo fare un confronto, possiede i metodi select(), from(), where() ecc… che corrispondono alle clausole della query JPQL
  - **Root<V>** → Oggetto rappresentativo dei dati da selezionare. Corrisponde alla variabile identificativa in una query JPQL
    - Nella query JPQL di esempio precedente, sarebbe la variabile «e»

Criteria builder

- L'interfaccia CriteriaBulder consente di creare un oggetto di tipo CriteriaQuery attraverso tre metodi
  - **createQuery(Class<T>)** → dove Class<T> indica la classe risultato della query
  - **createQuery()** → costruisce una query che restituisce un risultato di tipo Object
  - **createTupleQuery()** → costruisce una query il cui risultato (cioè quanto specificato nella clausola Select) è composto da più elementi diversi
    - Equivale a createQuery(Tuple.class) dove la classe Tuple raccoglie i vari elementi (oggetti o tipi base) risultato della query in unico oggetto

Interfacce coinvolte nella query criteria

- Schema delle interfacce coinvolte nella costruzione di una query



Costruzione di una query

- Come le query JPQL (ma anche SQL), una query criteria si compone di sei clausole base

Clausola JPQL	Interfaccia API Criteria	metodo
SELECT	CriteriaQuery Subquery	select() select()
FROM	AbstractQuery	from()
WHERE	AbstractQuery	where()
GROUP BY	AbstractQuery	groupBy()
HAVING	AbstractQuery	having()
ORDER BY	CriteriaQuery	orderBy()

- La maggior parte dei metodi sono immutabili: una volta invocato un metodo ed ottenuto un oggetto, non ci sono metodi tipo «set» da invocare per cambiare il contenuto dell'oggetto

Query Roots

- Nelle query JPQL uno degli elementi fondamentali è la variabile identificativa, cioè l'alias nella clausola FROM con cui identifichiamo il tipo di elemento (Entity, Embeddable ecc..) su cui stiamo lavorando

- Le query Criteria sono query ad oggetti e non avrebbe senso usare un alias

- Un oggetto Root<V> rappresenta il path di base su cui montare il resto della query. Nel costruire questo path, indichiamo il tipo risultato. Ad esempio:

```
CriteriaQuery<Employee> c = cb.createQuery(Employee.class);
Root<Employee> emp = c.from(Employee.class);
```

- Una query deve avere come minimo una root ma può averne più di una

Clausola FROM

- Il metodo form() è additivo: ogni invocazione aggiunge un tipo risultato al path della query.
  - L'effetto è quello di un prodotto cartesiano tra gli entity coinvolti

- In generale, nella clausola FROM di una query JPQL (o SQL) è possibile specificare più di un entity. Ad esempio:

```
SELECT DISTINCT d
FROM Department d, Employee e
WHERE d = e.department
```

- La corrispondente query criteria:

```
CriteriaQuery<Department> c = cb.createQuery(Department.class);
Root<Department> dept = c.from(Department.class);// attenti ai tipi
Root<Employee> emp = c.from(Employee.class);
c.select(dept).distinct(true).where(cb.equal(dept, emp.get("department")));
```

Path Expression

- Una volta costruito il root path della query, andiamo ad estenderlo «montando su» i pezzi necessari per inserire le nostre specifiche condizioni
  - Si parla quindi di path expression come estensione del root path

- Ad esempio, se vogliamo aggiungere una condizione nella where

```
SELECT e
FROM Employee e
WHERE e.address.city = 'New York'
```

la where si esprime attraverso i metodi get() sull'oggetto Root opportuno:

emp.get("address").get("city")

La query criteria diventa

```
CriteriaQuery<Employee> c = cb.createQuery(Employee.class);
Root<Employee> emp = c.from(Employee.class);
c.select(emp)
.where(cb.equal(emp.get("address").get("city"), "New York"));
```

Clausola Select

- E’ possibile creare select a singola espressione o ad espressione multipla

Select a singola espressione

- Si utilizza il metodo select() dell’interfaccia CriteriaQuery che richiede un argomento di tipo Selection o CompoundSelection nel caso il tipo risultato della query è Tuple
  - La versione del metodo select senza argomenti non è portabile
- L’attributo determina il tipo di dato risultato. Ad esempio, di seguito recuperiamo il nome di tutti gli impiegati (duplicati compresi)

Devono essere compatibili

```
CriteriaQuery<String> c = cb.createQuery(String.class);
Root<Employee> emp = c.from(Employee.class);
c.select(emp.<String>get("name"));
```

- Notare che il tipo dichiarato nella select deve essere compatibile con il tipo dichiarato come CriteriaQuery

Clausola Select

- La sintassi può sembrare strana ma è legale. Il metodo è dichiarato nel seguente modo:

```
CriteriaQuery<T> select(Selection<? extends T> selection);
```

Select ad espressione multipla

- Se la clausola select prevede diversi campi, questi vanno specificati al metodo select() attraverso un oggetto CompoundSelection
  - L'oggetto CompoundSelection è ritornato dal metodo tuple() come nell'esempio

```
CriteriaQuery<Tuple> c= cb.createTupleQuery(); //oppure cb.createQuery()
Root<Employee> emp = c.from(Employee.class);
c.select(cb.tuple(emp.get("id"), emp.get("name")));
```

  - In base al tipo specificato per CriteriaQuery, il metodo select tornerà il risultato in una determinata forma che può essere:
    - Tuple --> metodo tuple()
    - «Classe non persistente» → metodo construct()
    - Object[] → metodo array()

Clausola Select

- Un metodo più conveniente è multiselect()

```
CriteriaQuery<Object[]> c = cb.createQuery(Object[].class);
Root<Employee> emp = c.from(Employee.class);
c.multiselect(emp.get("id"), emp.get("name"));
```

- Questo metodo ritorna un vettore di tipo Obect[] se nella multiselect si esprimono più valori, un singolo Object se si esprime un solo valore.
  - E' possibile definire come tipo Object al posto di Object[] per QueryCriteria. In questo caso, se in multiselect si esprimono più valori, bisogna effettuare un downcast a Object[] e quindi accedere ai risultati

- Multiselect può tornare anche un oggetto Tuple, come nell'esempio

```
CriteriaQuery<Tuple> c = cb.createTupleQuery();
Root<Employee> emp = c.from(Employee.class);
c.multiselect(emp.get("id"), emp.get("name"));
```

Clausola Select

- Multiselect può tornare anche un oggetto di tipo non persistente, come nell'esempio

```
CriteriaQuery<EmployeeInfo> c = cb.createQuery(EmployeeInfo.class);
Root<Employee> emp = c.from(Employee.class);
c.multiselect(emp.get("id"), emp.get("name"));
```

In questo caso, viene invocato il costruttore di EmployeeInfo per costruire un oggetto di questo tipo con l'elenco dei valori recuperati da multiselect. L'esempio precedente è equivamente a:

```
CriteriaQuery<EmployeeInfo> c = cb.createQuery(EmployeeInfo.class);
Root<Employee> emp = c.from(Employee.class);
c.select(cb.construct(EmployeeInfo.class, emp.get("id"), emp.get("name")));
```

Se il risultato della query è un oggetto non persistente insieme ad altri campi, è possibile adottare la seguente soluzione

```
CriteriaQuery<Object[]> c = cb.createQuery(Object[].class);
Root<Employee> emp = c.from(Employee.class);
c.multiselect(emp.get("id"),
cb.construct(EmployeeInfo.class, emp.get("id"), emp.get("name")));
```

Alias

- Anche per le i criteria è possibile definire alias per i campi della query come nel seguente esempio:

```
CriteriaQuery<Tuple> c= cb.createTupleQuery();
Root<Employee> emp = c.from(Employee.class);
c.multiselect(emp.get("id").alias("id"),
emp.get("name").alias("fullName"));
```

Clausola FROM e JOIN

- Per creare delle query join viene utilizzato il metodo join() dell'interfaccia From (sottotipo di Root) che restituisce un oggetto Join
- Il metodo join prende come parametri una espressione e una costante che rappresenta il tipo di join INNER(default) o LEFT
  - La costante RIGHT esiste ma rende la query non portabile
- Le invocazioni al metodo join() sono additive.
- Esempio di join tra impiegato e progetto

```
Join<Employee,Project> project = emp.join("projects", JoinType.LEFT);
```

```
Join<Employee,Project> project = dept.join("employees").join("projects");
```

Clausola FROM e JOIN

- E’ possibile recuperare i dati di una mappa se usata come relazione molti-a-molti. Ad esempio, supponendo di avere una relazione molcome nel seguente esempio:

```
CriteriaQuery<Object> c = cb.createQuery();
Root<Employee> emp = c.from(Employee.class);
MapJoin<Employee,String,Phone> phone = emp.joinMap("phones");
c.multiselect(emp.get("name"), phone.key(), phone.value());
```

  il metodo join() non può essere riscritto perché è in overriding, qundi è sostituito con uno più appropriato

  I generics associati alla classe MapJoin sono:

  - Employee → oggetto originale
  - String → chiave
  - Phone → valore

In JPQL:

```
SELECT e.name, KEY(p), VALUE(p)
FROM Employee e JOIN e.phones p
```

- Per Collection,Set e List esistono metodi analoghi: joinCollection(), joinSet(), joinList()

Clausola FROM e JOIN

- Come per le Query JPQL è possibile attivare l'opzione FETCH. Il metodo fetch() va utilizzato al posto di join()

```
CriteriaQuery<Employee> c = cb.createQuery(Employee.class);
Root<Employee> emp = c.from(Employee.class);
emp.fetch("address"); //join fetch con address
c.select(emp);
```

  notare che il tipo di ritorno di fetch() e Fetch. Non è possibile specificare altro alla query attraverso questo oggetto

- Altro esempio di left join distinct:

```
CriteriaQuery<Employee> c = cb.createQuery(Employee.class);
Root<Employee> emp = c.from(Employee.class);
emp.fetch("phones", JoinType.LEFT);
c.select(emp).distinct(true);
```

Clausola WHERE

- Il metodo where() accetta singoli o multipli valori di tipo Predicate oppure un singolo ogetto di tipo Expression<Boolean>

- L'invocazione di where() NON E' ADDITIVA: invocando tale metodo si scartano automaticamente tutte le eventuali altre espressioni WHERE settate in precedenza

- La chiave per scrivere una condizione where anche complessa è utilizzare le API offerte dall'interfaccia CriteriaBuilder che possiede tutti i predicati, espressioni e funzionai supportate da JPQL

Clausola WHERE

- Tabella di corrispondenza tra JPQL e i metodi di CriteriaBuilder

JP QL Operator	CriteriaBuilder Method
AND	`and()`
OR	`or()`
NOT	`not()`
=	`equal()`
<>	`notEqual()`
>	`greaterThan()`, `gt()`
>=	`greaterThanOrEqualTo()`, `ge()`
<	`lessThan()`, `lt()`
<=	`lessThanOrEqualTo()`, `le()`
BETWEEN	`between()`
IS NULL	`isNull()`
IS NOT NULL	`isNotNull()`
EXISTS	`exists()`
NOT EXISTS	`not(exists())`

Se vi sono due metodi per lo stesso operatore JPQL, quello con il nome più lungo funziona per tutti i tipi, mentre quello più corto solo per i numeri

Clausola WHERE

JP QL Operator	CriteriaBuilder Method
IS EMPTY	`isEmpty()`
IS NOT EMPTY	`isNotEmpty()`
MEMBER OF	`isMember()`
NOT MEMBER OF	`isNotMember()`
LIKE	`like()`
NOT LIKE	`notLike()`
IN	`in()`
NOT IN	`not(in())`

Clausola WHERE

JP QL Expression	CriteriaBuilder Method
ALL	`all()`
ANY	`any()`
SOME	`some()`
-	`neg(), diff()`
+	`sum()`
*	`prod()`
/	`quot()`
COALESCE	`coalesce()`
NULLIF	`nullif()`
CASE	`selectCase()`

Clausola WHERE

JP QL Function	CriteriaBuilder Method
ABS	abs()
CONCAT	concat()
CURRENT_DATE	currentDate()
CURRENT_TIME	currentTime()
CURRENT_TIMESTAMP	currentTimestamp()
LENGTH	length()
LOCATE	locate()
LOWER	lower()
MOD	mod()
SIZE	size()
SQRT	sqrt()
SUBSTRING	substring()
UPPER	upper()
TRIM	trim()

Clausola WHERE

JP QL Aggregate Function	CriteriaBuilder Method
AVG	`avg()`
SUM	`sum()`, `sumAsLong()`, `sumAsDouble()`
MIN	`min()`, `least()`
MAX	`max()`, `greatest()`
COUNT	`count()`
COUNT DISTINCT	`countDistinct()`

Predicati

- E’ possibile creare un insieme di espressioni in AND o in OR logico da passare alla query attraverso i metodi and() e or() rispettivamente
- Tecnicamente, si crea una lista di oggetti Predicate e si inseriscono espressioni come risultante della chiamata ai metodi and() e or(). Ad esempio

```
List<Predicate> criteria = new ArrayList<Predicate>();
if (name != null) {
   ParameterExpression<String> p = cb.parameter(String.class, "name");
   criteria.add(cb.equal(emp.get("name"), p));
}
```

La costruzione delle espressioni può essere fatta anche incrementale, rinunciando alla lista ed utilizzando i metodi conjunction() e disjunction()

```
Predicate criteria = cb.conjunction();
if (name != null) {
   ParameterExpression<String> p = cb.parameter(String.class, "name");
   criteria = cb.and(criteria, cb.equal(emp.get("name"), p));
}
```

Parametri

- La gestione dei parametri differisce dalle queries JPQL
- E' necessario creare un oggetto ParameterExpression del tipo giusto da usare nelle espressioni condizionali.
  - L'oggetto ParameterExpression viene restituito dal metodo parameter() dell'interfaccia CriteriaBuilder.
  - I parametri così costruiti sono oggetti rappresentativi del parametro che va sostituito (via nome) prima di lanciare la query
- Ad esempio:

```
CriteriaQuery<Employee> c = cb.createQuery(Employee.class);
Root<Employee> emp = c.from(Employee.class);
c.select(emp);
ParameterExpression<String> deptName = cb.parameter(String.class, "deptName");
c.where(cb.equal(emp.get("dept").get("name"), deptName));

TypedQuery<Student> tq = em.createQuery(c);
tq.setParameter("deptName", "FIAT"); //setting del parametro nella query
```

Subquery

- Le sottoquery “semplici” vengono costruite come due query criteria separate e poi unite da un operatore. Ad esempio:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Employee> c = cb.createQuery(Employee.class);
Root<Employee> emp = c.from(Employee.class);
c.select(emp);
```

unita a

```
Subquery<Employee> sq = c.subquery(Employee.class);
Root<Project> project = sq.from(Project.class);
Join<Project,Employee> sqEmp = project.join("employees");
sq.select(sqEmp).where(cb.equal(project.get("name"),
cb.parameter(String.class, "project")));

c.add(cb.in(emp).value(sq)); //punto di unione
}
```

Domande

- Quali delle seguenti 2 possibile risposte sono vere per la ROOT di una query criteria?
  - **Una query deve definire una ROOT**
  - Una query deve avere obbligatoriamente una ROOT solo se si naviga verso la relativa entità
  - Una query può anche non avere una ROOT
  - **Una query può avere diverse ROOTs**

Una query deve avere come minimo una root ma può averne più di una

-

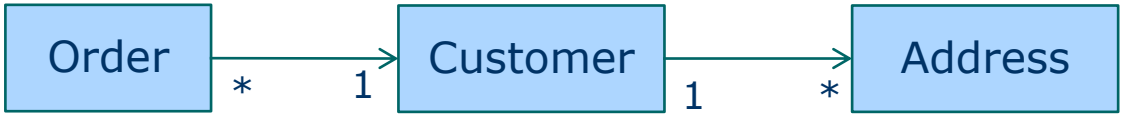
Domande

- Quale dei seguenti pezzi di codice mostra come creare correttamente una query criteria fortemente tipizzata? (cb e em sono oggetti validi)

1. **CriteriaQuery<Office> cq = cb.createQuery(Office.class);**
   …
   TypedQuery<Office> tq = em.createQuery(cq);
   List<Office> offices = tq.getResultList();
2. CriteriaQuery cq = cb.createQuery(Office.class); //mancano i generics
   …
   TypedQuery<Office> tq = em.createQuery(cq, Office.class); //la firma del metodo è errata
   List<Office> offices = tq.getResultList();
3. CriteriaQuery<Office> cq = em.createQuery(Office.class); //cb.createQuery(Office.class);
   …
   TypedQuery<Office> tq = em.createQuery(cq);
   List<Office> offices = tq.getResultList();
4. CriteriaQuery<Office> cq = cb.createQuery(Office.class);
   …
   TypedQuery<Office> tq = cb.createQuery(cq); // em.createQuery(cq);
   List<Office> offices = tq.getResultList();

Domande

- Dati 3 entities: Order, Customer e Address così definiti:

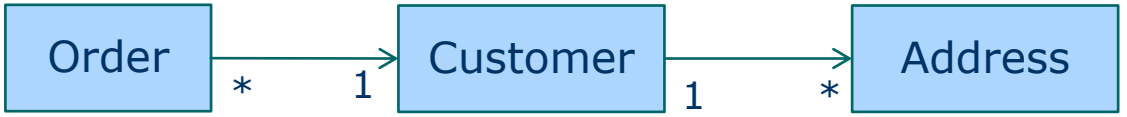


vogliamo scrivere una query criteria che ritorni gli ordini di tutti i customers che hanno un indirizzo il cui valore è espresso dal parametro String postalCode

```
CriteriaBuilder cb = …
CriteriaQuery<Order> cq = cb.createQuery(Order.class);
Root<Order> order = cq.from(Order.class);
Join<Order,Customer> customer = order.join(Order_.customer);
Join<Customer,Address> address= order.join(Customer_.addresses);
cq.where(cb.equals(address.get(Address_.postalCode), postalCode));
cq.select(order).distinct(true);
//esecuzione della query
```

Domande

- Dati 3 entities: Order, Customer e Address così definiti:



la seguente query seleziona il customer con l'ordine che ha il prezzo totale più alto (rispetto agli altri ordini degli altri customers)

```
CriteriaBuilder cb = …
CriteriaQuery<Customer> cq = cb.createQuery(Customer.class);
Root<Customer> c= cq.from(Customer.class);
Join<Customer,Order> o= c.join(Customer_.orders);
Join<Customer,Address> address= order.join(Customer_.addresses);
cq.select(c).distinct(true);

Subquery<Double> sq = cb.subquery(Double.class);
Root<Order> subo = sq.from(Order.class);
Join<Project,Employee> sqEmp = project.join("employees");
sq.select(cb.max(subo.get(Order_.totalPrice);

cq.where(cb.equal(o.get(Order_.totalePrice), cb.all(sq))); //punto di unione
```

Domande

- Dati 2 entities: Order, Customer e 1 embeddable class Address così definiti:

la seguente query seleziona GLI ordini effettuati da “James Brown” ORDINATI PER QUANTITA

```
CriteriaBuilder cb = …
CriteriaQuery<Order> cq = cb.createQuery(Order.class);
Root<Customer> c= cq.from(Customer.class);
Join<Customer,Order> o= c.join(Customer_.orders);
cq.where(cb.equals(c.get(Customer_.name), “James Brown”));
cq.orderBy(cb.asc(o.get(Order_.quantity));
cq.select(o);
```

Domande

- Dati 2 entities: Order, Customer e 1 embeddable class Address così definiti:

la seguente query seleziona il numero di ordini effettuati dai cliente raggruppati per country

```
CriteriaBuilder cb = …
CriteriaQuery<Order> cq = cb.createQuery(Order.class);
Root<Customer> c= cq.from(Customer.class);
Join<Customer,Order> o= c.join(Customer_.orders);
cq.multiselect(cb.count(o), c.get(Customer_.name).get(Address_.country));
cq.groupBy(c.get(Customer_.name).get(Address_.country));
```