Dynamic, typesafe queries in JPA 2.0

How the Criteria API builds dynamic queries and reduces run-time failures

Pinaki Poddar September 22, 2009

A query for persistent Java™ objects is typesafe if a compiler can verify it for syntactic correctness. Version 2.0 of the Java Persistence API (JPA) introduces the Criteria API, which brings the power of typesafe queries to Java applications for the first time and provides a mechanism for constructing queries dynamically at run time. This article describes how to write dynamic, typesafe queries using the Criteria API and the closely associated Metamodel API.

The Java developer community has welcomed JPA since its introduction in 2006. The next major update of the specification — version 2.0 (JSR 317) — will be finalized later in 2009 (see Resources). One of the key features introduced in JPA 2.0 is the Criteria API, which brings a unique capability to the Java language: a way to develop queries that a Java compiler can verify for correctness at compile time. The Criteria API also includes mechanisms for building queries dynamically at run time.

This article introduces the Criteria API and the closely associated *metamodel* concept. You will learn how to use the Criteria API to develop queries that a Java compiler can check for correctness to reduce run-time errors, in contrast to string-based Java Persistence Query Language (JPQL) queries. And through example queries that use database functions or match a template instance, I'll demonstrate the added power of programmatic query-construction mechanics compared to JPQL queries that use a predefined grammar. The article assumes you have a basic familiarity with Java language programming and common JPA usage such as `EntityManagerFactory` or `EntityManager`.

What's wrong with this JPQL query?

JPA 1.0 introduced JPQL, a powerful query language that's considered a major reason for JPA's popularity. However, JPQL — being a string-based query language with a definite grammar — has a few limitations. To understand one of the major limitations, consider the simple code fragment in Listing 1, which executes a JPQL query to select the list of `Person`s older than 20 years:

Listing 1. A simple (and wrong) JPQL query

```
EntityManager em = ...;
String jpql = "select p from Person where p.age > 20";
Query query = em.createQuery(jpql);
List result = query.getResultList();
```

This basic example shows the following key aspects of the query-execution model in JPA 1.0:

- A JPQL query is specified as a `String` (line 2).
- `EntityManager` is the factory that constructs an *executable* query instance given a JPQL string (line 3).
- The result of query execution consists of the elements of an untyped `java.util.List`.

But this simple example has a serious error. Effectively, the code will compile happily, but it will *fail* at run time because the JPQL query string is syntactically incorrect. The correct syntax for the second line of Listing 1 is:

```
String jpql = "select p from Person p where p.age > 20";
```

Unfortunately, the Java compiler has no way to detect such an error. The error will be encountered at run time at line 3 or line 4 (depending on whether the JPA provider parses a JPQL string according to JPQL grammar during query construction or execution).

How does a typesafe query help?

One of the major advantages of the Criteria API is that it prohibits the construction of queries that are syntactically incorrect. Listing 2 rewrites the JPQL query in Listing 1 using the `CriteriaQuery` interface:

Listing 2. Basic steps of writing a `CriteriaQuery`

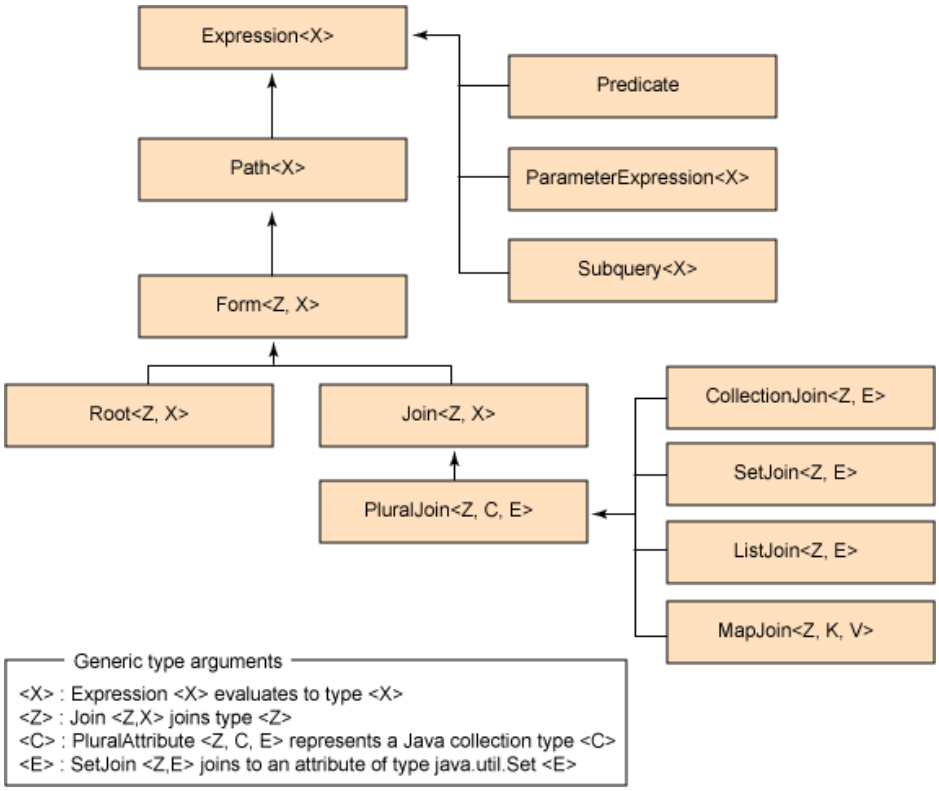
```
EntityManager em = ...
CriteriaBuilder qb = em.getCriteriaBuilder();
CriteriaQuery<Person> c = qb.createQuery(Person.class);
Root<Person> p = c.from(Person.class);
Predicate condition = qb.gt(p.get(Person_.age), 20);
c.where(condition);
TypedQuery<Person> q = em.createQuery(c);
List<Person> result = q.getResultList();
```

Listing 2 illustrates the Criteria API's core constructs and demonstrates its basic usage:

- Line 1 obtains an `EntityManager` instance by one of the several available means.
- In line 2, `EntityManager` creates an instance of `CriteriaBuilder`. `CriteriaBuilder` is the factory for `CriteriaQuery`.
- In line 3, the `CriteriaBuilder` factory constructs a `CriteriaQuery` instance. A `CriteriaQuery` is generically typed. The generic type argument declares the type of result this `CriteriaQuery` will return upon execution. You can supply various kinds of result-type arguments — from a persistent entity such as `Person.class` to a more free-form one such as `Object[]`— while constructing a `CriteriaQuery`.

- In line 4, *query expressions* are set on the `CriteriaQuery` instance. Query expressions are the core units or nodes that are assembled in a tree to specify a `CriteriaQuery`. Figure 1 shows the hierarchy of query expressions defined in the Criteria API:

Figure 1. Interface hierarchy of query expressions



To begin with, the `CriteriaQuery` is set to query *from* `Person.class`. As a result, a `Root<Person>` instance `p` is returned. `Root` is a query expression that denotes the extent of a persistent entity. `Root<T>` essentially says: "Evaluate this query across all instances of type `T`." It is similar to the `FROM` clause of a JPQL or SQL query. Also notice that `Root<Person>` is generically typed. (Actually, every expression is.) The type argument is the type of the value the expression evaluates to. So `Root<Person>` denotes an expression that evaluates to `Person.class`.

- Line 5 constructs a `Predicate`. `Predicate` is another common form of query expression that evaluates to either true or false. A predicate is constructed by the `CriteriaBuilder`, which is the factory not only for `CriteriaQuery`, but also for query expressions. `CriteriaBuilder` has API methods for constructing all kinds of query expressions that are supported in traditional JPQL grammar, plus a few more. In Listing 2, `CriteriaBuilder` is used to construct an expression that evaluates whether the value of its first expression argument is numerically greater than the value of the second argument. The method signature is:

```
Predicate gt(Expression<? extends Number> x, Number y);
```

This method signature is a fine example of how a strongly typed language such as the Java language can be judiciously used to define an API that allows you to express what is correct and prohibits what is not. The method signature specifies that it is possible to compare

an expression whose value is a `Number` only to another `Number` (and not, for example, to a `String`):

```
Predicate condition = qb.gt(p.get(Person_.age), 20);
```

But there is more to line 5. Notice the `qb.gt()` method's first input argument: `p.get(Person_.age),` where `p` is the `Root<Person>` expression obtained previously. `p.get(Person_.age)` is a *path expression*. A path expression is the result of navigation from a root expression via one or more persistent attribute(s). Hence `p.get(Person_.age)` denotes a path expression by navigating from the root expression `p` by the `age` attribute of `Person`. You may wonder what `Person_.age` is. For the time being, assume it is a way to denote the `age` attribute of `Person`. I'll elaborate on the meaning of `Person_.age` when I discuss the new Metamodel API introduced in JPA 2.0.

As I mentioned earlier, every query expression is generically typed to denote the type of the value the expression evaluates to. The path expression `p.get(Person_.age)` evaluates to an `Integer` if the `age` attribute in `Person.class` is declared to be of type `Integer` (or `int`). Because of the type safety inherent in the API, the compiler itself will raise an error for a meaningless comparison, such as:

```
Predicate condition = qb.gt(p.get(Person_.age, "xyz"));
```

- Line 6 sets the predicate on the `CriteriaQuery` as its `WHERE` clause.
- In line 7, `EntityManager` creates an executable query given an input `CriteriaQuery`. This is similar to constructing an executable query given a JPQL string as input. But because the input `CriteriaQuery` carries richer type information, the result is a `TypedQuery` that is an extension of the familiar `javax.persistence.Query`. The `TypedQuery`, as the name suggests, knows the type it returns as a result of its execution. It is defined as:

```
public interface TypedQuery<T> extends Query {
            List<T> getResultList();
}
```

As opposed to corresponding untyped super-interface:

```
public interface Query {
List getResultList();
}
```

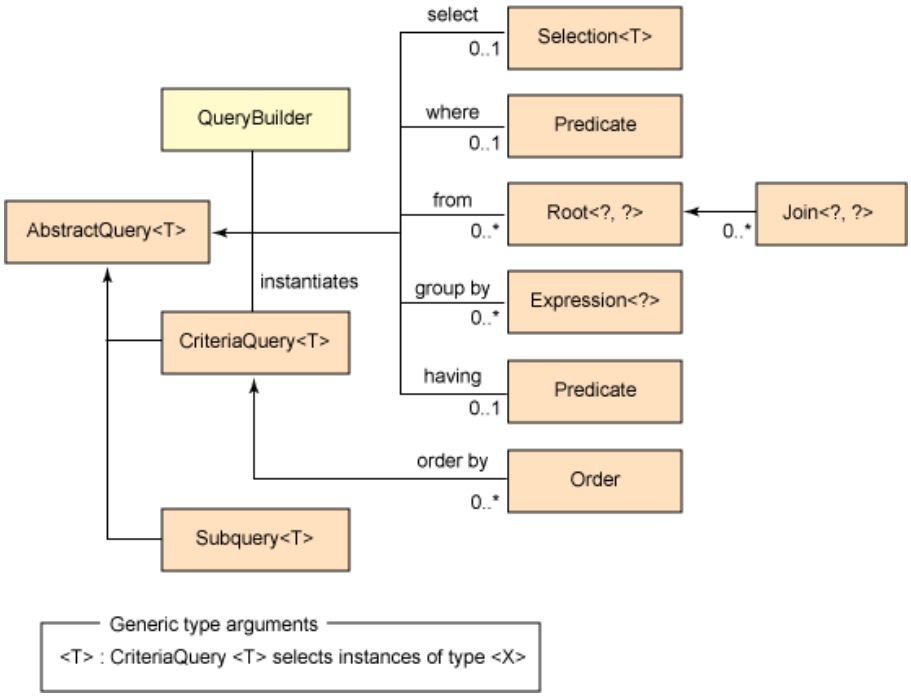
Naturally, the `TypedQuery` result has the same `Person.class` type specified during the construction of the input `CriteriaQuery` by a `CriteriaBuilder` (line 3).

- In line 8, the type information that is carried throughout shows its advantage when the query is finally executed to get a list of results. The result is a typed list of `Person`s that saves the developer of trouble of an extra (and often ugly) cast (and minimizes the risk of `ClassCastException` error at run time) while iterating through the resultant elements.

To summarize the basic aspects of the simple example in Listing 2:

- `CriteriaQuery` is a tree of query-expression nodes that are used to specify query clauses such as `FROM`, `WHERE`, and `ORDER BY` in a traditional string-based query language. Figure 2 shows the clauses related to a query:

Figure 2. `CriteriaQuery` encapsulates the clauses of a traditional query



- The query expressions are generically typed. A few typical expressions are:
  - `Root<T>`, which is equivalent to a `FROM` clause.
  - `Predicate`, which evaluates to a Boolean value of true or false. (In fact, it is declared as `interface Predicate extends Expression<Boolean>`.)
  - `Path<T>`, which denotes a persistent attribute navigated from a `Root<?>` expression. `Root<T>` is a special `Path<T>` with no parent.
- `CriteriaBuilder` is the factory for `CriteriaQuery` and query expressions of all sorts.
- `CriteriaQuery` is transferred to an executable query with its type information preserved so that the elements of the selected list can be accessed without any run-time casting.

Metamodel of a persistent domain

The discussion of Listing 2 points out an unusual construct: `Person_.age`, which is a designation for the persistent `age` attribute of `Person`. Listing 2 uses `Person_.age` to form a path expression navigating from a `Root<Person>` expression `p` by `p.get(Person_.age)`. `Person_.age` is a public static field in the `Person_` class, and `Person_` is the *static, instantiated, canonical metamodel class* corresponding to the original `Person` entity class.

A metamodel class describes the meta information of a persistent class. A metamodel class is *canonical* if the class describes the meta information of a persistent entity in the exact manner stipulated by the JPA 2.0 specification. A canonical metamodel class is *static* in the sense all its member variables are declared `static` (and `public`). The `Person_.age` is one such static member variable. You *instantiate* the canonical class by generating a concrete `Person_.java` at a source-code level at development time. Through such instantiation, it is possible to refer to persistent attributes of `Person` at compile time, rather than at run time, in a strongly typed manner.

This `Person_` metamodel class is an alternative means of referring to meta information of `Person`. This alternative is similar to the much-used (some may say, abused) Java Reflection API, but with a major conceptual difference. You can use reflection to obtain the meta information about an instance of a `java.lang.Class`, but meta information about `Person.class` cannot be referred to in a way that a compiler can verify. For example, using reflection, you'd refer to the field named `age` in `Person.class` with:

```
Field field = Person.class.getField("age");
```

However, this technique is fraught with a limitation similar to the one you observed in the case of the string-based JPQL query in Listing 1. The compiler feels happy about this piece of code but cannot verify whether it will work. The code can fail at run time if it includes even a simple typo. Reflection will not work for what JPA 2.0's typesafe query API wants to accomplish.

A typesafe query API must enable your code to refer to the persistent attribute named `age` in a `Person` class in a way that a compiler can verify at compile time. The solution that JPA 2.0 provides is the ability to instantiate a metamodel class named `Person_` that corresponds to `Person` by exposing the same persistent attributes statically.

Any discussion about meta or meta-meta information often induces somnolence. So I'll present a concrete example of a metamodel class for a familiar Plain Old Java Object (POJO) entity class — `domain.Person`, shown in Listing 3:

Listing 3. A simple persistent entity

```
package domain;
@Entity
public class Person {
  @Id
  private long ssn;
  private string name;
  private int age;

  // public gettter/setter methods
  public String getName() {...}
}
```

This is a typical definition of a POJO, with annotations — such as `@Entity` or `@Id` — that enable a JPA provider to manage instances of this class as persistent entities.

The corresponding static canonical metamodel class of `domain.Person` is shown in Listing 4:

Listing 4. Canonical metamodel for a simple entity

```
package domain;
import javax.persistence.metamodel.SingularAttribute;

@javax.persistence.metamodel.StaticMetamodel(domain.Person.class)

public class Person_ {
  public static volatile SingularAttribute<Person,Long> ssn;
  public static volatile SingularAttribute<Person,String> name;
  public static volatile SingularAttribute<Person,Integer> age;
}
```

The metamodel class declares each persistent attribute of the original `domain.Person` entity as a static public field of `SingularAttribute<Person,?>` type. Making use of this `Person_` metamodel class, I can refer to the persistent attribute of `domain.Person` named `age` — not via the Reflection API, but as a direct reference to the static `Person_.age` field — at compile time. The compiler can then enforce type checking based on the declared type of the attribute named `age`. I've already cited an example of such a restriction: `CriteriaBuilder.gt(p.get(Person_.age), "xyz")` will cause a compiler error because the compiler can determine from the signature of `CriteriaBuilder.gt(..)` and type of `Person_.age` that a `Person`'s `age` is a numeric field and cannot be compared against a `String`.

A few other key points to notice are:

- The metamodel `Person_.age` field is declared to be of type `javax.persistence.metamodel.SingularAttribute. SingularAttribute` is one of the interfaces defined in the JPA Metamodel API, which I'll describe in the next section. The generic type arguments of a `SingularAttribute<Person, Integer>` denote the class that declares the original persistent attribute and the type of the persistent attribute itself.
- The metamodel class is annotated as `@StaticMetamodel(domain.Person.class)` to designate it as a metamodel class corresponding to the original persistent `domain.Person` entity.

The Metamodel API

I've defined a metamodel class as a description of a persistent entity class. Just as the Reflection API requires other interfaces — such as `java.lang.reflect.Field` or `java.lang.reflect.Method` — to describe the constituents of `java.lang.Class`, so the JPA Metamodel API requires other interfaces, such as `SingularAttribute`, `PluralAttribute,` to describe a metamodel class's types and their attributes.

Figure 3 shows the interfaces defined in the Metamodel API to describe types:

Figure 3. Interface hierarchy for persistent types in the Metamodel API

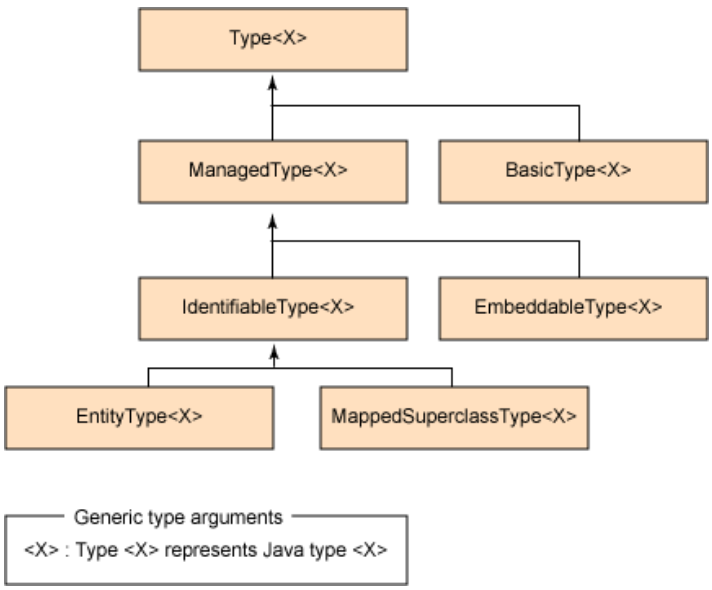
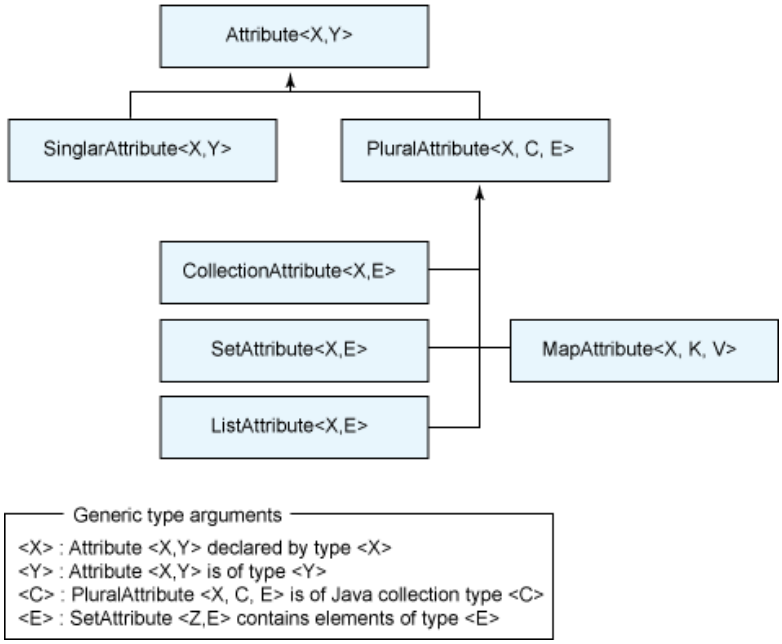


Figure 4 shows the interfaces defined in the Metamodel API to describe attributes:

Figure 4. Interface hierarchy of persistent attributes in the Metamodel API



The interfaces of JPA's Metamodel API are more specialized than those of the Java Reflection API. This finer distinction is required to express rich meta information about persistence. For example, the Java Reflection API represents all Java types as `java.lang.Class`. That is, no special distinction is made via separate definitions among concepts such as class, abstract class, and an interface. Of course, you can ask a `Class` whether it is an interface or if it is abstract — but that is not the same as representing the concept of interface differently from an abstract class via two separate definitions.

The Java Reflection API was introduced at the inception of the Java language (and was quite a pioneering concept at that time for a common general-purpose programming language), but awareness of the use and power of strongly typed systems has progressed over the years. The JPA Metamodel API harnesses that power to introduce strong typing for persistent entities. For example, persistent entities are semantically distinguished as `MappedSuperClass`, `Entity`, and `Embeddable`. Before JPA 2.0, this semantic distinction was represented via corresponding class-level annotations in the persistent-class definition. JPA Metamodel describes three separate interfaces — `MappedSuperclassType`, `EntityType`, and `EmbeddableType` — in the `javax.persistence.metamodel` package to bring their semantic specialties into sharper focus. Similarly, the persistent attributes are distinguished at type-definition level via interfaces such as `SingularAttribute`, `CollectionAttribute`, and `MapAttribute`.

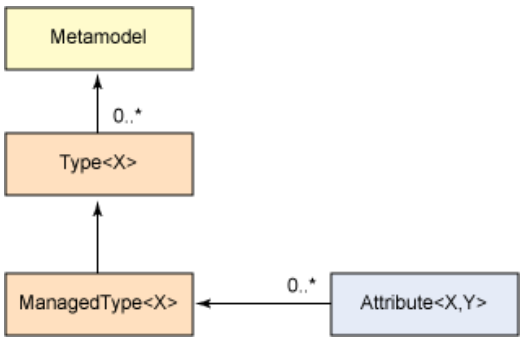
Aside from the aesthetics of description, these specialized metamodel interfaces have practical advantages that help to build typesafe queries and reduce the chance of run-time errors. You've seen some of these advantages in the earlier examples, and you will see more of them when I describe examples of joins using `CriteriaQuery`.

Run-time scope

Broadly speaking, one can draw some parallels between the traditional interfaces of the Java Reflection API and the interfaces of `javax.persistence.metamodel` specialized to describe persistence metadata. To further the analogy, an equivalent concept of run-time scope is needed for the metamodel interfaces. The `java.lang.Class` instances are scoped by `java.lang.ClassLoader` at run time. A set of Java class instances that reference one another must all be defined under the scope of a `ClassLoader`. The set boundaries are *strict* or *closed* in the sense that if a class `A` defined under the scope of `ClassLoader L` tries to refer to class `B`, which is not under the scope of `ClassLoader L`, the result is a dreaded `ClassNotFoundException` or `NoClassDef FoundError` (and often sleep deprivation for a developer or deployer for environments with multiple `ClassLoaders`).

This notion of run-time scope as a strict set of mutually referable classes is captured in JPA 1.0 as a *persistence unit*. The scope of a persistence unit in terms of its persistent entities is enumerated in the `<class>` clause of a META-INF/persistence.xml file. In JPA 2.0, the scope is made available to the developer at run time via the `javax.persistence.metamodel.Metamodel` interface. The `Metamodel` interface is the holder of all persistent entities known to a specific persistence unit, as illustrated in Figure 5:

Figure 5. Metamodel interface is the container of types in a persistence unit



This interface lets the metamodel elements be accessed by their corresponding persistent-entity class. For example, to obtain a reference to the persistent metadata for a `Person` persistent entity, you can write:

```
EntityManagerFactory emf = ...;
Metamodel metamodel = emf.getMetamodel();
EntityType<Person> pClass = metamodel.entity(Person.class);
```

This is analogous, with slightly different style and idioms, to obtaining a `Class` by its name via `ClassLoader`:

```
ClassLoader classloader =  Thread.currentThread().getContextClassLoader();
Class<?> clazz = classloader.loadClass("domain.Person");
```

`EntityType<Person>` can be browsed at run time to get the persistent attributes declared in the `Person` entity. If the application invokes a method on `pClass` such as `pClass.getSingularAttribute("age", Integer.class)`, it will return a

`SingularAttribute<Person, Integer>` instance that is effectively the same as the static `Person_.age` member of the instantiated canonical metamodel class. Essentially, the persistent attribute that the application can refer to at run time via the Metamodel API is made available to a Java compiler by instantiation of the static canonical metamodel `Person_` class.

Apart from resolving a persistent entity to its corresponding metamodel elements, the Metamodel API also allows access to all the known metamodel classes (`Metamodel.getManagedTypes()`) or access to a metamodel class by its persistence-specific information — for example `embeddable(Address.class)`, which returns a `EmbeddableType<Address>` instance that is a subinterface of `ManagedType<>`.

In JPA, the meta information about a POJO is further attributed with persistent specific meta information — such as whether a class is embedded or which fields are used as primary key — with source-code level annotations (or XML descriptors). The persistent meta information falls into two broad categories: for persistence (such as `@Entity`) and for mapping (such as `@Table`). In JPA 2.0, the metamodel captures the metadata *only* for persistence annotations — *not* for the mapping annotation. Hence, with the current version of the Metamodel API, it's possible to know which fields are persistent, but it's *not* possible to find out which database columns they are mapped to.

Canonical vs. non-canonical

Although the JPA 2.0 specification stipulates the exact shape of a canonical static metamodel class (including the metamodel class's fully qualified name and the names of its static fields), it is entirely possible for an application to write these metamodel classes as well. If the application developer writes the metamodel classes, they are called *non-canonical metamodel*. Currently, the specification for non-canonical metamodel is not very detailed, and support for non-canonical metamodel can be nonportable across the JPA providers. You may have noticed that the public static fields are only *declared* in canonical metamodel but not initialized. The declaration makes it possible to refer to these fields during development of a `CriteriaQuery`. But they must be assigned a value to be useful at run time. While it is the responsibility of the JPA provider to assign values to these fields for canonical metamodel, similar warranty is not extended for non-canonical metamodel. Applications that use non-canonical metamodel must either depend on specific vendor mechanisms or devise their own mechanics to initialize the field values to metamodel attributes at run time.

Code generation and usability

Automatic source-code generation often raises eyebrows. The case of generated source code for canonical metamodel adds some concerns. The generated classes are used during development, and other parts of the code that build the `CriteriaQuery` directly refer to them at compile time, leaving some usability questions:

- Should the source-code files be generated in the same directory as the original source, in a separate directory, or relative to the output directory?
- Should the source-code files be checked in a version-controlled configuration-management system?
- How should the correspondence between an original `Person` entity definition and its canonical `Person_` metamodel be maintained? For example, what if Person.java is edited to add an extra persistent attribute, or refactored to rename a persistent attribute?

The answers to these questions are not definitive at the time of this writing.

Annotation processing and metamodel generation

Quite naturally, if you have many persistent entities you will not be inclined to write the metamodel classes yourself. The persistence provider is *expected* to generate these metamodel classes for you. The specification doesn't mandate such a facility or the generation mechanics, but an implicit understanding among JPA providers is that they'll generate the canonical metamodel using the Annotation Processor facility integrated in the Java 6 compiler. Apache OpenJPA provides a utility to generate these metamodel classes either implicitly when you compile the source code for persistent entities or by explicitly invoking a script. Prior to Java 6, an annotation processor tool called `apt` was available and widely used — but with Java 6, the coupling between compiler and Annotation Processor is defined as part of the standard.

The process of generating these metamodel classes in OpenJPA as your persistence provider is as simple as compiling the POJO entity with OpenJPA class libraries in the compiler's classpath:

```
$ javac domain/Person.java
```

The canonical metamodel `Person_` class will be generated, written in the same source directory as Person.java, and compiled as a side-effect of this compilation.

Writing queries in a typesafe manner

So far, I've established the components of `CriteriaQuery` and its associated metamodel classes. Now I'll show you how to develop some queries with the Criteria API.

Functional expressions

Functional expressions apply a function to one or more input arguments to create a new expression. The functional expression's type depends on the nature of the function and type of its arguments. The input arguments themselves can be expressions or literal values. The compiler's type-checking rules coupled with the API's signature govern what constitutes legitimate input.

Consider a single-argument expression that applies averaging on its input expression. Listing 5 shows the `CriteriaQuery` to select the average balance of all `Accounts`:

Listing 5. Functional expression in `CriteriaQuery`

```
CriteriaQuery<Double> c = cb.createQuery(Double.class);
Root<Account> a = c.from(Account.class);

c.select(cb.avg(a.get(Account_.balance)));
```

An equivalent JPQL query would be:

```
String jpql = "select avg(a.balance) from Account a";
```

Fluent API

As this example shows, the Criteria API methods often return the type that can be directly used in a related method, thereby providing a popular programming style known as Fluent API.

In Listing 5, the `CriteriaBuilder` factory (represented by the variable `cb`) creates an `avg()` expression and uses that expression in the query's `select()` clause.

The query expression is a building block that can be assembled to define the final selection predicate for the query. The example in Listing 6 shows a `Path` expression created by navigating to the balance of `Account,` and then the `Path` expression is used as an input expression in a couple of binary functional expressions — `greaterThan()` and `lessThan()`— both of which result in a Boolean expression or simply a predicate. The predicates are then combined via an `and()` operation to form the final selection predicate to be evaluated by the query's `where()` clause:

Listing 6. `where()` predicate in `CriteriaQuery`

```
CriteriaQuery<Account> c = cb.createQuery(Account.class);
Root<Account> account = c.from(Account.class);
Path<Integer> balance = account.get(Account_.balance);
c.where(cb.and
       (cb.greaterThan(balance, 100),
        cb.lessThan(balance), 200)));
```

An equivalent JPQL query would be:

```
"select a from Account a where a.balance>100 and a.balance<200";
```

Complex predicates

Certain expressions — such as `in()`— apply to a variable number of expressions. Listing 7 shows an example:

Listing 7. Multivalued expression in `CriteriaQuery`

```
CriteriaQuery<Account> c = cb.createQuery(Account.class);
Root<Account> account = c.from(Account.class);
Path<Person> owner = account.get(Account_.owner);
Path<String> name = owner.get(Person_.name);
c.where(cb.in(name).value("X").value("Y").value("Z"));
```

This example navigates from `Account` via two steps to create a path expression representing the name of an account's owner. Then it creates an `in()` expression with the path expression as input. The `in()` expression evaluates if its input expression equals any of its variable number of arguments. These arguments are specified through the `value()` method on the `In<T>` expression, which has the following method signature:

```
In<T> value(T value);
```

Notice how Java generics are used to specify that an `In<T>` expression can be evaluated for membership only with values of type `T`. Because the path expression representing the `Account` owner's name is of type `String`, the only valid comparison is against `String`-valued arguments, which can be either a literal or another expression that evaluates to `String`.

Contrast the query in Listing 7 with the equivalent (correct) JPQL:

```
"select a from Account a where a.owner.name in ('X','Y','Z')";
```

A slight oversight in the JPQL will not only be undetected by the compiler but also will produce an unintended outcome. For example:

```
"select a from Account a where a.owner.name in (X, Y, Z)";
```

Joining relationships

Although the examples in Listing 6 and Listing 7 use expressions as the building blocks, the queries are based on a single entity and its attributes. But often queries involve more than one entity, which requires you to *join* two or more entities. `CriteriaQuery` expresses joining two entities by *typed join expressions*. A typed join expression has two type parameters: the type you are joining from and the bindable type of the attribute being joined. For example, if you want to query for the `Customer`s whose one or more `PurchaseOrder`(s) are not delivered yet, you need to express this by an expression that joins `Customer` to `PurchaseOrder`s, where `Customer` has a persistent attribute named `orders` of type `java.util.Set<PurchaseOrder>`, as shown in Listing 8:

Listing 8. Joining a multivalued attribute

```
CriteriaQuery<Customer> q = cb.createQuery(Customer.class);
Root<Customer> c = q.from(Customer.class);
SetJoin<Customer, PurchaseOrder> o = c.join(Customer_.orders);
```

The join expression created from the root expression `c` and the persistent `Customer.orders` attribute is parameterized by the source of join — that is, `Customer` — and the bindable type of the `Customer.orders` attribute, which is `PurchaseOrder` and *not* the declared type `java.util.Set<PurchaseOrder>`. Also notice that because the original attribute is of type `java.util.Set`, the resultant join expression is `SetJoin`, which is a specialized `Join` for an attribute of declared type `java.util.Set`. Similarly, for other supported multivalued persistent attribute types, the API defines `CollectionJoin`, `ListJoin`, and `MapJoin`. (Figure 1 shows the various join expressions.) There is no need for an explicit cast in the third line in Listing 8 because `CriteriaQuery` and the Metamodel API recognize and distinguish attribute types that are declared as `java.util.Collection` or `List` or `Set` or `Map` by overloaded methods for `join()`.

The joins are used in queries to form a predicate on the joined entity. So if you want to select the `Customer`s with one or more undelivered `PurchaseOrder`(s), you can define a predicate by navigating from the joined expression `o` via its status attribute, comparing it with `DELIVERED` status, and negating the predicate as:

```
Predicate p = cb.equal(o.get(PurchaseOrder_.status), Status.DELIVERED)
        .negate();
```

One noteworthy point about creating a join expression is that every time you join from an expression, it returns a new join expression, as shown in Listing 9:

Listing 9. Every join creates a unique instance

```
SetJoin<Customer, PurchaseOrder> o1 = c.join(Customer_.orders);
SetJoin<Customer, PurchaseOrder> o2 = c.join(Customer_.orders);
assert o1 == o2;
```

The assertion in Listing 9 for equality of two join expressions from the same expression `c` with the same attribute will fail. Thus, if a query involves a predicate for `PurchaseOrder`s that are not delivered and whose value is more than $200, then the correct construction is to join `PurchaseOrder` with the root `Customer` expression only once, assign the resultant join expression to a local variable (equivalent to a range variable in JPQL terminology), and use the local variable in forming the predicate.

Using parameters

Take a look back at this article's original JPQL query (the correct one):

```
String jpql = "select p from Person p where p.age > 20";
```

Though queries are often written with constant literals, it is not a good practice. The good practice is to parameterize a query, which allows the query to be parsed or prepared only once, cached, and reused. So a better way to write the query is to use a named parameter:

```
String jpql = "select p from Person p where p.age > :age";
```

A parameterized query binds the value of the parameter before the query execution:

```
Query query = em.createQuery(jpql).setParameter("age", 20);
List result = query.getResultList();
```

In a JPQL query, the parameters are encoded in the query string as either named (preceded by a colon — for example, `:age`) or positional (preceded by a question mark — for example, `?3`). In `CriteriaQuery`, the parameters themselves are query expressions. Like any other expression, they are strongly typed and constructed by the expression factory — namely, `CriteriaBuilder`. The query in Listing 2, then, can be parameterized as shown in Listing 10:

Listing 10. Using parameters in a `CriteriaQuery`

```
ParameterExpression<Integer> age = qb.parameter(Integer.class);
Predicate condition = qb.gt(p.get(Person_.age), age);
c.where(condition);
TypedQuery<Person> q = em.createQuery(c);
List<Person> result = q.setParameter(age, 20).getResultList();
```

To contrast the usage of parameters to that of JPQL: the parameter expression is created with explicit type information to be an `Integer` and is directly used to bind a value of `20` to the executable query. The extra type information is often useful for reducing run-time errors, because it prohibits the parameter from being compared against an expression of an incompatible type or being bound with a value of an inadmissible type. Neither form of compile-time safety is warranted for the parameters of a JPQL query.

The example in Listing 10 shows an unnamed parameter expression that is directly used for binding. It is also possible to assign a name to the parameter as the second argument during its construction. In that case, you can bind the parameter value to the query using that name. What is not possible, however, is use of positional parameters. Integral position in a (linear) JPQL query string makes some intuitive sense, but the notion of an integral position cannot be carried forward in the context of `CriteriaQuery`, where the conceptual model is a tree of query expressions.

Another interesting aspect of JPA query parameters is that they do not have intrinsic value. A value is bound to a parameter in the context of an executable query. So it is perfectly legal to create two separate executable queries from the same `CriteriaQuery` and bind two different integer values to the same parameter for these executable queries.

Projecting the result

You've seen that the type of result a `CriteriaQuery` will return upon execution is specified up front when a `CriteriaQuery` is constructed by `CriteriaBuilder`. The query's result is specified as one or more *projection terms*. There are two ways to specify the projection term on the `CriteriaQuery` interface:

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

The simplest and often used projection term is the candidate class of the query itself. It can be implicit, as shown in Listing 11:

Listing 11. `CriteriaQuery` selects candidate extent by default

```
CriteriaQuery<Account> q = cb.createQuery(Account.class);
Root<Account> account = q.from(Account.class);
List<Account> accounts = em.createQuery(q).getResultList();
```

In Listing 11, the query from `Account` does not explicitly specify its selection term and is the same as explicitly selecting the candidate class. Listing 12 shows a query that uses an explicit selection term:

Listing 12. `CriteriaQuery` with explicit single selection term

```
CriteriaQuery<Account> q = cb.createQuery(Account.class);
Root<Account> account = q.from(Account.class);
q.select(account);
List<Account> accounts = em.createQuery(q).getResultList();
```

When the projected result of the query is something other than the candidate persistent entity itself, several other constructs are available to shape the result of the query. These constructs are available in the `CriteriaBuilder` interface, as shown in Listing 13:

Listing 13. Methods to shape query result

```
<Y> CompoundSelection<Y> construct(Class<Y> result, Selection<?>... terms);
    CompoundSelection<Object[]> array(Selection<?>... terms);
    CompoundSelection<Tuple> tuple(Selection<?>... terms);
```

The methods in Listing 13 build a compound projection term composed of other selectable expressions. The `construct()` method creates an instance of the given class argument and invokes a constructor with values from the input selection terms. For example, if `CustomerDetails` — a nonpersistent entity — has a constructor that takes `String` and `int` arguments, then a `CriteriaQuery` can return `CustomerDetails` as its result by creating instances from name and age of selected `Customer` — a persistent entity — instances, as shown in Listing 14:

Listing 14. Shaping query result into instances of a class by `construct()`

```
CriteriaQuery<CustomerDetails> q = cb.createQuery(CustomerDetails.class);
Root<Customer> c = q.from(Customer.class);
q.select(cb.construct(CustomerDetails.class,
              c.get(Customer_.name), c.get(Customer_.age));
```

Multiple projection terms can also be combined into a compound term that represents an `Object[]` or `Tuple.` Listing 15 shows how to pack the result into an `Object[]:`

Listing 15. Shaping query result into an `Object[]`

```
CriteriaQuery<Object[]> q = cb.createQuery(Object[].class);
Root<Customer> c = q.from(Customer.class);
q.select(cb.array(c.get(Customer_.name), c.get(Customer_.age));
List<Object[]> result = em.createQuery(q).getResultList();
```

This query returns a result list in which each element is an `Object[]` of length 2, the zero-th array element is `Customer`'s name, and the first element is `Customer`'s age.

`Tuple` is a JPA-defined interface to denote a row of data. A `Tuple` is conceptually a list of `TupleElement`s — where `TupleElement` is the atomic unit and the root of all query expressions. The values contained in a `Tuple` can be accessed by either a 0-based integer index (similar to the familiar JDBC result), an alias name of the `TupleElement`, or directly by the `TupleElement`. Listing 16 shows how to pack the result into a `Tuple:`

Listing 16. Shaping query result into `Tuple`

```
CriteriaQuery<Tuple> q = cb.createTupleQuery();
Root<Customer> c = q.from(Customer.class);
TupleElement<String> tname = c.get(Customer_.name).alias("name");
q.select(cb.tuple(tname, c.get(Customer_.age).alias("age");
List<Tuple> result = em.createQuery(q).getResultList();
String name = result.get(0).get(name);
String age  = result.get(0).get(1);
```

Limitations on nesting

It is theoretically possible to compose complex result shapes by nesting terms such as a `Tuple` whose elements themselves are `Object[]`s or `Tuple`s. However, the JPA 2.0 specification prohibits such nesting. The input terms of a `multiselect()` cannot be an array or tuple-valued compound term. The only compound terms allowed as `multiselect()` arguments are ones created by the `construct()` method (which essentially represent a single element).

However, OpenJPA does not put any restriction on nesting one compound selection term inside another.

This query returns a result list each element of which is a `Tuple`. Each tuple, in turn, carries two elements — accessible either by index or by the alias, if any, of the individual `TupleElement`s, or directly by the `TupleElement`. Two other noteworthy points in Listing 16 are the use of `alias()`, which is a way to attach a name to any query expression (creating a new copy as a side-effect), and a `createTupleQuery()` method on `CriteriaBuilder`, which is merely an alternative to `createQuery(Tuple.class)`.

The behavior of these individual result-shaping methods and what is specified as the result type argument of the `CriteriaQuery` during construction are combined into the semantics of the `multiselect()` method. This method interprets its input terms based on the result type of the `CriteriaQuery` to arrive at the shape of the result. To construct `CustomerDetails` instances as in Listing 14 using `multiselect()`, you need to specify the `CriteriaQuery` to be of type `CustomerDetails` and simply invoke `multiselect()` with the terms that will compose the `CustomerDetails` constructor, as shown in Listing 17:

Listing 17. `multiselect()` interprets terms based on result type

```
CriteriaQuery<CustomerDetails> q = cb.createQuery(CustomerDetails.class);
Root<Customer> c = q.from(Customer.class);
q.multiselect(c.get(Customer_.name), c.get(Customer_.age));
```

Because the query result type is `CustomerDetails`, `multiselect()` interprets its argument projection terms as the constructor argument to `CustomerDetails`. If the query were specified to return a `Tuple`, the `multiselect()` method with the exact same arguments would create `Tuple` instances instead, as shown in Listing 18:

Listing 18. Creating `Tuple` instances with `multiselect()`

```
CriteriaQuery<Tuple> q = cb.createTupleQuery();
Root<Customer> c = q.from(Customer.class);
q.multiselect(c.get(Customer_.name), c.get(Customer_.age));
```

The behavior of `multiselect()` gets more interesting with `Object` as result type or if no type argument is specified. In such cases, if `multiselect()` is used with a single input term, then the return value is the selected term itself. But if `multiselect()` contains more than one input term, the result is an `Object[]`.

Advanced features

So far, I have mainly emphasized the strongly typed nature of the Criteria API and the fact that it helps to minimize syntactic errors that can creep into string-based JPQL queries. The Criteria API is also a mechanism for building queries programmatically and so is often referred to as a *dynamic* query API. The power of a programmable query construction API is limited only by the inventiveness of its user. I'll present four examples:

- Using a weakly typed version of the API to build dynamic queries
- Using a database-supported function as a query expression to extend the grammar
- Editing a query for search-within-result functionality

- Query-by-example — a familiar pattern popularized by the object-database community

Weak typing and dynamic query building

The Criteria API's strong type checking is based on the availability of instantiated metamodel classes at development time. However, for some use cases, the entities to be selected can only be determined at run time. To support such usage, the Criteria API methods provide a parallel version in which persistent attributes are referred by their names (similar to the Java Reflection API) rather than by reference to instantiated static metamodel attributes. This parallel version of the API can support truly dynamic query construction by sacrificing the type checking at compile time. Listing 19 rewrites the example in Listing 6 using the weakly typed version:

Listing 19. Weakly typed query

```
Class<Account> cls =Class.forName("domain.Account");
Metamodel model = em.getMetamodel();
EntityType<Account> entity = model.entity(cls);
CriteriaQuery<Account> c = cb.createQuery(cls);
Root<Account> account = c.from(entity);
Path<Integer> balance = account.<Integer>get("balance");
c.where(cb.and
       (cb.greaterThan(balance, 100),
        cb.lessThan(balance), 200)));
```

The weakly typed API, however, cannot return correct generically typed expressions, thereby generating a compiler warning for an unchecked cast. One way to get rid of these pesky warning messages is to use a relatively rare facility of Java generics: parameterized method invocation, as shown in Listing 19's invocation of the `get()` method to obtain a path expression.

Extensible datastore expressions

A distinct advantage of a dynamic query-construction mechanism is that the grammar is extensible. For example, you can use the `function()` method in the `CriteriaBuilder` interface to create an expression supported by the database:

```
<T> Expression<T> function(String name, Class<T> type, Expression<?>...args);
```

The `function()` method creates an expression of the given name and zero or more input expressions. The `function()` expression evaluates to the given type. This allows an application to create a query that evaluates a database function. For example, the MySQL database supports a `CURRENT_USER()` function that returns the user-name and host-name combination as a UTF-8 encoded string for the MySQL account that the server used to authenticate the current client. An application can use the `CURRENT_USER()` function, which takes no argument, in a `CriteriaQuery`, as Listing 20 demonstrates:

Listing 20. Using a database-specific function in a `CriteriaQuery`

```
CriteriaQuery<Tuple> q = cb.createTupleQuery();
Root<Customer> c = q.from(Customer.class);
Expression<String> currentUser =
    cb.function("CURRENT_USER", String.class, (Expression<?>[])null);
q.multiselect(currentUser, c.get(Customer_.balanceOwed));
```

Notice that an equivalent query is simply not possible to express in JPQL, because it has a defined grammar with a fixed number of supported expressions. A dynamic API is not strictly limited by a fixed set of expressions.

Editable query

A `CriteriaQuery` can be edited programmatically. The clauses of the query — such as its selection terms, the selection predicate in a `WHERE` clause, and ordering terms in an `ORDER BY` clause — can all be mutated. This editing capability can be used in a typical "search-within-result"-like facility whereby a query predicate is further refined in successive steps by adding more restrictions.

The example in Listing 21 creates a query that orders its result by name and then edits the query to order also by ZIP code:

Listing 21. Editing a `CriteriaQuery`

```
CriteriaQuery<Person> c = cb.createQuery(Person.class);
Root<Person> p = c.from(Person.class);
c.orderBy(cb.asc(p.get(Person_.name)));
List<Person> result = em.createQuery(c).getResultList();
// start editing
List<Order> orders = c.getOrderList();
List<Order> newOrders = new ArrayList<Order>(orders);
newOrders.add(cb.desc(p.get(Person_.zipcode)));
c.orderBy(newOrders);
List<Person> result2 = em.createQuery(c).getResultList();
```

In-memory evaluation in OpenJPA

With OpenJPA's extended features, the search-within-result example in Listing 21 can be made even more efficient by evaluating the edited query *in-memory*. This example dictates that the result of the edited query will be a strict subset of the original result. Because OpenJPA can evaluate a query in-memory when a candidate collection is specified, the only modification required is to the last line of Listing 21 to supply the result of the original query:

```
List<Person> result2 =
em.createQuery(c).setCandidateCollection(result).getResultList();
```

The setter methods on `CriteriaQuery` — `select()`, `where()`, or `orderBy()` — erase the previous values and replace them with new arguments. The list returned by the corresponding getter methods such as `getOrderList()` is not *live* — that is, adding or removing elements on the returned list does not modify the `CriteriaQuery;` furthermore, some vendors may even return an immutable list to prohibit inadvertent usage. So a good practice is to copy the returned list into a new list before adding or removing new expressions.

Query-by-example

Another useful facility in a dynamic query API is that it can support *query-by-example* with relative ease. Query-by-example (developed by IBM® Research in 1970) is often cited as an early example of end-user usability for software. The idea of query-by-example is that instead of specifying the exact predicates for a query, a template instance is presented. Given the template instance, a conjunction of predicates — where each predicate is a comparison for a nonnull,

nondefault attribute value of the template instance — is created. Execution of this query evaluates the predicate to find all instances that match the template instance. Query-by-example was considered for inclusion in the JPA 2.0 specification but is not included. OpenJPA supports this style of query through its extended `OpenJPACriteriaBuilder` interface, as shown in Listing 22:

Listing 22. Query-by-example using OpenJPA's extension of `CriteriaQuery`

```
CriteriaQuery<Employee> q = cb.createQuery(Employee.class);

Employee example = new Employee();
example.setSalary(10000);
example.setRating(1);

q.where(cb.qbe(q.from(Employee.class), example);
```

As this example shows, OpenJPA's extension of the `CriteriaBuilder` interface supports the following expression:

```
public <T> Predicate qbe(From<?, T> from, T template);
```

This expression produces a conjunction of predicates based on the attribute values of the given template instance. For example, this query will find all `Employee`s with a salary of `10000` and rating of `1`. The comparison can be further controlled by specifying an optional list of attributes to be excluded from comparison and the style of comparison for `String`-valued attributes. (See Resources for a link to the Javadoc for OpenJPA's `CriteriaQuery` extensions.)

Conclusion

This article has introduced the new Criteria API in JPA 2.0 as a mechanism for developing dynamic, typesafe queries in the Java language. A `CriteriaQuery` is constructed at run time as a tree of strongly typed query expressions whose use the article has illustrated with a series of code examples.

This article also establishes the critical role of the new Metamodel API and shows how instantiated metamodel classes enable the compiler to verify the correctness of the queries, thereby avoiding run-time errors caused by syntactically incorrect JPQL queries. Besides enforcing syntactic correctness, JPA 2.0's facilities for programmatic construction of queries can lead to more powerful usage, such as query-by-example, using database functions, and — I hope — many other innovative uses of these powerful new APIs that this article's readers will devise.

Acknowledgments

I acknowledge Rainer Kwesi Schweigkoffer for his careful review of this article and valuable suggestions, and fellow members of JPA 2.0 Expert Group for explaining the finer points of this powerful API. I also thank Fay Wang for her contribution, and Larry Kestila and Jeremy Bauer for their support during development of the Criteria API for OpenJPA.

Related topics

- JSR-000317 JavaTM Persistence 2.0
- Apache OpenJPA project
- Package org.apache.openjpa.persistence.criteria
- LIQUidFORM