# Java Persistence Criteria API

Linda DeMichiel
Oracle

JAZOON
THE INTERNATIONAL CONFERENCE ON JAVA TECHNOLOGY
1-3 JUNE 2010, ZURICH

ORACLE

ORACLE netcetera

# AGENDA

- Background: Where we started
- Criteria API objects and interfaces
- Examples and issues
- JPA Metamodel and metamodel objects
- CriteriaQuery definition
- CriteriaQuery execution
- Summary and what's next

# Background: Where We Started
# Java Persistence Query Language (JPQL)

- String-based query language
- SQL-like syntax
  - Operates over “abstract schema” of persistence unit
  - Path navigation syntax for relationships and state traversal
  - Translated by provider into native SQL
- Static queries
  - Metadata (using @NamedQuery annotations and/or XML)
  - In code
- Dynamic queries
  - Runtime string construction

# Background: Where We Started
# Java Persistence Query Language (JPQL)

- JPA 2.0 provides some much-needed improvements
  - Operations in SELECT list
  - Non-polymorphic queries
  - CASE statements
  - Collection-valued IN parameters
  - Date / time / timestamp literals
  - Operations over new features (ordered lists, maps, etc.)
- Syntax extended by vendors

# Background: Where We Started
# Java Persistence Query Language (JPQL)

- Positives
  - Easy to learn SQL-like string-based QL
  - Metadata-based queries provide for precompilation and/or deployment-time configurability
  - Dynamic query construction via straightforward string manipulation
- Negatives
  - Strings!
  - No type-safety
  - Usual security issues with SQL string construction
  - Pre-runtime semantic checks for metadata-based queries only

# Criteria API

- Addresses requests from community for object-based query API
- Several such APIs already available in community when we started
  - Hibernate Criteria API, TopLink Expression API, Cayenne, OJB, …
- Design went through MANY variations and much evolution in Expert Group
- Two main design principles emerged
  - "JPQL-completeness"
  - Type safety
- We heard from many people working on projects addressing similar goals
  - Squill, Querydsl, JaQu, LIQUidForm, …

# Criteria API

- Object-based API
- Strongly typed
  - Heavy use of Java generics
  - Based on type-safe metamodel of persistence unit
  - Typing carries through to query execution
- Designed to mirror JPQL semantics
  - Objects / methods mirror JPQL / SQL syntactic and semantic constructs
- Supports object-based or string-based navigation

# Criteria Query Objects

- CriteriaQuery objects are composites
- Component objects include:
  - Roots
  - Join objects
  - Expressions
  - Predicates
  - Selections
  - Orderings
  - Groupings
  - …
- Some of these are composites as well

# Key Interfaces: CriteriaBuilder

- Factory for CriteriaQuery objects
- Factory for Expression objects, including Predicate objects
- Methods for
  - comparisons
  - operations over strings, numbers, booleans, etc
  - operations over collections
  - operations over subqueries, …
  - CASE and IN-expression builders
  - creating parameters
  - …
- Obtained from EntityManager or EntityManagerFactory
  - getCriteriaBuilder() method

# Example: CriteriaBuilder Methods

```
<E, C extends Collection<E>> Predicate isMember(
        E elem, Expression<C> collection);

<E, C extends Collection<E>> Predicate isMember(
        Expression<E> elem, Expression<C> collection);

Expression<String> concat(Expression<String> x,
                          Expression<String> y);

Expression<String> concat(Expression<String> x, String y);

Expression<String> concat(String x, Expression<String> y);

<Y> Expression<Y> all(Subquery<Y> subquery);
```

# Key Interfaces: CriteriaQuery

- CriteriaQuery object represents a query definition
  - Can be constructed incrementally and/or reconstructed
  - Can be browsed; constituent objects can be extracted
- Methods to assign / replace constituent objects

  select(), multiselect()
  from()
  where()
  orderBy()
  groupBy(), having()
- Methods to browse CriteriaQuery objects

  getSelection(), getRoots(), getRestriction(), getOrderList(), getGroupList(), getGroupRestriction()
- Obtained from CriteriaBuilder
  - createQuery(), createTupleQuery()

# Interfaces that Model Query Components

- Root
  - Query roots
- Join
  - Joins from a root or existing join
  - CollectionJoin, SetJoin, ListJoin, MapJoin subinterfaces
- Path
  - Navigation from a root, join, or path
- Subquery
- Selection, CompoundSelection
- Expression
- Predicate
- …

# Example: The World's Simplest Query

JPQL:

```
SELECT c
FROM Customer c
```

Criteria API:

```
EntityManager em = …
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Customer> cq = cb.createQuery(Customer.class);
Root<Customer> customer = cq.from(Customer.class);
cq.select(customer);

TypedQuery<Customer> tq = em.createQuery(cq);
List<Customer> resultList = tq.getResultList();
```

# Example (with Join)

```
SELECT c
FROM Customer c join c.orders o

EntityManager em = …
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Customer> cq = cb.createQuery(Customer.class);
Root<Customer> customer = cq.from(Customer.class);
Join<Customer, Order> o = customer.join("orders");
cq.select(customer);
```

# Example (with Restriction)

```
SELECT c
FROM Customer c join c.orders o
WHERE c.name = 'Braun'

CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Customer> cq = cb.createQuery(Customer.class);
Root<Customer> customer = cq.from(Customer.class);
Join<Customer, Order> o = customer.join("orders");
cq.where(cb.equal(customer.get("name"), "Braun"))
  .select(customer);
```

# What's the Problem?

```
SELECT c
FROM Customer c join c.orders o
WHERE c.name = 'Braun'

CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Customer> cq = cb.createQuery(Customer.class);
Root<Customer> customer = cq.from(Customer.class);
Join<Customer, Order> o = customer.join("order");
cq.where(cb.equal(customer.get("names"), "Braun"))
  .select(customer);
```

# The Issue: A Hole in the Typing

- How to specify navigation in join() and get() methods
  - Strings vs objects
- With strings:
  - No guarantee referenced objects exist
  - Type safety falls apart
- But:
  - Strings are easier to use + familiar + intuitive
- Source of MUCH discussion in JPA Expert Group
  - Some developers may prefer string-based navigation, so we need to support it
  - However: not the design-center of the API

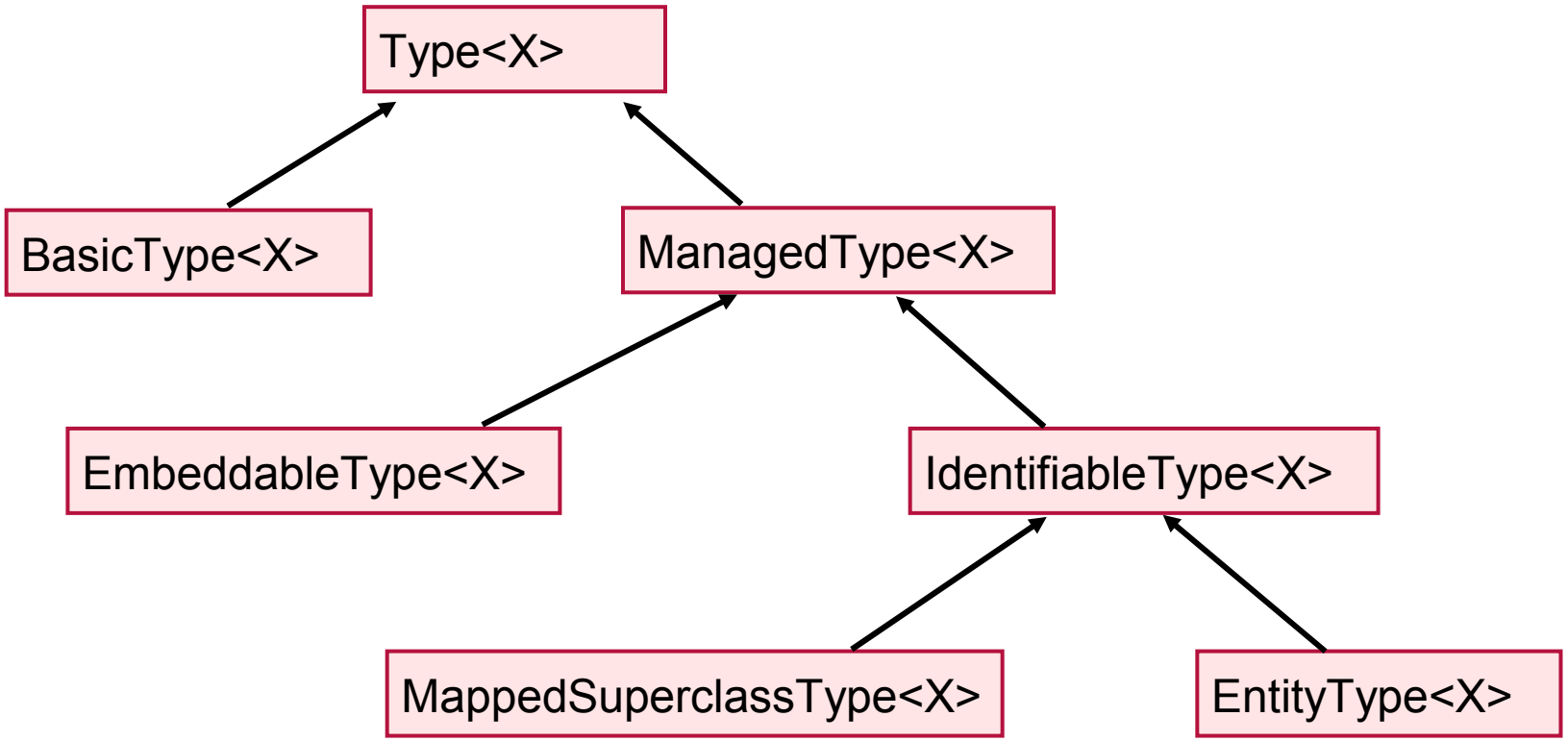
# What's the Solution to Type-safe Navigation?

- Class literals?  (Customer.class, etc.)
  - Used in creating CriteriaQuery objects, Root objects, TypedQuery objects, …
  - Don't work for specifying joins and paths
- Member literals?
  - Oops!!

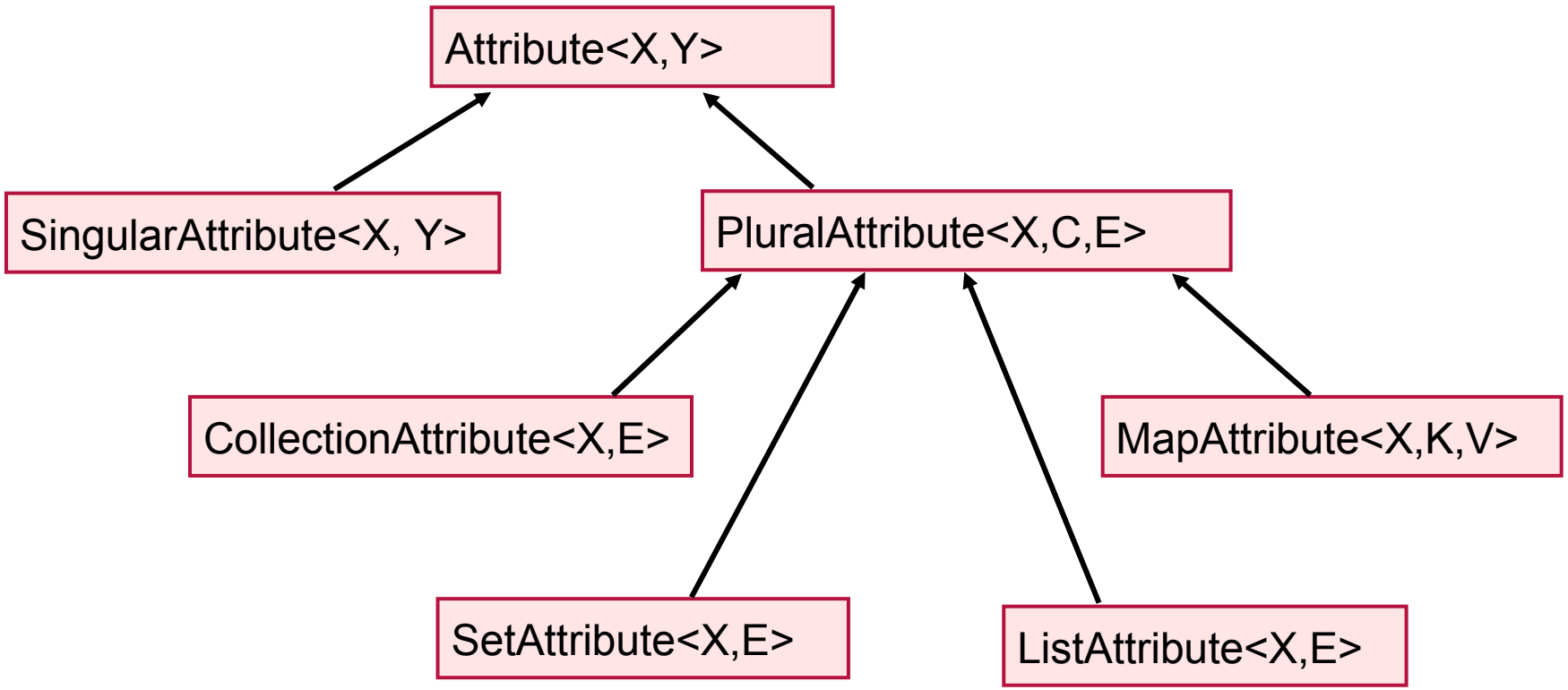
# Metamodel

- View over the “abstract schema” of the persistence unit
  - Entities, embeddables, mapped superclasses, and their attributes
- Accessed at runtime
  - EntityManager.getMetamodel()
  - EntityManagerFactory.getMetamodel()
- Useful for frameworks
  - Can dynamically discover all the managed types, their attributes, and their relationships
- Doesn’t yet cover O / R mapping level
  - Potential candidate for future release

# Metamodel Interfaces: Types



# Metamodel Interfaces: Attributes



# Example: Browsing the Metamodel

```
EntityManager em = ...
Metamodel mm = em.getMetamodel();
Set<EntityType<?>>  myEntities = mm.getEntities();
for (EntityType e : myEntities) {
     System.out.println(e.getName());
     Set<Attribute> entityAttributes = e.getAttributes();
     for (Attribute ea : entityAttributes) {
          System.out.println(ea.getName());
          System.out.println(ea.getDeclaringType());
          System.out.println(ea.isAssociation());
          ...
      }
}
```

# Static Metamodel Classes

- Metamodel can be captured in terms of static metamodel classes
- JPA specification defines canonical form
- Static metamodel classes can be materialized at compile time
  - Use javac together with annotation processor
  - Persistence provider initializes classes when persistence unit is deployed
- Alternatively:
  - You can write them by hand ☺
  - IDE can produce them for you
- Used to create strongly-typed criteria queries
  - The missing link!

# Example: Entity Class

```
@Entity
public class Customer {
    @Id int id;
    String name;
    Address address;
    ...
    @OneToMany Set<Order> orders;
    @ManyToOne SalesRep rep;
    ...
}
```

# Example: Canonical Static Metamodel Class

```
import javax.persistence.metamodel.*

@Generated("EclipseLink JPA 2.0 Canonical Model Generation")
@StaticMetamodel(Customer.class)
public class Customer_ {
    public static volatile SingularAttribute<Customer, Integer> id;
    public static volatile SingularAttribute<Customer, String> name;
    public static volatile SingularAttribute<Customer, Address> address;
    public static volatile SetAttribute<Customer, Order> orders;
    public static volatile SingularAttribute<Customer, SalesRep> rep;
    …
}
```

# Example: With Strings

```
SELECT c
FROM Customer c join c.orders o
WHERE c.name = 'Braun'

CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Customer> cq = cb.createQuery(Customer.class);
Root<Customer> customer = cq.from(Customer.class);
Join<Customer, Order> o = customer.join("orders");
cq.where(cb.equal(customer.get("name"), "Braun"))
  .select(customer);
```

# String-based Navigation Methods

```
<Y> Path<Y> get(String attributeName);

<X,Y> Join<X,Y> join(String attributeName);
```

# Strongly-Typed Navigation Methods

```
<Y> Path<Y> get(SingularAttribute<? super X, Y> attribute);

<Y> SetJoin<X,Y> join(SetAttribute<? super X, Y> attribute);
```

# Example: Using Static Metamodel Class

```
SELECT c
FROM Customer c join c.orders o
WHERE c.name = 'Braun'

CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Customer> cq = cb.createQuery(Customer.class);
Root<Customer> customer = cq.from(Customer.class);
Join<Customer, Order> o = customer.join(Customer_.orders);
cq.where(cb.equal(customer.get(Customer_.name), "Braun"))
  .select(customer);
```

# Example: Using Metamodel API Directly

```
Metamodel mm = em.getMetamodel();
EntityType<Customer> mcustomer = mm.entity(Customer.class);
SingularAttribute<Customer, String> nameAtt=
   mcustomer.getDeclaredSingularAttribute("name", String.class);
SetAttribute<Customer, Order> orderAtt =
   mcustomer.getDeclaredSet("orders", Order.class);

CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Customer> cq = cb.createQuery(Customer.class);
Root<Customer> customer = cq.from(Customer.class);
Join<Customer, Order> o = customer.join(orderAtt);
cq.where(cb.equal(customer.get(nameAtt), "Braun"))
  .select(customer);
```

# How to Build a Criteria Query

```
CriteriaBuilder cb = ...;
CriteriaQuery<ResultType> cq =
   cb.createQuery(ResultType.class);
Root<SomeEntity> e = cq.from(SomeEntity.class);
Join<SomeEntity,RelatedEntity> j = e.join(...);
...
// the following in any order:
cq.select(…)
   .where(…)
   .orderBy(…)
   .groupBy(…)
   .having(…);
TypedQuery<ResultType> tq = em.createQuery(cq);
List<ResultType> results = tq.getResultList();
```

# The World's Simplest Criteria Query

```
CriteriaBuilder cb = ...;
CriteriaQuery<Customer> cq = cb.createQuery(Customer.class);
Root<Customer> c = cq.from(Customer.class);
cq.select(c);
```

# Path Navigation

```
SELECT c.address.city
FROM Customer c

CriteriaBuilder cb = ...;
CriteriaQuery<String> cq = cb.createQuery(String.class);
Root<Customer> c = cq.from(Customer.class);
cq.select(c.get(Customer_.address).get(Address_.city));
```

# Equivalently, using Path Objects

```
SELECT c.address.city
FROM Customer c

CriteriaBuilder cb = ...;
CriteriaQuery<String> cq = cb.createQuery(String.class);
Root<Customer> c = cq.from(Customer.class);
Path<Address> addr = c.get(Customer_.address);
Path<String> city = addr.get(Address_.city);
cq.select(city);
```

# Joins and Predicates

```
SELECT c
FROM Customer c JOIN c.orders o
WHERE o.product.productType LIKE '%printer%'

CriteriaBuilder cb = ...;
CriteriaQuery<Customer> cq = cb.createQuery(Customer.class);
Root<Customer> c = cq.from(Customer.class);
Join<Customer, Order> o = c.join(Customer_.orders);
cq.where(cb.like(o.get(Order_.product)
                 .get(Product_.productType),
                "%printer%"));
cq.select(c);
```

# Compound Predicates

```
SELECT c
FROM Customer c JOIN c.orders o
WHERE c.name = 'Carl' AND o.quantity > 100

CriteriaBuilder cb = …;
CriteriaQuery<Customer> cq = cb.createQuery(Customer.class);
Root<Customer> c = cq.from(Customer.class);
Join<Customer, Order> o = c.join(Customer_.orders);
cq.where(cb.equal(c.get(Customer_.name), "Carl"),
         cb.gt(o.get(Order_.quantity), 100)
         );
cq.select(c);
```

# Equivalently….

```
SELECT c
FROM Customer c JOIN c.orders o
WHERE c.name = 'Carl' AND o.quantity > 100

CriteriaBuilder cb = ...;
CriteriaQuery<Customer> cq = cb.createQuery(Customer.class);
Root<Customer> c = cq.from(Customer.class);
Join<Customer, Order> o = c.join(Customer_.orders);
cq.where(cb.and(cb.equal(c.get(Customer_.name), "Carl"),
                cb.gt(o.get(Order_.quantity), 100))
         );
cq.select(c);
```

# Subqueries

```
SELECT e FROM Employee e
WHERE e.salary > ALL (
    SELECT m.salary FROM Manager m
    WHERE m.department = e.department)

CriteriaQuery<Employee> cq = cb.createQuery(Employee.class);
Root<Employee> emp = cq.from(Employee.class);
Subquery<BigDecimal> sq = cq.subquery(BigDecimal.class);
Root<Manager> m = sq.from(Manager.class);
sq.select(m.get(Manager_.salary))
  .where(cb.equal(m.get(Manager_.department),
                  emp.get(Employee_.department)));
cq.where(cb.gt(emp.get(Employee_.salary), cb.all(sq)));
cq.select(emp);
```

# Compound Selections: Constructors

```
SELECT NEW CustomerDetails(c.id, c.address, o.quantity)
FROM Customer c JOIN c.orders o
WHERE o.quantity > 100

CriteriaBuilder cb = ...;
CriteriaQuery<CustomerDetails> cq =
   cb.createQuery(CustomerDetails.class);
Root<Customer> c = cq.from(Customer.class);
Join<Customer, Order> o = c.join(Customer_.orders);
cq.where(cb.gt(o.get(Order_.quantity), 100));
cq.select(cb.construct(CustomerDetails.class,
                       c.get(Customer_.id),
                       c.get(Customer_.address),
                       o.get(Order_.quantity)));
```

# Compound Selections: Tuples

```
SELECT c.id, c.address, o.quantity
FROM Customer c JOIN c.orders o
WHERE o.quantity > 100

CriteriaBuilder cb = ...;
CriteriaQuery<Tuple> cq = cb.createTupleQuery();
Root<Customer> c = cq.from(Customer.class);
Join<Customer, Order> o = c.join(Customer_.orders);
cq.where(cb.gt(o.get(Order_.quantity), 100));
cq.select(cb.tuple(c.get(Customer_.id),
                   c.get(Customer_.address),
                   o.get(Order_.quantity)));
```

# Compound Selections: Tuples

```
SELECT c.id, c.address, o.quantity
FROM Customer c JOIN c.orders o
WHERE o.quantity > 100

CriteriaBuilder cb = ...;
CriteriaQuery<Tuple> cq = cb.createTupleQuery();
Root<Customer> c = cq.from(Customer.class);
Join<Customer, Order> o = c.join(Customer_.orders);
cq.where(cb.gt(o.get(Order_.quantity), 100));
cq.multiselect(c.get(Customer_.id),
               c.get(Customer_.address),
               o.get(Order_.quantity));
```

# Compound Selections: Using Aliases

```
SELECT c.id, c.address, o.quantity
FROM Customer c JOIN c.orders o
WHERE o.quantity > 100

CriteriaBuilder cb = ...;
CriteriaQuery<Tuple> cq = cb.createTupleQuery();
Root<Customer> c = cq.from(Customer.class);
Join<Customer, Order> o = c.join(Customer_.orders);
cq.where(cb.gt(o.get(Order_.quantity), 100));
cq.multiselect(c.get(Customer_.id).alias("id"),
               c.get(Customer_.address).alias("addr"),
               o.get(Order_.quantity).alias("quant"));
```

# Compound Selections: Arrays

```
SELECT c.id, c.address, o.quantity
FROM Customer c JOIN c.orders o
WHERE o.quantity > 100

CriteriaBuilder cb = ...;
CriteriaQuery<Object> cq = cb.createQuery();
Root<Customer> c = cq.from(Customer.class);
Join<Customer, Order> o = c.join(Customer_.orders);
cq.where(cb.gt(o.get(Order_.quantity), 100));
cq.select(cb.array(c.get(Customer_.id),
                   c.get(Customer_.address),
                   o.get(Order_.quantity)));
```

# Compound Selections: Arrays

```
SELECT c.id, c.address, o.quantity
FROM Customer c JOIN c.orders o
WHERE o.quantity > 100

CriteriaBuilder cb = ...;
CriteriaQuery<Object> cq = cb.createQuery();
Root<Customer> c = cq.from(Customer.class);
Join<Customer, Order> o = c.join(Customer_.orders);
cq.where(cb.gt(o.get(Order_.quantity), 100));
cq.multiselect(c.get(Customer_.id),
               c.get(Customer_.address),
               o.get(Order_.quantity));
```

# OrderBy

```
SELECT c.id, c.address, o.quantity
FROM Customer c JOIN c.orders o
ORDER BY o.quantity

CriteriaBuilder cb = ...;
CriteriaQuery<Tuple> cq = cb.createTupleQuery();
Root<Customer> c = cq.from(Customer.class);
Join<Customer, Order> o = c.join(Customer_.orders);
cq.where(cb.gt(o.get(Order_.quantity), 100));
cq.select(cb.tuple(c.get(Customer_.id),
                   c.get(Customer_.address),
                   o.get(Order_.quantity)));
cq.orderBy(cb.asc(o.get(Order_.quantity)));
```

# Aggregates, GroupBy

```
SELECT AVG(o.quantity), a.city
FROM Customer c JOIN c.orders o JOIN c.address a
WHERE a.country = 'Switzerland'
GROUP BY a.city

CriteriaQuery<Tuple> cq = cb.createTupleQuery();
Root<Customer> c = cq.from(Customer.class);
Join<Customer, Order>  o = c.join(Customer_.orders);
Join<Customer, Address> a = c.join(Customer_.address);
cq.where(cb.equal(a.get(Address_.country), "Switzerland"));
cq.multiselect(cb.avg(o.get(Order_.quantity)),
               a.get(Address_.city));
cq.groupBy(a.get(Address_.city));
```

# Extensibility with Functions

```
SELECT TRANSLATE(c.address.city)
FROM Customer c

CriteriaQuery<String> cq = cb.createQuery(String.class);
Root<Customer> c = cq.from(Customer.class);
cq.select(cb.function("TRANSLATE",
                      String.class,
                      c.get(Customer_.address)
                       .get(Address_.city)));
```

# Parameters Are Objects Too

```
SELECT DISTINCT o.product
FROM Customer c JOIN c.orders o
WHERE c.address.city = :city
```

```
CriteriaQuery<Product> cq = cb.createQuery(Product.class);
Root<Customer> c = cq.from(Customer.class);
Join<Customer, Order> o = c.join(Customer_.orders);
cq.select(o.get(Order_.product)).distinct(true);
ParameterExpression<String> city = cb.parameter(String.class);
cq.where(cb.equal(c.get(Customer_.address).get(Address_.city),
                  city));
```

# Parameters Objects Can Have Names

```
SELECT DISTINCT o.product
FROM Customer c JOIN c.orders o
WHERE c.address.city = :city
```

```
CriteriaQuery<Product> cq = cb.createQuery(Product.class);
Root<Customer> c = cq.from(Customer.class);
Join<Customer, Order> o = c.join(Customer_.orders);
cq.select(o.get(Order_.product)).distinct(true);
cq.where(cb.equal(c.get(Customer_.address).get(Address_.city),
                  cb.parameter(String.class, "city")));
```

# Criteria Query Execution

- TypedQuery interface
  - Extends Query interface
  - Strong typing carries through to query result
- Tuple interface
  - For extracting elements from a query result tuple
  - By position or by alias

# Query Execution Using Parameter Objects

```
EntityManager em = ...;
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Product> cq = cb.createQuery(Product.class);
Root<Customer> c = cq.from(Customer.class);
Join<Customer, Order> o = c.join(Customer_.orders);
cq.select(o.get(Order_.product)).distinct(true);
ParameterExpression<String> city = cb.parameter(String.class);
cq.where(cb.equal(c.get(Customer_.address).get(Address_.city),
                  city));

TypedQuery<Product> tq = em.createQuery(cq);
tq.setParameter(city, "Bern");
List<Product> products = tq.getResultList();
```

# Query Execution Using Named Parameters

```
EntityManager em =  ...;
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Product> cq = cb.createQuery(Product.class);
Root<Customer> c = cq.from(Customer.class);
Join<Customer, Order> o = c.join(Customer_.orders);
cq.select(o.get(Order_.product)).distinct(true);
cq.where(cb.equal(c.get(Customer_.address).get(Address_.city),
                  cb.parameter(String.class, "city")));

TypedQuery<Product> tq = em.createQuery(cq);
tq.setParameter("city", "Bern");
List<Product> products = tq.getResultList();
```

# Tuple Queries

```
CriteriaBuilder cb = ...;
CriteriaQuery<Tuple> cq = cb.createTupleQuery();
Root<Customer> c = cq.from(Customer.class);
Join<Customer, Order> o = c.join(Customer_.orders);
cq.where(cb.equal(c.get(Customer_.id), 10041));
cq.multiselect(c.get(Customer_.name),
               c.get(Customer_.address),
               o.get(Order_.quantity))
  .distinct(true);
```

# Tuple Query Execution

```
TypedQuery<Tuple> tq = em.createQuery(cq);
Tuple result = tq.getSingleResult();
String name = result.get(0, String.class);
Address address = result.get(1, Address.class);
Integer quantity = result.get(2, Integer.class);
```

# Tuple Queries Using Aliases

```
CriteriaBuilder cb = ...;
CriteriaQuery<Tuple> cq = cb.createTupleQuery();
Root<Customer> c = cq.from(Customer.class);
Join<Customer, Order> o = c.join(Customer_.orders);
cq.where(cb.equal(c.get(Customer_.id), 10041));
cq.multiselect(c.get(Customer_.name).alias("name"),
               c.get(Customer_.address).alias("addr"),
               o.get(Order_.quantity).alias("quant"))
  .distinct(true);
```

# Tuple Query Execution Using Aliases

```
TypedQuery<Tuple> tq = em.createQuery(cq);
Tuple result = tq.getSingleResult();
String name = result.get("name", String.class);
Address address = result.get("addr", Address.class);
Integer quantity = result.get("quant", Integer.class);
```

# Compatibility with JPQL

- TypedQuery interface available for use with JPQL as well
  - EntityManager methods createQuery, createNamedQuery added with result class arguments
- Parameter objects can be used with Query objects
  - Added methods to get Parameter objects by name or position and bind their values

# Browsing CriteriaQuery Objects

Methods:

- (on CriteriaQuery, Subquery)
  - getRoots, getSelection, getRestriction, getGroupList, getGroupRestriction
  - getOrderList, getParameters isDistinct, getResultType
- (on Selection)
  - isCompoundSelection, getCompoundSelectionItem
- (on Predicate)
  - getOperator, getExpressions
- (on Root, Join)
  - getJoins
- (on Path)
  - getParentPath

# Modifying CriteriaQuery Objects

Before or after TypedQuery objects have been created from them

```
CriteriaQuery<Customer> cq = cb.createQuery(Customer.class);
Root<Customer> c = cq.from(Customer.class);
Predicate p1 =
   cb.equal(c.get(Customer_.address).get(Address_.city),"Bern");
cq.where(p1).select(c);
TypedQuery<Customer> tq = em.createQuery(cq);
List<Customer> result = tq.getResultList();

Predicate p2 = cb.gt(c.get(Customer_.balanceOwed), 1000);
cq.where(p2);
TypedQuery<Customer> tq2 = em.createQuery(cq);
...
```

# Summary

- Object-based API for query definition
  - Queries are strongly-typed
  - Typing carries through to execution
  - Navigation via strings or objects
- Metamodel API
  - browsing of persistence unit
  - for frameworks
  - for strong-typing of criteria queries
- TypedQuery  interfaces for query execution

# Where Do We Go From Here?

- Query By Example ?
- Update and delete criteria queries?
- Metamodel API for O/R mapping ?
- Give us your opinions!!

- jsr-317-feedback@sun.com

Thank you!

**Linda DeMichiel**

**Oracle**

linda.demichiel@oracle.com