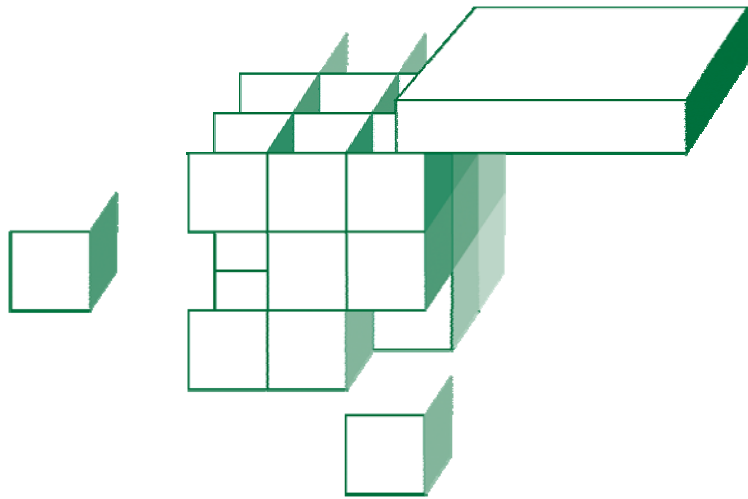


## CHAPTER 5

# THE CRITERIA QUERY API



## OBJECTIVES

*After completing “The Criteria Query API,” you will be able to:*

- Build **Criteria** queries.
- Use criteria restrictions.
- Illustrate sorting techniques.
- Work with projections and aggregations.
- Demonstrate Query by Example (QBE).

# Criteria Queries

---

- **Hibernate provides three different ways to retrieve data:**
  - The **Criteria API**, which is discussed in this chapter.
  - The **Hibernate Query Language**, which is the subject of the next chapter.
  - **Native SQL** queries, which can be used in HQL expressions.
- The **Criteria API** allows queries to be built at runtime without direct string manipulations.
- A **Criteria** object is a tree of **Criterion** instances, which are Java objects used to construct queries.
- The **Criteria** query API also includes **Query by Example (QBE)** functionality for supplying example objects.
- **Criteria** also includes **projection** and **aggregation** methods, such as **count**.
- Since **criteria** queries are built from library objects, the queries are parsed and validated at compile time, unlike **Hibernate Query Language** strings.

# The Criteria Interface

---

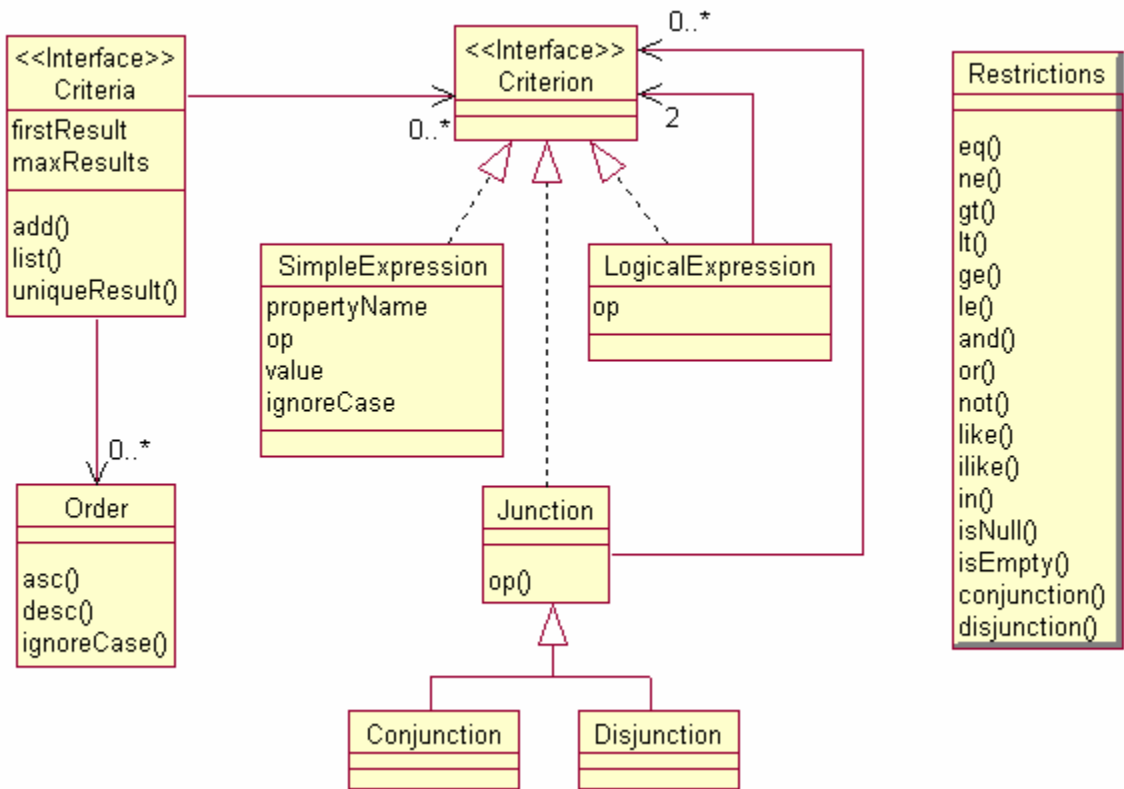
- The **Criteria** interface is in the **org.hibernate** package.
- It is a very convenient approach for applying a variable number of conditions to a search.
- Since **Criteria** is an interface, it can't be instantiated. Instead, the **Session** class has a factory method called **createCriteria**, which takes a reference of type **Class**.

```
Criteria crit = session.createCriteria(User.class);
```

- In fact there are several overloads of **createCriteria** in **Session**, which allow an entity name or an alias to be specified.
- The form that takes a **Class** reference is the most common, however.
- As with most Hibernate classes, the methods in **Criteria** return the **this** reference, so additional calls can be chained.

# The Criteria API

- The following UML diagram summarizes the fundamentals of the Criteria API:



- Queries primarily comprise **Criterion** instances, which say what objects to select.
- They can be ordered using **Order** instances.
- A query can be defined with **windowing**, including the index of the first row to return and the number of results to fetch.

# The Restriction Class Utility

---

- The **Restrictions** class is a static factory for **Criterion** instances.

- As an example, to find a **User** by name:

```
Criteria crit = session.createCriteria(User.class);
crit.add( Restrictions.eq("name", "buffy") );
User buffy = (User) crit.uniqueResult();
```

- Using method chaining, this becomes

```
User buffy =
    (User) session.createCriteria(User.class)
        .add(Restrictions.eq("name", "buffy"))
        .uniqueResult();
```

- The **org.hibernate.criterion.Restrictions** class is the new name for the **net.sf.hibernate.expression.Expression** class in Hibernate 2.1.
  - The JavaDoc API for the **Expression** class says, “This class is semi-deprecated. Use **Restrictions**.”
- The **Restrictions** class has a large number of static factory methods that return instances of **Criterion**.

# The Restriction Class Utility

---

- Criteria created using only a class name will provide all instances of that class when **list** is called.

```
List<User> users =  
    session.createCriteria(User.class).list();
```

- Using Java-5 generics in this manner will generate a warning message about unchecked casts.

- The factory methods in **Restrictions** allow developers to assemble complex expressions in an object-oriented way.
- Restrictions are applied to a Criteria object using the **add** method.

```
public Criteria add (Criterion restriction);
```

- Multiple **add** calls are enforced as **boolean AND** conditions.

- The **addOrder** method can be used to order the results.

```
public Criteria addOrder(Order order);
```

- The **Order** class has two static methods, **asc** and **desc**, which apply ascending or descending ordering constraints using a specified property as an input argument.

```
List emps = session.createCriteria(Employee.class)  
    .addOrder( Order.asc("lastName") )  
    .list();
```

- Criteria instances can also be chained to make subqueries. This is discussed further below.

# The Restriction Class Utility

---

- A sampling of the methods in the **Restrictions** class includes:
  - **Restrictions.eq**, which applies an equality constraint to the named property.

```
crit.add( Restrictions.eq("lastName", "Bigboote") );
```

- **Restrictions.ge** applies a greater-than-or-equal-to constraint. The methods **gt**, **lt**, **le** are similar for greater-than, less-than, and less-than-or-equal-to.

```
crit.add( Restrictions.gt("salary", 50000.0) );
```

- **Restrictions.like** and **Restrictions.ilike** apply SQL pattern matching constraints, where **ilike** is a case-insensitive version of like. Use the % character as a wildcard to match parts of a string.

```
crit.add( Restrictions.ilike("lastName", "Ca%") );
```

- **Restrictions.isNull** and **Restrictions.isNotNull** apply **null** and **not null** constraints on individual fields.

```
crit.add( Restrictions.isNotNull("email") );
```

- **Restrictions.between** looks for values between two limits, and **Restrictions.in** looks for values contained in a provided collection.

```
crit.add(
    Restrictions.in("address.state",
        new String[]{"CT", "GA", "VA"}) );
```

## Other Restrictions

---

- Applying more than one **Criterion** using **Criteria.add** requires them both equally, using a **boolean AND** operator.
- In order to apply an **OR** condition, **Restrictions** has an **or** method, which combines two **Criterion** conditions.
- Conditions can be combined using **Restrictions.disjunction** and **Restrictions.conjunction**.
  - No arguments are passed to either method.
  - Criterion instances are then added to the **conjunction** or **disjunction** using their respective **add** methods.
  - The **conjunction** or **disjunction** is then added to a **Criteria** instance and evaluated.

Evaluated Only

- As an example, see **Examples/Earthlings/Step7**, which holds a new class **CriteriaQueries**.

- Run this example as follows; the output includes the results of several Criteria API queries.

```
run cc.db.hibernate.CriteriaQueries
```

- The method **testConjunction** constructs a conjunction:

```
Criteria crit = s.createCriteria(Employee.class);
Criterion name =
    Restrictions.like("lastName", "Ca%");
Criterion salary =
    Restrictions.gt("salary", 30000.0);
Criterion state = Restrictions.in("address.state",
    new String[] {"MA", "NC"});
Conjunction cj = Restrictions.conjunction();
cj.add(name);
cj.add(salary);
cj.add(state);
crit.add(cj);
List<Employee> emps = crit.list();
```

- Note the usage of the **address** component in **Employee**.

- The result of this query is:

```
Walker Calhoun $38,000.00 MA
Hugh Campbell $32,000.00 MA
King Cardenas $84,000.00 NC
Mariana Castillo $67,000.00 NC
```

- In this case, the three restrictions could have been added directly to the criteria using its **add** method; see the **testMultipleCriteria** method. Either approach is acceptable.

# Conjunction and Disjunction

EXAMPLE

- Going the other way, the restrictions can be added to a disjunction instead.
- See the **testDisjunction** method:

```
Criteria crit = s.createCriteria(Employee.class);
Criterion name =
    Restrictions.like("lastName", "Ca%");
Criterion salary =
    Restrictions.gt("salary", 30000.0);
Criterion state =
    Restrictions.in("address.state",
        new String[] {"MA", "NC"});
```

```
Disjunction dj = Restrictions.disjunction();
dj.add(name);
dj.add(salary);
dj.add(state);
crit.add(dj);
List<Employee> emps = crit.list();
```

- In this case, many more (actually, 101) **Employees** are matched, since each row only needs to satisfy one of the restrictions.

```
Patrick Acosta $61,000.00 GA
Letitia Anderson $21,000.00 MA
...
Hubert Young $50,000.00 NC
Rebecca Zimmerman $97,000.00 MA
Total: 101
```

# Windowing Results

---

- The Criteria API can be used to retrieve a limited **window** of results from a large result set.
- The **setMaxResults** method limits the total number returned.

```
crit.setMaxResults(10);
```

- The **setFirstResult** method takes an integer to indicate the starting row.

```
crit.setFirstResult(i*10) // i counts windows
```

- Each window results in a separate query.

- **CriteriaQueries** also includes a method **printWindowedResults**, as follows:

```
Criteria crit =
    session.createCriteria(Employee.class);
crit.setMaxResults(10);

int k = 0;
for (int i = 0; i < 20; i++) {

    crit.setFirstResult(i*10);
    List<Employee> emps = crit.list();

    if (emps.size() == 0) break;
    System.out.println("\nResults from " +
        (i*10) + " to " + (i*10 + emps.size() - 1));
    for (Employee e : emps) {
        System.out.println(k++ + ": " + e);
    }
}
```

- This loop prints the **Employee** instances in groups of 10.
- Each group results in a SQL **SELECT** statement.

Results from 0 to 9

```
0: Acosta
1: Amdell
2: Anderson
3: Angel
4: Ayer
5: Bailey
6: Barrell
7: Baxter
8: Beard
9: Berger
...
```

# Sorting Query Results

---

- As mentioned above, the **Order** class can be used to sort results.
- **Order** has two static methods, each of which take a **String** property as an argument.

```
public static Order asc(String propertyName);  
public static Order desc(String propertyName);
```

- Each is applied through the **addOrder** method in **Criteria**.
- Multiple ordering conditions can be applied. The results will be sorted by the first order, then by the second, and so on.

- Constraints on associations are applied by chaining **Criteria**.
- For example, in the **Earthlings** schema, an **Employee** is a member of a **Department** which has a given **Location**.

- So, to find all employees that work in Massachusetts, it is necessary to traverse the **Employee** association from its **Department** to the associated **Location**.

- See **printEmpsWorkInMA**:

```
Criteria crit = s.createCriteria(Employee.class)
    .createCriteria("department")
    .createCriteria("location")
    .add( Restrictions.eq("state", "MA") );
```

```
List<Employee> emps = crit.list();
```

```
for (Employee e : emps) {
    System.out.println(e.getLastName() + " works for "
        + e.getDepartment().getName() + " in "
        + e.getDepartment().getLocation().getState());
}
```

- The result gives the 22 employees that work in Massachusetts.

Anderson works for Administration in MA

Baxter works for Administration in MA

...

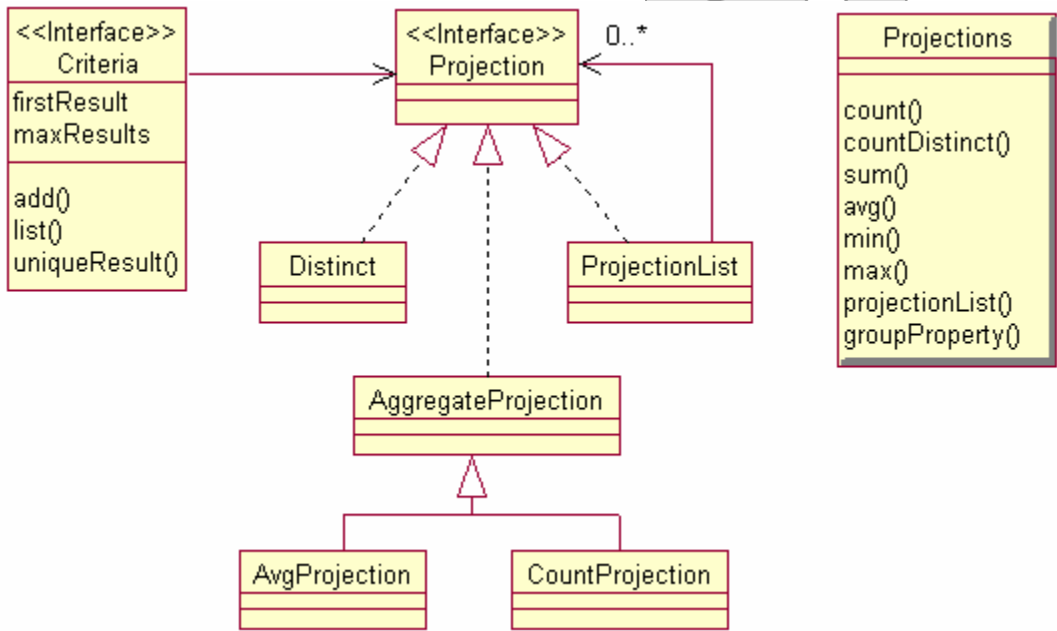
Rollins works for Facilities in MA

Watts works for Facilities in MA

- By default, the employees are sorted alphabetically by department and within each department by last name. This can easily be modified using the **Order** methods above.

# Projections and Aggregates

- Each of the above queries has functioned in terms of objects.
- Instead, the results can be interpreted using **projections**, **aggregate functions**, and **group by** functionality.



# The Projections Class Utility

---

- Projections come from the **org.hibernate.criterion.Projections** factory class.

- As simple example of a projection is counting rows.

```
Criteria crit = s.createCriteria(Employee.class);
crit.setProjection( Projections.rowCount() );
List results = crit.list();
```

- The result list contains a single Integer with the results. This is equivalent to a **COUNT(\*)** function in SQL.
- Other available aggregate functions are:

```
public static AggregateProjection avg(String);
public static CountProjection count(String);
public static CountProjection
    countDistinct(String);
public static AggregateProjection min(String);
public static AggregateProjection max(String);
```

# The ProjectionList Class

---

- More than one projection can be applied to a criteria instance.
- The result is a **List** with an **Object** array as its first element. The array contains all the resulting values in the order requested.
- To add multiple projections, the **Projections** class has a method called **projectionList** that returns an instance of the **ProjectionList** class.

```
public static ProjectionList projectionList();
```

- The **ProjectionList** class has an **add** method that takes a **Projection** as an argument.

```
public ProjectionList add(Projection);
```

- Finally, the projection list is added to a **Criteria** through the **setProjection** method.

```
public Criteria setProjection(Projection proj);
```

- The method **printMultipleProjections** shows how to set up and execute multiple projections. The key lines of code are:

```
Criteria crit = s.createCriteria(Employee.class);
List results = crit.setProjection(
    Projections.projectionList()
        .add(Projections.rowCount())
        .add(Projections.avg("salary"))
        .add(Projections.min("salary"))
        .add(Projections.max("salary")))
    .list();
Object[] res = (Object[]) results.get(0);
for (Object o : res) {
    System.out.println(o);
}
```

- The generated SQL is then:

```
select
  count(*) as y0_,
  avg(this_.salary) as y1_,
  min(this_.salary) as y2_,
  max(this_.salary) as y3_
from
  EARTHLINGS.EMPLOYEES this_
```

- ... and the printed results are:

```
Row count: 135
Average salary: $43,866.67
Minimum salary: $15,000.00
Maximum salary: $120,000.00
```

- The returned **List** contains only an **Object** array which has the desired answers in the order they were requested.

- Projections can be used to retrieve individual properties rather than entire objects.
- The Projections class has a **property** method takes a **String** property name as an argument and returns a **PropertyProjection**.

```
public static PropertyProjection property(String);
```

- As before, add the resulting projection to projection list, which is then set on a **Criteria**.
- The result is a **List** containing an **Object** array for each row in the result set. See **printProjectionsWithProperties**:

```
Criteria crit = s.createCriteria(Employee.class);
List results = crit.setProjection(
    Projections.projectionList()
        .add(Projections.property("firstName"))
        .add(Projections.property("lastName"))).list();
for (Object o : results) {
    Object[] names = (Object[]) o;
    System.out.println(names[0] + " " + names[1]);
}
```

- The result shows the first and last names of each **Employee**:

```
Patrick Acosta
...
Rebecca Zimmerman
```

- Property projections are useful when the number of columns in a table is very large, or when a large set of joins may return a very large result set and you're only interested in a few columns.

- The **Projections** class has a method called **groupByProperty** that takes a string property name.
- This is equivalent to the **GROUP BY** clause in SQL.
- For example, to count how many employees live in each state, use the following expression.
  - See **projectionWithGroupBy**:

```
List output = s.createCriteria(Employee.class)
    .setProjection(Projections.projectionList()
    .add(Projections.rowCount())
    .add(Projections.groupProperty("address.state")))
    .list();
for (Object o : output) {
    System.out.println(Arrays.asList((Object[]) o));
}
```

- The grouped results are found at the bottom of the program output:

```
Number of employees in each department
[32, GA]
[22, MA]
[46, NC]
[35, NJ]
```

# Query By Example

---

- **Query By Example** provides another style of searching.
- Rather than build a query by programmatically adding conditions to a **Criteria** object, partially populate an instance of the desired object.
  - The partially populated instance is an example.
  - Hibernate then builds the **Criteria** query from the example.
- To convert an object into an example, use the create method in the **org.hibernate.criterion.Example** class.

```
public static Example create(Object entity);
```

- The trick with QBE is that when the query is translated into SQL, all properties of the example that are not null are used in the query.
  - To drop a property from the list, use the **excludeProperty** method.
  - For numerical properties, use the **excludeZeros** method.
- **Example** also has the **ignoreCase** method, which does what it sounds like, and the **enableLike** method, which is used for string comparisons.

- The following code is used to find all Employees who live in the state of Massachusetts.

```
Criteria crit = s.createCriteria(Employee.class);
Employee ex = new Employee();
Address addr = new Address();
addr.setState("MA");
ex.setAddress(addr);

crit.add(Example.create(ex).excludeZeroes());

List<Employee> emps = crit.list();
for (Employee e : emps) {
    System.out.println(e);
}
System.out.println(emps.size() + " found");
```

- Without using **excludeZeroes**, the result set would be empty. This is because the example employee has a **salary** attribute and a **commissionPct** attribute, each with value zero.

- QBE associations are done the same way **Criteria** associations were implemented above.
- For example, to find all **Departments** that are located in Massachusetts, we need to follow the association from **Department** to **Location**.
  - See **printDeptsInMAUsingQBE**:

```
Department dept = new Department();  
Location loc = new Location();  
loc.setState("MA");
```

```
List<Department> depts =  
    s.createCriteria(Department.class)  
      .add(Example.create(dept))  
      .createCriteria("location")  
        .add(Example.create(loc))  
      .list();
```

```
for (Department d : depts) {  
    System.out.println(d.getName() + " "  
        + d.getLocation().getState());  
}
```

- The result is:

```
Administration MA  
HR MA  
Facilities MA
```

**Suggested time: 30 minutes**

In this lab you will create **Criteria** queries to answer various questions about the **Earthlings** database.

Detailed instructions are found at the end of the chapter.

Evaluation  
Only

## SUMMARY

- The **Criteria** Query API provides an object-oriented way to build complex queries.
- **Criteria** queries begin with the **createCriteria** factory method in the **Session** class. Of its overloads, the most commonly used takes a **Class** reference as an argument.
- The **list** method and the **uniqueResult** method return all instances of a class and a single instance, respectively.
- Constraints on a query are applied using static factory methods from the **Restrictions** class.
- Commonly used **Restrictions** include **eq**, **ne**, **gt**, **lt**, **like**, and **between**. Many variations are available.
- **Projections** are used to compute scalar quantities from results, like **count**, **max**, or **min**.
- Projections can be applied to properties, associations, and more.
- The Query by Example capability lets Hibernate construct a query based on a sample object.