

# Hibernate Query Language

**“Code with Passion!”**



# Topics

- Criteria Query
- Query By Example
- Query Language (HQL) Query
- SQL Query

# Sub-Topics of Hibernate Criteria Query

- What is Criteria query?
- How to use Criteria query API?
- Pagination
- Restrictions
- Ordering
- Aggregate functions
- Fetch modes
- Query By Example (QBE)



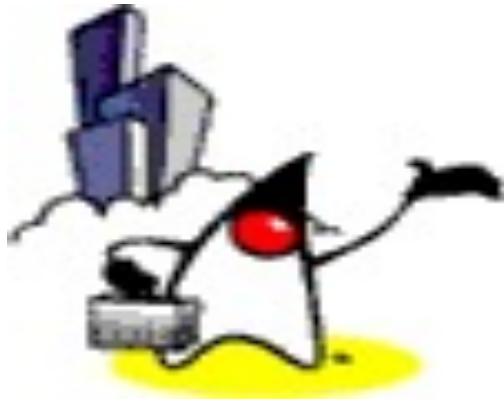
# What is Criteria Query?

# Multiple ways of retrieving data in Hibernate

- Criteria query API
  - The easiest way to retrieve data (from Java developer standpoint)
  - Pure Java language based
- Hibernate Query Language (HQL)
- Native SQL query

# Criteria Query API

- Uses a set of Java objects for constructing queries
  - Instead of using SQL-like query language
- Lets you build nested, structured query expressions in Java programming language
  - Compile time syntax/type checking is possible
  - Polymorphic behavior – get instances of X & subclass(X)
- Supports Query By Example (QBE)
  - Performing a query by providing an example Java object that contain properties that need to be retrieved
- Supports aggregation methods (from Hibernate 3)
  - Count

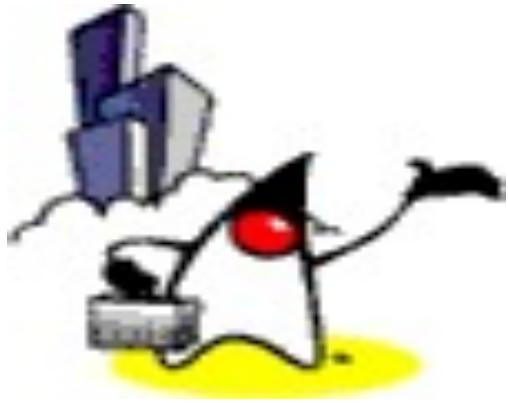


# How to use Criteria Query API

# How to use Criteria Query API

- Create *org.hibernate.Criteria* object via *createCriteria()* factory method of the *Session*
  - Pass persistent object's class or its entity name to the *createCriteria()* method
- Call *list()* method of the *Criteria* object to perform a query

```
// Get all instances of Person class and its subclasses  
Criteria crit = sess.createCriteria(Person.class);  
List results = crit.list();
```

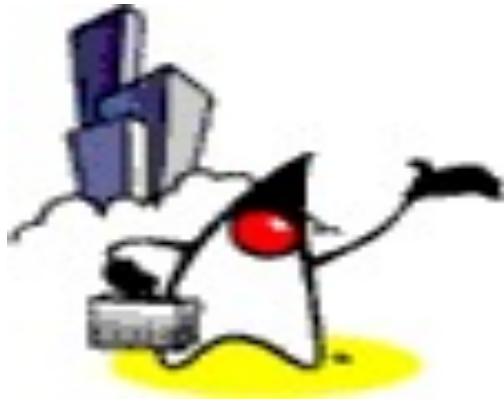


# Pagination

# Pagination through the Result Set

- Hibernate handles the pagination
  - Retrieving fixed number of rows
- Two methods of Criteria class
  - *setFirstResult()* - set the first row in the result
  - *setMaxResults()* - number of rows to retrieve

```
Criteria crit = sess.createCriteria(Person.class);
crit.setFirstResult(2); // Starting from 3rd row
crit.setMaxResults(50); // Number of rows
List results = crit.list();
```



# Narrowing the Result Set via Criteria Restrictions

# Restrictions class

- Used to selectively retrieve objects
  - Example: Person objects whose age field's value is greater than 20
- Add restrictions to the *Criteria* query object with *add()* method
  - The *add()* method of the *Criteria* object takes an *org.hibernate.criterion.Criterion* object that represents an individual restriction
- You can have more than one restriction for a Criteria query

# Methods of Restrictions class

- *Restrictions.eq("name", "Shin")*
- *Restrictions.ne("name", "NoName")*
- *Restrictions.like("name", "Sa%")*
- *Restrictions.ilike("name", "sa%")*
- *Restrictions.isNull("name");*
- *Restrictions.gt("price", new Double(30.0))*
- *Restrictions.between("age", new Integer(2), new Integer(10))*
- *Restrictions.or(criterion1, criterion2)*
- *Restrictions.disjunction()*

# Add a restriction

- *Restrictions.like()* - pattern based restriction

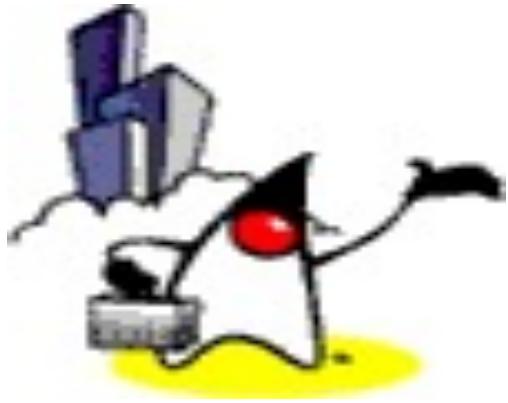
```
// Create a Criteria object from Session object
Criteria criteria = session.createCriteria(Person.class);

// Create Criterion object first and then add it to Criteria
Criterion nameRestriction = Restrictions.like("name", "Shin%");
crit.add( nameRestriction );
List results = criteria.list();
```

# Logical Grouping of Restrictions

- Restrictions can be logically grouped with `.and` and `.or`

```
// Retrieve Person objects whose name has a pattern  
// and whose age is 10 or null  
  
List people = sess.createCriteria(Person.class)  
    .add( Restrictions.like("name", "Shin%") )  
    .add( Restrictions.or(  
        Restrictions.eq( "age", new Integer(10) ),  
        Restrictions.isNull("age")  
    ))  
    .list();
```



# Ordering the Result Set

# Ordering the results

- You may order the results using  
*org.hibernate.criterion.Order*

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "F%"))
    .addOrder( Order.asc("name") )
    .addOrder( Order.desc("age") )
    .setMaxResults(50)
    .list();
```



# Projections & Aggregates

# Aggregate functions available through Projections factory class

- *rowCount()*
- *avg(String propertyName)*
  - average of a property's value
- *count(String propertyName)*
  - number of times a property has a value
- *countDistinct(String propertyName)*
  - number of unique values the property contains
- *max(String propertyName)*
- *min(String propertyName)*
- *sum(String propertyName)*
  - sum of the property values

# Projections

- Projections.rowCount()

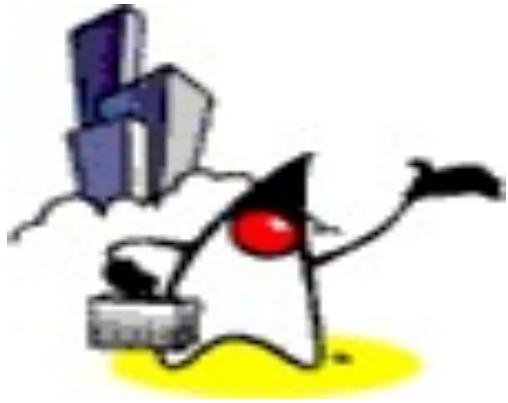
```
// The result will contain one object, an Integer that  
// contains the results of executing COUNT SQL  
// statement
```

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

# Multiple Projections

- Projections.projectionList()

```
// You will get a List with an Object array
// as the first element. The Object array
// contains all the values in order
Criteria crit = sess.createCriteria(Product.class);
ProjectionList projectList = Projections.projectionList();
projectList.add(Projections.avg("price"));
projectList.add(Projections.sum("price"));
crit.setProjection( projectList );
List results = crit.list();
```



# Fetch Modes

# Fetching Modes (How it is fetched)

- `FetchMode.DEFAULT`
  - Default to the setting configured in the mapping file.
- `FetchMode.JOIN`
  - Hibernate retrieves the associated instance or collection in the same SELECT, using an OUTER JOIN.
- `FetchMode.SELECT`
  - A second SELECT is used to retrieve the associated entity or collection.
  - Unless you explicitly disable lazy fetching by specifying `lazy="false"`, this second select will only be executed when you actually access the association.

# Setting the Fetch Mode

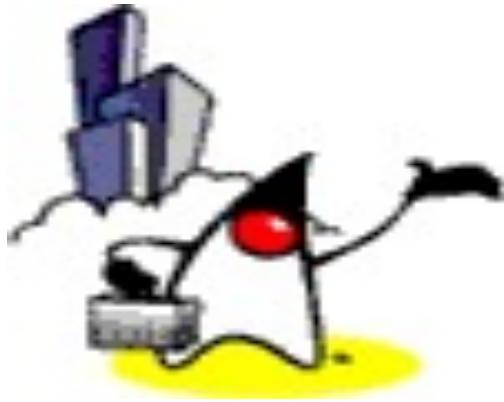
- *setFetchMode("permissions", FetchMode.JOIN)*

```
User user = (User) session.createCriteria(User.class)
    .setFetchMode("permissions", FetchMode.JOIN)
    .add( Restrictions.idEq(userId) )
    .uniqueResult();
```

# Lab:

**Exercise 1: Criteria Query API**  
**3517\_hibernate\_query.zip**





# Query By Example (QBE)

# What is Query By Example (QBE)?

- Provides another style of query
- How to perform QBE based query
  - Partially populate an instance of an object
  - Let Hibernate build behind the scene a criteria using the instance as an example
- *org.hibernate.criterion.Example* class implements *Criterion* interface
  - You can use it like any other restrictions

# Query By Example

- Use *Example.create()* to create a restriction

```
// Retrieve person objects via example object
Criteria crit = sess.createCriteria(Person.class);
Person person = new Person();
person.setName("Shin");
Example exampleRestriction = Example.create(person);
crit.add( exampleRestriction );
List results = crit.list();
```

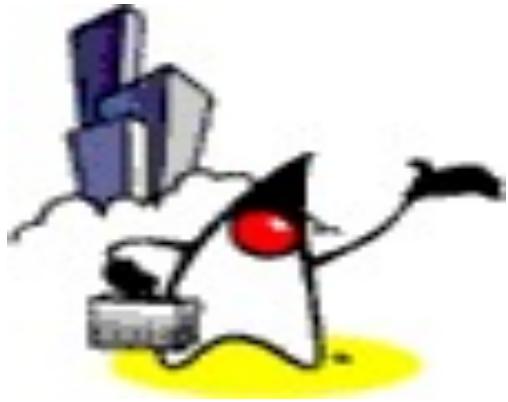
# Lab:

**Exercise 2: Query By Example**  
**3517\_hibernate\_query.zip**



# Sub-Topics of HQL

- What is HQL?
- “from” clause
- Associations and join
- “select” clause
- “where” clause
- Named query
- Polymorphic query



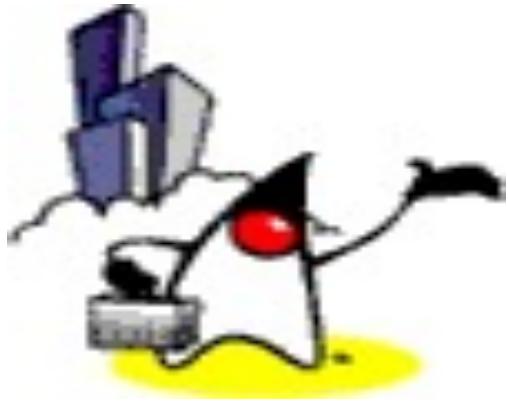
# What is HQL?

# Hibernate Query Language (HQL)

- Very similar to SQL but less verbose
- Understands OO – inheritance, polymorphism, association
  - Selection: *from, as*
  - Associations and joins: *inner join, outer join, right outer join, full join*
  - Projection: *select, elements*
  - Constraints: *where*
  - Other constructs: *aggregate functions, expressions, order by clauses, group by clauses, polymorphic selections, sub-queries*

# Differences of HQL from SQL

- HQL is fully object-oriented, understanding notions like inheritance, polymorphism and association
- Queries are case-insensitive, except for names of Java classes and properties



# “from” clause

## “from” clause

- Return all instances of the class, eg.Cat.

*from eg.Cat*

- Usually don't need to qualify the class name, since auto-import is the default.

*from Cat* is same as *from eg.Cat*

- Most of the time, you will want to assign an alias, since you will want to refer to the Cat in other parts of the query

*from Cat as cat* (“cat” is the alias of “Cat”)

*from Cat cat* (“as” can be omitted)

## “from” clause for multiple classes

- Multiple classes may appear, resulting in a Cartesian product or "cross" join.

*from Formula, Parameter*

*from Formula as form, Parameter as param*

- Local variable naming recommendation - Name query aliases using an initial lowercase, consistent with Java naming standards for local variables



# Associations and joins

# join

- We may also assign aliases to associated entities, or even to elements of a collection of values, using a join

*from Cat as cat*

*inner join cat.mate as mate*

*left outer join cat.kittens as kitten*

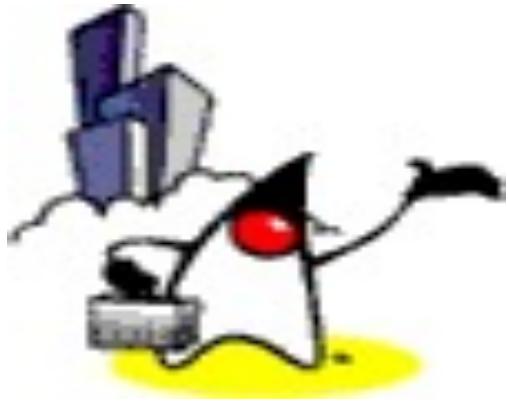
*from Cat as cat*

*left join cat.mate.kittens as kittens*

## join types supported

- inner join
- left outer join
- right outer join
- full join (not usually useful)

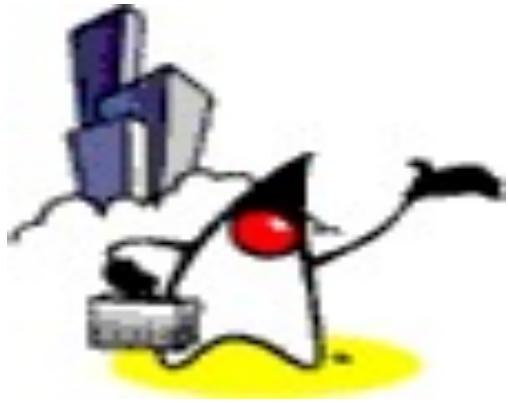
P.S.: If you are not familiar with join types, please see “mysql\_join.pdf” presentation of “MySQL” JPassion.com (JavaPassion.com) online course



# “where” clause

## where clause

- The *where* clause allows you to narrow the list of instances returned.
- If no alias exists, you may refer to properties by name  
*from Cat where name='Fritz'*
- If there is an alias, use a qualified property name:  
*from Cat as cat where cat.name='Fritz'*



# “select” clause

# select clause

- The select clause picks which objects and properties to return in the query result set

*select mate*

*from Cat as cat*

*inner join cat.mate as mate*

- Compact form

*select cat.mate from Cat cat*

## select clause: Returning Properties

- Queries may return properties of any value type including properties of component type

*select cat.name from DomesticCat cat*

*where cat.name like 'fri%*

*select cust.name.firstName from Customer as cust*

## select clause: Returning Multiple Objects as type Object[]

- Queries may return multiple objects and/or properties as an array of type Object[]

*select mother, offspr, mate.name*

*from DomesticCat as mother*

*inner join mother.mate as mate*

*left outer join mother.kittens as offspr*

## select clause: Returning Multiple Objects as a List type

- Queries may return multiple objects and/or properties as a List

```
select new list(mother, offspr, mate.name)
```

```
from DomesticCat as mother
```

```
inner join mother.mate as mate
```

```
left outer join mother.kittens as offspr
```

# select clause: Returning a Java Object

- Returns a typesafe Java object

*select new Family(mother, mate, offspr)*

*from DomesticCat as mother*

*join mother.mate as mate*

*left join mother.kittens as offspr*

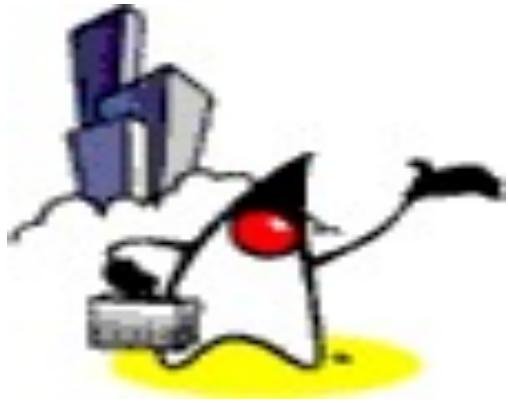
## where clause

- Return all instances of *Foo* for which there exists an instance of *bar* with a *date* property equal to the *startDate* property of the *Foo*

```
select foo  
from Foo foo, Bar bar  
where foo.startDate = bar.date
```

- Compound path expressions make the where clause extremely powerful.

```
from Cat cat where cat.mate.name is not null
```



# Named Query

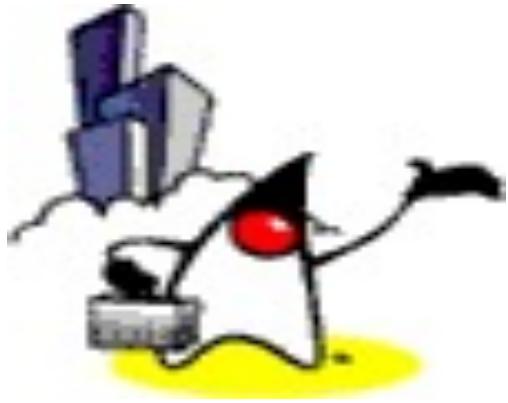
# Named queries

- Get Named HQL Query from Mapping File

```
<query name="HQLpricing"><![CDATA[  
    select product.price from Product product where  
    product.price > 25.0]]>  
</query>
```

- Use the named query

```
Query query = session.getNamedQuery("HQLpricing");  
List results = query.list();  
displayObjectList(results);
```



# Polymorphic Query

# Polymorphic queries

- The query below returns instances not only of Cat, but also of subclasses like DomesticCat

*from Cat as cat*

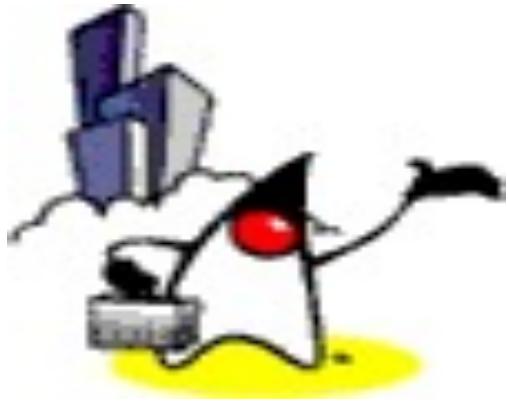
- Hibernate queries may name any Java class or interface in the from clause.
- The query will return instances of all persistent classes that extend that class or implement the interface.
- The following query would return all persistent objects

*from java.lang.Object o*

# Lab:

Exercise 3: HQL Query  
[3517\\_hibernate\\_query.zip](#)





# SQL Query

# SQL Query

- Use native SQL for query operation

# Lab:

**Exercise 4: SQL Query  
3517\_hibernate\_query.zip**



# Code with Passion!

