## Table of Contents

## Course Overview

JPA, or the Java Persistence API, is focused on persistence. Persistence can refer to any mechanism by which Java objects outlive the applications that create them. JPA is not a tool or a framework or an actual implementation. It defines a set of concepts that can be implemented by any tool or framework. JPA supports object relational mapping. A mapping from objects and high-level programming languages to tables which are the fundamental units of databases.

JPA's model was originally based on Hibernate, and initially only focused on relational databases. Today there are multiple persistent providers that support JPA, but Hibernate remains the most widely used. Because of their intertwined relationship and JPA's origins from Hibernate, they're often spoken off in the same breath. JPA today has many provider implementations and Hibernate is just one amongst them.

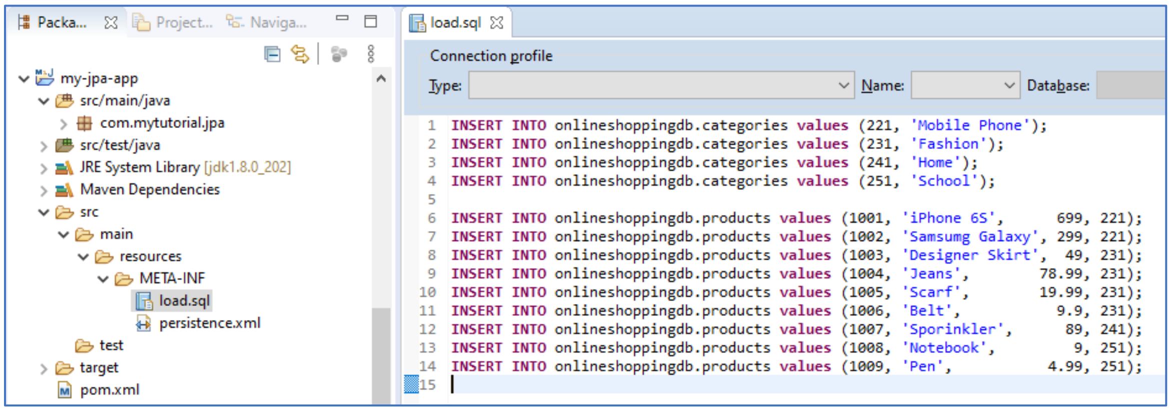
In this course, we'll explore a variety of ways in which you can execute queries using JPA. We'll run native queries, queries using JPQL, and queries using the criteria API, which helps avoid syntax errors in your query specification. We'll round off this course by seeing how you can have entity-specific callback methods that can be invoked at different parts in the entity's lifecycle. When you're done with this course, you will have the knowledge needed to perform complex queries on your database tables and manage your entity's lifecycles in a very granular manner.

## Executing Native Queries

Here we are on a brand new demo and in this demo, we'll study how the Java Persistence API queries work.

We'll first focus on native queries. Now we'll query this database throughout this particular demo, the **onlineshoppingdb**.

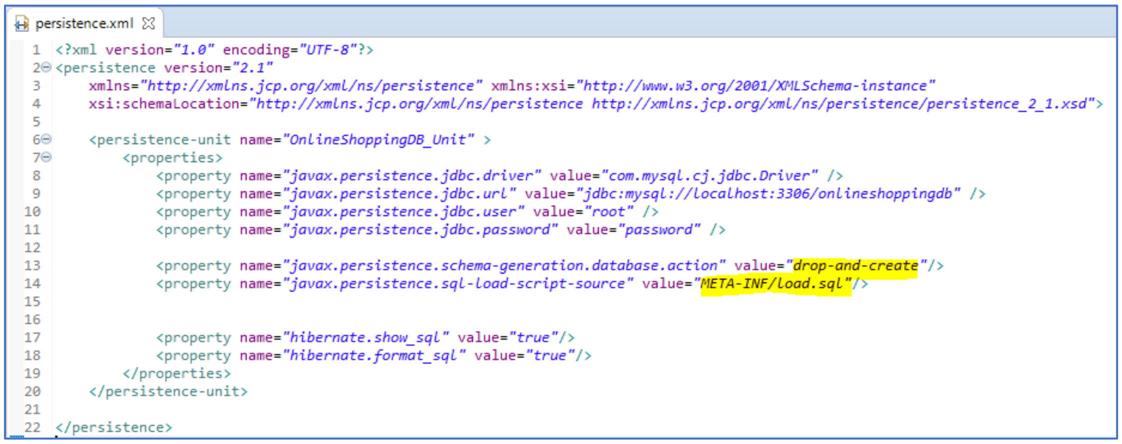
Now let's switch over to our Eclipse project where I'll define entities that we'll use in this demo. Here I am in the **load.sql** file that is under resources META-INF.



```sql
INSERT INTO onlineshoppingdb.categories values (221, 'Mobile Phone');
INSERT INTO onlineshoppingdb.categories values (231, 'Fashion');
INSERT INTO onlineshoppingdb.categories values (241, 'Home');
INSERT INTO onlineshoppingdb.categories values (251, 'School');

INSERT INTO onlineshoppingdb.products values (1001, 'iPhone 6S',      699, 221);
INSERT INTO onlineshoppingdb.products values (1002, 'Samsumg Galaxy', 299, 221);
INSERT INTO onlineshoppingdb.products values (1003, 'Designer Skirt',  49, 231);
INSERT INTO onlineshoppingdb.products values (1004, 'Jeans',        78.99, 231);
INSERT INTO onlineshoppingdb.products values (1005, 'Scarf',        19.99, 231);
INSERT INTO onlineshoppingdb.products values (1006, 'Belt',           9.9, 231);
INSERT INTO onlineshoppingdb.products values (1007, 'Sporinkler',      89, 241);
INSERT INTO onlineshoppingdb.products values (1008, 'Notebook',         9, 251);
INSERT INTO onlineshoppingdb.products values (1009, 'Pen',           4.99, 251);
```

- I'm going to define a bunch of insert operations here to load data into my database tables which I'll then query using my JPA application.
- I have a table called categories with two columns, id, and name of category.
- And I insert four records into this table, Mobile Phones, Fashion, Home, and School. These are e-commerce product categories in our online shopping database.
- Next, I'm going to insert some products that we can query. So insert into the products table all of these products that you see here.
- Every product has a unique ID, name, price, and finally, a category ID that is a reference to one of the four categories that we had set up earlier, Mobile Phones, Fashion, Home, and School.
- So for example, the first two products, the iPhone 6S and the Samsung Galaxy, belongs to the category with ID 221, which is mobile phones.

With our load.sql configured, let's set up our **persistence.xml** files.



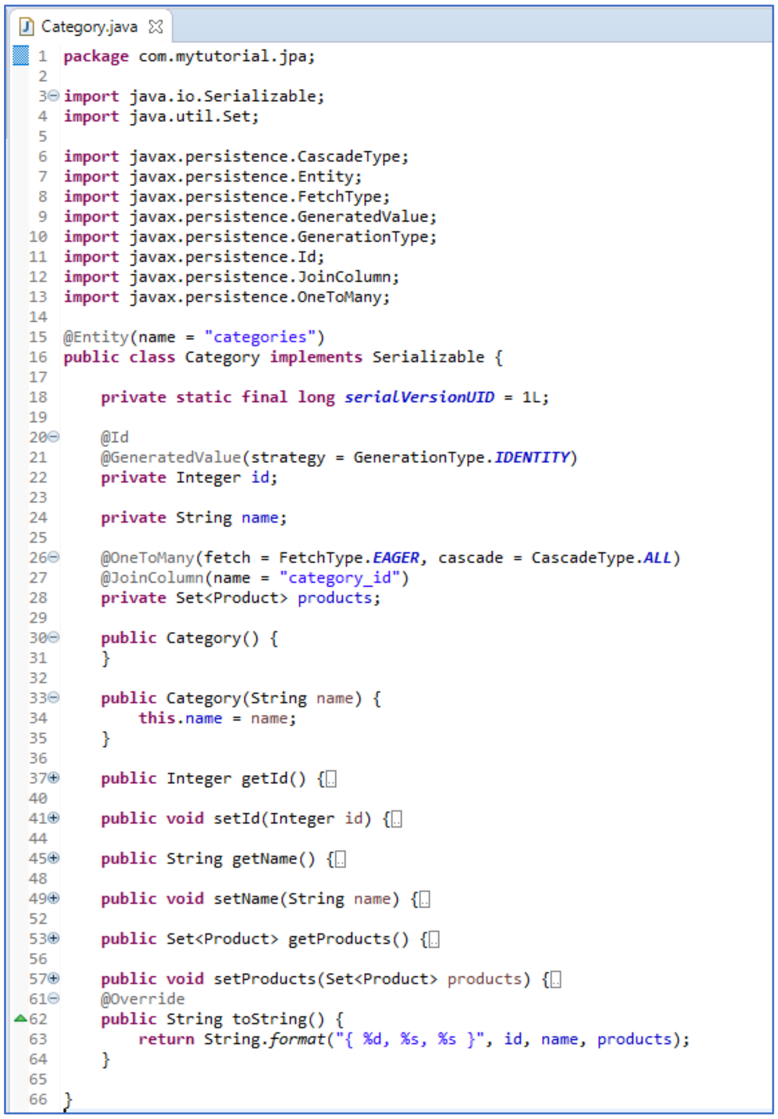
persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
    xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">

    <persistence-unit name="OnlineShoppingDB_Unit" >
        <properties>
            <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/onlineshoppingdb" />
            <property name="javax.persistence.jdbc.user" value="root" />
            <property name="javax.persistence.jdbc.password" value="password" />

            <property name="javax.persistence.schema-generation.database.action" value="drop-and-create"/>
            <property name="javax.persistence.sql-load-script-source" value="META-INF/load.sql"/>


            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
        </properties>
    </persistence-unit>

</persistence>
```

- I've set the database action to be drop-and-create and
- I've specified the SQL load script source to be set to META-INF/load.sql.
- The database that we'll connect to is the onlineshoppingdb
- And our persistence-unit is the OnlineShoppingDB_Unit.

Now, I'm going to create a number of classes that correspond to entities. The `Category` class that I've created corresponding to the Category entity, and The `Product` class that corresponding to the Product entity.

Now I'm going to go back and define these classes, starting with the `Category` class.

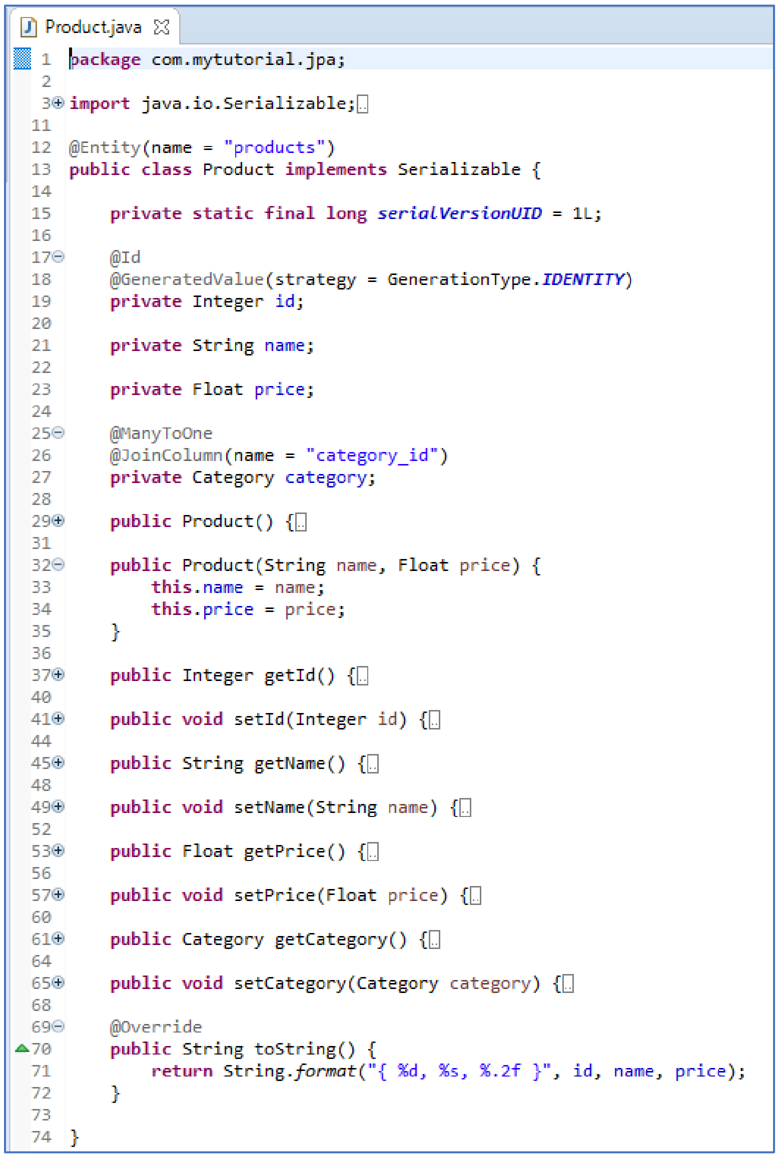


Category.java

```java
package com.mytutorial.jpa;

import java.io.Serializable;
import java.util.Set;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToMany;

@Entity(name = "categories")
public class Category implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String name;

    @OneToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    @JoinColumn(name = "category_id")
    private Set<Product> products;

    public Category() {
    }

    public Category(String name) {
        this.name = name;
    }

    public Integer getId() {

    public void setId(Integer id) {

    public String getName() {

    public void setName(String name) {

    public Set<Product> getProducts() {

    public void setProducts(Set<Product> products) {
    @Override
    public String toString() {
        return String.format("{ %d, %s, %s }", id, name, products);
    }

}
```

- Set up the import statements and let's have the category class annotated with the @Entity annotation and implement the Serializable interface.
- This category class, is what will represent an e-commerce product category. Every category has a unique ID that is generated and a name.
- A category is also related to the products associated with that category using a one-to-many relationship mapping.
- The join column on the Products table is category_id.
- In this one-to-many mapping, our FetchType is EAGER and our CascadeType is ALL. We'll cascade all operations from this parent category entity to child entities which are the products.
- Fill in the rest of the code for this category entity getters and setters for the different member variables.
- I've also implemented this string representation for each category using toString().

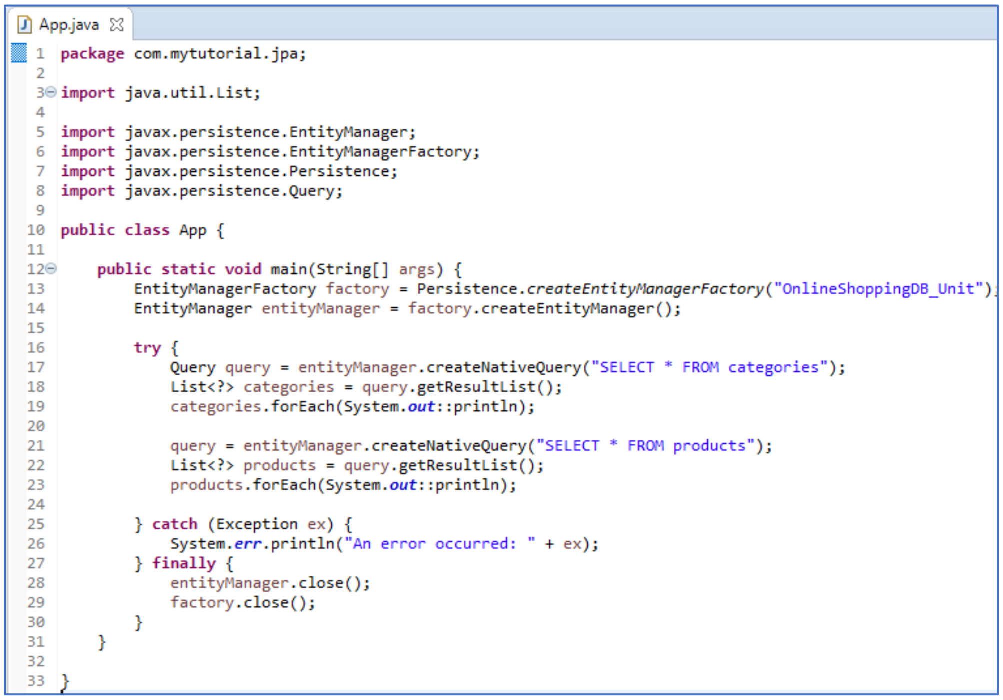
Now let's head over to the `Product` class and define the various details of this Product entity.



```java
package com.mytutorial.jpa;

import java.io.Serializable;

@Entity(name = "products")
public class Product implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String name;

    private Float price;

    @ManyToOne
    @JoinColumn(name = "category_id")
    private Category category;

    public Product() {

    public Product(String name, Float price) {
        this.name = name;
        this.price = price;
    }

    public Integer getId() {

    public void setId(Integer id) {

    public String getName() {

    public void setName(String name) {

    public Float getPrice() {

    public void setPrice(Float price) {

    public Category getCategory() {

    public void setCategory(Category category) {

    @Override
    public String toString() {
        return String.format("{ %d, %s, %.2f }", id, name, price);
    }

}
```

- Set up the import statements, tag the Product class with the @Entity annotation, and have Product implement Serializable.
- Product entities will be stored in the Products table.
- Let's set up the member variables here. Every product has a unique ID, a name, and a price that is a floating point.
- Every product is also related to the associated category using a many-to-one annotation. This is the inverse mapping.
- Product is the owned entity. The rest of the code includes getters and setters for the various member variables here, and also a **toString()** representation for products.

Now it's time to head over to our main class that is App.java, and query products and categories that exist within our underlying database table. And the way we'll perform this query is using **native queries**.

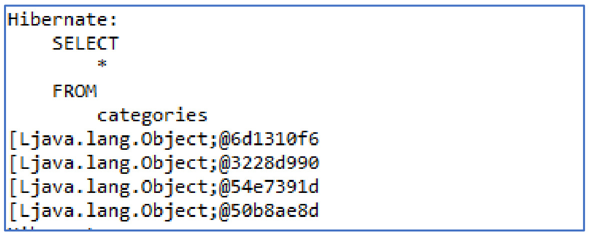


App.java

```java
package com.mytutorial.jpa;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class App {

    public static void main(String[] args) {
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {
            Query query = entityManager.createNativeQuery("SELECT * FROM categories");
            List<?> categories = query.getResultList();
            categories.forEach(System.out::println);

            query = entityManager.createNativeQuery("SELECT * FROM products");
            List<?> products = query.getResultList();
            products.forEach(System.out::println);

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
    }

}
```

- Now the Java Persistence API supports queries known as JPQL queries. We'll cover JPQL queries in a lot of detail in just a bit.
- Native queries are queries which depend on the underlying databases SQL scripting support. For example, if you're connecting to a MySQL database, these are queries that are valid MySQL queries. If you're connecting to a PostgreSQL database, these queries should be valid PostgreSQL queries. So to write valid native queries, you need to know the type of database that you're using.
- Native queries are created using the `createNativeQuery()` method. I've created two native queries here. The first is a `"SELECT * FROM categories"` and the second is `"SELECT * FROM products"`.
- `categories` and `products` are the names of the database tables that exist.
- Once we've created the query, we can call `query.getResultList()` to get the entities returned by each query.
- With these raw native queries, Hibernate has a no way to know what kind of entities are returned. Basically these queries will return a list of objects. So categories and products are lists of an unknown type, just objects. We then run a forEach operation on each of the categories and products returned and print them out to screen.

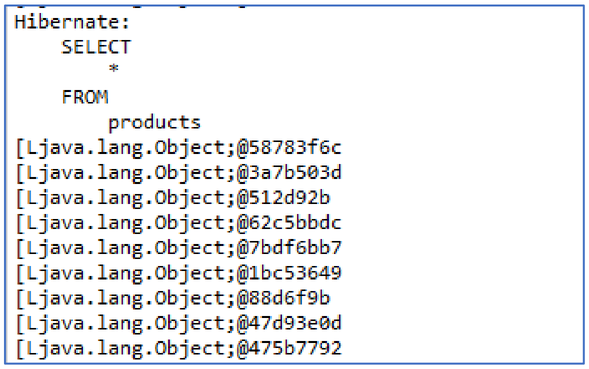
Let's run this code and see how these native queries work to retrieve objects.



```
Hibernate: 
    SELECT
        * 
    FROM
        categories
[Ljava.lang.Object;@6d1310f6
[Ljava.lang.Object;@3228d990
[Ljava.lang.Object;@54e7391d
[Ljava.lang.Object;@50b8ae8d
```

At the very bottom here, here is our first query executed, *select * from categories*. The four categories are retrieved, but printing them out to screen, only printed out object and the memory location of the object. **That's because these categories were retrieved as objects, not actual entities** belonging to the category class.

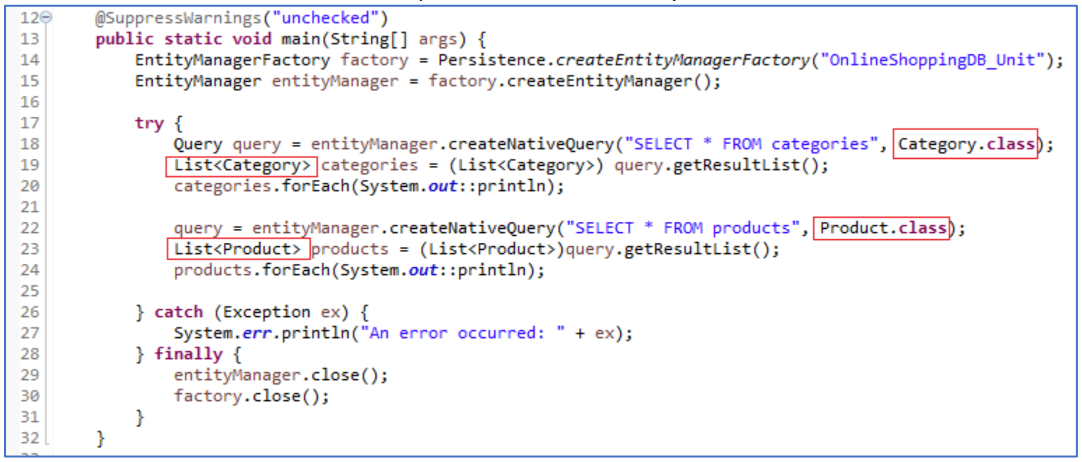
If you scroll down below, you'll see the same is true for the products that were retrieved as well.



```
Hibernate: 
    SELECT
        * 
    FROM
        products
[Ljava.lang.Object;@58783f6c
[Ljava.lang.Object;@3a7b503d
[Ljava.lang.Object;@512d92b
[Ljava.lang.Object;@62c5bbdc
[Ljava.lang.Object;@7bdf6bb7
[Ljava.lang.Object;@1bc53649
[Ljava.lang.Object;@88d6f9b
[Ljava.lang.Object;@47d93e0d
[Ljava.lang.Object;@475b7792
```

*Select * from products*, the products were retrieved as objects. Hibernate and JPA had no idea what kind of entities these were because we use raw native queries.

We can fix this, we can run native queries in a better way

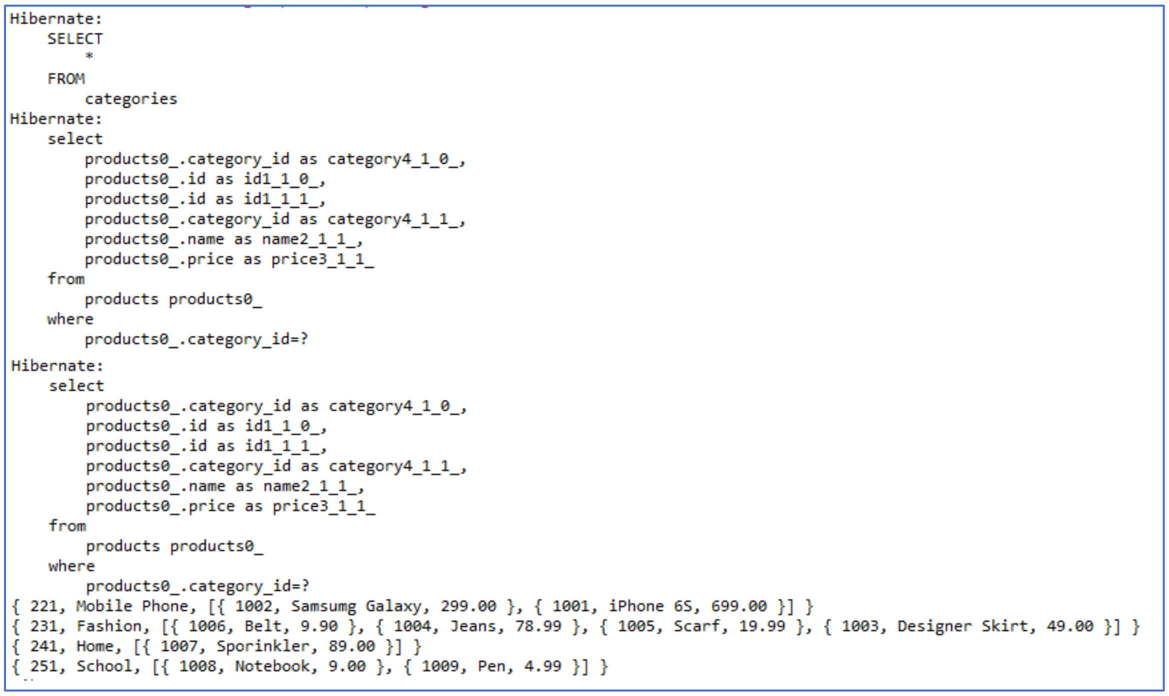


```
12      @SuppressWarnings("unchecked")
13      public static void main(String[] args) {
14          EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
15          EntityManager entityManager = factory.createEntityManager();
16  
17          try {
18              Query query = entityManager.createNativeQuery("SELECT * FROM categories", Category.class);
19              List<Category> categories = (List<Category>) query.getResultList();
20              categories.forEach(System.out::println);
21  
22              query = entityManager.createNativeQuery("SELECT * FROM products", Product.class);
23              List<Product> products = (List<Product>)query.getResultList();
24              products.forEach(System.out::println);
25  
26          } catch (Exception ex) {
27              System.err.println("An error occurred: " + ex);
28          } finally {
29              entityManager.close();
30              factory.close();
31          }
32      }
```

- I'm adding an **@SuppressWarning** annotation to the main method, and I've updated my query.
- Notice how we create native queries, `entityManager.createNativeQuery()` on line 18.
- *Select * from categories* and we specified the type of entity that will be returned from this query.
- We've said that it will belong to the `Category.class`.
- Now, when we call `query.getResultList()`, we can cast it to be of type `List Category`.

- We'll run a `forEach` on the `categories`, return, and print them out to screen.
- And we'll do the same thing for `products`. When we create a NativeQuery, we specify the entity type that it belongs to the `Product.class`.
- The result list can then be cast to a `List` of `products`, and we can print out individual products to screen.

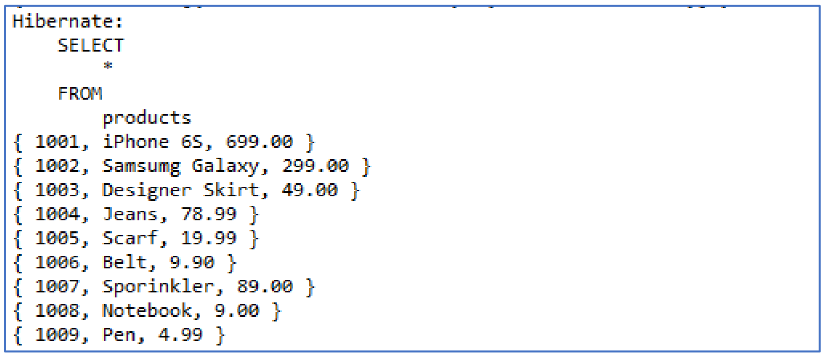
Let's run this code and see how this works.



```
Hibernate:
    SELECT
        *
    FROM
        categories
Hibernate:
    select
        products0_.category_id as category4_1_0_,
        products0_.id as id1_1_0_,
        products0_.id as id1_1_1_,
        products0_.category_id as category4_1_1_,
        products0_.name as name2_1_1_,
        products0_.price as price3_1_1_
    from
        products products0_
    where
        products0_.category_id=?
Hibernate:
    select
        products0_.category_id as category4_1_0_,
        products0_.id as id1_1_0_,
        products0_.id as id1_1_1_,
        products0_.category_id as category4_1_1_,
        products0_.name as name2_1_1_,
        products0_.price as price3_1_1_
    from
        products products0_
    where
        products0_.category_id=?
{ 221, Mobile Phone, [{ 1002, Samsumg Galaxy, 299.00 }, { 1001, iPhone 6S, 699.00 }] }
{ 231, Fashion, [{ 1006, Belt, 9.90 }, { 1004, Jeans, 78.99 }, { 1005, Scarf, 19.99 }, { 1003, Designer Skirt, 49.00 }] }
{ 241, Home, [{ 1007, Sporinkler, 89.00 }] }
{ 251, School, [{ 1008, Notebook, 9.00 }, { 1009, Pen, 4.99 }] }
```

Here is our *`select * from categories`* and these will now return `Category` objects.

If you scroll down below, and take a look at the individual categories printed out to screen, you can see that our two string representations of categories have been used to print out category details along with the products that belong to each category.

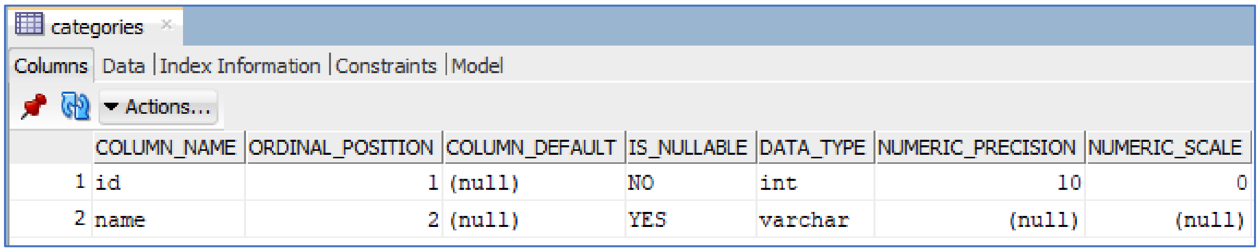


```
Hibernate:
    SELECT
        *
    FROM
        products
{ 1001, iPhone 6S, 699.00 }
{ 1002, Samsumg Galaxy, 299.00 }
{ 1003, Designer Skirt, 49.00 }
{ 1004, Jeans, 78.99 }
{ 1005, Scarf, 19.99 }
{ 1006, Belt, 9.90 }
{ 1007, Sporinkler, 89.00 }
{ 1008, Notebook, 9.00 }
{ 1009, Pen, 4.99 }
```

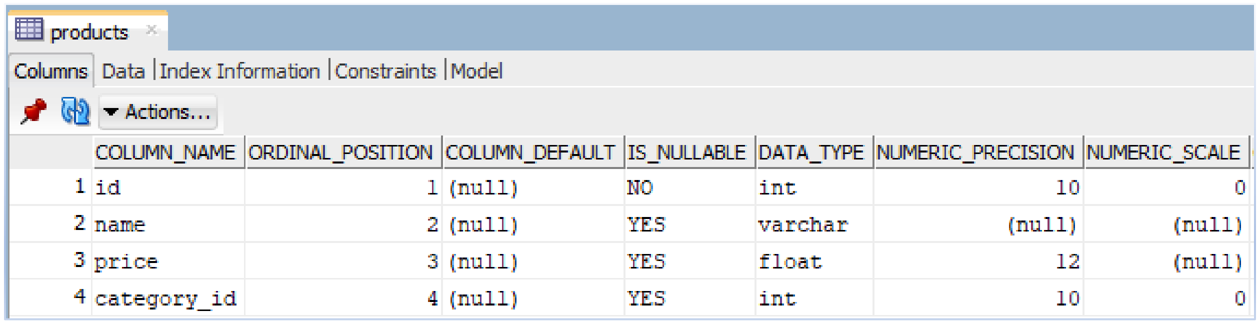
JPA and Hibernate have correctly identified these entities as category entities.

And the same is true for products as well. *`Select * from products`* and the product details have been printed out to screen.

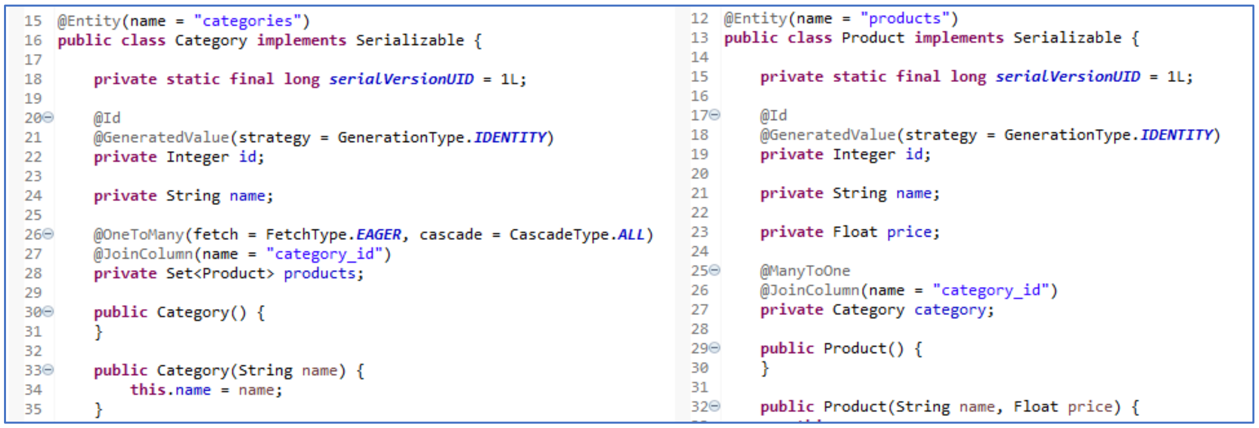
Native queries are not often used, but they're useful if the query that you want to run is a complex one, which is not supported by JPQL, that is the Java Persistence Query Language.



| | COLUMN_NAME | ORDINAL_POSITION | COLUMN_DEFAULT | IS_NULLABLE | DATA_TYPE | NUMERIC_PRECISION | NUMERIC_SCALE |
|---|---|---|---|---|---|---|---|
| 1 | id | 1 | (null) | NO | int | 10 | 0 |
| 2 | name | 2 | (null) | YES | varchar | (null) | (null) |

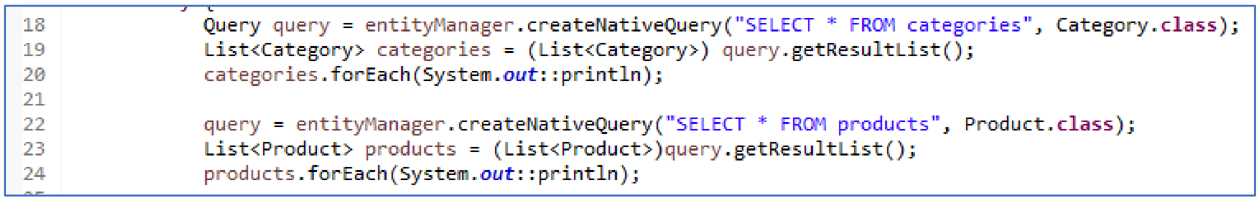


| | COLUMN_NAME | ORDINAL_POSITION | COLUMN_DEFAULT | IS_NULLABLE | DATA_TYPE | NUMERIC_PRECISION | NUMERIC_SCALE |
|---|---|---|---|---|---|---|---|
| 1 | id | 1 | (null) | NO | int | 10 | 0 |
| 2 | name | 2 | (null) | YES | varchar | (null) | (null) |
| 3 | price | 3 | (null) | YES | float | 12 | (null) |
| 4 | category_id | 4 | (null) | YES | int | 10 | 0 |



```java
@Entity(name = "categories")
public class Category implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String name;

    @OneToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    @JoinColumn(name = "category_id")
    private Set<Product> products;

    public Category() {
    }

    public Category(String name) {
        this.name = name;
    }
```

```java
@Entity(name = "products")
public class Product implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String name;

    private Float price;

    @ManyToOne
    @JoinColumn(name = "category_id")
    private Category category;

    public Product() {
    }

    public Product(String name, Float price) {
```



```java
        Query query = entityManager.createNativeQuery("SELECT * FROM categories", Category.class);
        List<Category> categories = (List<Category>) query.getResultList();
        categories.forEach(System.out::println);

        query = entityManager.createNativeQuery("SELECT * FROM products", Product.class);
        List<Product> products = (List<Product>)query.getResultList();
        products.forEach(System.out::println);
```

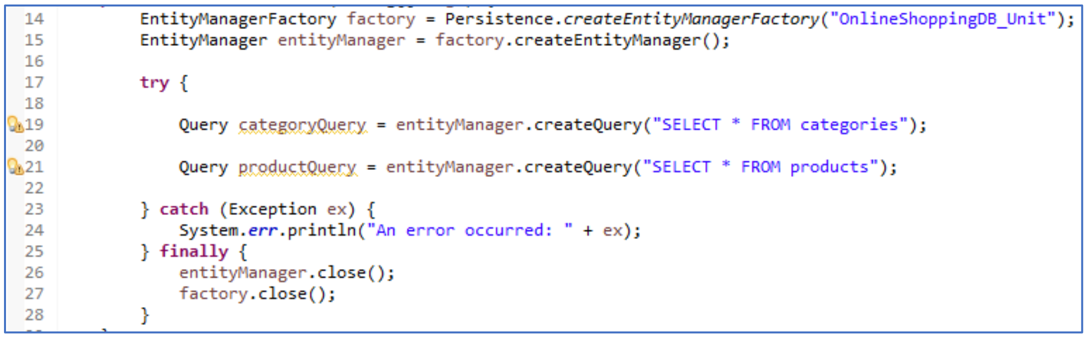
## Executing JPQL Queries

Starting from this demo onwards, we'll work with JPQL Queries. These are special queries that are understood by the Java Persistence API.

Even though in many cases, JPQL queries might look like regular native SQL queries. They are actually different because they are understood by JPA and the persistence provider that implements JPA. **JPQL queries are platform-independent**, you can use the same query with any underlying database.

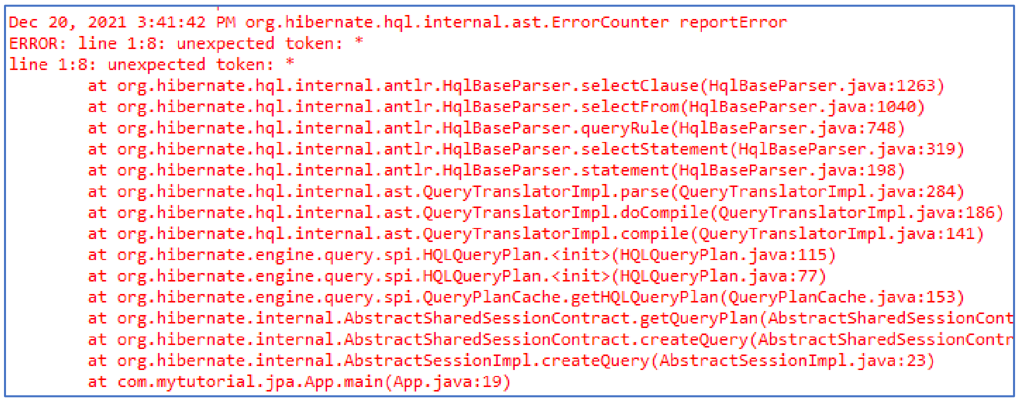
They are also dynamic queries, where the logic of the query is defined within your application's business logic, as you shall see.

JPQL queries are created using the `createQuery()` method on the entity manager. I've used createQuery, but notice my query, it's still a native query.



```java
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {

            Query categoryQuery = entityManager.createQuery("SELECT * FROM categories");

            Query productQuery = entityManager.createQuery("SELECT * FROM products");

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

*Select * from categories*, this will not work with JPQL. I'll show you just by running this code, you will find that we'll encounter an exception. Here at the bottom you can see that we have an exception which says unexpected token *.

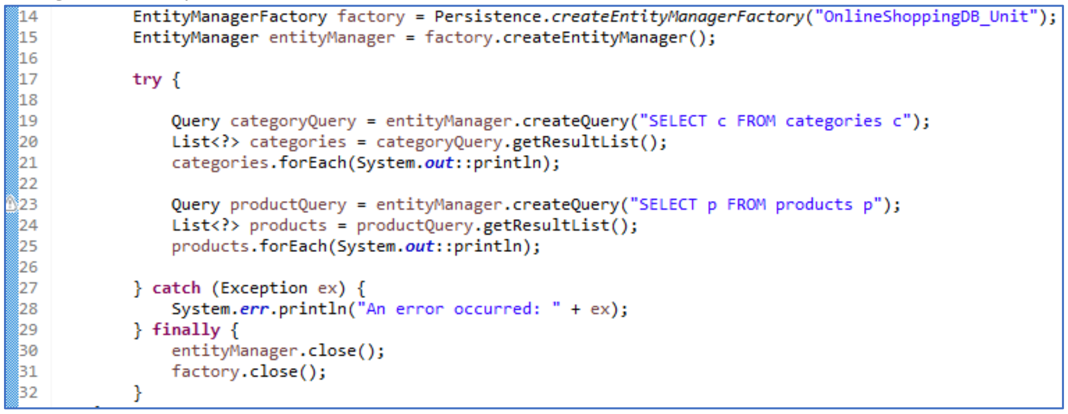


```
Dec 20, 2021 3:41:42 PM org.hibernate.hql.internal.ast.ErrorCounter reportError
ERROR: line 1:8: unexpected token: *
line 1:8: unexpected token: *
        at org.hibernate.hql.internal.antlr.HqlBaseParser.selectClause(HqlBaseParser.java:1263)
        at org.hibernate.hql.internal.antlr.HqlBaseParser.selectFrom(HqlBaseParser.java:1040)
        at org.hibernate.hql.internal.antlr.HqlBaseParser.queryRule(HqlBaseParser.java:748)
        at org.hibernate.hql.internal.antlr.HqlBaseParser.selectStatement(HqlBaseParser.java:319)
        at org.hibernate.hql.internal.antlr.HqlBaseParser.statement(HqlBaseParser.java:198)
        at org.hibernate.hql.internal.ast.QueryTranslatorImpl.parse(QueryTranslatorImpl.java:284)
        at org.hibernate.hql.internal.ast.QueryTranslatorImpl.doCompile(QueryTranslatorImpl.java:186)
        at org.hibernate.hql.internal.ast.QueryTranslatorImpl.compile(QueryTranslatorImpl.java:141)
        at org.hibernate.engine.query.spi.HQLQueryPlan.<init>(HQLQueryPlan.java:115)
        at org.hibernate.engine.query.spi.HQLQueryPlan.<init>(HQLQueryPlan.java:77)
        at org.hibernate.engine.query.spi.QueryPlanCache.getHQLQueryPlan(QueryPlanCache.java:153)
        at org.hibernate.internal.AbstractSharedSessionContract.getQueryPlan(AbstractSharedSessionCont
        at org.hibernate.internal.AbstractSharedSessionContract.createQuery(AbstractSharedSessionContr
        at org.hibernate.internal.AbstractSessionImpl.createQuery(AbstractSessionImpl.java:23)
        at com.mytutorial.jpa.App.main(App.java:19)
```

JPQL queries have a specific syntax which are similar to regular SQL queries, but they are different. So make sure you get the differences right, otherwise you'll end up with an exception like you see here on screen.

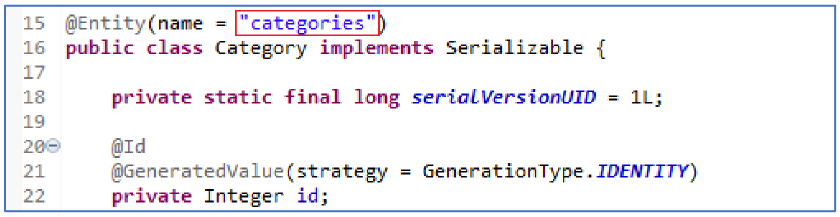
## How to use JPQL Queries

Let's fix our queries so they're actually JPQL queries. I want to do a select of all entities from the categories and products table.



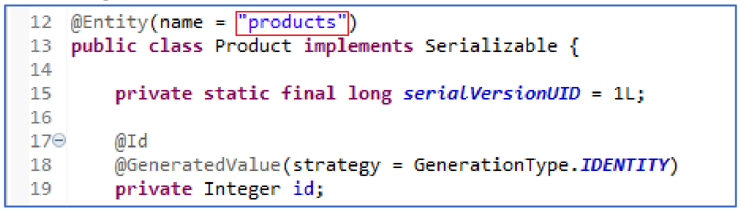
```
14          EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
15          EntityManager entityManager = factory.createEntityManager();
16
17          try {
18
19              Query categoryQuery = entityManager.createQuery("SELECT c FROM categories c");
20              List<?> categories = categoryQuery.getResultList();
21              categories.forEach(System.out::println);
22
23              Query productQuery = entityManager.createQuery("SELECT p FROM products p");
24              List<?> products = productQuery.getResultList();
25              products.forEach(System.out::println);
26
27          } catch (Exception ex) {
28              System.err.println("An error occurred: " + ex);
29          } finally {
30              entityManager.close();
31              factory.close();
32          }
```

- Let's take a look at the first query on line 19. I've used `entityManager.createQuery()`, create a new query object.
- And notice the query, `"SELECT c FROM categories c"`.
  You need to have a variable to represent your underlying table. And `categories` here refers to the name of the entity that you had specified in the **@Entity** annotation for your `Category` class.



```
@Entity(name = "categories")
public class Category implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
```

- `categories` here refers to the *name* property, make sure you get your capitalization, plural, everything exactly correct.
- Category entities are represented using the variable `c` and you want to `select` all `c` s, that is all entities from this `categories` table.
- Once you've created your category query, you can call **`getResultList()`** to get a `List` of all the result entities, which I stored in the `Categories List`.
- I then run a `forEach()` operation and print out each `Category`.
- Similarly, I've created a query `"SELECT p FROM products p"`. The `from` clause here once again references the *name* property that we had specified in the **@Entity** annotation for the `Product` entity. Make sure you get the capitalization, the plural form, everything exactly correct and matching.

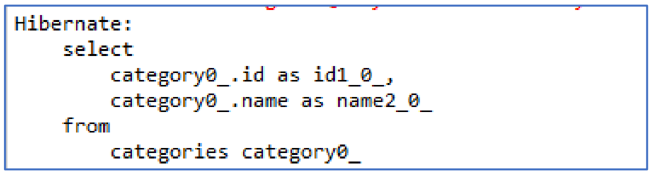


```
@Entity(name = "products")
public class Product implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
```

- Once you've created a `product` query, you can execute this query by calling `productQuery.getResultList()`. This will return a `List` of `Products`.
- We run a `forEach()` operation over all elements of this `List` and print them out to screen.

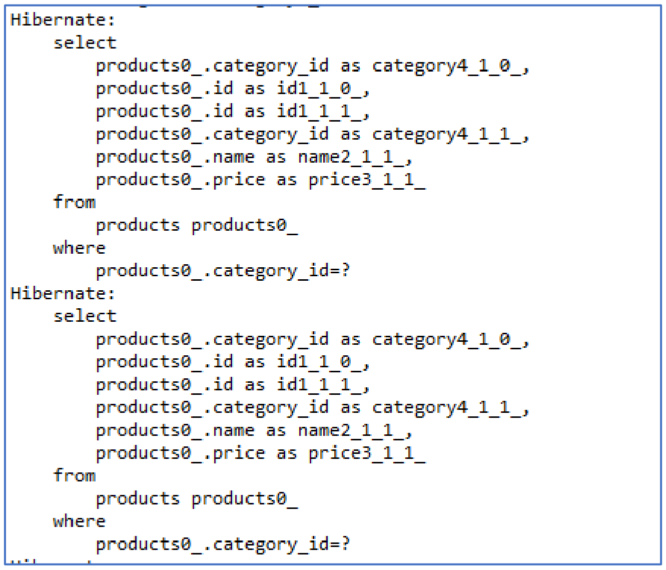
Let's run this JPQL queries and take a look at the results.

- Here is the select statement which selects all of the categories from our category table.



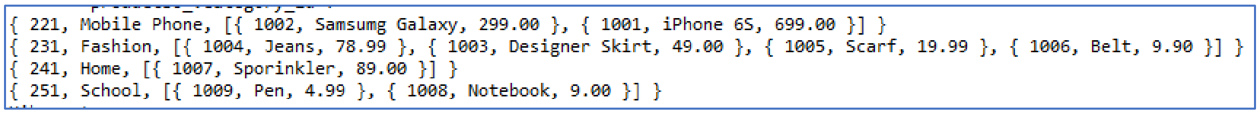
```
Hibernate:
    select
        category0_.id as id1_0_,
        category0_.name as name2_0_
    from
        categories category0_
```

- For each category, we also retrieve the products that belong to that category. Which is why you see all of these select statements for products belonging to each category.



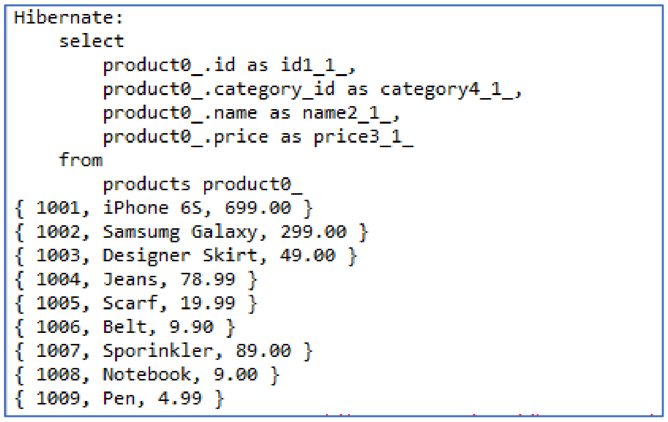
```
Hibernate:
    select
        products0_.category_id as category4_1_0_,
        products0_.id as id1_1_0_,
        products0_.id as id1_1_1_,
        products0_.category_id as category4_1_1_,
        products0_.name as name2_1_1_,
        products0_.price as price3_1_1_
    from
        products products0_
    where
        products0_.category_id=?
Hibernate:
    select
        products0_.category_id as category4_1_0_,
        products0_.id as id1_1_0_,
        products0_.id as id1_1_1_,
        products0_.category_id as category4_1_1_,
        products0_.name as name2_1_1_,
        products0_.price as price3_1_1_
    from
        products products0_
    where
        products0_.category_id=?
```

- And here at the bottom printed out to screen are category details, “Mobile Phones” with products “iPhone” and “Samsung Galaxy”. The “Fashion” category with the “Scarf”, “Jeans”, “Belt” and a “Designer Skirt”. The “Home” and "School” categories printed out as well.



```
{ 221, Mobile Phone, [{ 1002, Samsumg Galaxy, 299.00 }, { 1001, iPhone 6S, 699.00 }] }
{ 231, Fashion, [{ 1004, Jeans, 78.99 }, { 1003, Designer Skirt, 49.00 }, { 1005, Scarf, 19.99 }, { 1006, Belt, 9.90 }] }
{ 241, Home, [{ 1007, Sporinkler, 89.00 }] }
{ 251, School, [{ 1009, Pen, 4.99 }, { 1008, Notebook, 9.00 }] }
```

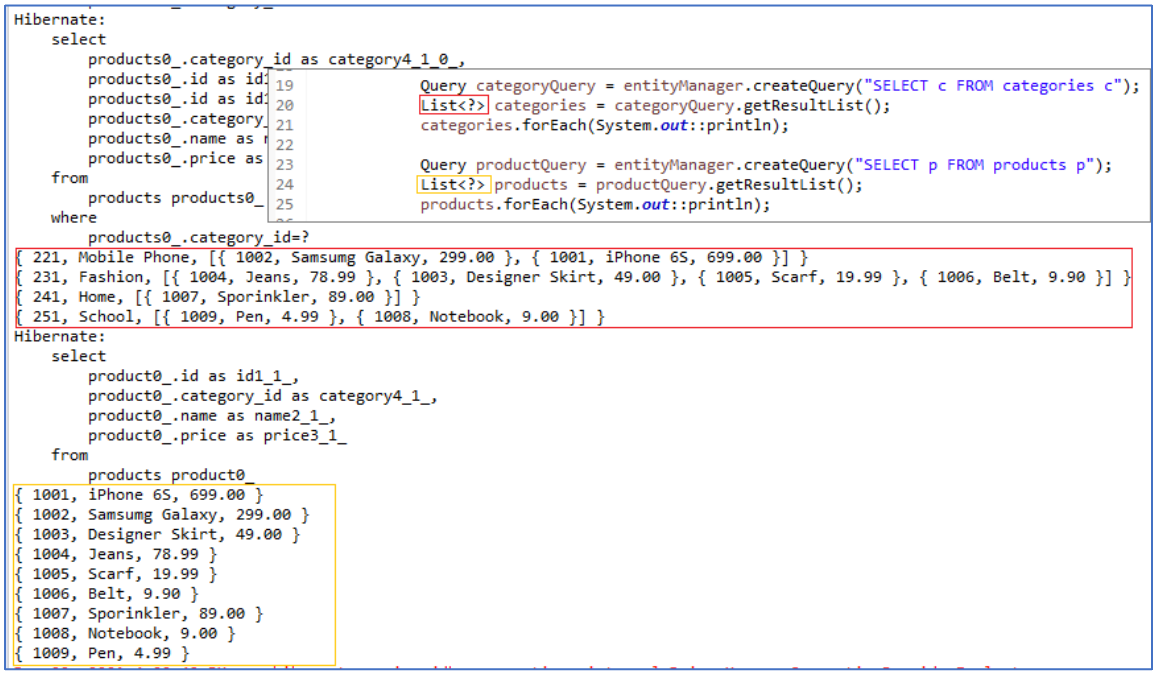
- Next, we have a second select operation to select all products from the underlying products table and all of the products are printed out to screen.



```
Hibernate:
    select
        product0_.id as id1_1_,
        product0_.category_id as category4_1_,
        product0_.name as name2_1_,
        product0_.price as price3_1_
    from
        products product0_
{ 1001, iPhone 6S, 699.00 }
{ 1002, Samsumg Galaxy, 299.00 }
{ 1003, Designer Skirt, 49.00 }
{ 1004, Jeans, 78.99 }
{ 1005, Scarf, 19.99 }
{ 1006, Belt, 9.90 }
{ 1007, Sporinkler, 89.00 }
{ 1008, Notebook, 9.00 }
{ 1009, Pen, 4.99 }
```

- Notice, even if we haven't explicitly specified that `categories` correspond to the `category` entity and `products` correspond to the `product` entity.

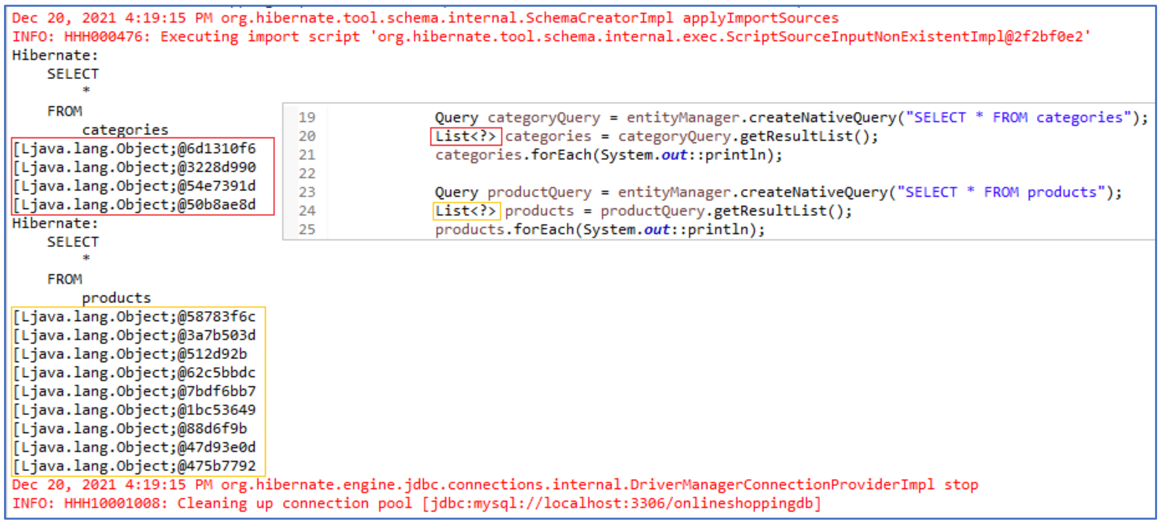
Uses JPQL



```
Query categoryQuery = entityManager.createQuery("SELECT c FROM categories c");
List<?> categories = categoryQuery.getResultList();
categories.forEach(System.out::println);

Query productQuery = entityManager.createQuery("SELECT p FROM products p");
List<?> products = productQuery.getResultList();
products.forEach(System.out::println);
```

```
Hibernate:
    select
        products0_.category_id as category4_1_0_,
        products0_.id as id1
        products0_.id as id1
        products0_.category_
        products0_.name as n
        products0_.price as
    from
        products products0_
    where
        products0_.category_id=?
{ 221, Mobile Phone, [{ 1002, Samsumg Galaxy, 299.00 }, { 1001, iPhone 6S, 699.00 }] }
{ 231, Fashion, [{ 1004, Jeans, 78.99 }, { 1003, Designer Skirt, 49.00 }, { 1005, Scarf, 19.99 }, { 1006, Belt, 9.90 }] }
{ 241, Home, [{ 1007, Sporinkler, 89.00 }] }
{ 251, School, [{ 1009, Pen, 4.99 }, { 1008, Notebook, 9.00 }] }
Hibernate:
    select
        product0_.id as id1_1_,
        product0_.category_id as category4_1_,
        product0_.name as name2_1_,
        product0_.price as price3_1_
    from
        products product0_
{ 1001, iPhone 6S, 699.00 }
{ 1002, Samsumg Galaxy, 299.00 }
{ 1003, Designer Skirt, 49.00 }
{ 1004, Jeans, 78.99 }
{ 1005, Scarf, 19.99 }
{ 1006, Belt, 9.90 }
{ 1007, Sporinkler, 89.00 }
{ 1008, Notebook, 9.00 }
{ 1009, Pen, 4.99 }
```

Vs. Use Native Query



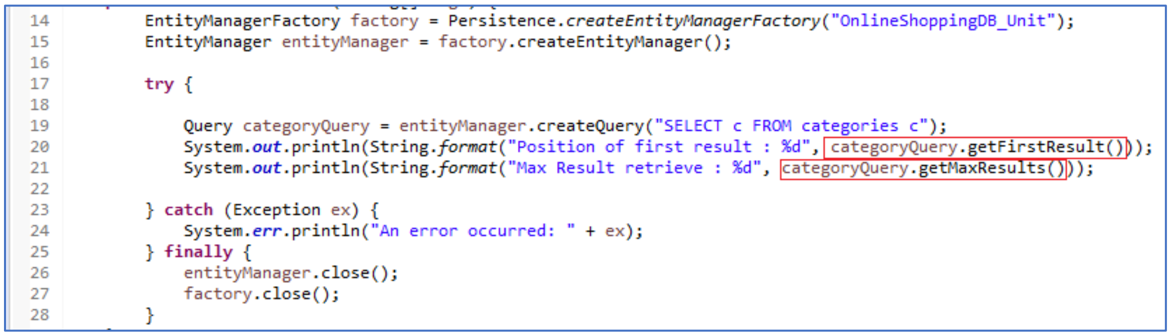
```
Query categoryQuery = entityManager.createNativeQuery("SELECT * FROM categories");
List<?> categories = categoryQuery.getResultList();
categories.forEach(System.out::println);

Query productQuery = entityManager.createNativeQuery("SELECT * FROM products");
List<?> products = productQuery.getResultList();
products.forEach(System.out::println);
```

```
Dec 20, 2021 4:19:15 PM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInputNonExistentImpl@2f2bf0e2'
Hibernate:
    SELECT
        *
    FROM
        categories
[Ljava.lang.Object;@6d1310f6
[Ljava.lang.Object;@3228d990
[Ljava.lang.Object;@54e7391d
[Ljava.lang.Object;@50b8ae8d
Hibernate:
    SELECT
        *
    FROM
        products
[Ljava.lang.Object;@58783f6c
[Ljava.lang.Object;@3a7b503d
[Ljava.lang.Object;@512d92b
[Ljava.lang.Object;@62c5bbdc
[Ljava.lang.Object;@7bdf6bb7
[Ljava.lang.Object;@1bc53649
[Ljava.lang.Object;@88d6f9b
[Ljava.lang.Object;@47d93e0d
[Ljava.lang.Object;@475b7792
Dec 20, 2021 4:19:15 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl stop
INFO: HHH10001008: Cleaning up connection pool [jdbc:mysql://localhost:3306/onlineshoppingdb]
```

This JPQL query knows the results have been retrieved by creating the correct entity object. And that's why you see the correct **`toString()`** representations when the entities are printed out to screen.

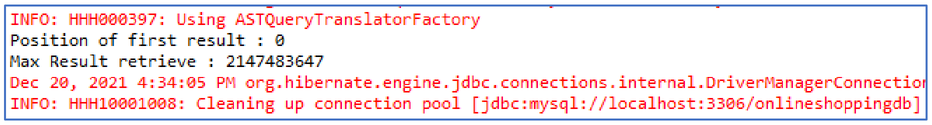
## Additional Parameters that provided by JPQL

When you create a JPQL query, there are additional parameters that you can access.



```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {

            Query categoryQuery = entityManager.createQuery("SELECT c FROM categories c");
            System.out.println(String.format("Position of first result : %d", categoryQuery.getFirstResult()));
            System.out.println(String.format("Max Result retrieve : %d", categoryQuery.getMaxResults()));

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

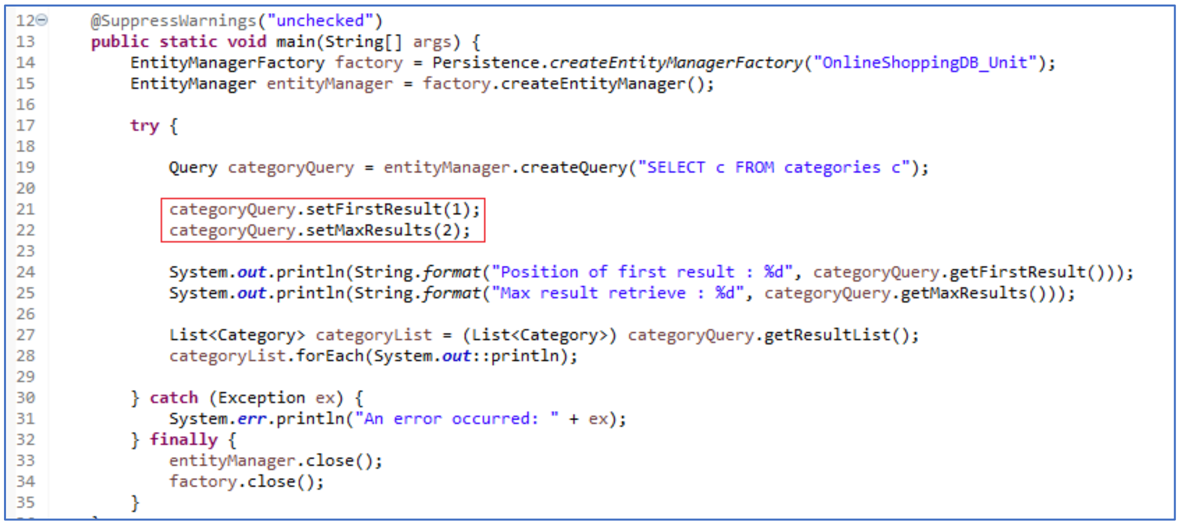
- Here is a create query to retrieve all `category` entities from the underlying table.
  `entityManager.createQuery("SELECT c FROM categories c")`
- Using the query, I access the first result, that is the index of the first result.
  `categoryQuery.getFirstResult()`
- And I access the total number of results that can be retrieved, that is max results retrieved.
  `categoryQuery.getMaxResults()`
- When you run this code, you'll see by default, the position of the first result is always the result at index 0 and max result is this very large number.



```
INFO: HHH000397: Using ASTQueryTranslatorFactory
Position of first result : 0
Max Result retrieve : 2147483647
Dec 20, 2021 4:34:05 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectio
INFO: HHH10001008: Cleaning up connection pool [jdbc:mysql://localhost:3306/onlineshoppingdb]
```

- By default JPA Hibernate tries to retrieve all results that exist in the underlying table.

But you can configure this and that's exactly what I'll do next.

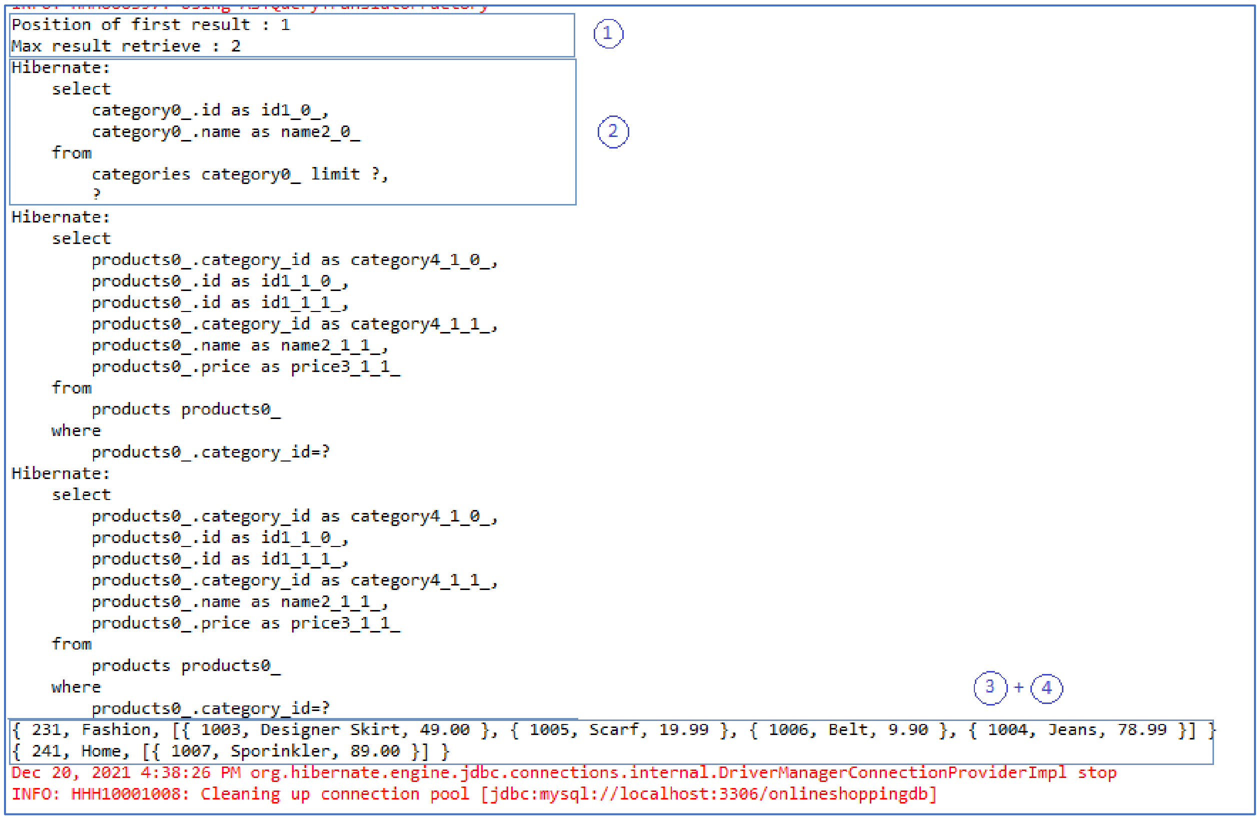


```
12      @SuppressWarnings("unchecked")
13      public static void main(String[] args) {
14          EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
15          EntityManager entityManager = factory.createEntityManager();
16
17          try {
18
19              Query categoryQuery = entityManager.createQuery("SELECT c FROM categories c");
20
21              categoryQuery.setFirstResult(1);
22              categoryQuery.setMaxResults(2);
23
24              System.out.println(String.format("Position of first result : %d", categoryQuery.getFirstResult()));
25              System.out.println(String.format("Max result retrieve : %d", categoryQuery.getMaxResults()));
26
27              List<Category> categoryList = (List<Category>) categoryQuery.getResultList();
28              categoryList.forEach(System.out::println);
29
30          } catch (Exception ex) {
31              System.err.println("An error occurred: " + ex);
32          } finally {
33              entityManager.close();
34              factory.close();
35          }
```

- I'll still create the same category query to query all categories that exist in the underlying table.
- But I want to limit the number of categories that I retrieve. So I'll set the first result to be equal to 1 and max results to 2, which means I want to retrieve two results, starting at index 1.
- I'll print out the first result and max results, this is on lines 24 and 25.

- I'll then run `categoryQuery.getResultList()` and cast this list to a `List` of `category` objects and print out the `category List`.

Let's run this and see whether this query works as we would expect it to. If you scroll down below, you can see the parameters that we have specified for our query.



```
Position of first result : 1
Max result retrieve : 2
Hibernate:
    select
        category0_.id as id1_0_,
        category0_.name as name2_0_
    from
        categories category0_ limit ?,
        ?
Hibernate:
    select
        products0_.category_id as category4_1_0_,
        products0_.id as id1_1_0_,
        products0_.id as id1_1_1_,
        products0_.category_id as category4_1_1_,
        products0_.name as name2_1_1_,
        products0_.price as price3_1_1_
    from
        products products0_
    where
        products0_.category_id=?
Hibernate:
    select
        products0_.category_id as category4_1_0_,
        products0_.id as id1_1_0_,
        products0_.id as id1_1_1_,
        products0_.category_id as category4_1_1_,
        products0_.name as name2_1_1_,
        products0_.price as price3_1_1_
    from
        products products0_
    where
        products0_.category_id=?
{ 231, Fashion, [{ 1003, Designer Skirt, 49.00 }, { 1005, Scarf, 19.99 }, { 1006, Belt, 9.90 }, { 1004, Jeans, 78.99 }] }
{ 241, Home, [{ 1007, Sporinkler, 89.00 }] }
Dec 20, 2021 4:38:26 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl stop
INFO: HHH10001008: Cleaning up connection pool [jdbc:mysql://localhost:3306/onlineshoppingdb]
```

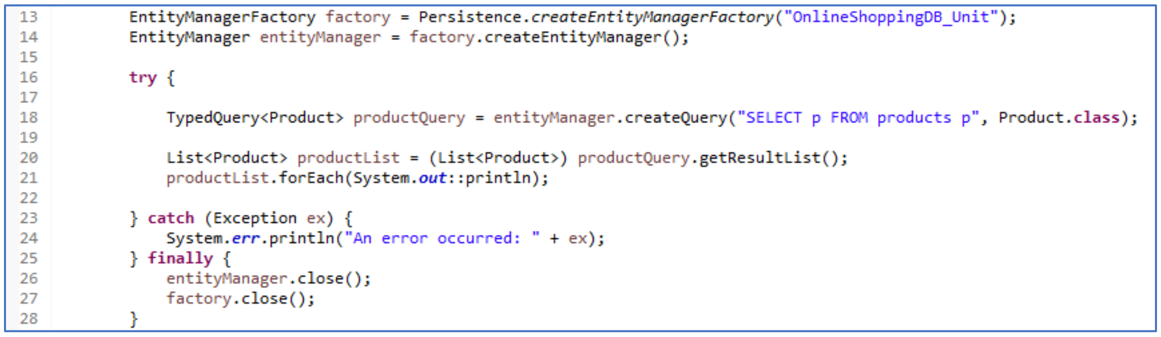
1) Position of the first result is 1 and the max results retrieved is 2.
2) Notice also that when we perform a `select` operation on the `category` table, we limit what we retrieve. We parameterize our query based on the number of results that we want to retrieve.
3) Scroll down below and here is the first of our two results, the “Fashion” `category` and all `products` that belong to the “Fashion” `category`.
4) And if you scroll further down, you'll see the second result, the “Home” category with only the “Sprinkler” product belonging to this category.

## TYPED Query

When you are executing JPQL queries, rather than use the plain vanilla query object, which is not a generic type, you can use the `javax.persistence.TypedQuery`.

The **TypedQuery** avoids you having to perform all of the cast operations to cast the retrieved objects to the right kind of entity.

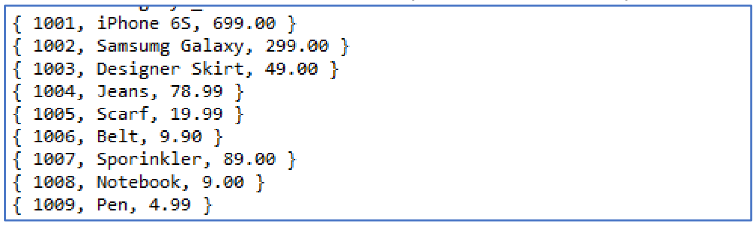
Let's take a look at how you can create and use a type query.



```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {

            TypedQuery<Product> productQuery = entityManager.createQuery("SELECT p FROM products p", Product.class);

            List<Product> productList = (List<Product>) productQuery.getResultList();
            productList.forEach(System.out::println);

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

- I've used `entityManager.createQuery()` as I've done before. The query is select p from Products p, and I have also specify the entity type, `Product.class`.
- Notice that the query that is created, `productQuery`, is a generic type. It's a `TypedQuery` with the generic data type set to `Product`.
- Once I have a `TypedQuery`, when I call `productQuery.getResultList()`, I don't have to do any casting and I get a `List` of `Product` entities.
- And with this `List` of `Products`, I simply print them out to screen, typed queries make things easier overall.

I'm going to run my code and scroll down to the very bottom to take a look at the list of products that were returned. Here are all of the products that are present in the underlying database table.



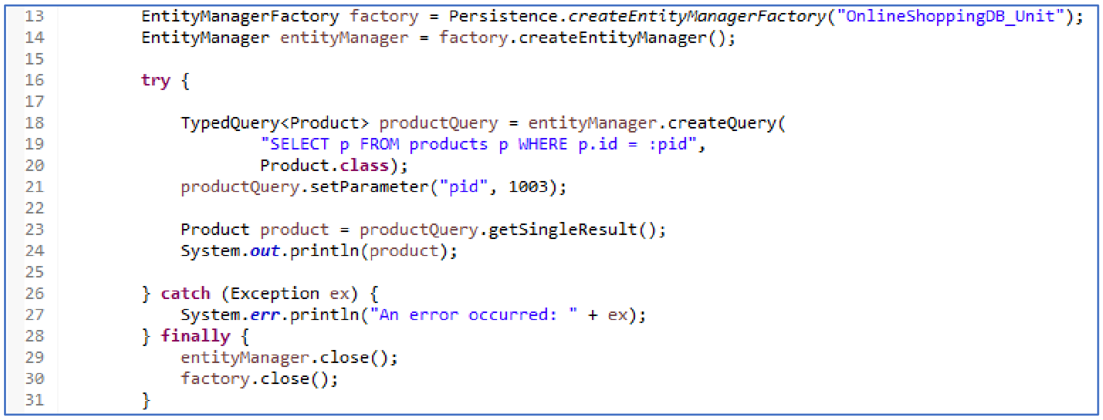
```
{ 1001, iPhone 6S, 699.00 }
{ 1002, Samsumg Galaxy, 299.00 }
{ 1003, Designer Skirt, 49.00 }
{ 1004, Jeans, 78.99 }
{ 1005, Scarf, 19.99 }
{ 1006, Belt, 9.90 }
{ 1007, Sporinkler, 89.00 }
{ 1008, Notebook, 9.00 }
{ 1009, Pen, 4.99 }
```

## Using Named Parameters and Positional Parameters

In this demo, we'll see how we can parametrize the queries that we run using named parameters and positional parameters.

### Named Parameters in JPQL

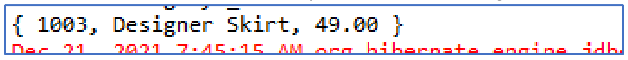
We'll start off by first understanding how named parameters work.



```
13        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
14        EntityManager entityManager = factory.createEntityManager();
15
16        try {
17
18            TypedQuery<Product> productQuery = entityManager.createQuery(
19                    "SELECT p FROM products p WHERE p.id = :pid",
20                    Product.class);
21            productQuery.setParameter("pid", 1003);
22
23            Product product = productQuery.getSingleResult();
24            System.out.println(product);
25
26        } catch (Exception ex) {
27            System.err.println("An error occurred: " + ex);
28        } finally {
29            entityManager.close();
30            factory.close();
31        }
```

- Here is a TypedQuery that I have created, `"SELECT p FROM products p WHERE p.id = :pid"`. I haven't specified the value for `pid` yet. That's something that is a parameter that I can set after I have created the query.
- Another thing to note here in this JPQL query is how I access the id field of product entities. `p` is the variable reference to every product that we select `p.id`. The dot notation is what I used to access the id field.
- The `id` here should be the name of the member variable in your product class.
- I do this on line 21, `productQuery.setParameter()` `"pid"` to `1003`. This will automatically replace the `:pid` named parameter in our query with the value `1003`. And the query will explicitly seek for this parameter while performing the SELECT clause.
  So I want to retrieve the product with id 1003.
- Because this is just a single product, I use `productQuery.getSingleResult()` in order to execute the query.
- Once we've retrieved the product with this id, we print it out to screen.

Let's run this code, and if you scroll through to the very bottom,

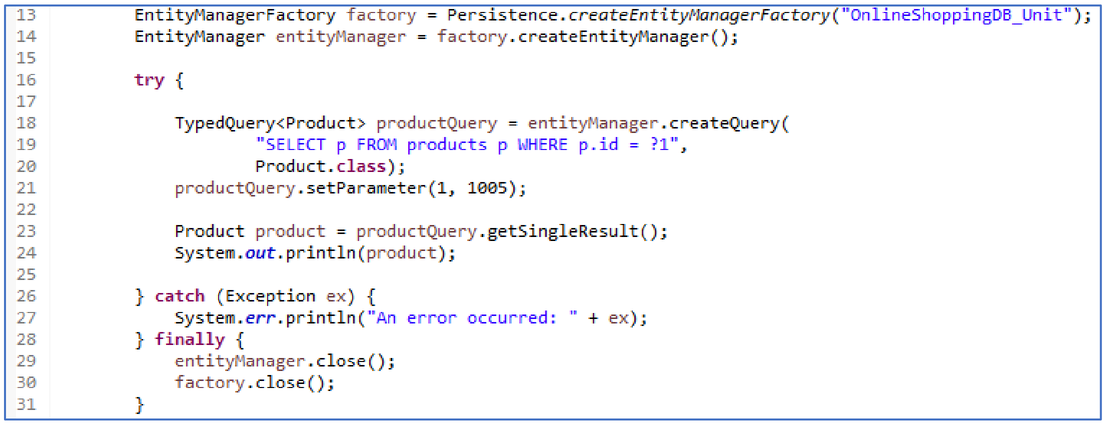


```
{ 1003, Designer Skirt, 49.00 }
Dec 21, 2021 7:45:15 AM org.hibernate.engine.jdb
```

here is the product, a designer skirt for $49. The id is 1003.
This is an example of running a query using named parameters.

## Positional Argument in JPQL

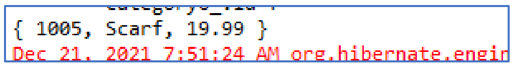
You can also specify parameters in a query using positional arguments.



```java
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {

            TypedQuery<Product> productQuery = entityManager.createQuery(
                    "SELECT p FROM products p WHERE p.id = ?1",
                    Product.class);
            productQuery.setParameter(1, 1005);

            Product product = productQuery.getSingleResult();
            System.out.println(product);

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

- Here is the same query that we ran before. `"SELECT p FROM products p WHERE p.id = ?1"`. `?1` here is the position of the argument. A value of 1 is the first position, a value of 2 is the second position and so on.
- We call `productQuery.setParameter()`. Specify 1 as the position where the parameter has to be replaced and specify an ID of `1005`.
- Notice how the parameter key is an **integer** and not a **String**.
- This is an example of a position parameter.

I've run this code and here at the bottom,

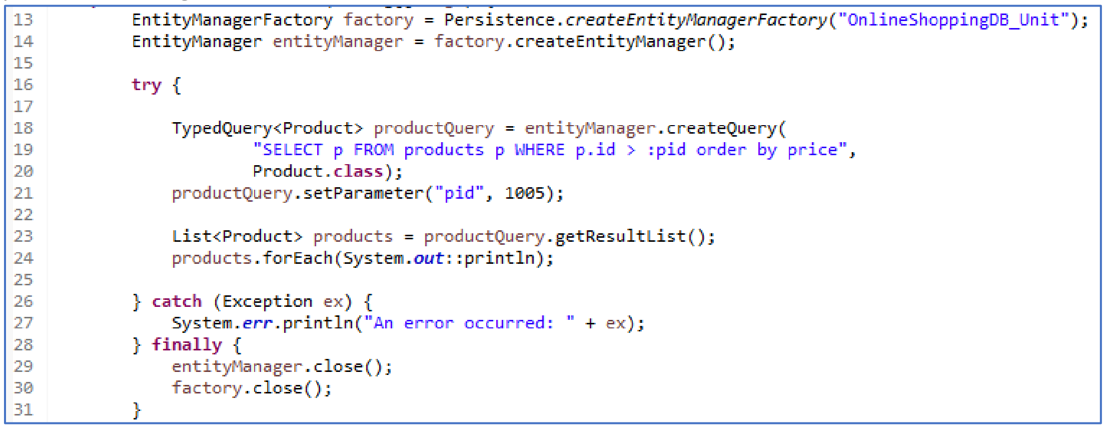


```
{ 1005, Scarf, 19.99 }
Dec 21, 2021 7:51:24 AM org.hibernate.engin
```

here is the product with id 1005, the scarf.

## Ascending Sorting Order in JPQL

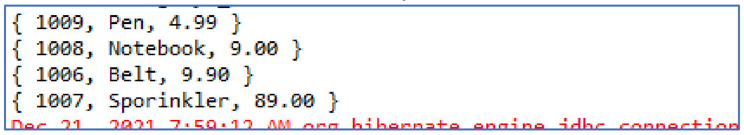
We've seen how we can use parameters and the `getSingleResult()` method. Let's now use parameters to get a `List` of results.



```java
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {

            TypedQuery<Product> productQuery = entityManager.createQuery(
                    "SELECT p FROM products p WHERE p.id > :pid order by price",
                    Product.class);
            productQuery.setParameter("pid", 1005);

            List<Product> products = productQuery.getResultList();
            products.forEach(System.out::println);

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

- In addition, I've also tweaked the query to include an order by clause.
- I want to select all products from the products table where p.id is greater than the id that I've specified. `:pid` indicates that I'm using named parameters and whatever results are returned, I want the results ordered by price.
- After creating the query, I have set the named parameter to be the value 1005.
- When you invoke set parameter with named parameters, notice that we always specify the parameter as a string here. It's the string `pid`. This will retrieve more than one product, it's quite likely, so I call `productQuery.getResultList()` and print out the list of products.

Let's run this code, and at the very bottom,

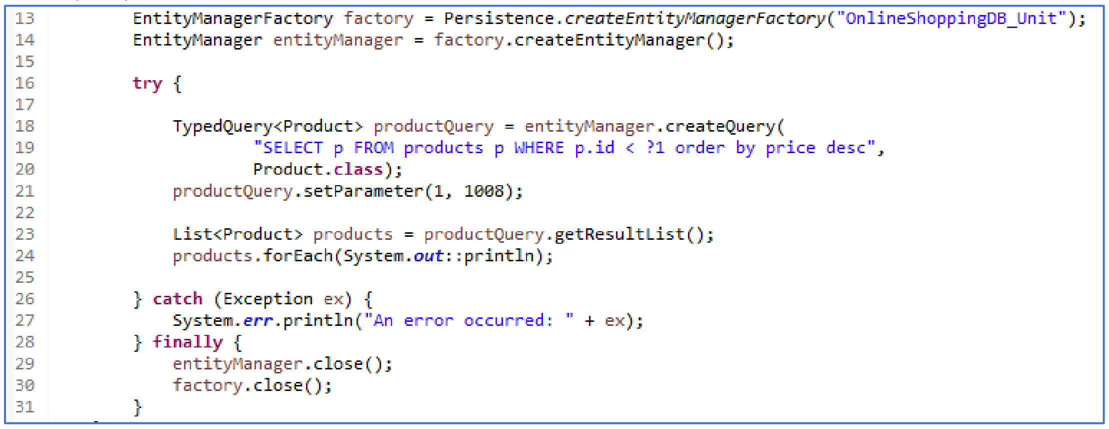


```
{ 1009, Pen, 4.99 }
{ 1008, Notebook, 9.00 }
{ 1006, Belt, 9.90 }
{ 1007, Sporinkler, 89.00 }
Dec 21, 2021 7:59:12 AM org.hibernate.engine.jdbc.connection
```

- You can see the `List` of `products` ordered in the ascending order of the `price` of the `product`.
- All products have ids greater than 1005.

## Descending Sorting Order in JPQL

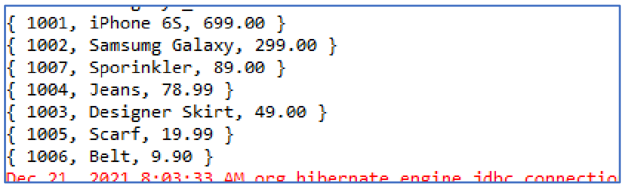
If you want your products in the descending order, you can simply change the order by clause in your JPQL query.



```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {

            TypedQuery<Product> productQuery = entityManager.createQuery(
                    "SELECT p FROM products p WHERE p.id < ?1 order by price desc",
                    Product.class);
            productQuery.setParameter(1, 1008);

            List<Product> products = productQuery.getResultList();
            products.forEach(System.out::println);

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

- I've tweaked the query here so that products are returned in the descending order of `price`, from the highest price to the lowest price.
- I've also used positional parameters, `pid < ?1`.
- I want to return all products less than a certain id in the descending order of price. And the id that I've specified using `setParameter()` is 1008. I've used the integer 1 to indicate that the value 1008 should be associated with the first positional parameter.

I've run this code and somewhere at the very bottom, there will be the list of products that I have retrieved.

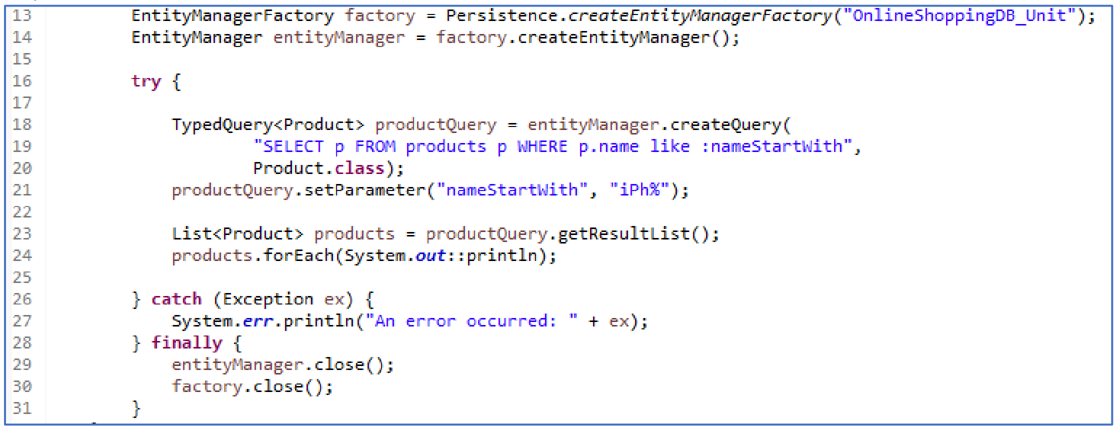


```
{ 1001, iPhone 6S, 699.00 }
{ 1002, Samsumg Galaxy, 299.00 }
{ 1007, Sporinkler, 89.00 }
{ 1004, Jeans, 78.99 }
{ 1003, Designer Skirt, 49.00 }
{ 1005, Scarf, 19.99 }
{ 1006, Belt, 9.90 }
Dec 21, 2021 8:03:33 AM org.hibernate.engine.jdbc.connectio
```

- The most expensive product is displayed first, that is the iPhone 6S, $699, and the least expensive last.
- All IDs are less than a 1008.

## Named Parameter with LIKE Filter in JPQL

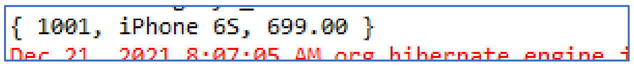
I'll now try running a slightly different query, this is a query that uses named parameters and the like keyword.



```
13          EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
14          EntityManager entityManager = factory.createEntityManager();
15
16          try {
17
18              TypedQuery<Product> productQuery = entityManager.createQuery(
19                      "SELECT p FROM products p WHERE p.name like :nameStartWith",
20                      Product.class);
21              productQuery.setParameter("nameStartWith", "iPh%");
22
23              List<Product> products = productQuery.getResultList();
24              products.forEach(System.out::println);
25
26          } catch (Exception ex) {
27              System.err.println("An error occurred: " + ex);
28          } finally {
29              entityManager.close();
30              factory.close();
31          }
```

- I want to select products from the products table where p.name like what I specify. `:nameStartWith` is my named parameter.
- The parameter value that I've specified on line 21 is `iPh%`, that is a wild card.

Let's retrieve the result list

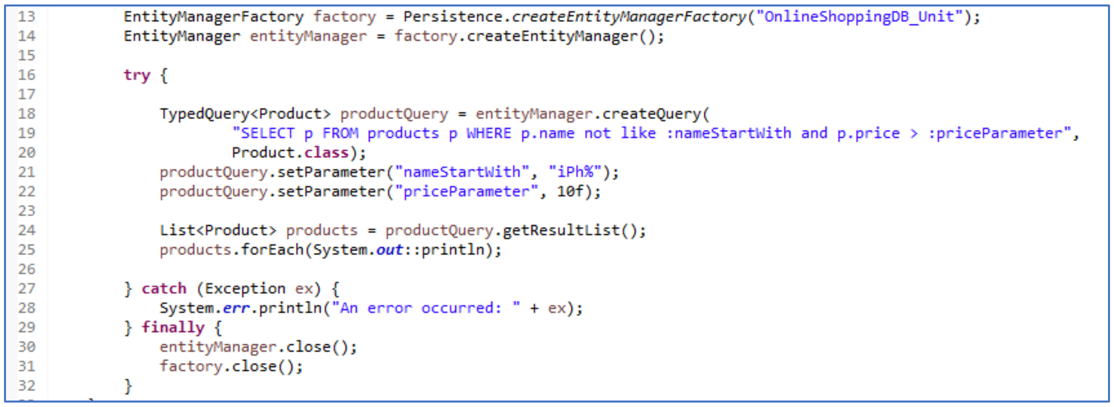


```
{ 1001, iPhone 6S, 699.00 }
Dec 21, 2021 8:07:05 AM org.hibernate.engine.i
```

- and take a look at which product has a name that starts at "iPh".
- There is just one, the iPhone 6S.

## Named Parameter with NOT Filter in JPQL

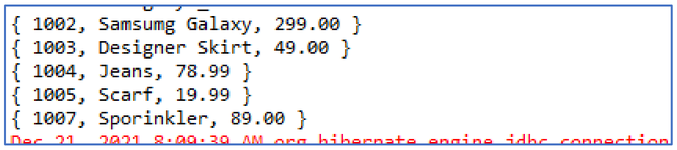
Just like you can use the like keyword in JPQL queries, you can use the not like keyword as well.



```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {

            TypedQuery<Product> productQuery = entityManager.createQuery(
                    "SELECT p FROM products p WHERE p.name not like :nameStartWith and p.price > :priceParameter",
                    Product.class);
            productQuery.setParameter("nameStartWith", "iPh%");
            productQuery.setParameter("priceParameter", 10f);

            List<Product> products = productQuery.getResultList();
            products.forEach(System.out::println);

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

- Here I want to select those products where `p.name` is **not** like `:nameStartWith`. That is the first of my named parameters.
- And I have a second named parameter as well. I want `p.price` to be greater than `:priceParameter`. This is the first query where we've use two named parameters, so you can see it's possible.
- Once again, and notice how we access the attributes of the product entity using the dot notation, p.name and `p.price`.
- Now you can set these two named parameters using their names, `productQuery.setParameter()`. The name substring that I'm looking for is iPh%. The product name should not start with this prefix, the priceParameter I've set to 10. Note that I've specified this as `10f` because this is a floating point value. So I want products with price greater than 10, but which don't have names that start with iPh.

Let's run this code.

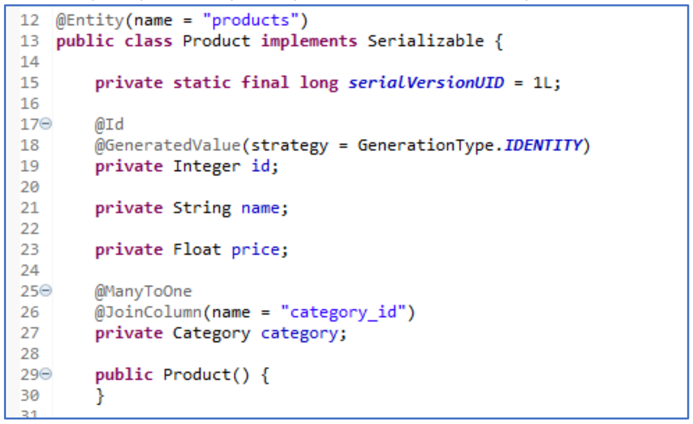


```
{ 1002, Samsumg Galaxy, 299.00 }
{ 1003, Designer Skirt, 49.00 }
{ 1004, Jeans, 78.99 }
{ 1005, Scarf, 19.99 }
{ 1007, Sporinkler, 89.00 }
Dec 21, 2021 8:09:39 AM org.hibernate.engine.jdbc.connection
```

And there are several products that meet this criteria and those are displayed here on screen.

## Referencing Foreign Keys

In this demo, we'll see how you can reference foreign keys in your queries. But before we get to the actual query, let's quickly take a look at the product class.

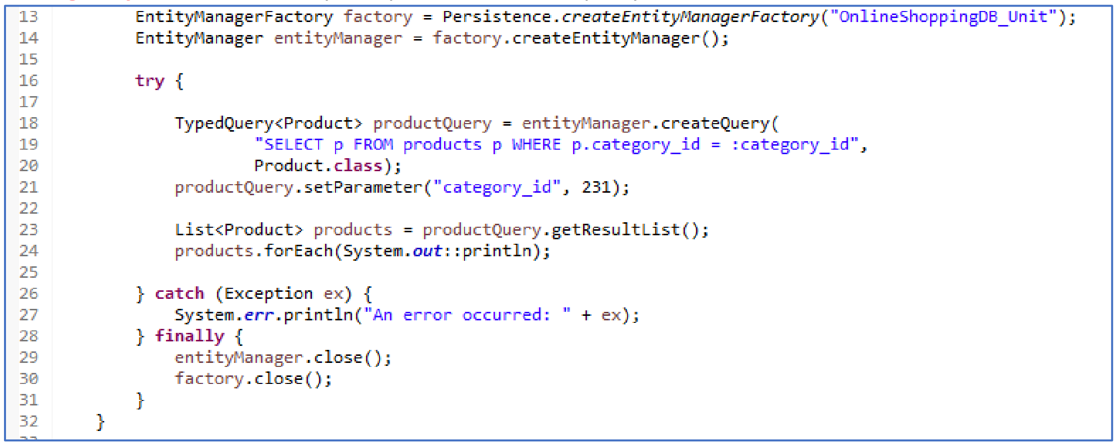


```
@Entity(name = "products")
public class Product implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String name;

    private Float price;

    @ManyToOne
    @JoinColumn(name = "category_id")
    private Category category;

    public Product() {
    }
```

- Notice that the member variables are ID, name, and price. We reference these member variables when we're accessing product attributes in our query.
- Notice that we have a reference here to the category of a product. And this reference is stored in the variable category.
- The join column is category_id. So one category has many products within it, and this is set up as a foreign key reference in the products table. Where the category ID references the primary key that is the ID of the categories table.

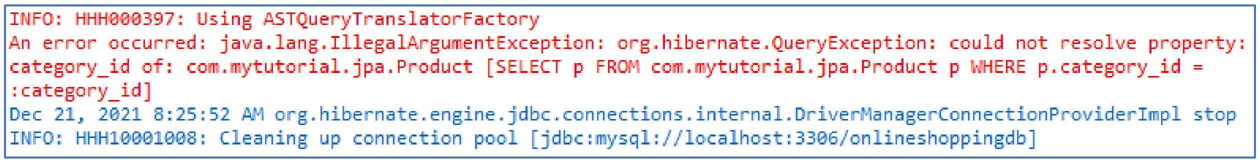
## The Wrong Way to Reference Foreign Key No. 1

Let's see how you can select products from the products table based on their category. Now here is a **wrong way** to do it. I'll tell you upfront that this query is **not correct**.



```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {

            TypedQuery<Product> productQuery = entityManager.createQuery(
                    "SELECT p FROM products p WHERE p.category_id = :category_id",
                    Product.class);
            productQuery.setParameter("category_id", 231);

            List<Product> products = productQuery.getResultList();
            products.forEach(System.out::println);

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
    }
```

- Notice the where clause of this query where I've tried to access the category ID present in the products table `p.category_id` is = `:category_id`.
- `p.category_id` is the name of the joint column, but **you cannot reference column names directly** in JPQL queries.
- You have to use the member variables of the product entity. **Product has no member variable called category_id**, which is why when you run this code and search for the category 231, you end up with an error.

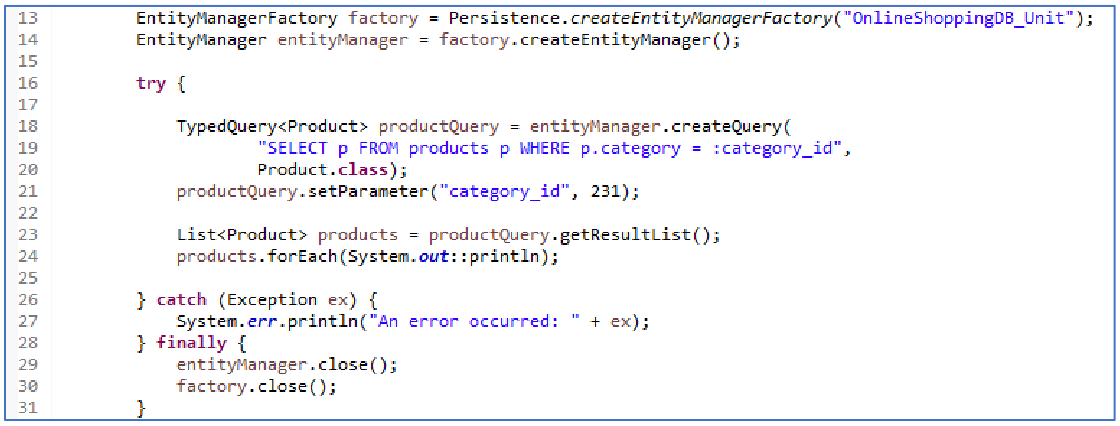


```
INFO: HHH000397: Using ASTQueryTranslatorFactory
An error occurred: java.lang.IllegalArgumentException: org.hibernate.QueryException: could not resolve property:
category_id of: com.mytutorial.jpa.Product [SELECT p FROM com.mytutorial.jpa.Product p WHERE p.category_id =
:category_id]
Dec 21, 2021 8:25:52 AM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl stop
INFO: HHH10001008: Cleaning up connection pool [jdbc:mysql://localhost:3306/onlineshoppingdb]
```

Notice the exception here at the very bottom, QueryException. The exception basically says that the **category_id** property **could not be found** for the product entity. That's because there is no member variable in product called `category_id`.

## The Wrong Way to Reference Foreign Key No. 2

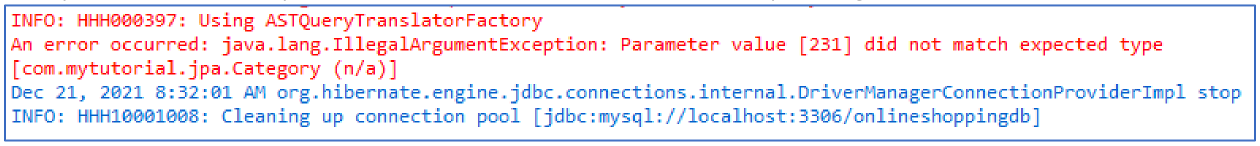
I'm now going to rewrite the query with the foreign key reference but once again, this is wrong.



```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {

            TypedQuery<Product> productQuery = entityManager.createQuery(
                    "SELECT p FROM products p WHERE p.category = :category_id",
                    Product.class);
            productQuery.setParameter("category_id", 231);

            List<Product> products = productQuery.getResultList();
            products.forEach(System.out::println);

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

- Notice that `category` is a member variable to reference the category from within a product. I've used `p.category` in my query, but I've tried to see whether this category is equal to an id 231.
- This is once again **wrong**. This is because when you access `p.category`, you're actually referencing the category object within your product entity, not the category ID.

So if you run this code, you will find that we'll encounter an exception again,

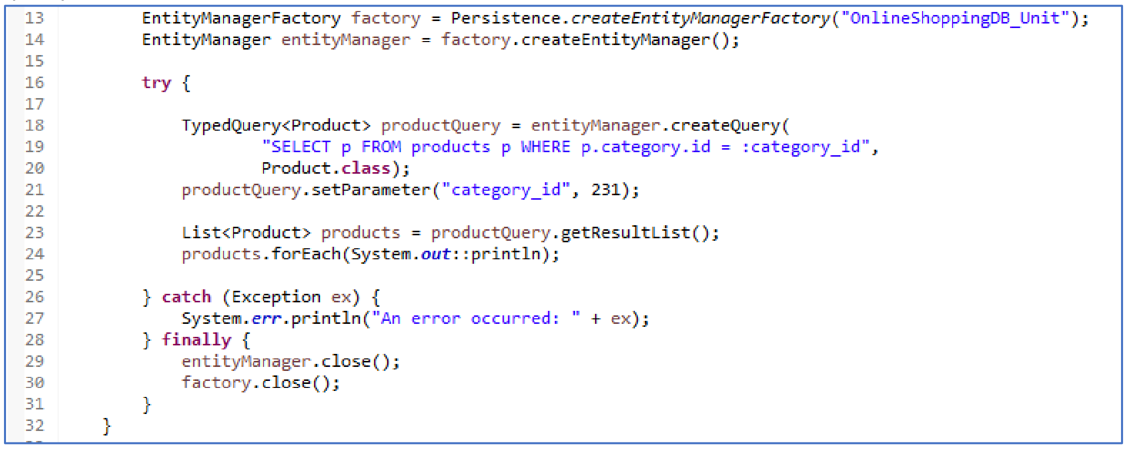


```
INFO: HHH000397: Using ASTQueryTranslatorFactory
An error occurred: java.lang.IllegalArgumentException: Parameter value [231] did not match expected type
[com.mytutorial.jpa.Category (n/a)]
Dec 21, 2021 8:32:01 AM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl stop
INFO: HHH10001008: Cleaning up connection pool [jdbc:mysql://localhost:3306/onlineshoppingdb]
```

right there at the very bottom of our screen. Parameter value 231 did not match the expected type.

## The Correct way to Reference Foreign Key

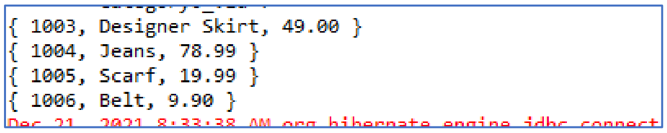
This time around when we reference the category corresponding to a product, let's do it right. Here is a query that is correct.



```java
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {

            TypedQuery<Product> productQuery = entityManager.createQuery(
                    "SELECT p FROM products p WHERE p.category.id = :category_id",
                    Product.class);
            productQuery.setParameter("category_id", 231);

            List<Product> products = productQuery.getResultList();
            products.forEach(System.out::println);

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
    }
```

- Select a product where `p.category.id` is equal to the named parameter, the `category_id` that we have specified. So `p.category` will get us a reference to the category entity and then a further `.id` will allow us to access the ID of that category entity.
- This is how you specify a foreign key reference within your JPQL query. I'm looking for category ID 231.

Let's run this code.

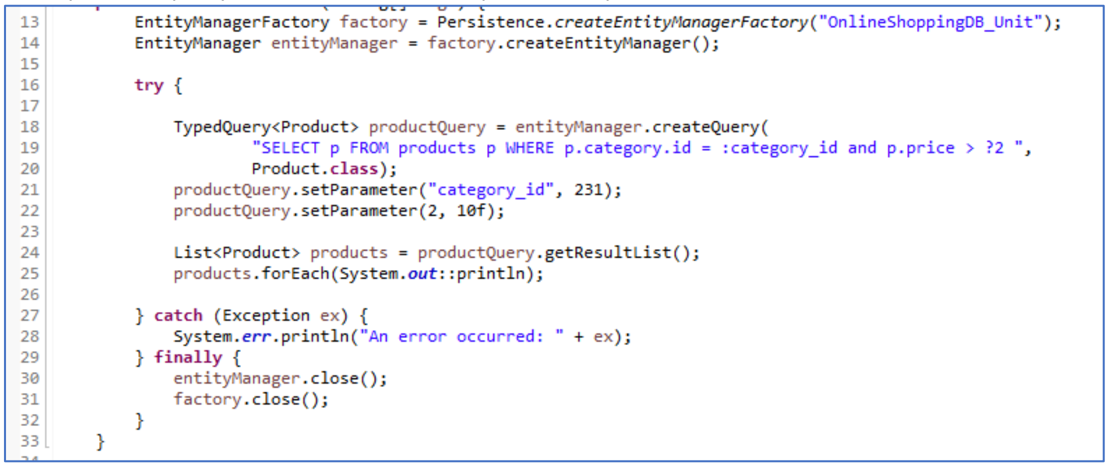


```
{ 1003, Designer Skirt, 49.00 }
{ 1004, Jeans, 78.99 }
{ 1005, Scarf, 19.99 }
{ 1006, Belt, 9.90 }
Dec 21, 2021 8:33:38 AM org.hibernate.engine.jdbc.connect
```

And at the very bottom, you'll see the products in this category. The designer skirt, jeans, scarf and belt.

## Filter with Foreign Key Reference

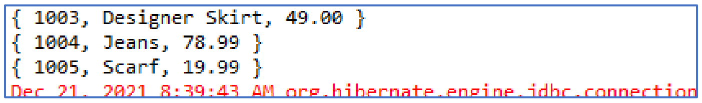
Lets tweak our query a little bit once again, we'll use a foreign key reference but we'll also see an example of a query with more than one positional parameter.



```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {

            TypedQuery<Product> productQuery = entityManager.createQuery(
                    "SELECT p FROM products p WHERE p.category.id = :category_id and p.price > ?2 ",
                    Product.class);
            productQuery.setParameter("category_id", 231);
            productQuery.setParameter(2, 10f);

            List<Product> products = productQuery.getResultList();
            products.forEach(System.out::println);

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
    }
```

- I want products. Where the p.category.id = ?1 and p.price greater than ?2.
- There are two positional parameters corresponding to 1 and 2.
- I call productQuery.setParameter for position 1 that is the ID of the category and
- productQuery.setParameter for the position 2 which is $10 expression floating point.

Let's run this code.



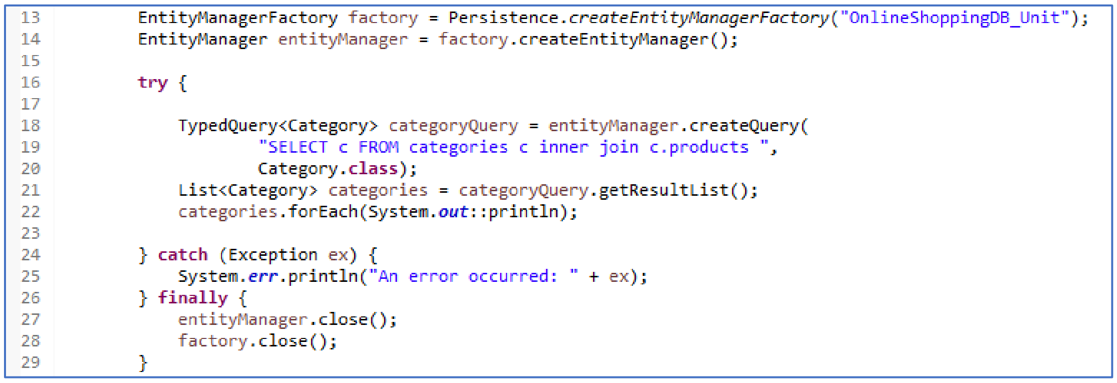
```
{ 1003, Designer Skirt, 49.00 }
{ 1004, Jeans, 78.99 }
{ 1005, Scarf, 19.99 }
Dec 21, 2021 8:39:43 AM org.hibernate.engine.jdbc.connection
```

And here are products in the fashion category with price greater than $10.

## Performing Joins

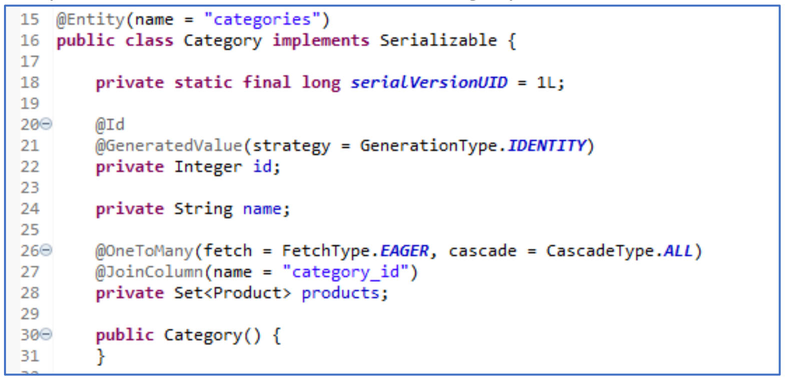
Let's perform a few join operations with JPQL queries. JPQL supports the full range of joins. You can have inner joins, outer joins, left outer, right outer joins, and so on. In this example, we'll just focus on the inner join and the fetch join. Which is a more performant variant of the join.

### Inner Join



```java
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {

            TypedQuery<Category> categoryQuery = entityManager.createQuery(
                    "SELECT c FROM categories c inner join c.products ",
                    Category.class);
            List<Category> categories = categoryQuery.getResultList();
            categories.forEach(System.out::println);

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

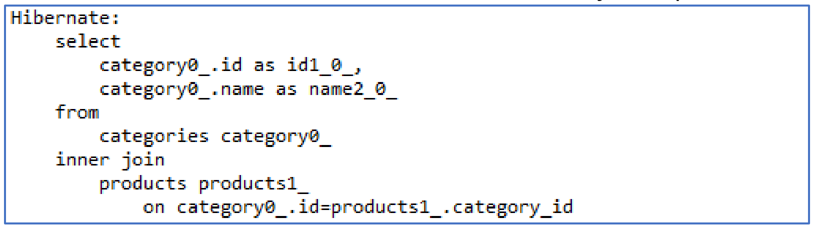
- Here I've selected all of the categories from Category c and performed an inner join with all of the products associated with each category.



```java
@Entity(name = "categories")
public class Category implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String name;

    @OneToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    @JoinColumn(name = "category_id")
    private Set<Product> products;

    public Category() {
    }
```

- Notice how I set up the join, inner join c.products. Remember, the reference to the products is available as a member variable in the Category entity.
- And this is what I have used to perform the join. Because categories hold a reference to the corresponding products, the entities returned by this inner join operation are category entities.
- And that's what I print out to screen.

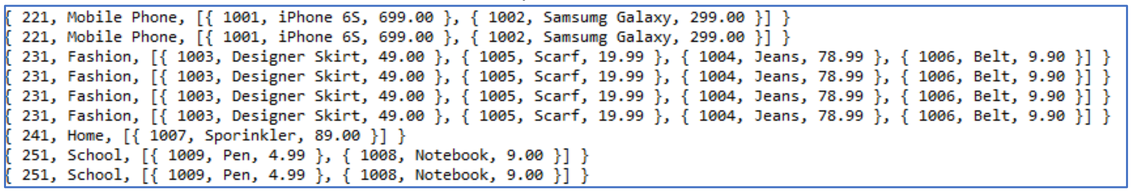
Let's run this code and you will find that for every product associated with the category, a category object has been returned.
Here is the select statement, and notice the inner join is performed under the hood.



```
Hibernate:
    select
        category0_.id as id1_0_,
        category0_.name as name2_0_
    from
        categories category0_
    inner join
        products products1_
            on category0_.id=products1_.category_id
```

The categories table is joined with the products table based on the category id associated with each product.

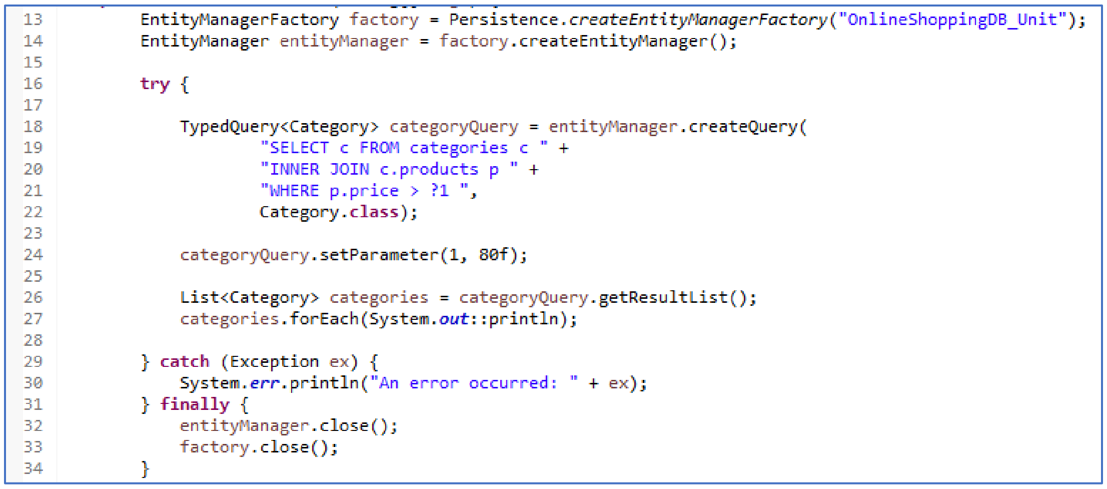
Now let's take a look at the results that are printed out.



```
{ 221, Mobile Phone, [{ 1001, iPhone 6S, 699.00 }, { 1002, Samsumg Galaxy, 299.00 }] }
{ 221, Mobile Phone, [{ 1001, iPhone 6S, 699.00 }, { 1002, Samsumg Galaxy, 299.00 }] }
{ 231, Fashion, [{ 1003, Designer Skirt, 49.00 }, { 1005, Scarf, 19.99 }, { 1004, Jeans, 78.99 }, { 1006, Belt, 9.90 }] }
{ 231, Fashion, [{ 1003, Designer Skirt, 49.00 }, { 1005, Scarf, 19.99 }, { 1004, Jeans, 78.99 }, { 1006, Belt, 9.90 }] }
{ 231, Fashion, [{ 1003, Designer Skirt, 49.00 }, { 1005, Scarf, 19.99 }, { 1004, Jeans, 78.99 }, { 1006, Belt, 9.90 }] }
{ 231, Fashion, [{ 1003, Designer Skirt, 49.00 }, { 1005, Scarf, 19.99 }, { 1004, Jeans, 78.99 }, { 1006, Belt, 9.90 }] }
{ 241, Home, [{ 1007, Sporinkler, 89.00 }] }
{ 251, School, [{ 1009, Pen, 4.99 }, { 1008, Notebook, 9.00 }] }
{ 251, School, [{ 1009, Pen, 4.99 }, { 1008, Notebook, 9.00 }] }
```

- Here is the Mobile Phones category with two phones, that is two products associated with this category.  Because there are two products in this Mobile Phones category. We have two category objects along with the corresponding products that have been returned. That's because of the structure of our category entity.
- If you scroll down, you will see the Fashion category, there are four products associated with the Fashion category. The Fashion category entity has been returned four times and printed out four times to screen. We've got a category entity for each product associated with that category.

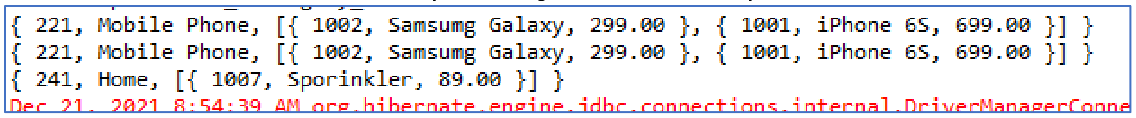
We may be interested in applying an additional filter to this inner join operation and that's exactly what we'll see now.



```java
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {

            TypedQuery<Category> categoryQuery = entityManager.createQuery(
                    "SELECT c FROM categories c " +
                    "INNER JOIN c.products p " +
                    "WHERE p.price > ?1 ",
                    Category.class);

            categoryQuery.setParameter(1, 80f);

            List<Category> categories = categoryQuery.getResultList();
            categories.forEach(System.out::println);

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

- We `SELECT c FROM categories c`, we perform an inner join with `c.products`, which we reference using the `p` variable.
- And we add an additional where clause, `p.price` should be greater than what we have specified.
- Any category which has some product that is greater than the price that we have specified, that is $80, will be returned. That category will be returned in our join operation.

Let's run this code and take a look by scrolling down to the very bottom.

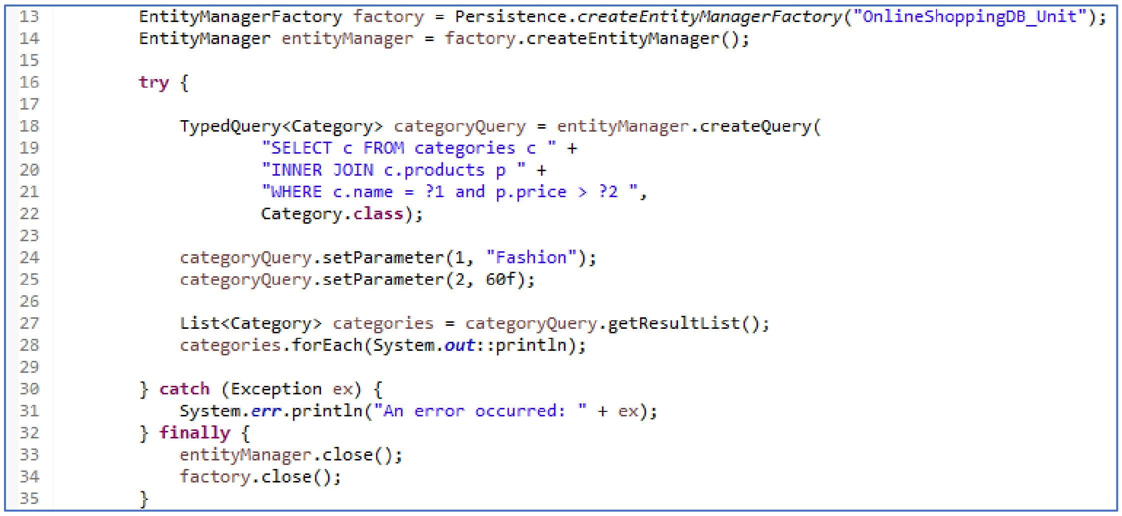


```
{ 221, Mobile Phone, [{ 1002, Samsumg Galaxy, 299.00 }, { 1001, iPhone 6S, 699.00 }] }
{ 221, Mobile Phone, [{ 1002, Samsumg Galaxy, 299.00 }, { 1001, iPhone 6S, 699.00 }] }
{ 241, Home, [{ 1007, Sporinkler, 89.00 }] }
Dec 21, 2021 8:54:39 AM org.hibernate.engine.jdbc.connections.internal.DriverManagerConne
```

- There are just two categories that match the criteria that we have specified.
  The first of these is the Mobile Phones category. Two category entities have been returned, one corresponding to each mobile phone product.
  The second category that matches is the Home category.
- Here is a sprinkler for $89.
- It satisfies our price constraint. The other categories did not satisfy our price constraint, they haven't been returned.

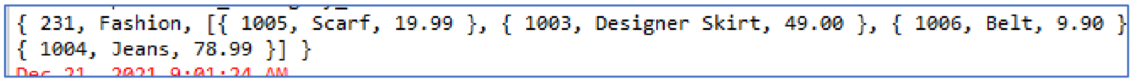
## Inner Join with multiple filter

In a join operation, it's possible for you to filter based on multiple constraints. Here is an inner join.



```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {

            TypedQuery<Category> categoryQuery = entityManager.createQuery(
                    "SELECT c FROM categories c " +
                    "INNER JOIN c.products p " +
                    "WHERE c.name = ?1 and p.price > ?2 ",
                    Category.class);

            categoryQuery.setParameter(1, "Fashion");
            categoryQuery.setParameter(2, 60f);

            List<Category> categories = categoryQuery.getResultList();
            categories.forEach(System.out::println);

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

- We `SELECT c FROM categories c`, we perform an inner join with `c.products` reference using variable `p`.
- We want `c.name` to be a certain value, and we want `p.price` to be greater than a certain value. Both of these are positional parameters. The category name at position 1 I've specified as `"Fashion"`. The product price at position 2 I've specified as $60.



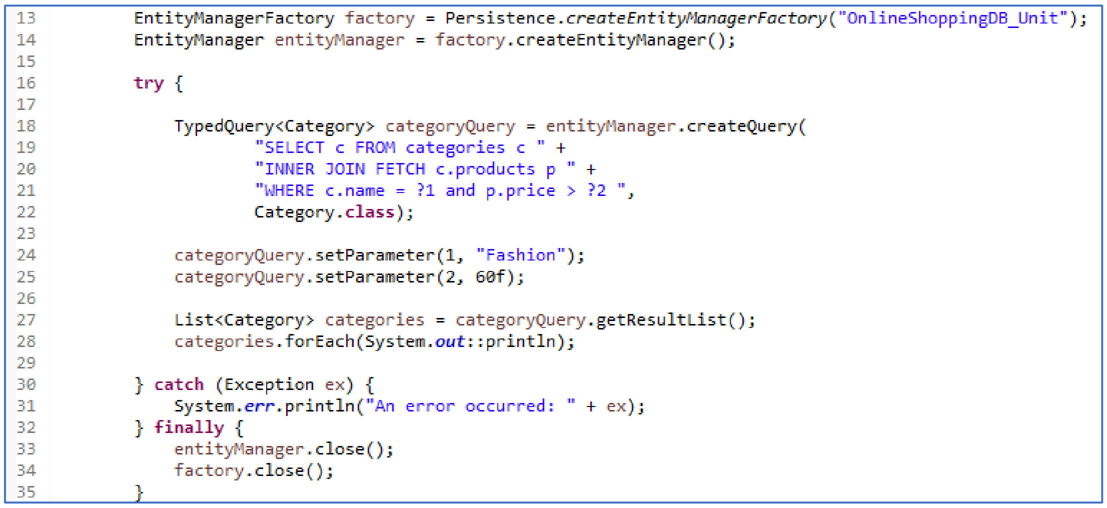
```
{ 231, Fashion, [{ 1005, Scarf, 19.99 }, { 1003, Designer Skirt, 49.00 }, { 1006, Belt, 9.90 }
{ 1004, Jeans, 78.99 }] }
Dec 21, 2021 9:01:24 AM
```

There is exactly one category that matches both of our constraints. The category with name “Fashion”, and you can see that there is at least one product here greater than *$60, Jeans*.

## Fetch Join

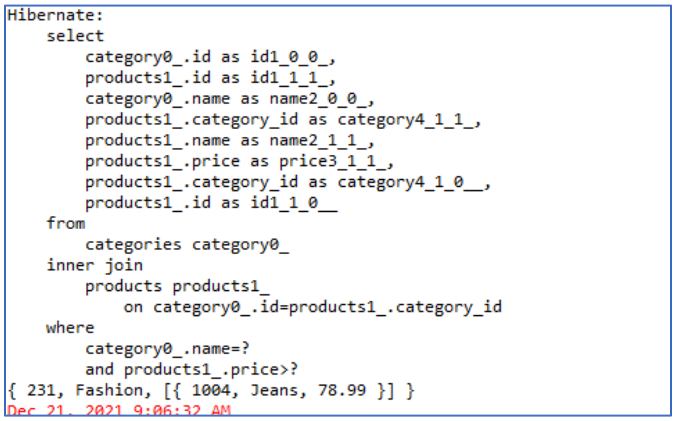
When you're performing join operations using JPQL queries, it's more performant to use the **fetch join** rather than plain joins. Whether they are inner joins, outer joins, and so on.

The **fetch join** allows you to fetch related entities in a single query under the hood. Instead of using additional queries for each access of the object's lazy relationships, this implementation is abstracted away from you. You should know though that a **fetch join is more performant and** the way you perform a fetch join is by using the fetch keyword.



```java
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {

            TypedQuery<Category> categoryQuery = entityManager.createQuery(
                    "SELECT c FROM categories c " +
                    "INNER JOIN FETCH c.products p " +
                    "WHERE c.name = ?1 and p.price > ?2 ",
                    Category.class);

            categoryQuery.setParameter(1, "Fashion");
            categoryQuery.setParameter(2, 60f);

            List<Category> categories = categoryQuery.getResultList();
            categories.forEach(System.out::println);

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

Notice my inner join here, `SELECT c FROM categories c`, `INNER JOIN FETCH c.products p`. I've used parameters to specify the category name and the product price that I'm interested in.

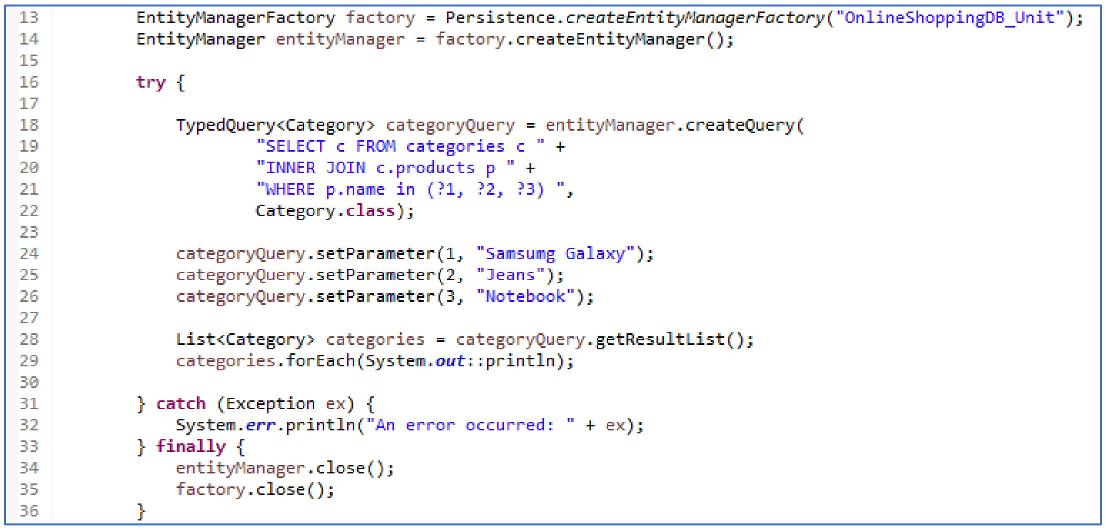


```
Hibernate:
    select
        category0_.id as id1_0_0_,
        products1_.id as id1_1_1_,
        category0_.name as name2_0_0_,
        products1_.category_id as category4_1_1_,
        products1_.name as name2_1_1_,
        products1_.price as price3_1_1_,
        products1_.category_id as category4_1_0__,
        products1_.id as id1_1_0__
    from
        categories category0_
    inner join
        products products1_
            on category0_.id=products1_.category_id
    where
        category0_.name=?
        and products1_.price>?
{ 231, Fashion, [{ 1004, Jeans, 78.99 }] }
Dec 21, 2021 9:06:32 AM
```

The **fetch** keyword makes this a **fetch join**. And you'll find that a single query gives us our result. It's much more performant overall. Here is the single query performing the fetch join. And if we scroll down further, you'll find the final result.

## Join with multiple positional parameters

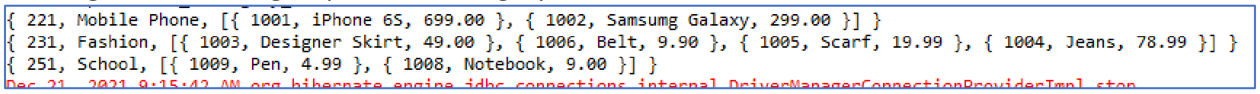
Let's complete joins in JPQL by performing one last join operation with multiple positional parameters.



```java
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {

            TypedQuery<Category> categoryQuery = entityManager.createQuery(
                    "SELECT c FROM categories c " +
                    "INNER JOIN c.products p " +
                    "WHERE p.name in (?1, ?2, ?3) ",
                    Category.class);

            categoryQuery.setParameter(1, "Samsumg Galaxy");
            categoryQuery.setParameter(2, "Jeans");
            categoryQuery.setParameter(3, "Notebook");

            List<Category> categories = categoryQuery.getResultList();
            categories.forEach(System.out::println);

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

`SELECT c FROM categories c`, `inner join c.products p WHERE p.name` is in one of these three options, the “Samsung Galaxy”, “Jeans” or “Notebook”.

Running this code will give you three category entities that match our criteria.



```
{ 221, Mobile Phone, [{ 1001, iPhone 6S, 699.00 }, { 1002, Samsumg Galaxy, 299.00 }] }
{ 231, Fashion, [{ 1003, Designer Skirt, 49.00 }, { 1006, Belt, 9.90 }, { 1005, Scarf, 19.99 }, { 1004, Jeans, 78.99 }] }
{ 251, School, [{ 1009, Pen, 4.99 }, { 1008, Notebook, 9.00 }] }
Dec 21, 2021 9:15:42 AM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl stop
```

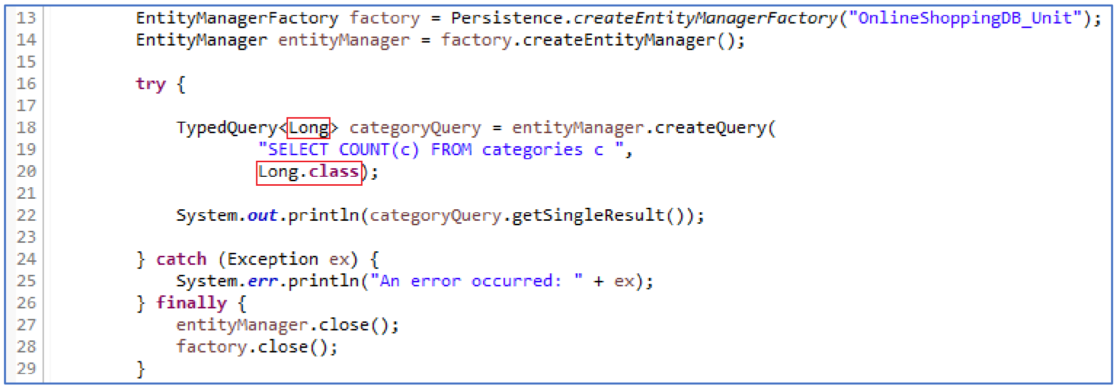
Here is the “Mobile Phones” category that has the “Samsung Galaxy”.
Here's the “Fashion” category that has the “Jeans” product.
And here is the “School” category that has the “Notebook” product.

## Selecting Multiple Fields Including Aggregates

Let's continue exploring queries that you can run using JPQL. In this example, we'll see how we can run aggregate queries.

### Count Aggregation

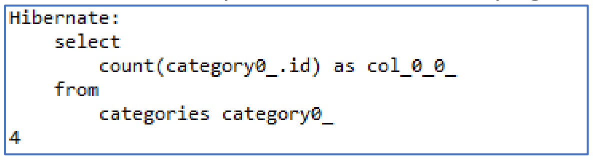
I want to select the count of categories from my categories table, `SELECT COUNT(c)`.



```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {

            TypedQuery<Long> categoryQuery = entityManager.createQuery(
                    "SELECT COUNT(c) FROM categories c ",
                    Long.class);

            System.out.println(categoryQuery.getSingleResult());

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

- Now, notice that the field that I'm extracting here is of type **Long**. And that is the class of the result that I specify for my typed query `Long.class`.
- And because we know that our aggregate query returns a single result, we can invoke `categoryQuery.getSingleResult()`to run this query.

Run this code and you can see the underlying SQL query run by Hibernate,

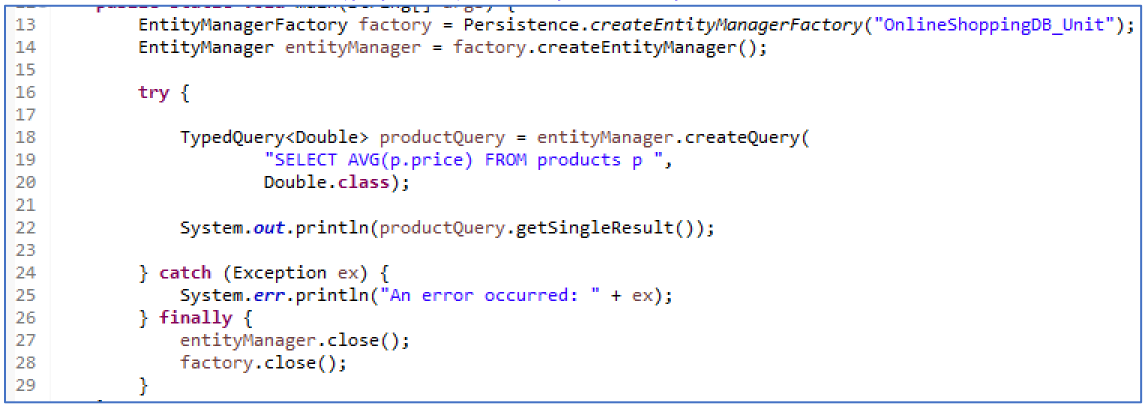


```
Hibernate:
    select
        count(category0_.id) as col_0_0_
    from
        categories category0_
4
```

select count category ID from Categories. And the total count of categories present in our underlying table is equal to 4, and that's printed out to screen.

## Average Aggregation

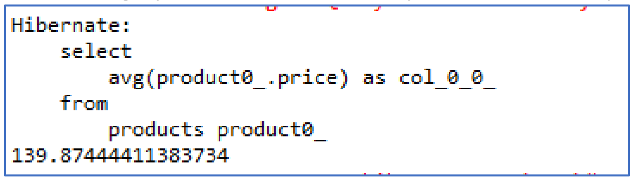
Let's try one more aggregate operation. This time I want to select the average price of products from the Products table. `SELECT AVG(p.price) FROM producs p`.



```java
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {

            TypedQuery<Double> productQuery = entityManager.createQuery(
                    "SELECT AVG(p.price) FROM products p ",
                    Double.class);

            System.out.println(productQuery.getSingleResult());

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

- Our select statement here is for a single aggregate field, and we have to specify the class that our result will return. `Double.class` is what we have specified to this typed query.
- There's just a single result returned. We use `getSingleResult()` to print out the average price.

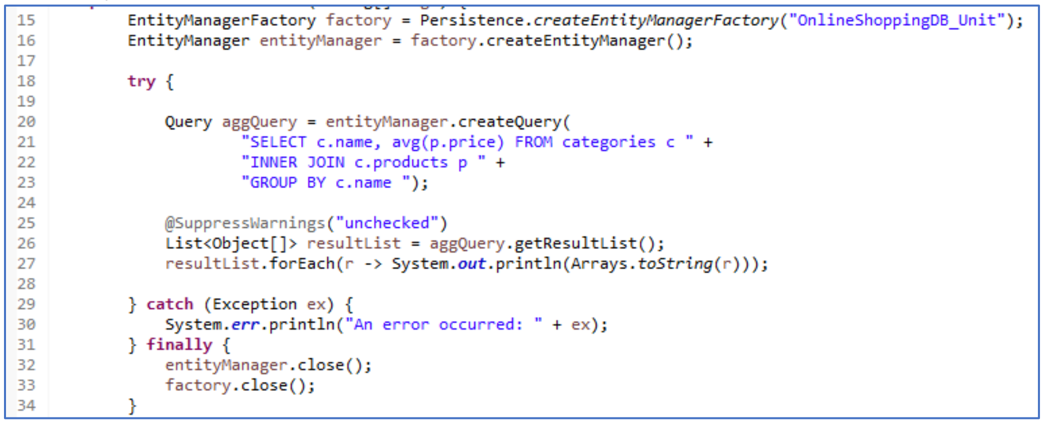
The average price from Products, you can see the query, and the result returned here is $139.87.



```
Hibernate:
    select
        avg(product0_.price) as col_0_0_
    from
        products product0_
139.87444411383734
```

## Multiple Aggregation

So far we've seen how JPQL queries can return entity objects and single field values. But what if there are multiple fields that you want to return from your query? How do you express that in code? Let's see an example of this.



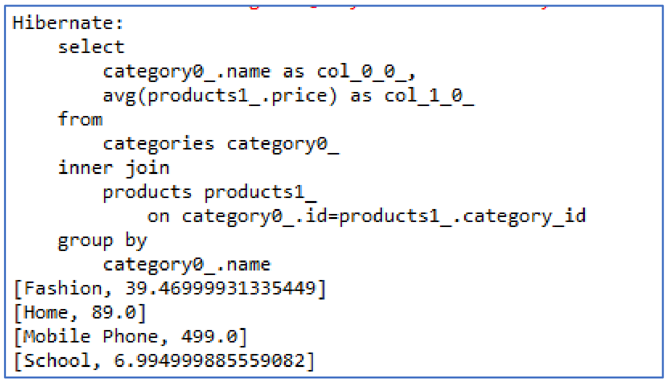
```
15          EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
16          EntityManager entityManager = factory.createEntityManager();
17  
18          try {
19  
20              Query aggQuery = entityManager.createQuery(
21                      "SELECT c.name, avg(p.price) FROM categories c " +
22                      "INNER JOIN c.products p " +
23                      "GROUP BY c.name ");
24  
25              @SuppressWarnings("unchecked")
26              List<Object[]> resultList = aggQuery.getResultList();
27              resultList.forEach(r -> System.out.println(Arrays.toString(r)));
28  
29          } catch (Exception ex) {
30              System.err.println("An error occurred: " + ex);
31          } finally {
32              entityManager.close();
33              factory.close();
34          }
```

- Along with selecting multiple fields, we'll also perform a `GROUP BY` operation using a JPQL query.
- Notice my select statement, `SELECT c.name`, that is the name of the category.
- And the average price of products, `AVG(p.price)` from Category `c`.
- You perform an `INNER JOIN` with `c.products p`,
- and we `GROUP BY` category name.

For every category, we want to extract the average price of products in that category. We have two fields that we have selected here, `c.name` and `AVG(p.price)`.

- By default, JPA Hibernate will return a list of object arrays where every object array in this list corresponds to the two fields that we have selected. Notice the return type on line 26.
- Because we're working with raw objects directly, we need to add in that **@SuppressWarnings** annotation.
- Now, for every object array, I'm going to run a `forEach` operation and print out the contents of that array using `Arrays.toString()`.
- The first element in the object array will be the category name, the second element, the average price of products in that category.

Let's run this code and take a look at the results.



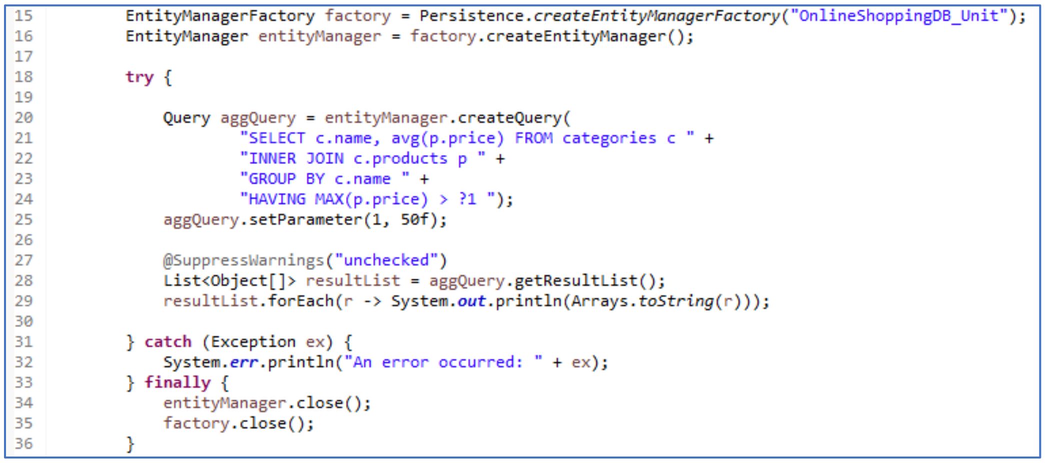
```
Hibernate: 
    select
        category0_.name as col_0_0_,
        avg(products1_.price) as col_1_0_ 
    from
        categories category0_ 
    inner join
        products products1_ 
            on category0_.id=products1_.category_id 
    group by
        category0_.name
[Fashion, 39.46999931335449]
[Home, 89.0]
[Mobile Phone, 499.0]
[School, 6.994999885559082]
```

And here are the object arrays returned by the underlying database.

“Mobile Phones” is the first category, average price, $499,
then “Fashion”, average price about $39.46.
“Home”, $89,
“School”, $6.99.

## Multiple Aggregation using Having Clause

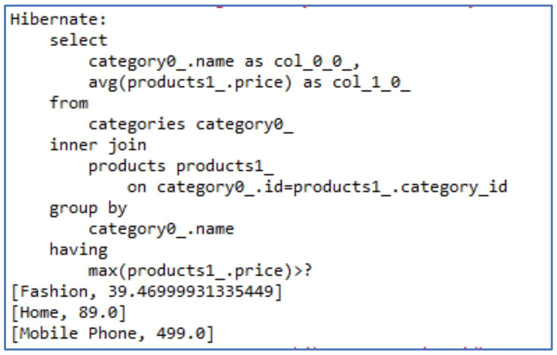
Let's try this once again. We'll perform an aggregation using the group by clause. We'll return multiple field values which will be returned as an array of objects. In addition, we'll specify an additional constraint using the having clause, along with our group by.



```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {

            Query aggQuery = entityManager.createQuery(
                    "SELECT c.name, avg(p.price) FROM categories c " +
                    "INNER JOIN c.products p " +
                    "GROUP BY c.name " +
                    "HAVING MAX(p.price) > ?1 ");
            aggQuery.setParameter(1, 50f);

            @SuppressWarnings("unchecked")
            List<Object[]> resultList = aggQuery.getResultList();
            resultList.forEach(r -> System.out.println(Arrays.toString(r)));

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

- The two fields that we want to return in our query is the name of the category and the maximum price of products that belong to that category. So `SELECT c.name, AVG(p.price) FROM categories c`.
- In order to do this, we need to perform an `INNER JOIN` with `c.product p` and group by the category name, `GROUP BY c.name`.
- And we also want to ensure that the maximum price of any product in that category is greater than a certain value. That is the additional filtering that we perform on our grouped result using the having clause. And the max price should be greater than $50.

Let's run this code,



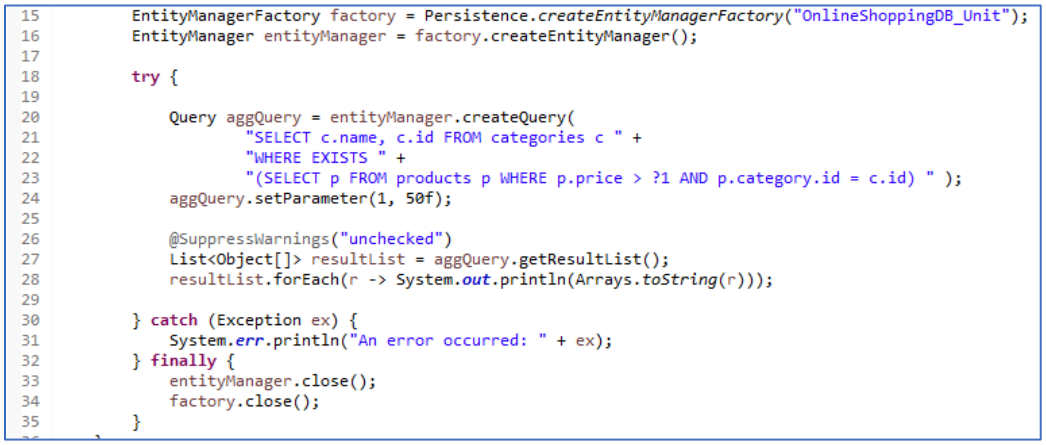
```
Hibernate: 
    select
        category0_.name as col_0_0_,
        avg(products1_.price) as col_1_0_ 
    from
        categories category0_ 
    inner join
        products products1_ 
            on category0_.id=products1_.category_id 
    group by
        category0_.name 
    having
        max(products1_.price)>?
[Fashion, 39.46999931335449]
[Home, 89.0]
[Mobile Phone, 499.0]
```

and You'll see that there are just three categories that satisfy this additional condition.

“Mobile Phones”, “Fashion”, and “Home” have products that have a maximum price greater than $50.

## Using Exists Clause and Sub Query

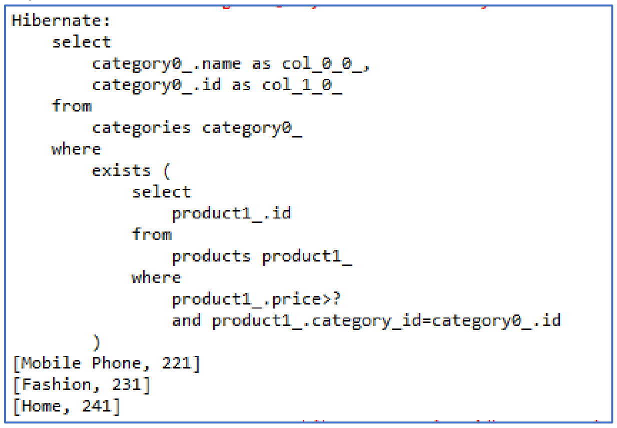
Let's look at another example of a JPQL query. This time, one that uses the exists clause and a subquery.



```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {

            Query aggQuery = entityManager.createQuery(
                    "SELECT c.name, c.id FROM categories c " +
                    "WHERE EXISTS " +
                    "(SELECT p FROM products p WHERE p.price > ?1 AND p.category.id = c.id) " );
            aggQuery.setParameter(1, 50f);

            @SuppressWarnings("unchecked")
            List<Object[]> resultList = aggQuery.getResultList();
            resultList.forEach(r -> System.out.println(Arrays.toString(r)));

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

- So I'm also selecting multiple fields here, but we are already familiar with that. We `SELECT` category name and category ID `FROM categories c WHERE EXISTS`, and here is our subquery, `SELECT p FROM products p WHERE p.price` is greater than what we have specified.
- And the category ID is equal to the ID of the category entity that we select.
- The price parameter I've specified using positional arguments is $50.
- So I want the name and ID of categories where exists at least one product that is greater than $50.

If you run this code,

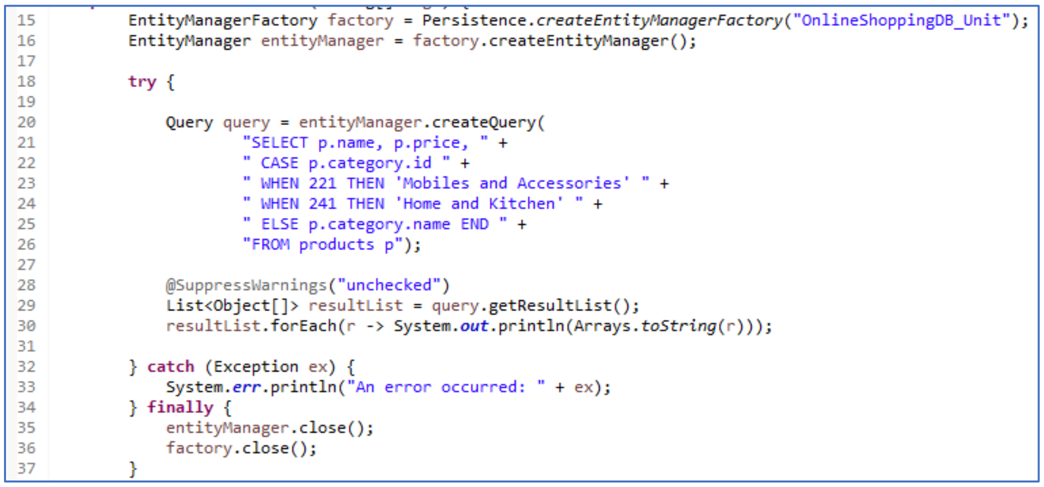


```
Hibernate:
    select
        category0_.name as col_0_0_,
        category0_.id as col_1_0_
    from
        categories category0_
    where
        exists (
            select
                product1_.id
            from
                products product1_
            where
                product1_.price>?
                and product1_.category_id=category0_.id
        )
[Mobile Phone, 221]
[Fashion, 231]
[Home, 241]
```

You'll see that there are exactly three categories that meet our criteria, "Mobile Phones", "Fashion", and "Home".
These are the only three categories that have at least one product greater than $50.

## Queries with Case and Statement

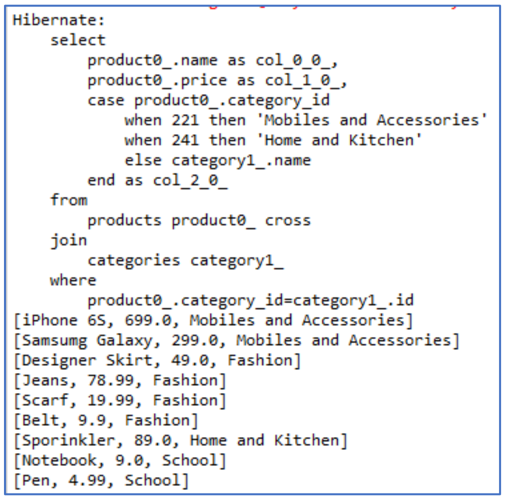
If you want to be able to run complex queries that involve case end statements, that's possible in JPQL as well.



```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {

            Query query = entityManager.createQuery(
                    "SELECT p.name, p.price, " +
                    " CASE p.category.id " +
                    " WHEN 221 THEN 'Mobiles and Accessories' " +
                    " WHEN 241 THEN 'Home and Kitchen' " +
                    " ELSE p.category.name END " +
                    "FROM products p");

            @SuppressWarnings("unchecked")
            List<Object[]> resultList = query.getResultList();
            resultList.forEach(r -> System.out.println(Arrays.toString(r)));

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

- Notice this query with a case end statement.
- I'm going to `SELECT p.name` and `p.price`. That's what I'm going to display.
- The third field I want to select and display is specified using the `CASE END` statement. I've started the case here, and the case is on the `p.category.id` column.
  My case condition is based on the category that a particular product belongs to.
  If the category is 221, then I want to display `"Mobiles And Accessories"`,
  if it's 241 then `"Home and Kitchen"`.
  Otherwise, I'll continue using the name of the category as specified in the database, else `p.category.name`.
- And I perform this case end selection from the products table.
- This select query returns three fields, the product name, the product price, and whatever is returned based on the case end statement.
- This is a list of object arrays, and we print out the object arrays to screen.

Run this code and you'll get three bits of information for every product in our Products table.



```
Hibernate: 
    select
        product0_.name as col_0_0_,
        product0_.price as col_1_0_,
        case product0_.category_id 
            when 221 then 'Mobiles and Accessories' 
            when 241 then 'Home and Kitchen' 
            else category1_.name 
        end as col_2_0_ 
    from
        products product0_ cross 
    join
        categories category1_ 
    where
        product0_.category_id=category1_.id
[iPhone 6S, 699.0, Mobiles and Accessories]
[Samsumg Galaxy, 299.0, Mobiles and Accessories]
[Designer Skirt, 49.0, Fashion]
[Jeans, 78.99, Fashion]
[Scarf, 19.99, Fashion]
[Belt, 9.9, Fashion]
[Sporinkler, 89.0, Home and Kitchen]
[Notebook, 9.0, School]
[Pen, 4.99, School]
```

We have the name of the product, the price of the product.

And in the case of category ID 221, the category ID is displayed as “Mobiles and Accessories”.

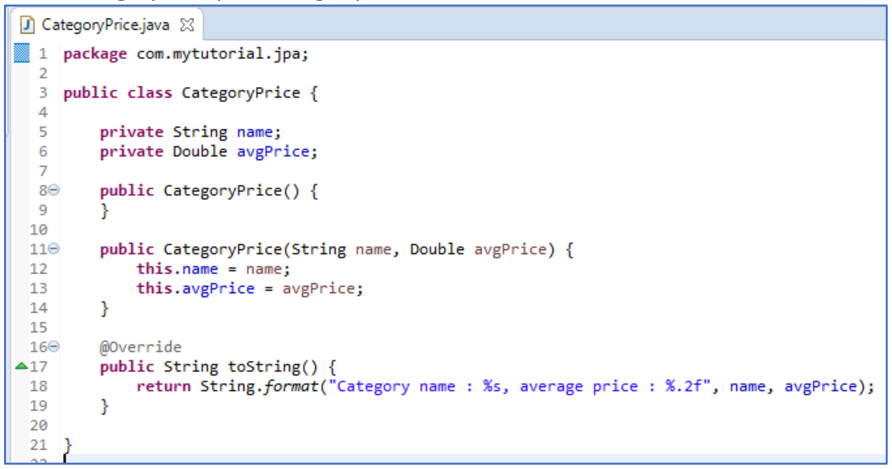
For category ID 241, we display “Home and Kitchen”.

Otherwise, we stick to the category name in the case of “Fashion” and “School”.

## Constructing Objects in Queries

Now, it's quite common for you to run queries which return multiple fields rather than entire entities. And it's kind of annoying to be dealing with those fields as object arrays. You can fix this by creating a new class to represent the fields that you might work with.

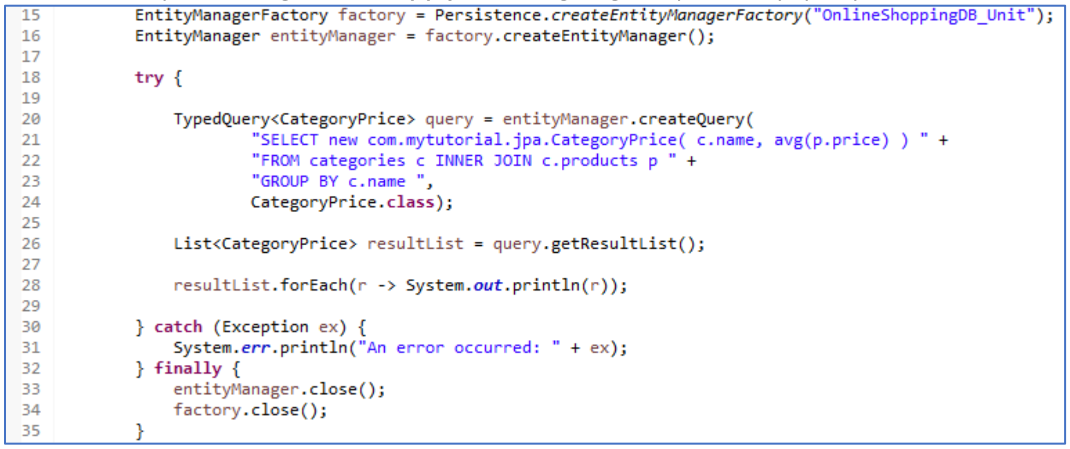
Here is a class that I've created called **`CategoryPrice`**. This is what is going to hold the categories and the average price per category.


CategoryPrice.java

```java
package com.mytutorial.jpa;

public class CategoryPrice {

    private String name;
    private Double avgPrice;

    public CategoryPrice() {
    }

    public CategoryPrice(String name, Double avgPrice) {
        this.name = name;
        this.avgPrice = avgPrice;
    }

    @Override
    public String toString() {
        return String.format("Category name : %s, average price : %.2f", name, avgPrice);
    }

}
```

- So I set up member variables accordingly.
- We have the name member variable that is the `name` of the category, and the average `price` that will hold the average product `price` for each category.
- I have constructors and I have a **`toString()`** representation. The **`toString()`** simply prints out the category name and the average product price for that category.

This is the object that I'm going to use to hold the fields that I query using JPQL queries. Let's see how this is done by switching over to **App.java**. I'm going to update my query here.



```java
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {

            TypedQuery<CategoryPrice> query = entityManager.createQuery(
                    "SELECT new com.mytutorial.jpa.CategoryPrice( c.name, avg(p.price) ) " +
                    "FROM categories c INNER JOIN c.products p " +
                    "GROUP BY c.name ",
                    CategoryPrice.class);

            List<CategoryPrice> resultList = query.getResultList();

            resultList.forEach(r -> System.out.println(r));

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

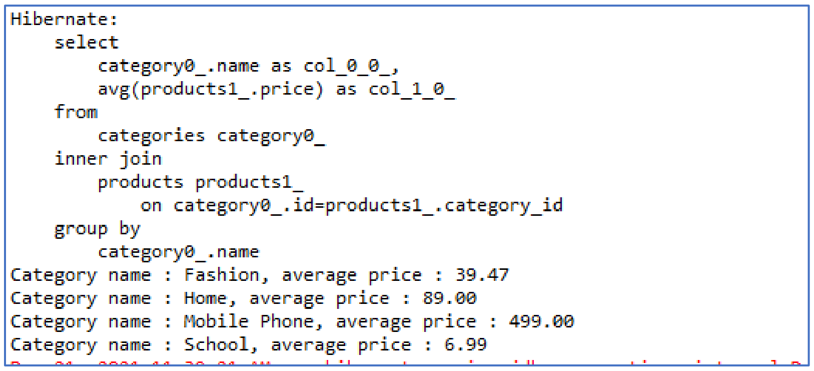
- And this will be an aggregate query that will query two fields. This query will get the `name` of the category and the `average` product `price` per category.
- Notice how I use `new com.mytutorial.jpa.CategoryPrice()` to instantiate a new object containing these bits of information right within the query.

```
SELECT new com.mytutorial.jpa.CategoryPrice( c.name, avg(p.price) )
FROM categories c INNER JOIN c.products p
GROUP BY c.name
```

Now this would definitely not have been possible had you been running regular SQL. Notice you're mixing up Java syntax along with your queries, and this is what JPQL offers. Note that the object that we have constructed within a query is not an entity. It's just another object that holds the information that we have retrieved.

- Notice that the TypedQuery also accepts the datatype of the object returned `CategoryPrice.class`.
- The rest of the query is familiar to us `FROM categories c INNER JOIN c.products` we group by category `FROM`, and calculate the average product `price`. Except that both of the fields, instead of being returned as a list of object, arrays will now be returned as a list of `CategoryPrice` objects.
- Notice the return value data type on line 26, a `List` of `CategoryPrice` objects.
- I'll run a forEach through every object in this list and print out the list.

Let's run this code.



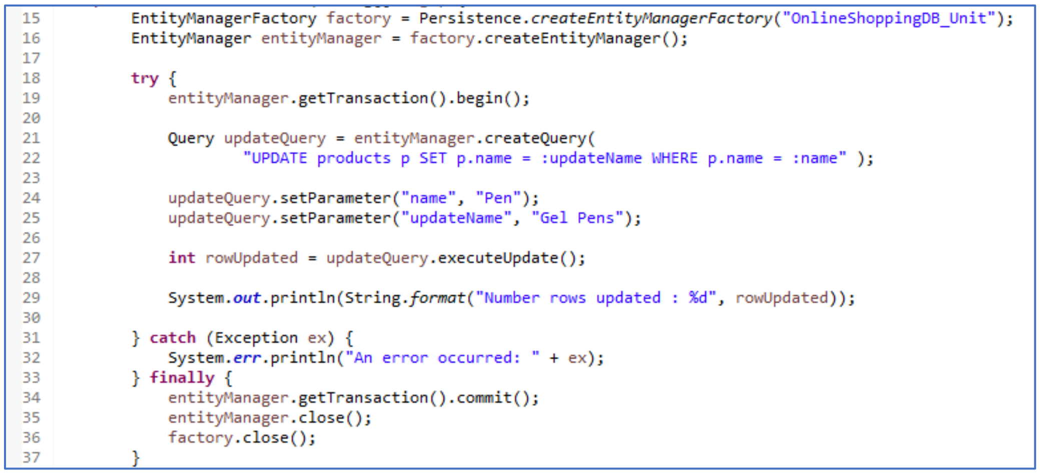
```
Hibernate:
    select
        category0_.name as col_0_0_,
        avg(products1_.price) as col_1_0_
    from
        categories category0_
    inner join
        products products1_
            on category0_.id=products1_.category_id
    group by
        category0_.name
Category name : Fashion, average price : 39.47
Category name : Home, average price : 89.00
Category name : Mobile Phone, average price : 499.00
Category name : School, average price : 6.99
```

And notice the **`toString()`** of the category price objects have been invoked to print out the category name, and the average product price per category.

## Executing Update and Delete Queries

So far we've seen a number of select queries that retrieves data from our database. Let's look at some update and delete queries as well, to round out our knowledge of JPQL queries.

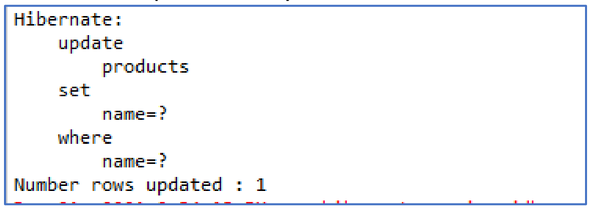
### Update Operation



```
15          EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
16          EntityManager entityManager = factory.createEntityManager();
17
18          try {
19              entityManager.getTransaction().begin();
20
21              Query updateQuery = entityManager.createQuery(
22                      "UPDATE products p SET p.name = :updateName WHERE p.name = :name" );
23
24              updateQuery.setParameter("name", "Pen");
25              updateQuery.setParameter("updateName", "Gel Pens");
26
27              int rowUpdated = updateQuery.executeUpdate();
28
29              System.out.println(String.format("Number rows updated : %d", rowUpdated));
30
31          } catch (Exception ex) {
32              System.err.println("An error occurred: " + ex);
33          } finally {
34              entityManager.getTransaction().commit();
35              entityManager.close();
36              factory.close();
37          }
```

- Here is how we perform an update query. The query that I have specified here updates the product's entities and sets the product name to be “Gel Pens”, where the product name was something else previously.
- I have specified what the product name was previously using a named parameter.
- Previously if the product name was Pen, I want it to be renamed to Gel Pens.
- Just like with select queries, you can have parameterize update queries, and you'll set your parameters using the `setParameter()` method.
- Updates are executed using the `executeUpdate()` method which I invoke on line 27, `updateQuery.executeUpdate()`. This will return the number of rows that were updated, and I'll print that out to screen.

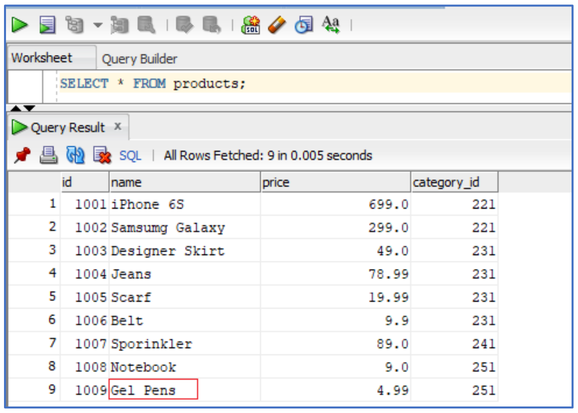
Let's run this code, see whether the update is performed and see how many rows have been updated. Here is the updateQuery that Hibernate has run under the hood.



```
Hibernate:
    update
        products
    set
        name=?
    where
        name=?
Number rows updated : 1
```

Number of rows updated is exactly one, there was exactly one product which had the name Pens, we've updated that to Gel Pens.

Let's confirm this by switching over to the MySQL Workbench, and run a `select *` on the `products` table.

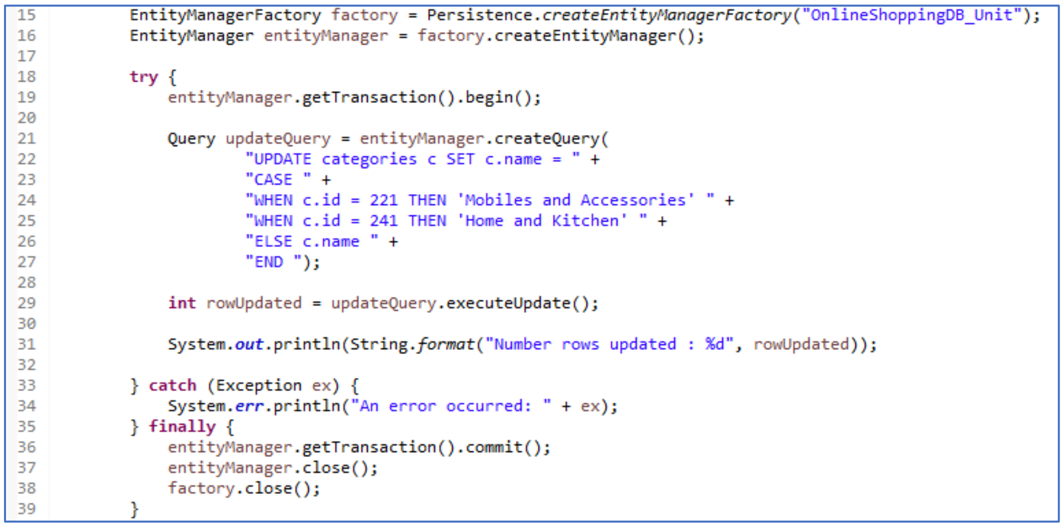


```
SELECT * FROM products;
```

| | id | name | price | category_id |
|---|---|---|---|---|
| 1 | 1001 | iPhone 6S | 699.0 | 221 |
| 2 | 1002 | Samsumg Galaxy | 299.0 | 221 |
| 3 | 1003 | Designer Skirt | 49.0 | 231 |
| 4 | 1004 | Jeans | 78.99 | 231 |
| 5 | 1005 | Scarf | 19.99 | 231 |
| 6 | 1006 | Belt | 9.9 | 231 |
| 7 | 1007 | Sporinkler | 89.0 | 241 |
| 8 | 1008 | Notebook | 9.0 | 251 |
| 9 | 1009 | Gel Pens | 4.99 | 251 |

Notice at the very bottom with id 1009, we have a product named “Gel Pens” that was previously named “Pens”.

You can also update your database using more complex queries, queries that include the case and statement.

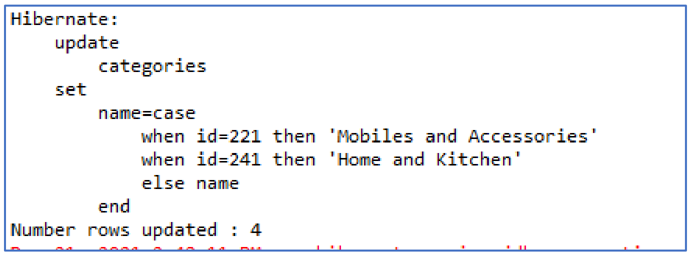


```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {
            entityManager.getTransaction().begin();

            Query updateQuery = entityManager.createQuery(
                    "UPDATE categories c SET c.name = " +
                    "CASE " +
                    "WHEN c.id = 221 THEN 'Mobiles and Accessories' " +
                    "WHEN c.id = 241 THEN 'Home and Kitchen' " +
                    "ELSE c.name " +
                    "END ");

            int rowUpdated = updateQuery.executeUpdate();

            System.out.println(String.format("Number rows updated : %d", rowUpdated));

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.getTransaction().commit();
            entityManager.close();
            factory.close();
        }
```

- Here is an update query, where I update the name of the category based on certain conditions that I've specified using the case and the statement.
- Update Categories c set c.name equal to, and then I begin my case statement to evaluate multiple conditions.
- Case when c.id is 221, then I'll update the category name to be Mobiles and Accessories.
- When c.id is 241, then I'll update the category name to be Home and Kitchen.
- In all other cases, I'll keep the original category name, c.name.

This will update every category in my categories table, either keeping the original name or updating the name based on the case called `updateQuery.executeUpdate()`, and print out the number of rows updated.

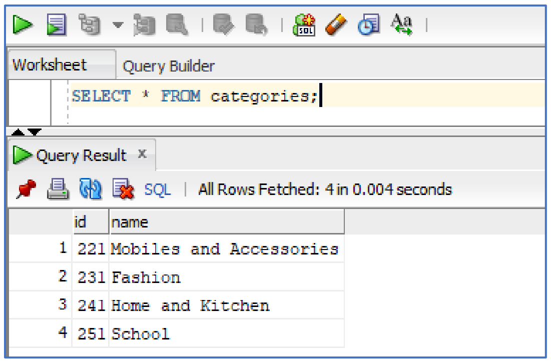
When you run this code,



```
Hibernate: 
    update
        categories 
    set
        name=case 
            when id=221 then 'Mobiles and Accessories' 
            when id=241 then 'Home and Kitchen' 
            else name 
        end
Number rows updated : 4
```

- Here is the SQL query that Hibernate actually executes it updates the Categories table,
- and then sets the name based on the key statement,
- the number of rows updated equal to four.

We'll head over to MySQL Workbench to see whether these rows have actually been updated, run a *select * from* the *categories* table.



```
SELECT * FROM categories;
```

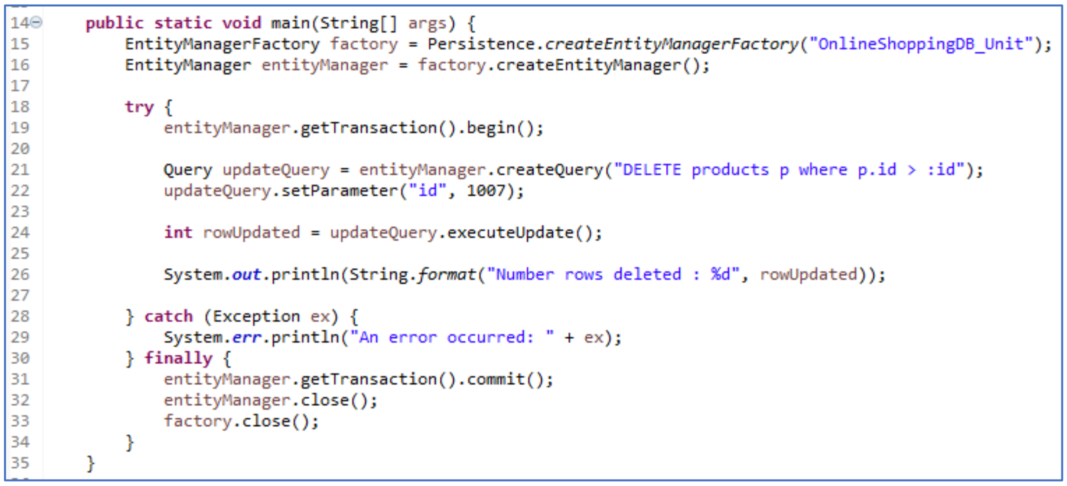
All Rows Fetched: 4 in 0.004 seconds

```
   id  name
1  221 Mobiles and Accessories
2  231 Fashion
3  241 Home and Kitchen
4  251 School
```

Notice that mobile phones is now called Mobiles and Accessories, home is called Home and Kitchen. The remaining two categories Fashion and School, haven't been updated, they retain their original names.

## Delete Operation

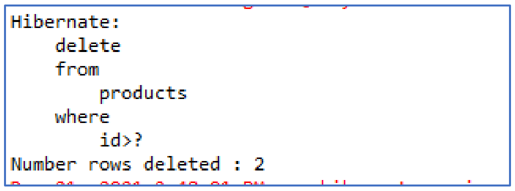
Just like you can update records in your table using JPQL queries, you can also delete records using JPQL queries.



```
14      public static void main(String[] args) {
15          EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
16          EntityManager entityManager = factory.createEntityManager();
17
18          try {
19              entityManager.getTransaction().begin();
20
21              Query updateQuery = entityManager.createQuery("DELETE products p where p.id > :id");
22              updateQuery.setParameter("id", 1007);
23
24              int rowUpdated = updateQuery.executeUpdate();
25
26              System.out.println(String.format("Number rows deleted : %d", rowUpdated));
27
28          } catch (Exception ex) {
29              System.err.println("An error occurred: " + ex);
30          } finally {
31              entityManager.getTransaction().commit();
32              entityManager.close();
33              factory.close();
34          }
35      }
```

- Here is a delete query, delete Products p where p.id is greater than the specified name parameter,and the parameter id that I've specified is 1007.
- In order for the delete operation to be executed you need to call `executeUpdate()`, which is what we do on line 24,
- and we'll print out the number of rows deleted.

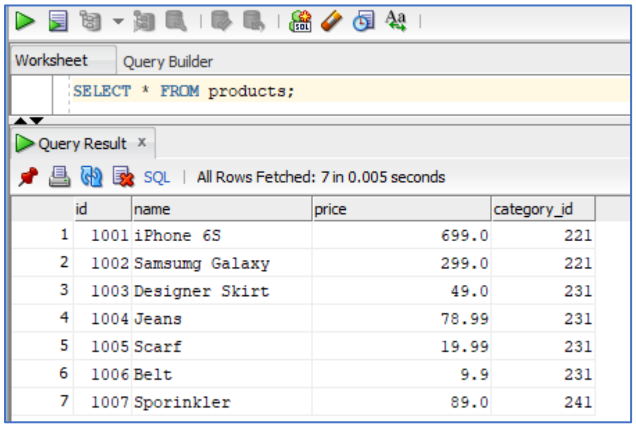
Go ahead and run this code,



```
Hibernate:
    delete
    from
        products
    where
        id>?
Number rows deleted : 2
```

here is the delete statement in SQL as run by Hibernate, number of rows deleted equal to two.

Check in MySQL, whether products with id 1008 and 1009 have been deleted.



```
SELECT * FROM products;
```

All Rows Fetched: 7 in 0.005 seconds

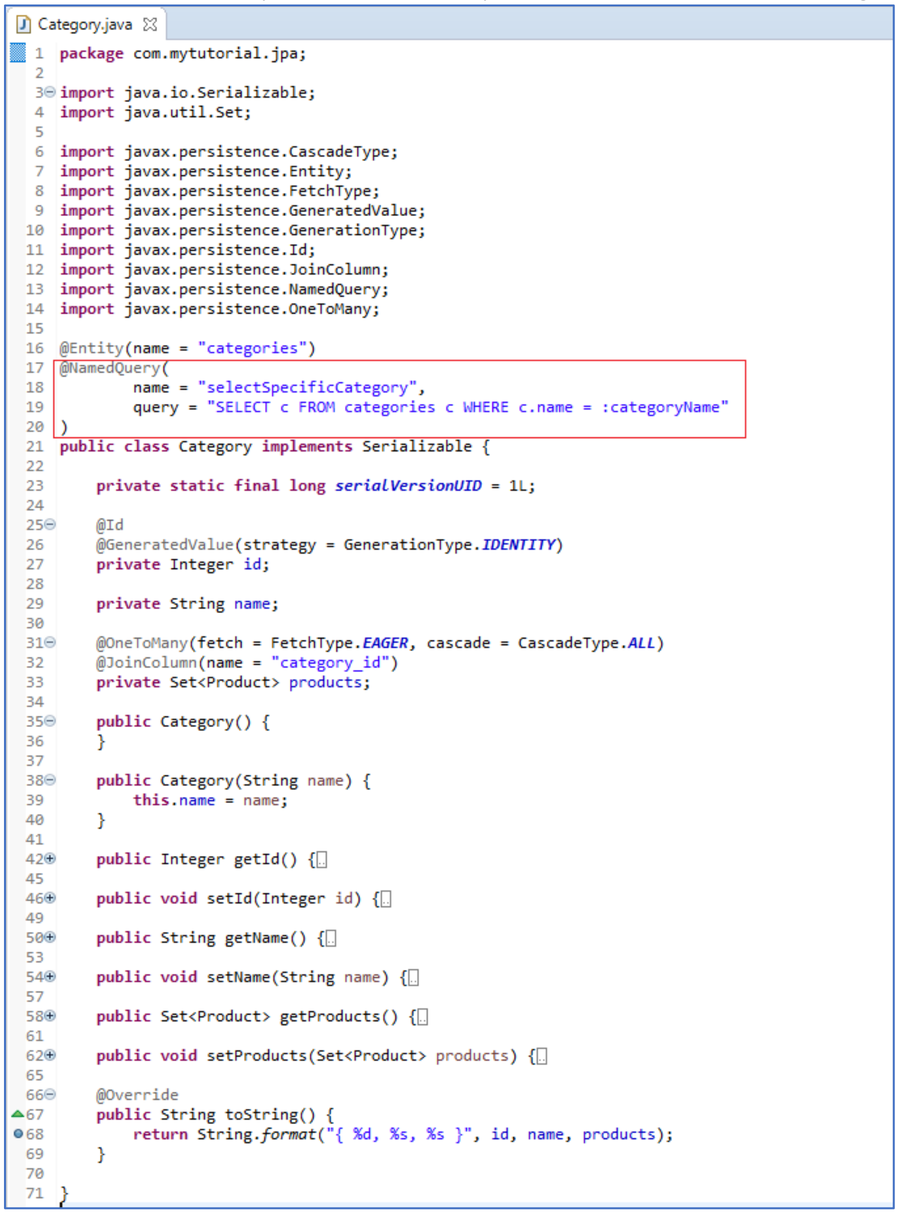
| | id | name | price | category_id |
|---|---|---|---|---|
| 1 | 1001 | iPhone 6S | 699.0 | 221 |
| 2 | 1002 | Samsumg Galaxy | 299.0 | 221 |
| 3 | 1003 | Designer Skirt | 49.0 | 231 |
| 4 | 1004 | Jeans | 78.99 | 231 |
| 5 | 1005 | Scarf | 19.99 | 231 |
| 6 | 1006 | Belt | 9.9 | 231 |
| 7 | 1007 | Sporinkler | 89.0 | 241 |

## Using Named Queries

A discussion of JPQL queries is not complete till we discuss named queries. Now typically in an application, there are certain queries that you run over and over again. Maybe you run these queries from different parts of the application.

It's worth your while then to predefine these queries. And that's what named queries are all about. And these are predefined queries associated with a specific entity. And once you define these queries, these queries are stored by name in the entity manager. So when you use the entity manager to create a query, you can specify the query by name. That's what makes it a named query.

Here I am in the `Category` class and I've imported on line 13, **`import`** `javax.persistence.NamedQuery` in order to be able to predefine named queries associated with the category entity.

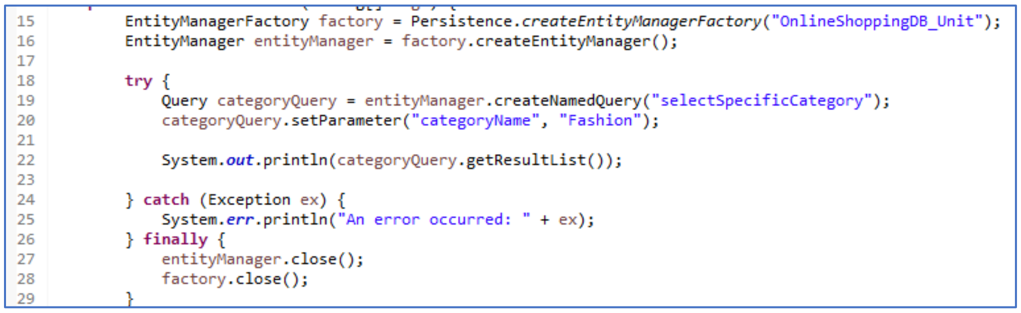


```
 1 package com.mytutorial.jpa;
 2
 3 import java.io.Serializable;
 4 import java.util.Set;
 5
 6 import javax.persistence.CascadeType;
 7 import javax.persistence.Entity;
 8 import javax.persistence.FetchType;
 9 import javax.persistence.GeneratedValue;
10 import javax.persistence.GenerationType;
11 import javax.persistence.Id;
12 import javax.persistence.JoinColumn;
13 import javax.persistence.NamedQuery;
14 import javax.persistence.OneToMany;
15
16 @Entity(name = "categories")
17 @NamedQuery(
18         name = "selectSpecificCategory",
19         query = "SELECT c FROM categories c WHERE c.name = :categoryName"
20 )
21 public class Category implements Serializable {
22
23     private static final long serialVersionUID = 1L;
24
25     @Id
26     @GeneratedValue(strategy = GenerationType.IDENTITY)
27     private Integer id;
28
29     private String name;
30
31     @OneToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
32     @JoinColumn(name = "category_id")
33     private Set<Product> products;
34
35     public Category() {
36     }
37
38     public Category(String name) {
39         this.name = name;
40     }
41
42     public Integer getId() {□
45
46     public void setId(Integer id) {□
49
50     public String getName() {□
53
54     public void setName(String name) {□
57
58     public Set<Product> getProducts() {□
61
62     public void setProducts(Set<Product> products) {□
65
66     @Override
67     public String toString() {
68         return String.format("{ %d, %s, %s }", id, name, products);
69     }
70
71 }
```

- Now the named query definition is in the form of an annotation **@NamedQuery** that is associated with the class.
- For every named query, you need to specify the name that we'll use to reference this query and the actual query itself. The name of this query is `"selectSpecificCategory"`.
- And the query is `"SELECT c FROM categories c WHERE c.name = :categoryName"`. It's a parameterized query that accepts a named parameter, the `categoryName`.

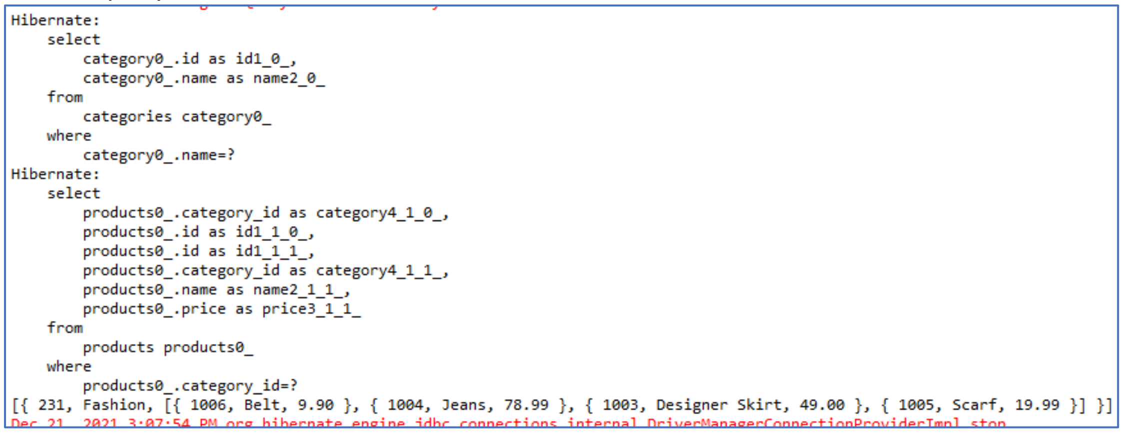
Once you've defined this annotation, this particular query is associated with the category entity that is it'll return category entities. And you can look up this query by name using the entity manager.

Let's switch over to **App.java**. And here I use the entity manager to create a new query using `createNamedQuery()`.



```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {
            Query categoryQuery = entityManager.createNamedQuery("selectSpecificCategory");
            categoryQuery.setParameter("categoryName", "Fashion");

            System.out.println(categoryQuery.getResultList());

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

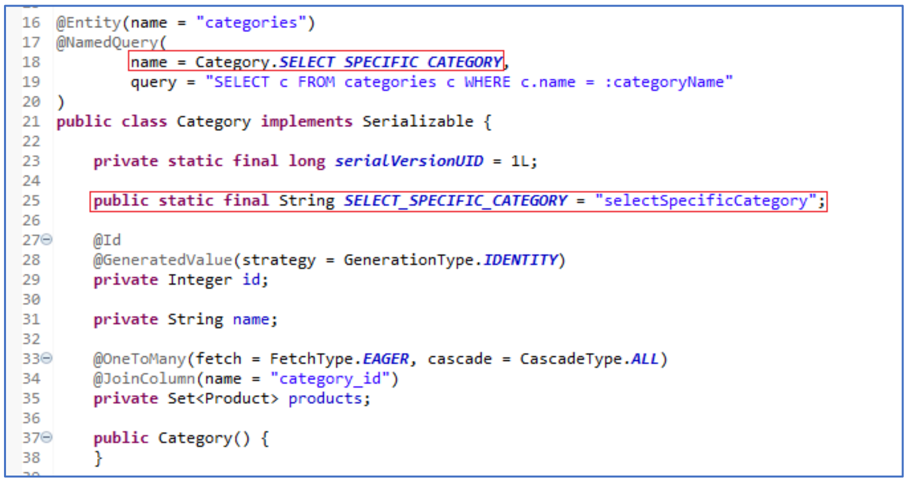
- The named query associated with the entity manager will be looked up and a query object returned, that is the `categoryQuery`. Remember, this is a parameterized query.
- I have set the `categoryName` parameter to `"Fashion"`
- and I invoke `geResultList()` on this category query to get all of the categories that match my query.



```
Hibernate:
    select
        category0_.id as id1_0_,
        category0_.name as name2_0_
    from
        categories category0_
    where
        category0_.name=?
Hibernate:
    select
        products0_.category_id as category4_1_0_,
        products0_.id as id1_1_0_,
        products0_.id as id1_1_1_,
        products0_.category_id as category4_1_1_,
        products0_.name as name2_1_1_,
        products0_.price as price3_1_1_
    from
        products products0_
    where
        products0_.category_id=?
[{ 231, Fashion, [{ 1006, Belt, 9.90 }, { 1004, Jeans, 78.99 }, { 1003, Designer Skirt, 49.00 }, { 1005, Scarf, 19.99 }] }]
Dec 21, 2021 3:07:54 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl stop
```

There's exactly one category that has been returned here in the result, the “Fashion” category with id 231 and four products that are associated with this category.

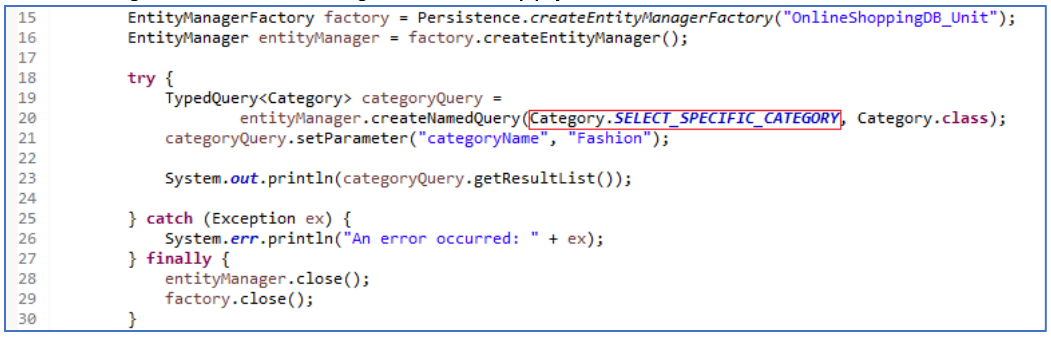
Here is how you use named queries. The way you'd specify the named query in the real world is to use a **String Constant** defined within your entity class that holds the name of your query.



```
@Entity(name = "categories")
@NamedQuery(
        name = Category.SELECT_SPECIFIC_CATEGORY,
        query = "SELECT c FROM categories c WHERE c.name = :categoryName"
)
public class Category implements Serializable {

    private static final long serialVersionUID = 1L;

    public static final String SELECT_SPECIFIC_CATEGORY = "selectSpecificCategory";

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String name;

    @OneToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    @JoinColumn(name = "category_id")
    private Set<Product> products;

    public Category() {
    }
```

- Here the String is `SELECT_SPECIFIC_CATEGORY`, I've defined it within my category entity as `public static final String`. And the value of this **String** is the name of my query.
- Within my **@NamedQuery** annotation, I reference this string constant, `Category.SELECT_SPECIFIC_CATEGORY`. And the query that I have defined is the same, `"SELECT c FROM categories c WHERE c.name = :categoryName"`.

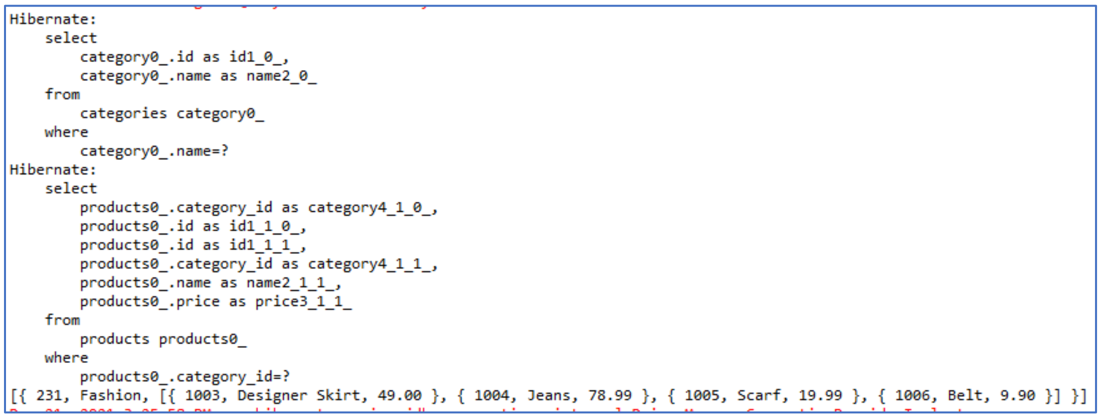
Using a constant to define the name of the query makes it easier to access and look up that query. You won't get the name wrong. So within App.java,



```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {
            TypedQuery<Category> categoryQuery =
                    entityManager.createNamedQuery(Category.SELECT_SPECIFIC_CATEGORY, Category.class);
            categoryQuery.setParameter("categoryName", "Fashion");

            System.out.println(categoryQuery.getResultList());

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

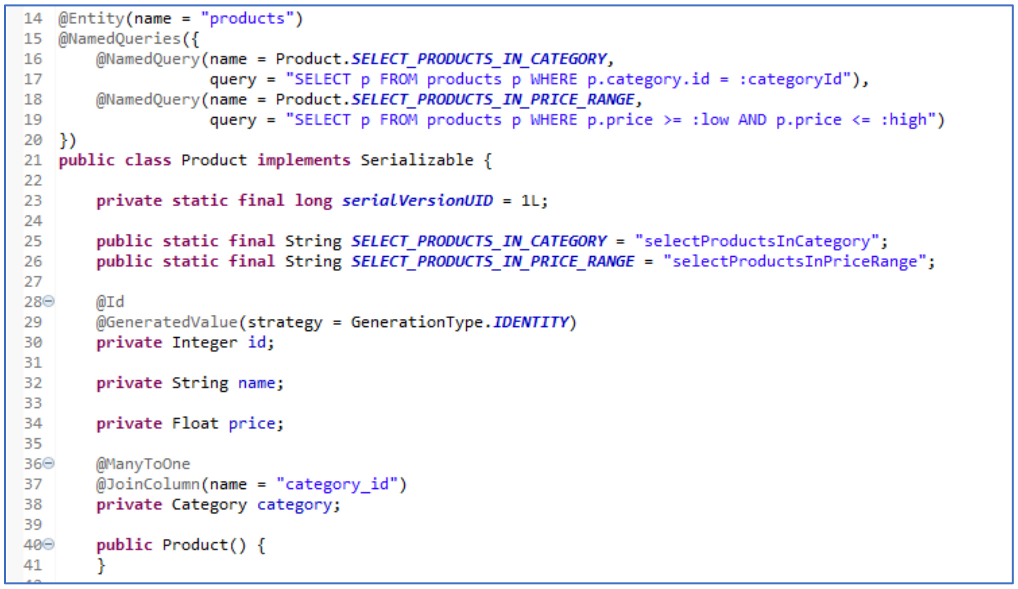
- I'll now create a named query using the same constant, `Category.SELECT_SPECIFIC_CATEGORY`.
- Notice that we've invoked the `createNamedQuery()` method as a `TypedQuery`. We've specified that this will return objects of the `Category.class`.
- Then we set the `categoryName` parameter to fashion as before,
- and invoke get single result because we know there's exactly one category with this name.

And here are the details of the “Fashion” category along with its products printed out to screen within our console window.



```
Hibernate: 
    select
        category0_.id as id1_0_,
        category0_.name as name2_0_ 
    from
        categories category0_ 
    where
        category0_.name=?
Hibernate: 
    select
        products0_.category_id as category4_1_0_,
        products0_.id as id1_1_0_,
        products0_.id as id1_1_1_,
        products0_.category_id as category4_1_1_,
        products0_.name as name2_1_1_,
        products0_.price as price3_1_1_ 
    from
        products products0_ 
    where
        products0_.category_id=?
[{ 231, Fashion, [{ 1003, Designer Skirt, 49.00 }, { 1004, Jeans, 78.99 }, { 1005, Scarf, 19.99 }, { 1006, Belt, 9.90 }] }]
```

We are ready to see yet another example of a named query, in fact, multiple named queries that we'll associate with the products entity. We are in **Product.java**.

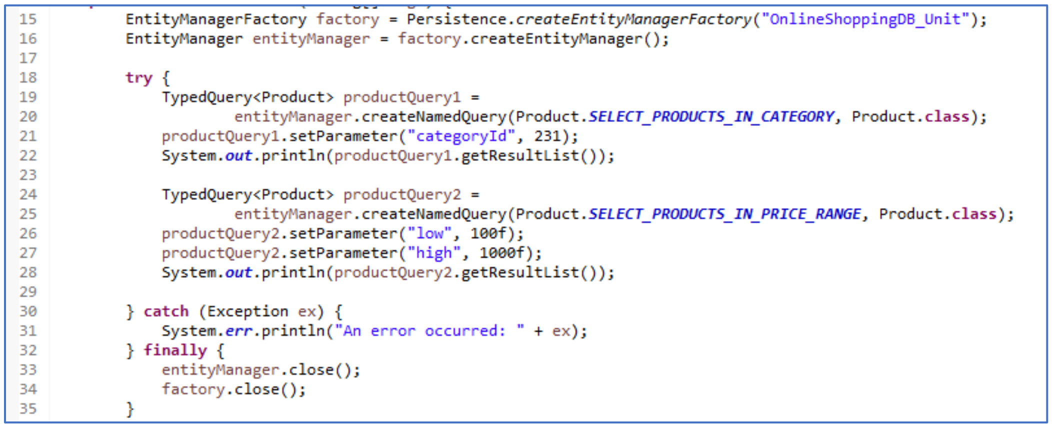


```
14 @Entity(name = "products")
15 @NamedQueries({
16     @NamedQuery(name = Product.SELECT_PRODUCTS_IN_CATEGORY,
17                 query = "SELECT p FROM products p WHERE p.category.id = :categoryId"),
18     @NamedQuery(name = Product.SELECT_PRODUCTS_IN_PRICE_RANGE,
19                 query = "SELECT p FROM products p WHERE p.price >= :low AND p.price <= :high")
20 })
21 public class Product implements Serializable {
22 
23     private static final long serialVersionUID = 1L;
24 
25     public static final String SELECT_PRODUCTS_IN_CATEGORY = "selectProductsInCategory";
26     public static final String SELECT_PRODUCTS_IN_PRICE_RANGE = "selectProductsInPriceRange";
27 
28     @Id
29     @GeneratedValue(strategy = GenerationType.IDENTITY)
30     private Integer id;
31 
32     private String name;
33 
34     private Float price;
35 
36     @ManyToOne
37     @JoinColumn(name = "category_id")
38     private Category category;
39 
40     public Product() {
41     }
```

- I set up the import for named queries and named query.
  `import javax.persistence.NamedQueries;`
  `import javax.persistence.NamedQuery;`
- When you want to associate multiple named queries with the same entity, you will need to use a repeating annotation and named queries is our repeating annotation.
- Notice **@NamedQueries** on line 15. And within that, we have specified two named queries using the **@NamedQuery** annotation.
- The names of each query I have defined as public static final string within my product class, ***SELECT_PRODUCTS_IN_CATEGORY*** and ***SELECT_PRODUCTS_IN_PRICE_RANGE***.
- Let's take a look at the actual queries. Now, I've defined the name of both of these queries using these constants strings.
  The first query ***SELECT_PRODUCTS_IN_CATEGORY*** does a `"SELECT p FROM products p WHERE p.category.id = :categoryId"` is equal to the `categoryId` we specify.

The second query contains two named parameters, `"SELECT p FROM products p WHERE p.price` is greater than equal to the low price specified and `p.price` is less than equal to the high price that we specified.

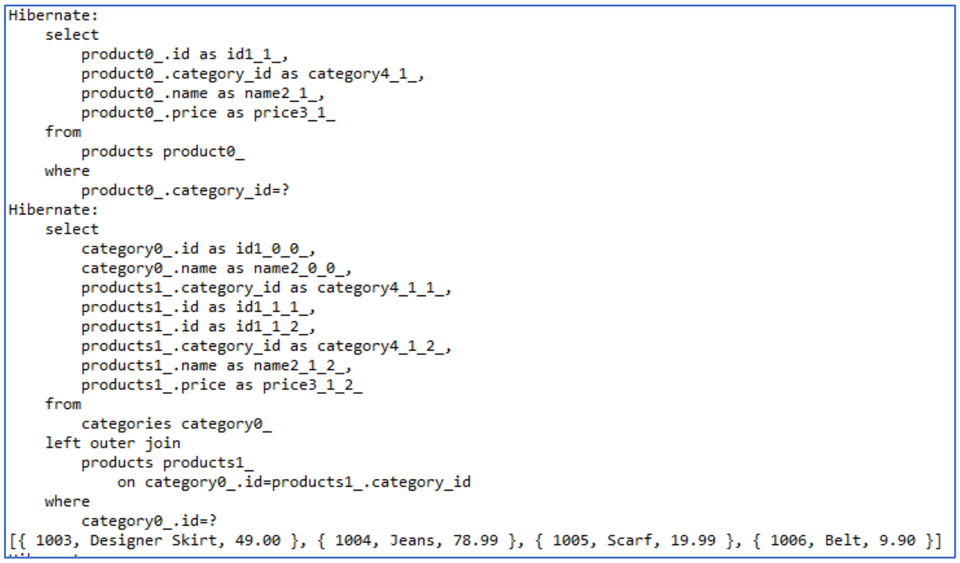
Let's see how we can use these named queries from our `entityManager`.



```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {
            TypedQuery<Product> productQuery1 =
                    entityManager.createNamedQuery(Product.SELECT_PRODUCTS_IN_CATEGORY, Product.class);
            productQuery1.setParameter("categoryId", 231);
            System.out.println(productQuery1.getResultList());

            TypedQuery<Product> productQuery2 =
                    entityManager.createNamedQuery(Product.SELECT_PRODUCTS_IN_PRICE_RANGE, Product.class);
            productQuery2.setParameter("low", 100f);
            productQuery2.setParameter("high", 1000f);
            System.out.println(productQuery2.getResultList());

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

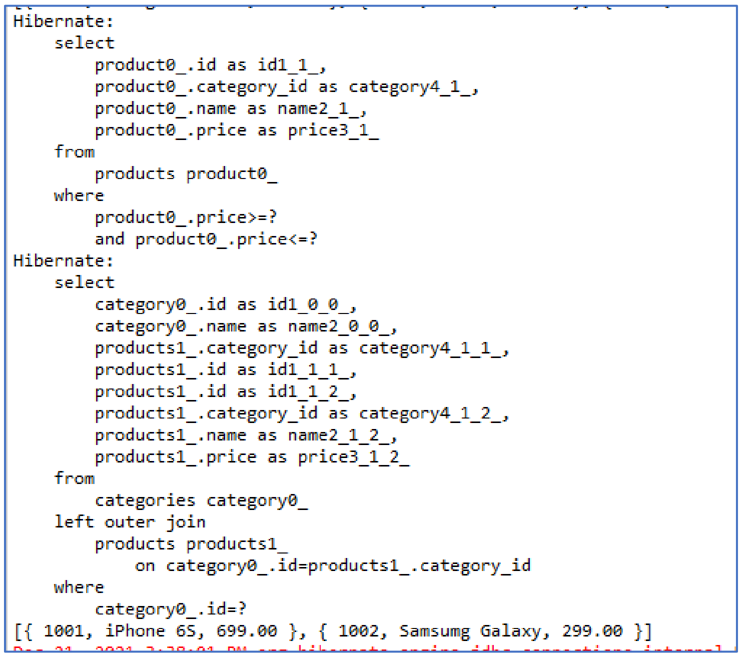
- I've created two named queries here which return products. The first of these queries, ***`SELECT_PRODUCTS_IN_CATEGORY`***, which returns the product class. I set the `categoryId` parameter to 231 and I print out the result list.
- The second query is ***`SELECT_PRODUCTS_IN_PRICE_RANGE`***. For the second query, I set the `low` parameter to $100 and `high` parameter to $1,000 and I print out the result list.

Let's run this code and take a look at the results within our console window. Here are the products within the fashion category, the one with id 231.



```
Hibernate: 
    select
        product0_.id as id1_1_,
        product0_.category_id as category4_1_,
        product0_.name as name2_1_,
        product0_.price as price3_1_ 
    from
        products product0_ 
    where
        product0_.category_id=?
Hibernate: 
    select
        category0_.id as id1_0_0_,
        category0_.name as name2_0_0_,
        products1_.category_id as category4_1_1_,
        products1_.id as id1_1_1_,
        products1_.id as id1_1_2_,
        products1_.category_id as category4_1_2_,
        products1_.name as name2_1_2_,
        products1_.price as price3_1_2_ 
    from
        categories category0_ 
    left outer join
        products products1_ 
            on category0_.id=products1_.category_id 
    where
        category0_.id=?
[{ 1003, Designer Skirt, 49.00 }, { 1004, Jeans, 78.99 }, { 1005, Scarf, 19.99 }, { 1006, Belt, 9.90 }]
```

If you scroll down further, you'll find the results of the second query, products within a price range. The only two products are the iPhone 6s and the Samsung Galaxy.

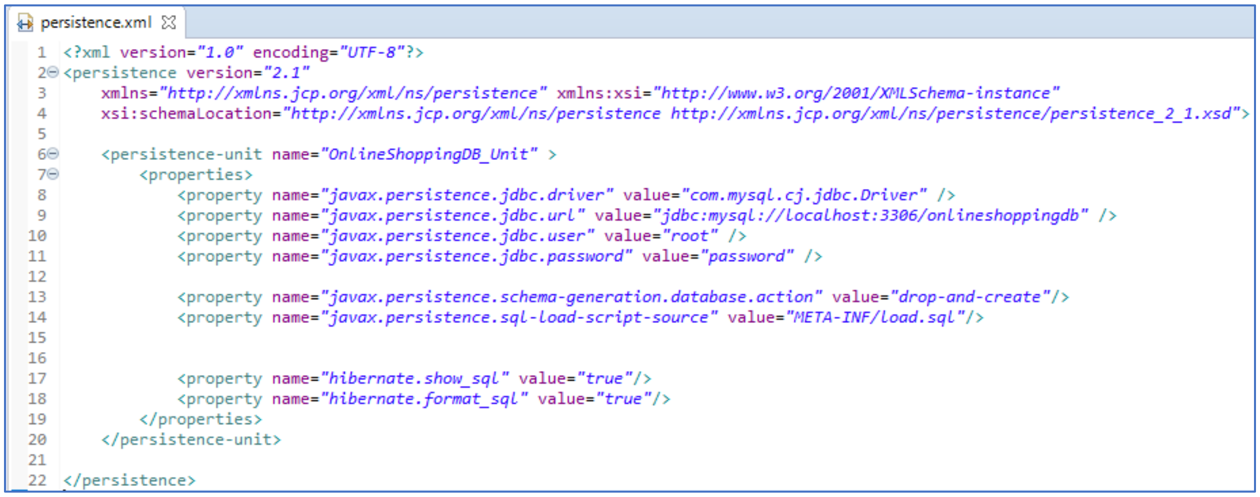


```
Hibernate:
    select
        product0_.id as id1_1_,
        product0_.category_id as category4_1_,
        product0_.name as name2_1_,
        product0_.price as price3_1_
    from
        products product0_
    where
        product0_.price>=?
        and product0_.price<=?
Hibernate:
    select
        category0_.id as id1_0_0_,
        category0_.name as name2_0_0_,
        products1_.category_id as category4_1_1_,
        products1_.id as id1_1_1_,
        products1_.id as id1_1_2_,
        products1_.category_id as category4_1_2_,
        products1_.name as name2_1_2_,
        products1_.price as price3_1_2_
    from
        categories category0_
    left outer join
        products products1_
            on category0_.id=products1_.category_id
    where
        category0_.id=?
[{ 1001, iPhone 6S, 699.00 }, { 1002, Samsumg Galaxy, 299.00 }]
```

## The Criteria API

In this demo and in the demos that follow, we'll get a quick overview of the Criteria API in JPA in order to build queries. The Criteria API offers a programmatic way to create typed queries, which allows us to avoid syntax errors in query specification. The Criteria API can be used along with the Metamodel API in order to perform compile time checks on your queries.

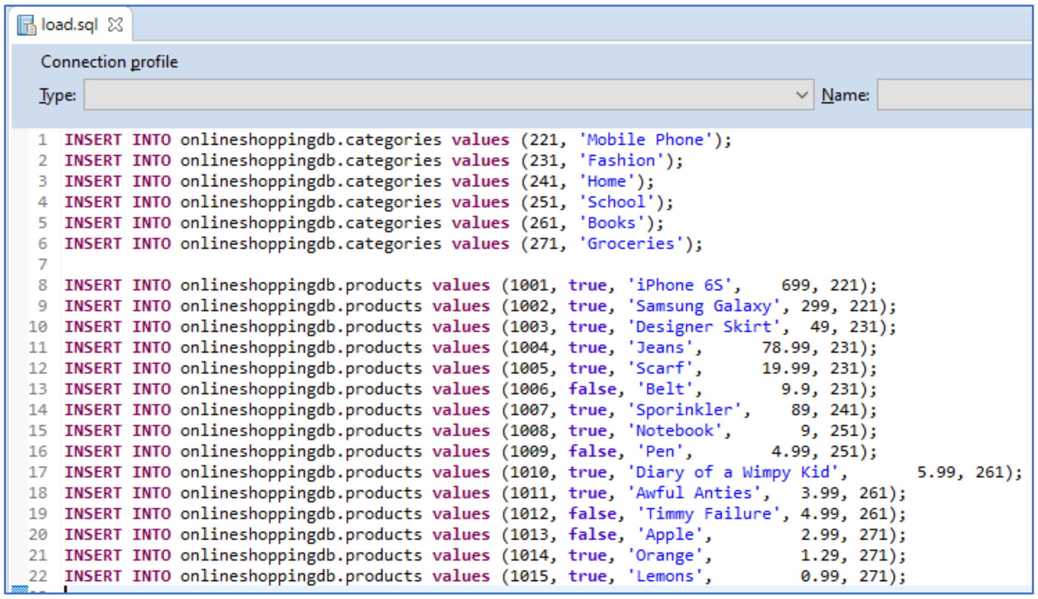
Now we won't be exploring the Metamodel API in our demos, but we will explore the basics of the Criteria API that will give you a strong foundation to move on to the Metamodel API later. Here is the **persistence.xml** that we'll use.



persistence.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
    xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">

    <persistence-unit name="OnlineShoppingDB_Unit" >
        <properties>
            <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/onlineshoppingdb" />
            <property name="javax.persistence.jdbc.user" value="root" />
            <property name="javax.persistence.jdbc.password" value="password" />

            <property name="javax.persistence.schema-generation.database.action" value="drop-and-create"/>
            <property name="javax.persistence.sql-load-script-source" value="META-INF/load.sql"/>


            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
        </properties>
    </persistence-unit>

</persistence>
```

We'll continue to use the "onlineshoppingdb". We'll also use the database action drop-and-create and I have a **load.sql** that will load categories and products into my database.



load.sql

Connection profile

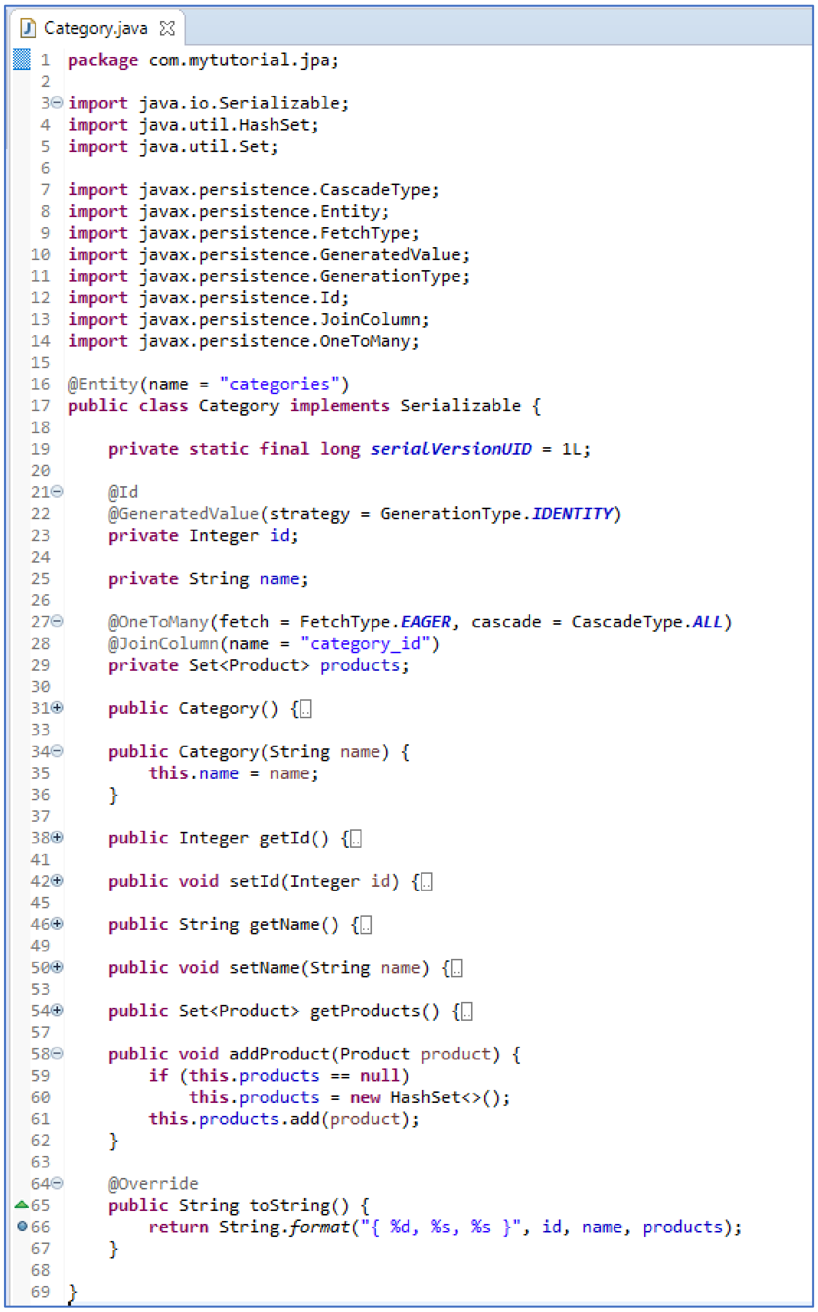
Type: Name:

```sql
INSERT INTO onlineshoppingdb.categories values (221, 'Mobile Phone');
INSERT INTO onlineshoppingdb.categories values (231, 'Fashion');
INSERT INTO onlineshoppingdb.categories values (241, 'Home');
INSERT INTO onlineshoppingdb.categories values (251, 'School');
INSERT INTO onlineshoppingdb.categories values (261, 'Books');
INSERT INTO onlineshoppingdb.categories values (271, 'Groceries');

INSERT INTO onlineshoppingdb.products values (1001, true, 'iPhone 6S',    699, 221);
INSERT INTO onlineshoppingdb.products values (1002, true, 'Samsung Galaxy', 299, 221);
INSERT INTO onlineshoppingdb.products values (1003, true, 'Designer Skirt',  49, 231);
INSERT INTO onlineshoppingdb.products values (1004, true, 'Jeans',      78.99, 231);
INSERT INTO onlineshoppingdb.products values (1005, true, 'Scarf',      19.99, 231);
INSERT INTO onlineshoppingdb.products values (1006, false, 'Belt',        9.9, 231);
INSERT INTO onlineshoppingdb.products values (1007, true, 'Sporinkler',    89, 241);
INSERT INTO onlineshoppingdb.products values (1008, true, 'Notebook',       9, 251);
INSERT INTO onlineshoppingdb.products values (1009, false, 'Pen',        4.99, 251);
INSERT INTO onlineshoppingdb.products values (1010, true, 'Diary of a Wimpy Kid',       5.99, 261);
INSERT INTO onlineshoppingdb.products values (1011, true, 'Awful Anties',    3.99, 261);
INSERT INTO onlineshoppingdb.products values (1012, false, 'Timmy Failure', 4.99, 261);
INSERT INTO onlineshoppingdb.products values (1013, false, 'Apple',         2.99, 271);
INSERT INTO onlineshoppingdb.products values (1014, true, 'Orange',         1.29, 271);
INSERT INTO onlineshoppingdb.products values (1015, true, 'Lemons',         0.99, 271);
```

We'll continue working with the category and product entities. I have expanded the number of categories that I have. I've added two more categories, Books and Groceries.

Here are the insert statements to load our underlying database with products. I've also expanded the number of products that we have. I have books and groceries at the very bottom.

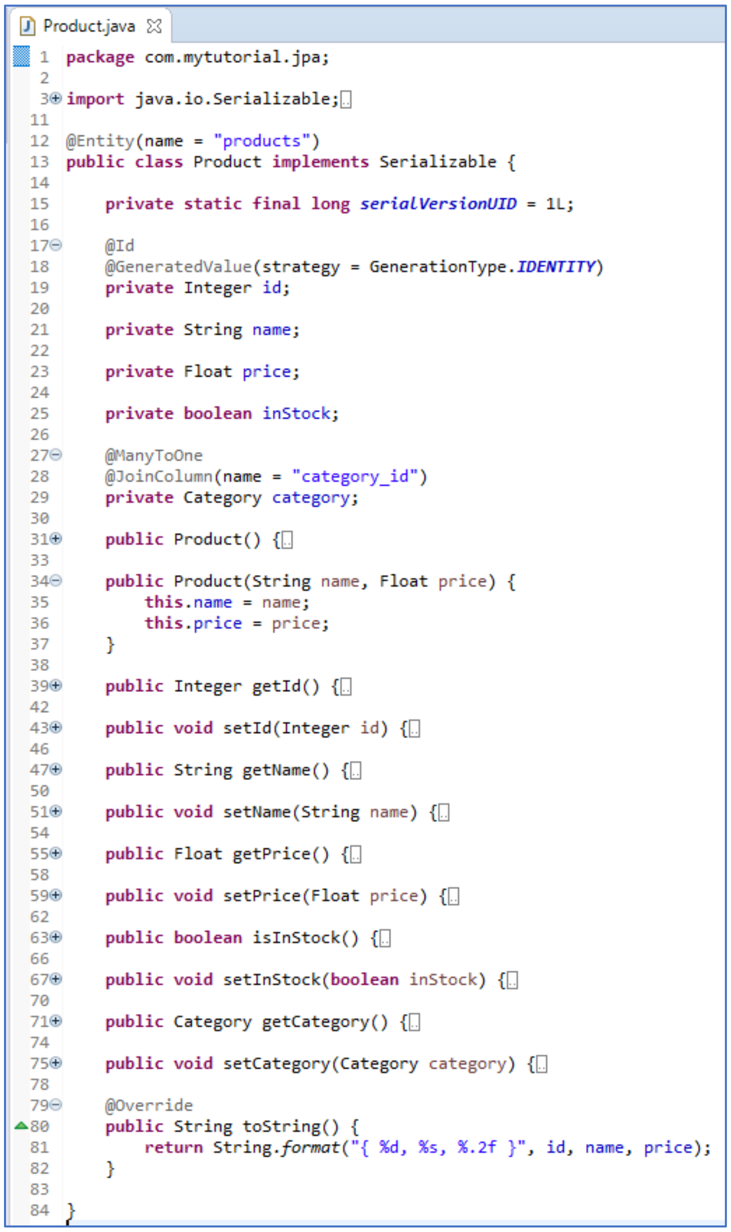
Here is the `Category` entity that we'll continue working with. It hasn't changed much.



```
package com.mytutorial.jpa;

import java.io.Serializable;
import java.util.HashSet;
import java.util.Set;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToMany;

@Entity(name = "categories")
public class Category implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String name;

    @OneToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    @JoinColumn(name = "category_id")
    private Set<Product> products;

    public Category() {

    public Category(String name) {
        this.name = name;
    }

    public Integer getId() {

    public void setId(Integer id) {

    public String getName() {

    public void setName(String name) {

    public Set<Product> getProducts() {

    public void addProduct(Product product) {
        if (this.products == null)
            this.products = new HashSet<>();
        this.products.add(product);
    }

    @Override
    public String toString() {
        return String.format("{ %d, %s, %s }", id, name, products);
    }

}
```

- I've tagged it with the **@Entity** interface and it has the ID, name, and products member variables.
- There is a one to many relationship between a category and the associated products. Set up the getter and setters exactly as we've done before for the category entity.

Let's move on and now take a look at the `Product` entity.


Product.java

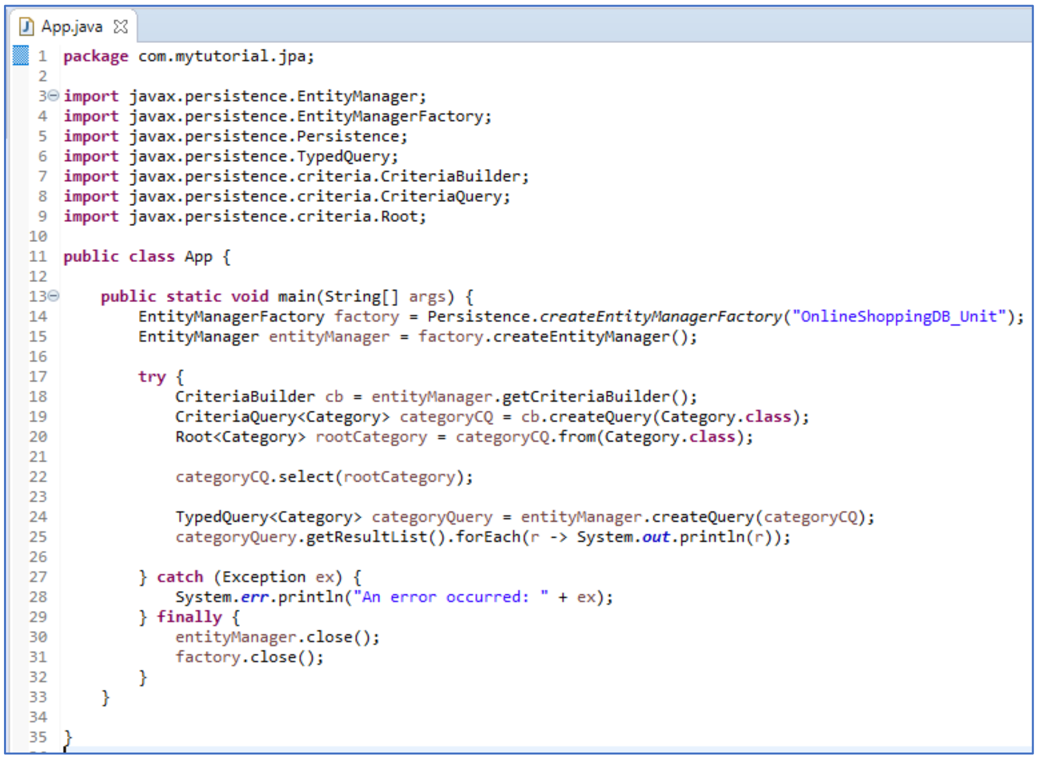
```java
package com.mytutorial.jpa;

import java.io.Serializable;

@Entity(name = "products")
public class Product implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String name;

    private Float price;

    private boolean inStock;

    @ManyToOne
    @JoinColumn(name = "category_id")
    private Category category;

    public Product() {

    public Product(String name, Float price) {
        this.name = name;
        this.price = price;
    }

    public Integer getId() {

    public void setId(Integer id) {

    public String getName() {

    public void setName(String name) {

    public Float getPrice() {

    public void setPrice(Float price) {

    public boolean isInStock() {

    public void setInStock(boolean inStock) {

    public Category getCategory() {

    public void setCategory(Category category) {

    @Override
    public String toString() {
        return String.format("{ %d, %s, %.2f }", id, name, price);
    }

}
```

Once again, this hasn't changed much. In fact, there's exactly one change that I point out.

- I've tagged it with the **@Entity** annotation. Here are the member variables, ID, name, price, and category which is a reference to the category that is associated with a product.
- I have an additional boolean variable here called inStock, which tells me whether a particular product is in stock or not.
- Set up the getters and setters for this product entities all of the member variables in here

## CriteriaBuilder, CriteriaQuery, Root and select() clause

Let's move on to **App.java**.



App.java

```
 1 package com.mytutorial.jpa;
 2 
 3 import javax.persistence.EntityManager;
 4 import javax.persistence.EntityManagerFactory;
 5 import javax.persistence.Persistence;
 6 import javax.persistence.TypedQuery;
 7 import javax.persistence.criteria.CriteriaBuilder;
 8 import javax.persistence.criteria.CriteriaQuery;
 9 import javax.persistence.criteria.Root;
10 
11 public class App {
12 
13     public static void main(String[] args) {
14         EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
15         EntityManager entityManager = factory.createEntityManager();
16 
17         try {
18             CriteriaBuilder cb = entityManager.getCriteriaBuilder();
19             CriteriaQuery<Category> categoryCQ = cb.createQuery(Category.class);
20             Root<Category> rootCategory = categoryCQ.from(Category.class);
21 
22             categoryCQ.select(rootCategory);
23 
24             TypedQuery<Category> categoryQuery = entityManager.createQuery(categoryCQ);
25             categoryQuery.getResultList().forEach(r -> System.out.println(r));
26 
27         } catch (Exception ex) {
28             System.err.println("An error occurred: " + ex);
29         } finally {
30             entityManager.close();
31             factory.close();
32         }
33     }
34 
35 }
```

Here's where I'll use the CriteriaBuilder API to construct my query. Now the CriteriaBuilder is great, because you can create queries programmatically, rather than using JPQL commands. This ensures that you will avoid syntax errors in your queries.

However, the Criteria API tends to be very verbose. There's a lot of boilerplate code that you need to write, as you'll see in just a bit.

- In order to use the Criteria API, the first thing that you do is obtain an instance of the `CriteriaBuilder`. This I do on line 18. The `CriteriaBuilder` is the entry point to the Criteria API and is used to construct query objects and their expressions.
  ```
  CriteriaBuilder cb = entityManager.getCriteriaBuilder();
  ```
  And you can obtain the `CriteriaBuilder` from the entity manager.

- Next, using the `CriteriaBuilder`, you create a `CriteriaQuery` which represents a query.
  ```
  CriteriaQuery<Category> categoryCQ = cb.createQuery(Category.class);
  ```
  This is a TypedQuery, this query will return category entities. The CriteriaQuery contains a number of interface methods that you can use to specify clauses for your query.
  For the time being, we're working with very simple queries with no additional clauses.

- Once you have the CriteriaQuery, you'll access the `javax.persistence.criteria.Root` type of the target entity.
  ```
  Root<Category> rootCategory = categoryCQ.from(Category.class);
  ```
  This you do using `categoryCQ.from()`. This Root type allows you to construct queries.

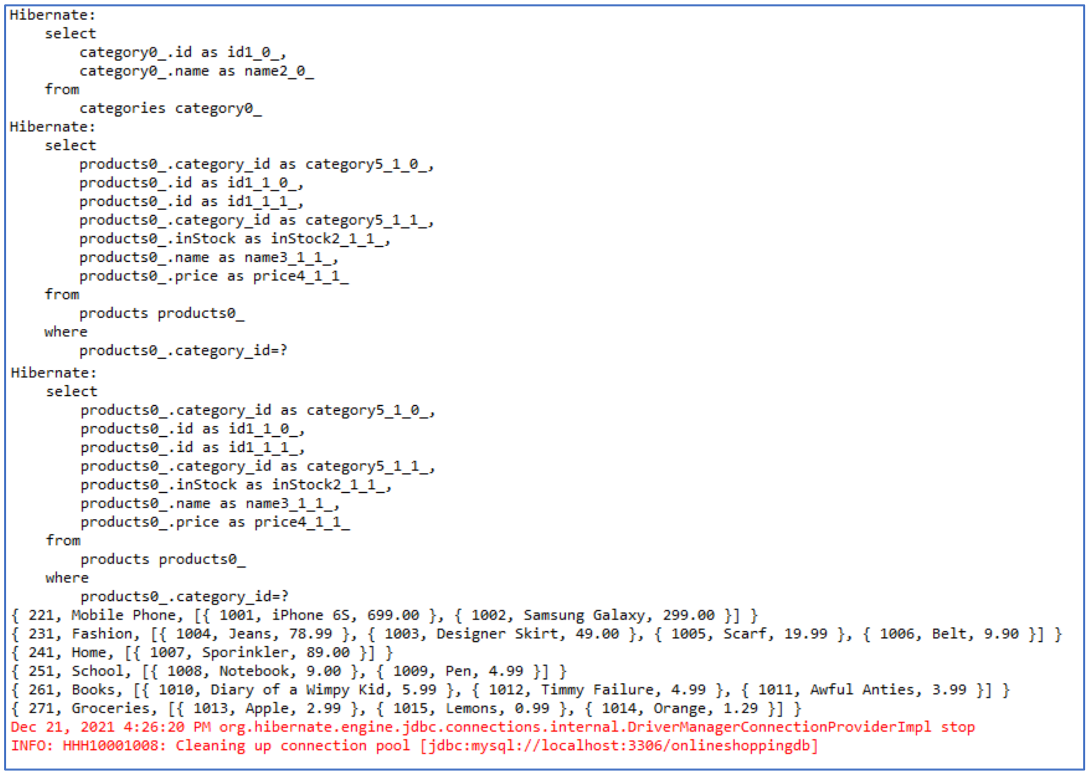
- Our query is going to be a simple **`select()`** for all category. So I invoke on line 22
  ```
  categoryCQ.select(rootCategory)
  ```

- And then I use this category query to create an ordinary typed query.
  ```
  TypedQuery<Category> categoryQuery =
  entityManager.createQuery(categoryCQ);
  ```

- Notice that the end result with the Criteria API is an ordinary typed query. But the way I've constructed my query ensures that there are no syntax errors in my query. Then I'll call `getResultList()` and get all of the categories present in my underlying database table.
  ```
  categoryQuery.getResultList()
  ```

Notice how much work I had to do to create a simple query using the Criteria API.

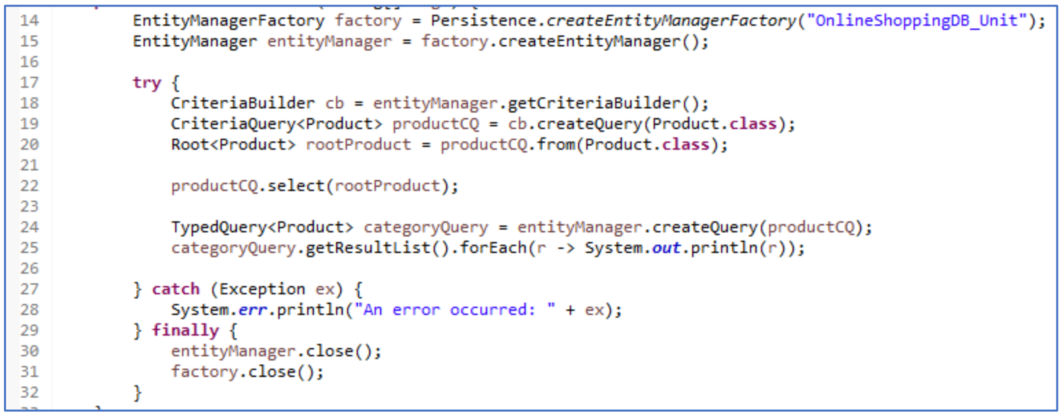
Let's run this code.



```
Hibernate:
    select
        category0_.id as id1_0_,
        category0_.name as name2_0_
    from
        categories category0_
Hibernate:
    select
        products0_.category_id as category5_1_0_,
        products0_.id as id1_1_0_,
        products0_.id as id1_1_1_,
        products0_.category_id as category5_1_1_,
        products0_.inStock as inStock2_1_1_,
        products0_.name as name3_1_1_,
        products0_.price as price4_1_1_
    from
        products products0_
    where
        products0_.category_id=?
Hibernate:
    select
        products0_.category_id as category5_1_0_,
        products0_.id as id1_1_0_,
        products0_.id as id1_1_1_,
        products0_.category_id as category5_1_1_,
        products0_.inStock as inStock2_1_1_,
        products0_.name as name3_1_1_,
        products0_.price as price4_1_1_
    from
        products products0_
    where
        products0_.category_id=?
{ 221, Mobile Phone, [{ 1001, iPhone 6S, 699.00 }, { 1002, Samsung Galaxy, 299.00 }] }
{ 231, Fashion, [{ 1004, Jeans, 78.99 }, { 1003, Designer Skirt, 49.00 }, { 1005, Scarf, 19.99 }, { 1006, Belt, 9.90 }] }
{ 241, Home, [{ 1007, Sporinkler, 89.00 }] }
{ 251, School, [{ 1008, Notebook, 9.00 }, { 1009, Pen, 4.99 }] }
{ 261, Books, [{ 1010, Diary of a Wimpy Kid, 5.99 }, { 1012, Timmy Failure, 4.99 }, { 1011, Awful Anties, 3.99 }] }
{ 271, Groceries, [{ 1013, Apple, 2.99 }, { 1015, Lemons, 0.99 }, { 1014, Orange, 1.29 }] }
Dec 21, 2021 4:26:20 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl stop
INFO: HHH10001008: Cleaning up connection pool [jdbc:mysql://localhost:3306/onlineshoppingdb]
```

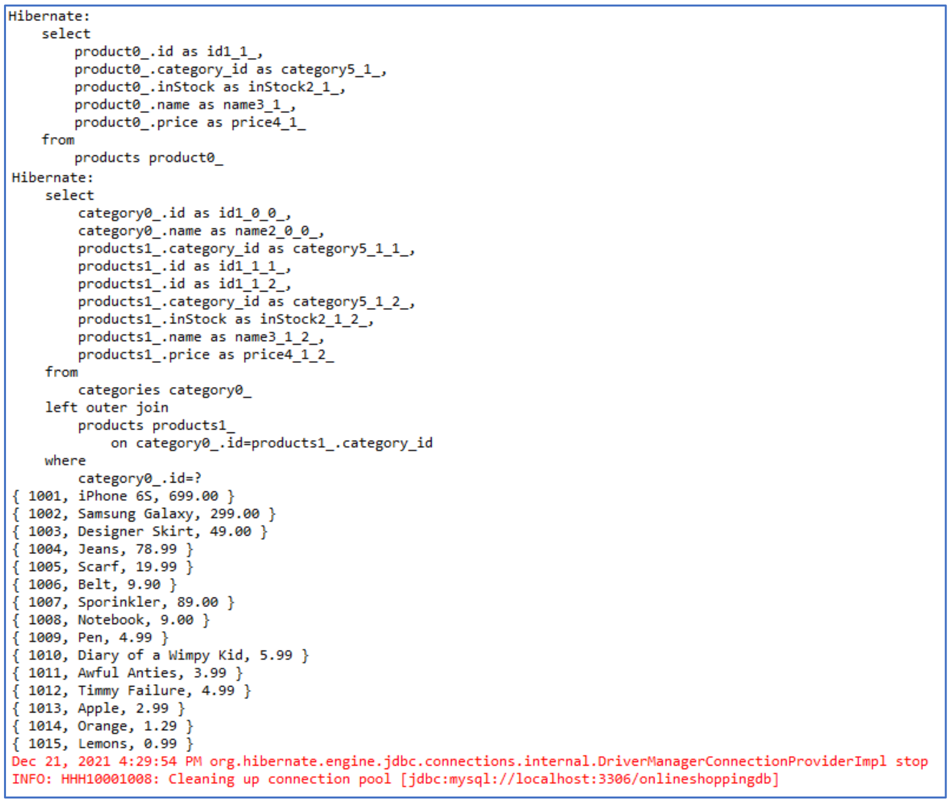
And if you scroll down to the very bottom, you'll see the category entries retrieved by this `CriteriaQuery` that we used to query the underlying database. All of the categories that are present in our tables.

In exactly the same way, let's use the Criteria API to query all products that exists in the `products` table.



```java
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {
            CriteriaBuilder cb = entityManager.getCriteriaBuilder();
            CriteriaQuery<Product> productCQ = cb.createQuery(Product.class);
            Root<Product> rootProduct = productCQ.from(Product.class);

            productCQ.select(rootProduct);

            TypedQuery<Product> categoryQuery = entityManager.createQuery(productCQ);
            categoryQuery.getResultList().forEach(r -> System.out.println(r));

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

- We get the `CriteriaBuilder`, we create a query which returns the `Product.class`.
- We then get the `javax.persistence.criteria.Root` entity type using the `from()` method.
- Then using the `CriteriaQuery`, we `.select()` the `Root` type of the entity returned.
- We then create a typed query using `entityManager.createQuery()`, pass in the `CriteriaQuery` that we've created, and print out the result list, all of the `products` that exists in our database table.



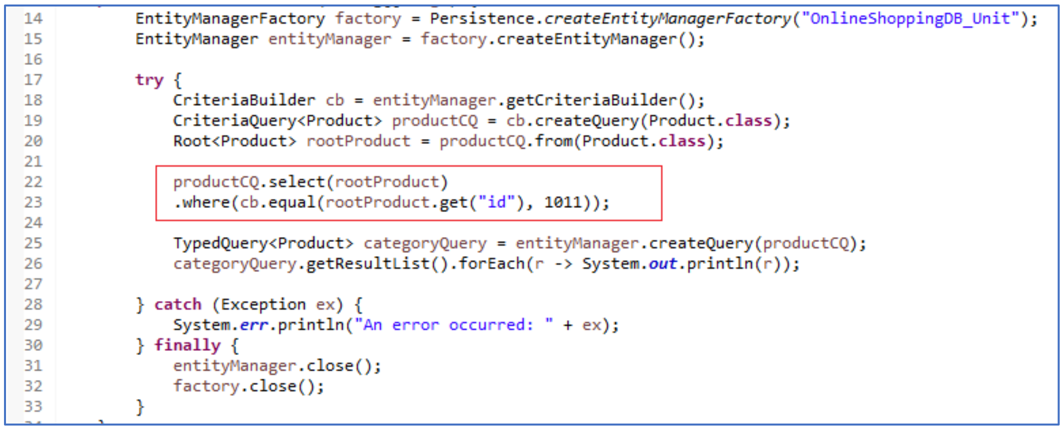
```
Hibernate: 
    select
        product0_.id as id1_1_,
        product0_.category_id as category5_1_,
        product0_.inStock as inStock2_1_,
        product0_.name as name3_1_,
        product0_.price as price4_1_ 
    from
        products product0_
Hibernate: 
    select
        category0_.id as id1_0_0_,
        category0_.name as name2_0_0_,
        products1_.category_id as category5_1_1_,
        products1_.id as id1_1_1_,
        products1_.id as id1_1_2_,
        products1_.category_id as category5_1_2_,
        products1_.inStock as inStock2_1_2_,
        products1_.name as name3_1_2_,
        products1_.price as price4_1_2_ 
    from
        categories category0_ 
    left outer join
        products products1_ 
            on category0_.id=products1_.category_id 
    where
        category0_.id=?
{ 1001, iPhone 6S, 699.00 }
{ 1002, Samsung Galaxy, 299.00 }
{ 1003, Designer Skirt, 49.00 }
{ 1004, Jeans, 78.99 }
{ 1005, Scarf, 19.99 }
{ 1006, Belt, 9.90 }
{ 1007, Sporinkler, 89.00 }
{ 1008, Notebook, 9.00 }
{ 1009, Pen, 4.99 }
{ 1010, Diary of a Wimpy Kid, 5.99 }
{ 1011, Awful Anties, 3.99 }
{ 1012, Timmy Failure, 4.99 }
{ 1013, Apple, 2.99 }
{ 1014, Orange, 1.29 }
{ 1015, Lemons, 0.99 }
Dec 21, 2021 4:29:54 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl stop
INFO: HHH10001008: Cleaning up connection pool [jdbc:mysql://localhost:3306/onlineshoppingdb]
```

You can scroll down to the very bottom and here are all of the products listed out on screen. The Criteria API is hard to work with at the very beginning, but as you use it more and more often, you'll find that it's extremely useful and it helps you avoid making mistakes with your query syntax.

## Running Basic Criteria Queries

So far we've used the Criteria API to run basic queries that selects all entities in a particular table. Now, let's make our query slightly more complicated, not too much.

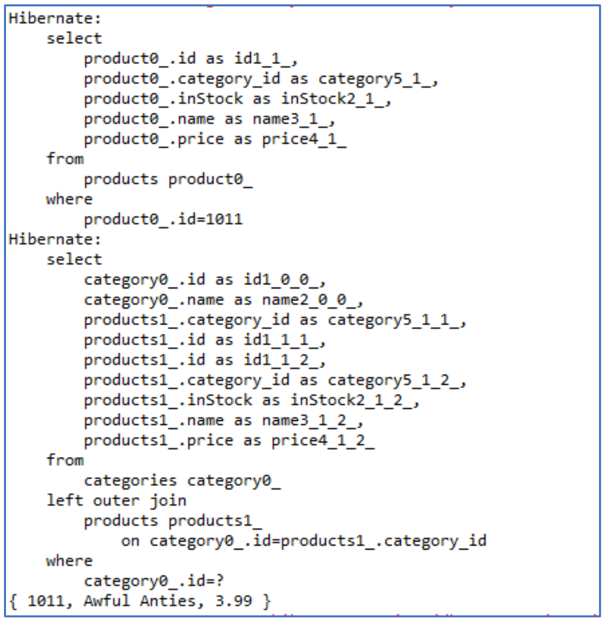
### where() and equal() clause



```java
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {
            CriteriaBuilder cb = entityManager.getCriteriaBuilder();
            CriteriaQuery<Product> productCQ = cb.createQuery(Product.class);
            Root<Product> rootProduct = productCQ.from(Product.class);

            productCQ.select(rootProduct)
            .where(cb.equal(rootProduct.get("id"), 1011));

            TypedQuery<Product> categoryQuery = entityManager.createQuery(productCQ);
            categoryQuery.getResultList().forEach(r -> System.out.println(r));

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

- This is a `CriteriaQuery`, which uses the `.where()` clause.
- We get an instance of the `CriteriaBuilder` create a query that returns a `Product` entity.
- We get the `javax.persistence.criteria.Root` Entity Data Type in a `rootProduct`.
- We want the productCQ to `select()` products where the rootProduct id is equal to `1011`, that is, select the product that has this specific id.
- `cb.equal()` is a method in the criteria builder that returns a predicate and a predicate is what we use to represent a condition.
  We use the predicate returned by `cb.equal()` for our `.where()` clause.

There's no change in the rest of the code. We create a type query using the Criteria API, and get the result list.



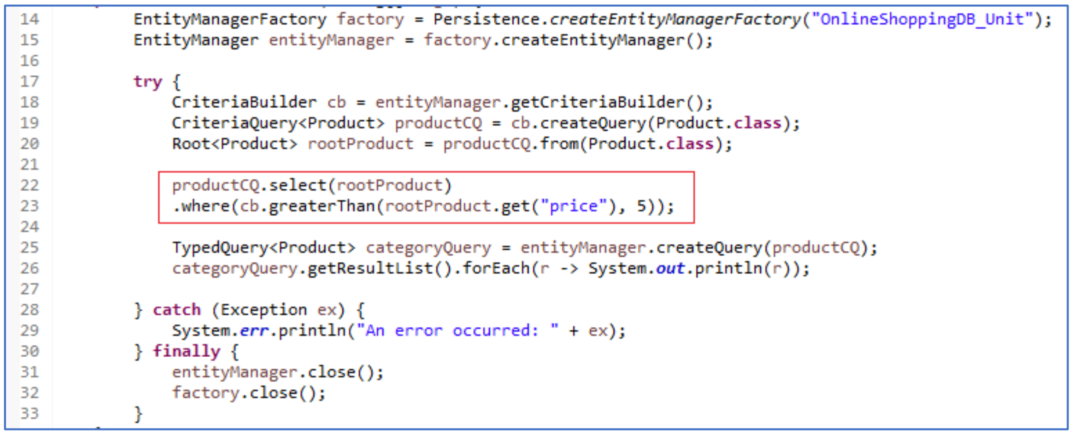
```
Hibernate:
    select
        product0_.id as id1_1_,
        product0_.category_id as category5_1_,
        product0_.inStock as inStock2_1_,
        product0_.name as name3_1_,
        product0_.price as price4_1_
    from
        products product0_
    where
        product0_.id=1011
Hibernate:
    select
        category0_.id as id1_0_0_,
        category0_.name as name2_0_0_,
        products1_.category_id as category5_1_1_,
        products1_.id as id1_1_1_,
        products1_.id as id1_1_2_,
        products1_.category_id as category5_1_2_,
        products1_.inStock as inStock2_1_2_,
        products1_.name as name3_1_2_,
        products1_.price as price4_1_2_
    from
        categories category0_
    left outer join
        products products1_
            on category0_.id=products1_.category_id
    where
        category0_.id=?
{ 1011, Awful Anties, 3.99 }
```

Here is the product with id 1011. The book called "Awful Aunties".

## gretherThan() clause

The criteria builder offers a large number of interface methods that return predicates that you can use to set up different conditions.

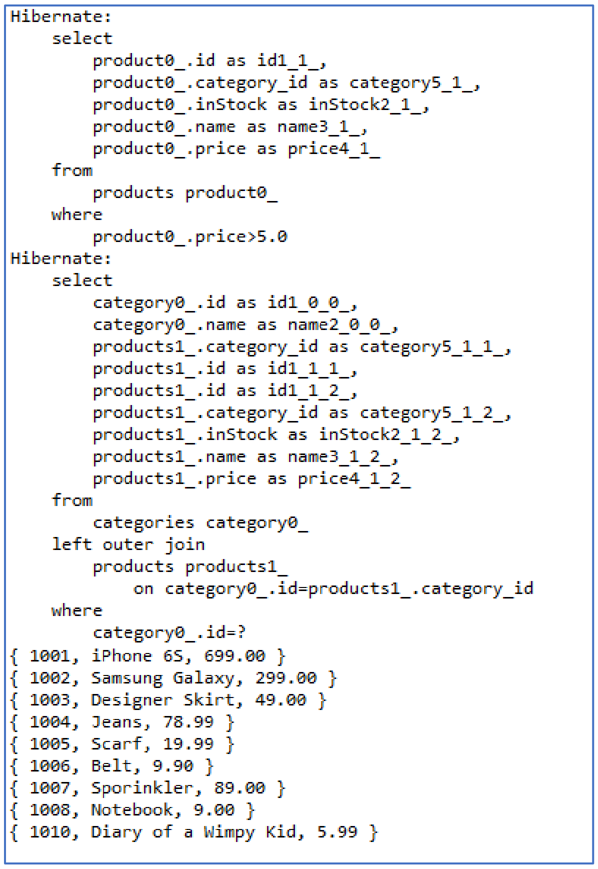
Here is a predicate, where you can specify that you want price greater than 5.



```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {
            CriteriaBuilder cb = entityManager.getCriteriaBuilder();
            CriteriaQuery<Product> productCQ = cb.createQuery(Product.class);
            Root<Product> rootProduct = productCQ.from(Product.class);

            productCQ.select(rootProduct)
            .where(cb.greaterThan(rootProduct.get("price"), 5));

            TypedQuery<Product> categoryQuery = entityManager.createQuery(productCQ);
            categoryQuery.getResultList().forEach(r -> System.out.println(r));

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

`cb.greaterThan(rootProduct.get("price"), 5).`

This is the condition that we have specified in the where clause of our product query to return all products, which have price greater than 5.



```
Hibernate:
    select
        product0_.id as id1_1_,
        product0_.category_id as category5_1_,
        product0_.inStock as inStock2_1_,
        product0_.name as name3_1_,
        product0_.price as price4_1_
    from
        products product0_
    where
        product0_.price>5.0
Hibernate:
    select
        category0_.id as id1_0_0_,
        category0_.name as name2_0_0_,
        products1_.category_id as category5_1_1_,
        products1_.id as id1_1_1_,
        products1_.id as id1_1_2_,
        products1_.category_id as category5_1_2_,
        products1_.inStock as inStock2_1_2_,
        products1_.name as name3_1_2_,
        products1_.price as price4_1_2_
    from
        categories category0_
    left outer join
        products products1_
            on category0_.id=products1_.category_id
    where
        category0_.id=?
{ 1001, iPhone 6S, 699.00 }
{ 1002, Samsung Galaxy, 299.00 }
{ 1003, Designer Skirt, 49.00 }
{ 1004, Jeans, 78.99 }
{ 1005, Scarf, 19.99 }
{ 1006, Belt, 9.90 }
{ 1007, Sporinkler, 89.00 }
{ 1008, Notebook, 9.00 }
{ 1010, Diary of a Wimpy Kid, 5.99 }
```

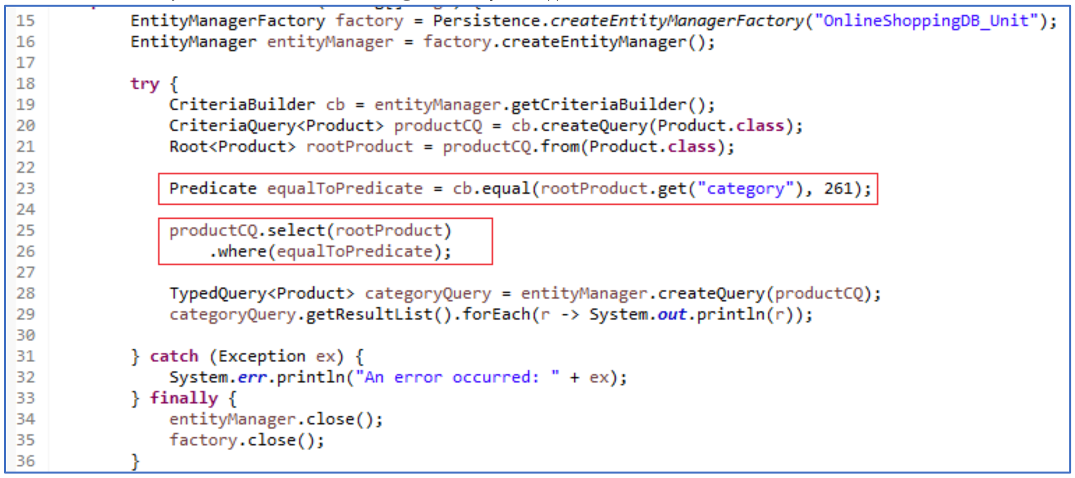
Here are all of the products. The iPhone 6S, Sprinkler, the notebook, all have price greater than 5.

## Predicates Condition

Conditions that you specify typically in the where clause are **Predicates**. I'm going to import the predicate library explicitly.

```
import javax.persistence.criteria.Predicate;
```

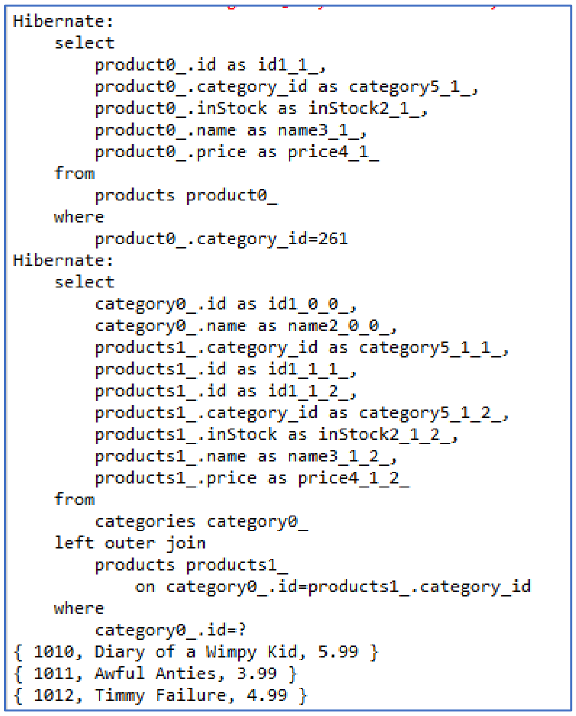
And let's set up a `Predicate` using `cb.equal()`.



```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {
            CriteriaBuilder cb = entityManager.getCriteriaBuilder();
            CriteriaQuery<Product> productCQ = cb.createQuery(Product.class);
            Root<Product> rootProduct = productCQ.from(Product.class);

            Predicate equalToPredicate = cb.equal(rootProduct.get("category"), 261);

            productCQ.select(rootProduct)
                .where(equalToPredicate);

            TypedQuery<Product> categoryQuery = entityManager.createQuery(productCQ);
            categoryQuery.getResultList().forEach(r -> System.out.println(r));

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

- Notice that I first use `cb.equal()` to check whether `"category"`is equal to 261.
- And I assigned that to a `Predicate` data type that is equal to `Predicate`.
- I then use this `equal()` to `Predicate` within my select from where clause.
  `select(rootProduct).where(equalToPredicate);` is satisfied.
  That is where category is equal to 261.

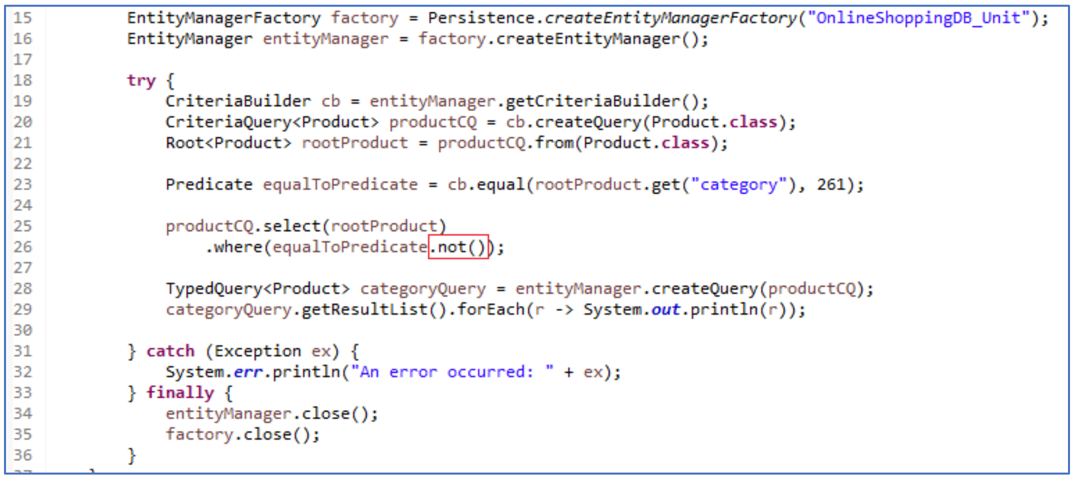
This makes it clear that the where clause accepts predicates. And here are the products that satisfy our query.



```
Hibernate: 
    select
        product0_.id as id1_1_,
        product0_.category_id as category5_1_,
        product0_.inStock as inStock2_1_,
        product0_.name as name3_1_,
        product0_.price as price4_1_ 
    from
        products product0_ 
    where
        product0_.category_id=261
Hibernate: 
    select
        category0_.id as id1_0_0_,
        category0_.name as name2_0_0_,
        products1_.category_id as category5_1_1_,
        products1_.id as id1_1_1_,
        products1_.id as id1_1_2_,
        products1_.category_id as category5_1_2_,
        products1_.inStock as inStock2_1_2_,
        products1_.name as name3_1_2_,
        products1_.price as price4_1_2_ 
    from
        categories category0_ 
    left outer join
        products products1_ 
            on category0_.id=products1_.category_id 
    where
        category0_.id=?
{ 1010, Diary of a Wimpy Kid, 5.99 }
{ 1011, Awful Anties, 3.99 }
{ 1012, Timmy Failure, 4.99 }
```

261 is the book category, and we have three books in our underlying database table.

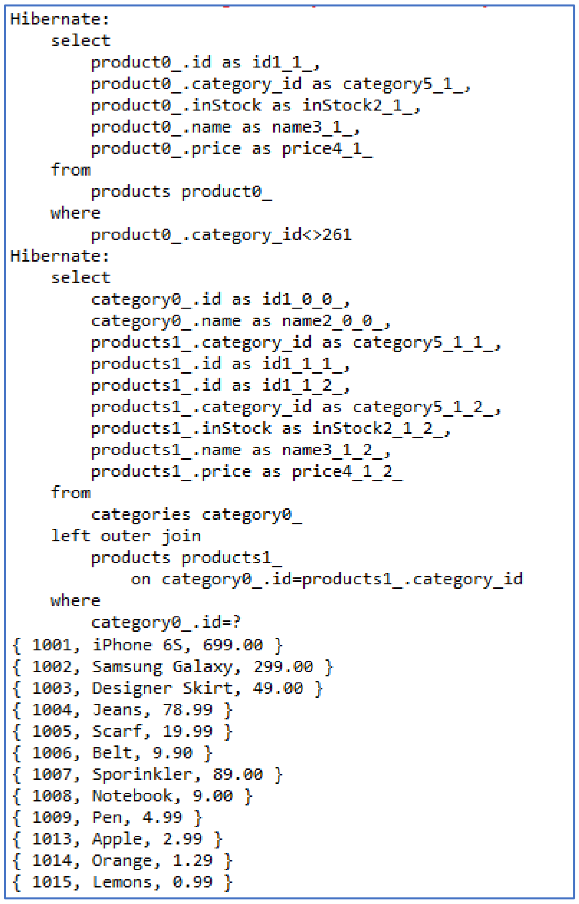
Once you have a predicate, you can also specify the negation of that predicate within your where clause. Here's our where clause. It's the same equal to predicate as before, but I invoke the dot not function to specify its negation.



```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {
            CriteriaBuilder cb = entityManager.getCriteriaBuilder();
            CriteriaQuery<Product> productCQ = cb.createQuery(Product.class);
            Root<Product> rootProduct = productCQ.from(Product.class);

            Predicate equalToPredicate = cb.equal(rootProduct.get("category"), 261);

            productCQ.select(rootProduct)
                .where(equalToPredicate.not());

            TypedQuery<Product> categoryQuery = entityManager.createQuery(productCQ);
            categoryQuery.getResultList().forEach(r -> System.out.println(r));

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

As a result, our query has changed. We want all products where the category is not equal to 261, where the category is not books.

Let's run this code.

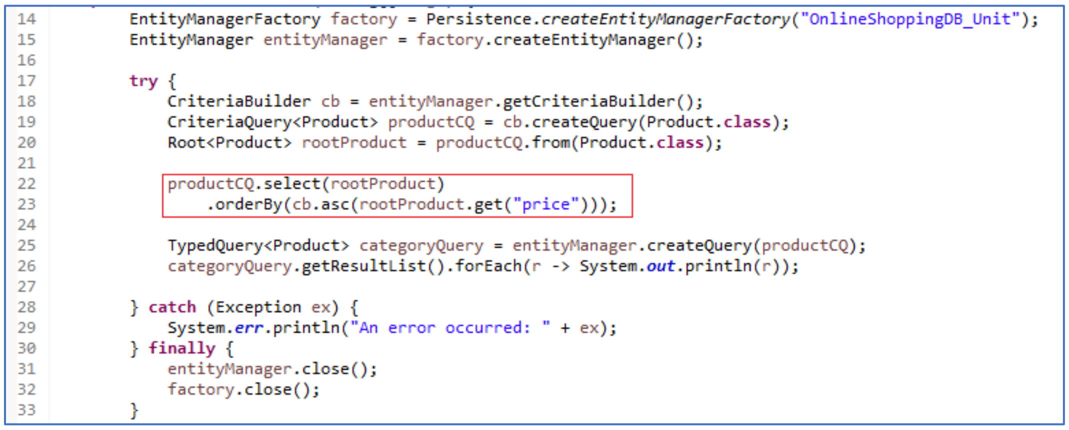


```
Hibernate:
    select
        product0_.id as id1_1_,
        product0_.category_id as category5_1_,
        product0_.inStock as inStock2_1_,
        product0_.name as name3_1_,
        product0_.price as price4_1_
    from
        products product0_
    where
        product0_.category_id<>261
Hibernate:
    select
        category0_.id as id1_0_0_,
        category0_.name as name2_0_0_,
        products1_.category_id as category5_1_1_,
        products1_.id as id1_1_1_,
        products1_.id as id1_1_2_,
        products1_.category_id as category5_1_2_,
        products1_.inStock as inStock2_1_2_,
        products1_.name as name3_1_2_,
        products1_.price as price4_1_2_
    from
        categories category0_
    left outer join
        products products1_
            on category0_.id=products1_.category_id
    where
        category0_.id=?
{ 1001, iPhone 6S, 699.00 }
{ 1002, Samsung Galaxy, 299.00 }
{ 1003, Designer Skirt, 49.00 }
{ 1004, Jeans, 78.99 }
{ 1005, Scarf, 19.99 }
{ 1006, Belt, 9.90 }
{ 1007, Sporinkler, 89.00 }
{ 1008, Notebook, 9.00 }
{ 1009, Pen, 4.99 }
{ 1013, Apple, 2.99 }
{ 1014, Orange, 1.29 }
{ 1015, Lemons, 0.99 }
```

And here are all of the products starting from the mobile phones category, fashion, etc. But the products corresponding to the book category haven't been returned.

## Sorting clause

Now that we've seen a few examples of using the where clause, let's set up a query with a slightly different clause. We'll use the **orderBy** clause.

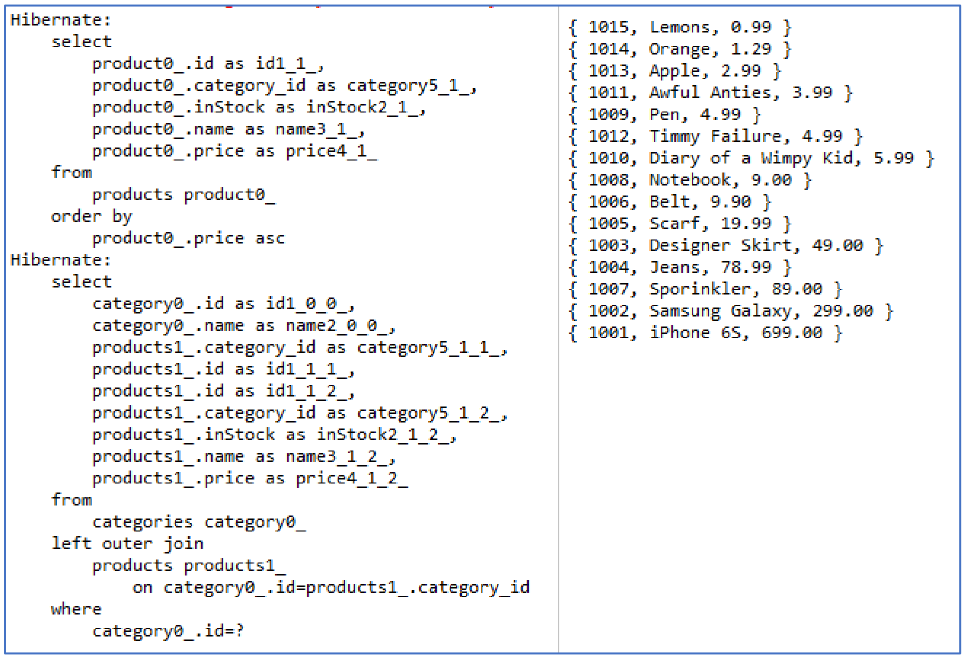


```
14          EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
15          EntityManager entityManager = factory.createEntityManager();
16
17          try {
18              CriteriaBuilder cb = entityManager.getCriteriaBuilder();
19              CriteriaQuery<Product> productCQ = cb.createQuery(Product.class);
20              Root<Product> rootProduct = productCQ.from(Product.class);
21
22              productCQ.select(rootProduct)
23                  .orderBy(cb.asc(rootProduct.get("price")));
24
25              TypedQuery<Product> categoryQuery = entityManager.createQuery(productCQ);
26              categoryQuery.getResultList().forEach(r -> System.out.println(r));
27
28          } catch (Exception ex) {
29              System.err.println("An error occurred: " + ex);
30          } finally {
31              entityManager.close();
32              factory.close();
33          }
```

- Once again I'm going to select products.
- Notice on line 27 I use `productCQ.select(rootProduct)`
- I've used the `.orderBy()` clause here and the `cb.asc()` method to get a predicate.
- I want my results ordered by `price`.

`cb.asc()` gives us an ordering predicate in the *ascending* order. `cb.desc()` will give us the corresponding predicate in the *descending* order.

Let's run this query,



```
Hibernate:
    select
        product0_.id as id1_1_,
        product0_.category_id as category5_1_,
        product0_.inStock as inStock2_1_,
        product0_.name as name3_1_,
        product0_.price as price4_1_
    from
        products product0_
    order by
        product0_.price asc
Hibernate:
    select
        category0_.id as id1_0_0_,
        category0_.name as name2_0_0_,
        products1_.category_id as category5_1_1_,
        products1_.id as id1_1_1_,
        products1_.id as id1_1_2_,
        products1_.category_id as category5_1_2_,
        products1_.inStock as inStock2_1_2_,
        products1_.name as name3_1_2_,
        products1_.price as price4_1_2_
    from
        categories category0_
    left outer join
        products products1_
            on category0_.id=products1_.category_id
    where
        category0_.id=?
```

```
{ 1015, Lemons, 0.99 }
{ 1014, Orange, 1.29 }
{ 1013, Apple, 2.99 }
{ 1011, Awful Anties, 3.99 }
{ 1009, Pen, 4.99 }
{ 1012, Timmy Failure, 4.99 }
{ 1010, Diary of a Wimpy Kid, 5.99 }
{ 1008, Notebook, 9.00 }
{ 1006, Belt, 9.90 }
{ 1005, Scarf, 19.99 }
{ 1003, Designer Skirt, 49.00 }
{ 1004, Jeans, 78.99 }
{ 1007, Sporinkler, 89.00 }
{ 1002, Samsung Galaxy, 299.00 }
{ 1001, iPhone 6S, 699.00 }
```

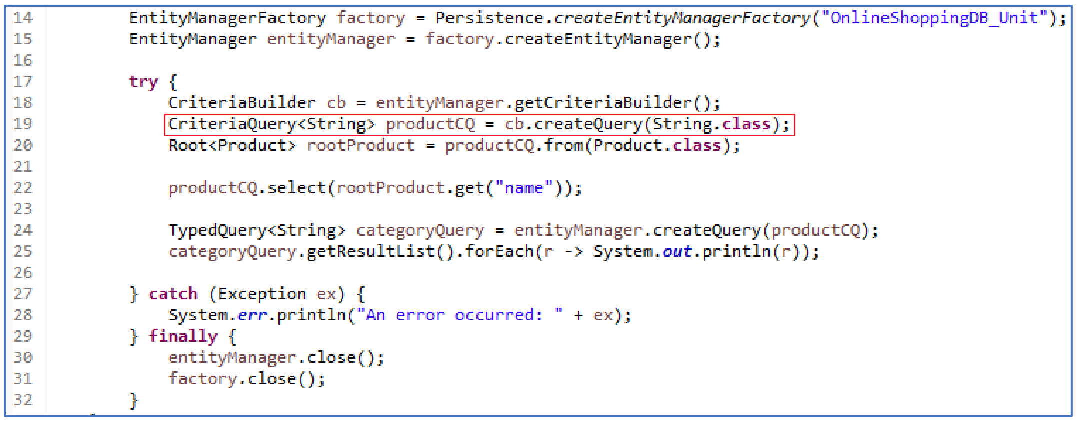
and you can see the lowest price item, a pen at 4.99, is listed first. Then we have a couple of books, and if you scroll down to the very bottom, you'll find the iPhone and the Samsung Galaxy phones.

## Running Advanced Criteria Queries

So far the criteria queries that we've been working with have returned entities in their results.

### Return Single Field

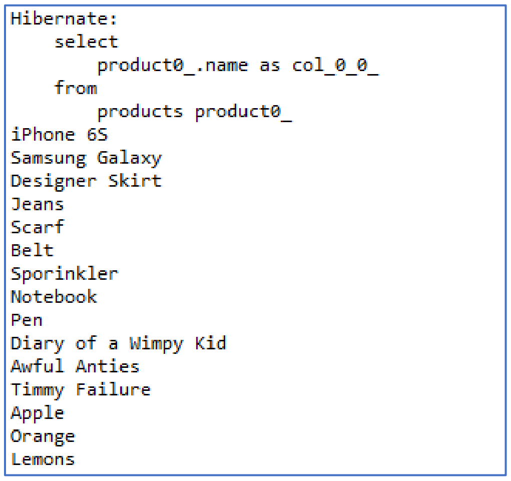
Here we'll look at an example of a criteria query that returns the value in a single field. There are a bunch of changes that I'd like you to observe here. Take a look at line 19.



```
14          EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
15          EntityManager entityManager = factory.createEntityManager();
16
17          try {
18              CriteriaBuilder cb = entityManager.getCriteriaBuilder();
19              CriteriaQuery<String> productCQ = cb.createQuery(String.class);
20              Root<Product> rootProduct = productCQ.from(Product.class);
21
22              productCQ.select(rootProduct.get("name"));
23
24              TypedQuery<String> categoryQuery = entityManager.createQuery(productCQ);
25              categoryQuery.getResultList().forEach(r -> System.out.println(r));
26
27          } catch (Exception ex) {
28              System.err.println("An error occurred: " + ex);
29          } finally {
30              entityManager.close();
31              factory.close();
32          }
```

- When you use the `CriteriaBuilder` to create the criteria query, you need to specify the class of the return type. We are going to have our query return **`String`**s, so call `createQuery` with `String.`**`class`**.
- We are still querying the product table here. So the `Root Product` remains the same. Pass in `Product.`**`class`** to the from method.
- We perform a select operation. Specify the field that you want selected, `rootProduct.get("name")`. That is the field. The *name* attribute of the `Product`.
- We then use this `criteriaQuery` that we have set up to create a `TypedQuery`. The `TypedQuery` expects a `String` result.
- We call `query.getResultList()`. And print out the strings that have been returned.

And when you run this query,

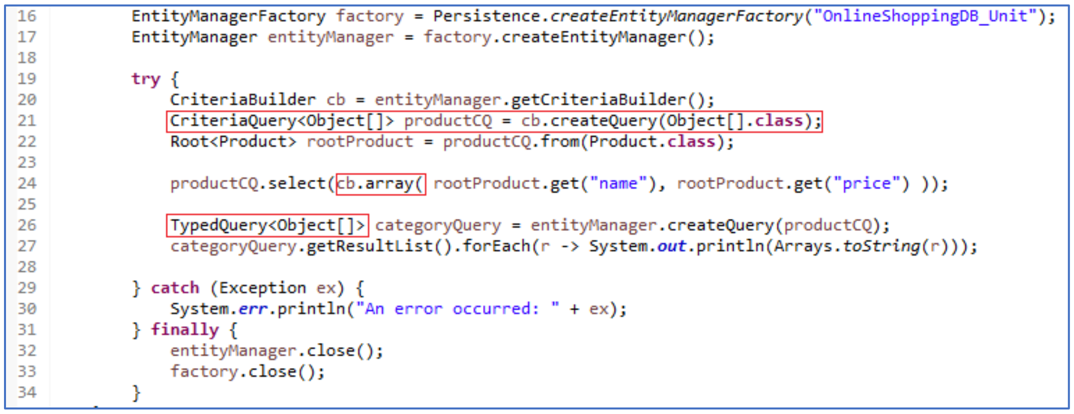


```
Hibernate:
    select
        product0_.name as col_0_0_
    from
        products product0_
iPhone 6S
Samsung Galaxy
Designer Skirt
Jeans
Scarf
Belt
Sporinkler
Notebook
Pen
Diary of a Wimpy Kid
Awful Anties
Timmy Failure
Apple
Orange
Lemons
```

You'll see the names of all of the products. Starting from the iPhone and ending at Apples, Oranges, and Lemons.

## Return Multiple Fields

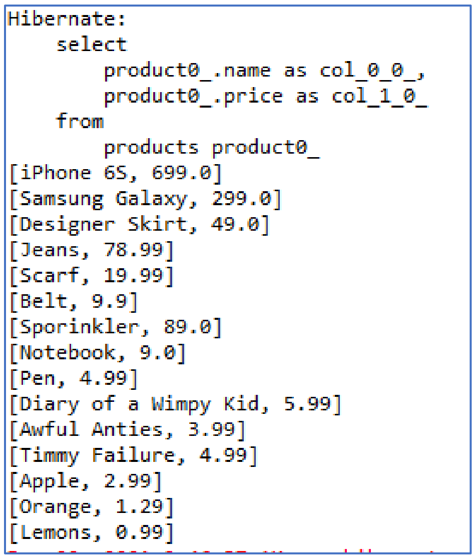
It's also possible to use the `CriteriaQuery` to select and return multiple field values. Once again, there are a bunch of small changes that you need to pay attention to.



```
16          EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
17          EntityManager entityManager = factory.createEntityManager();
18
19          try {
20              CriteriaBuilder cb = entityManager.getCriteriaBuilder();
21              CriteriaQuery<Object[]> productCQ = cb.createQuery(Object[].class);
22              Root<Product> rootProduct = productCQ.from(Product.class);
23
24              productCQ.select(cb.array( rootProduct.get("name"), rootProduct.get("price") ));
25
26              TypedQuery<Object[]> categoryQuery = entityManager.createQuery(productCQ);
27              categoryQuery.getResultList().forEach(r -> System.out.println(Arrays.toString(r)));
28
29          } catch (Exception ex) {
30              System.err.println("An error occurred: " + ex);
31          } finally {
32              entityManager.close();
33              factory.close();
34          }
```

- Notice on line 21, when we create the `CriteriaQuery`, we create the query using the **`Object array`** class.
- When you select multiple fields for every record, an array of objects is returned. And that's why we create the query by passing the Object array class.
- The `Root Product` remains the same.
- What's different is the way we have specified the `select()` clause.
  We're selecting two columns here, a `name` and `price`. That is two fields I want these columns in an array, *an array of objects*. So I use `cb.array()`. That is a helper method.
  `cb.array()` will return the values in these two columns in the form of an `Object Array`.
- I use `entityManager.createQuery()` using the criteria query.
- The `TypedQuery` will return a `List` of `Object Arrays`.
- And I then print out these arrays to screen using `Arrays.toString()`.

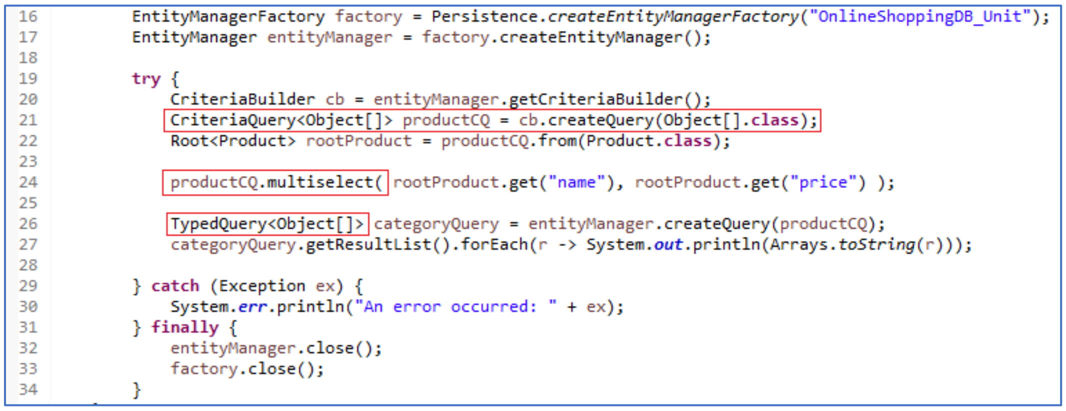
Time to run this code and see the results.



```
Hibernate:
    select
        product0_.name as col_0_0_,
        product0_.price as col_1_0_
    from
        products product0_
[iPhone 6S, 699.0]
[Samsung Galaxy, 299.0]
[Designer Skirt, 49.0]
[Jeans, 78.99]
[Scarf, 19.99]
[Belt, 9.9]
[Sporinkler, 89.0]
[Notebook, 9.0]
[Pen, 4.99]
[Diary of a Wimpy Kid, 5.99]
[Awful Anties, 3.99]
[Timmy Failure, 4.99]
[Apple, 2.99]
[Orange, 1.29]
[Lemons, 0.99]
```

Notice the `name` and `price` for every product in our underlying database table are now displayed here within this console window.

For multiple field selection, rather than use `cb.array()` to create an object array which holds the multiple field results. You can use the `multiselect()` option.

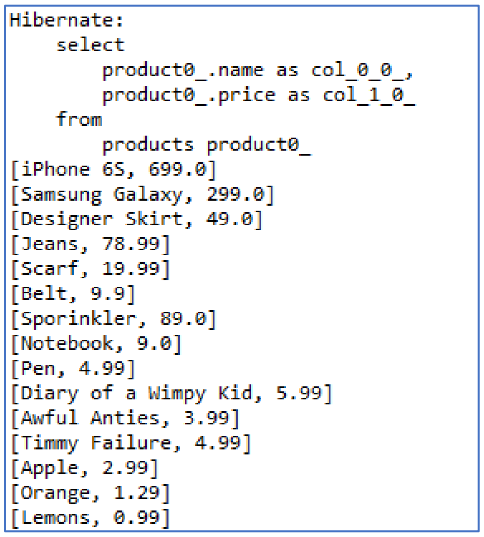


```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {
            CriteriaBuilder cb = entityManager.getCriteriaBuilder();
            CriteriaQuery<Object[]> productCQ = cb.createQuery(Object[].class);
            Root<Product> rootProduct = productCQ.from(Product.class);

            productCQ.multiselect( rootProduct.get("name"), rootProduct.get("price") );

            TypedQuery<Object[]> categoryQuery = entityManager.createQuery(productCQ);
            categoryQuery.getResultList().forEach(r -> System.out.println(Arrays.toString(r)));

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

```
productCQ.multiselect( rootProduct.get("name"), rootProduct.get("price") );
```

The `multiselect()` method in the criteria query API allows you to select multiple fields in a single query. The fields remain the same, the `name` and `price` for every `Product`. But the query itself is simpler and easier to read and maintain.

An `array` of `Objects` is still returned. We haven't explicitly specified that though using `cb.array()`.

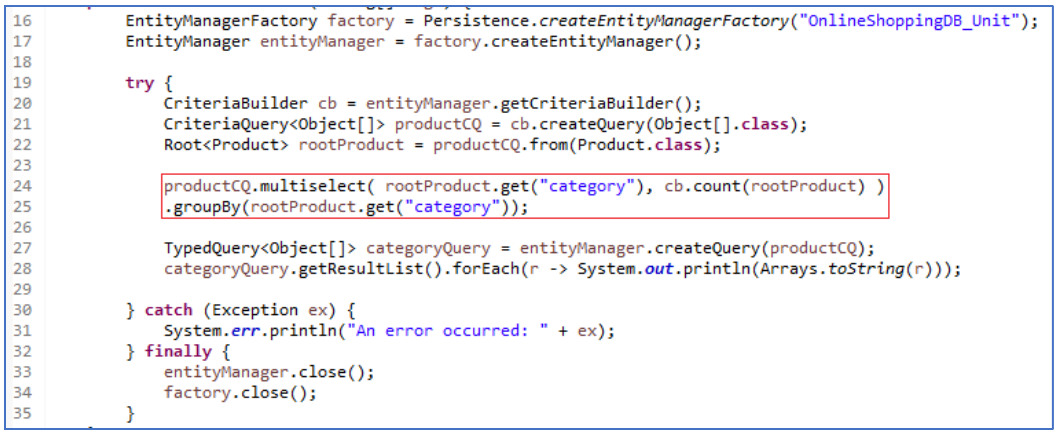


```
Hibernate:
    select
        product0_.name as col_0_0_,
        product0_.price as col_1_0_
    from
        products product0_
[iPhone 6S, 699.0]
[Samsung Galaxy, 299.0]
[Designer Skirt, 49.0]
[Jeans, 78.99]
[Scarf, 19.99]
[Belt, 9.9]
[Sporinkler, 89.0]
[Notebook, 9.0]
[Pen, 4.99]
[Diary of a Wimpy Kid, 5.99]
[Awful Anties, 3.99]
[Timmy Failure, 4.99]
[Apple, 2.99]
[Orange, 1.29]
[Lemons, 0.99]
```

If you run this code, you can see the name and price for every product will be displayed within the console window.

## Group By Clause

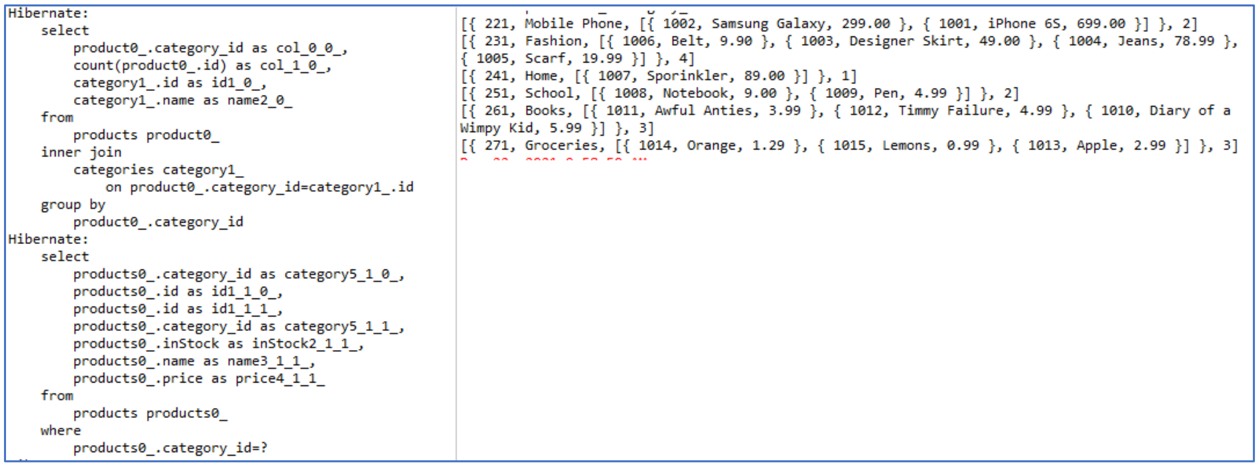
We'll now move on to our last example here using the criteria API. This will be a query that performs a groupBy operation.



```java
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {
            CriteriaBuilder cb = entityManager.getCriteriaBuilder();
            CriteriaQuery<Object[]> productCQ = cb.createQuery(Object[].class);
            Root<Product> rootProduct = productCQ.from(Product.class);

            productCQ.multiselect( rootProduct.get("category"), cb.count(rootProduct) )
            .groupBy(rootProduct.get("category"));

            TypedQuery<Object[]> categoryQuery = entityManager.createQuery(productCQ);
            categoryQuery.getResultList().forEach(r -> System.out.println(Arrays.toString(r)));

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

- This is a multiselect query.
- We are selecting multiple fields. The first is the `category` field. And the second field is an *aggregate*.
- The category field we access using `rootProduct.get("category")`
- The count aggregation we perform using `cb.count(rootProduct)`.
  For every category, we want a count of products in that category.
- The `groupBy()` method in the criteria query allows us to perform the grouping operation.
- Because we're selecting multiple fields, these fields are returned as an `Array` of `Object` types.

Let's run this code.



```
Hibernate:
    select
        product0_.category_id as col_0_0_,
        count(product0_.id) as col_1_0_,
        category1_.id as id1_0_,
        category1_.name as name2_0_
    from
        products product0_
    inner join
        categories category1_
            on product0_.category_id=category1_.id
    group by
        product0_.category_id
Hibernate:
    select
        products0_.category_id as category5_1_0_,
        products0_.id as id1_1_0_,
        products0_.id as id1_1_1_,
        products0_.category_id as category5_1_1_,
        products0_.inStock as inStock2_1_1_,
        products0_.name as name3_1_1_,
        products0_.price as price4_1_1_
    from
        products products0_
    where
        products0_.category_id=?
```

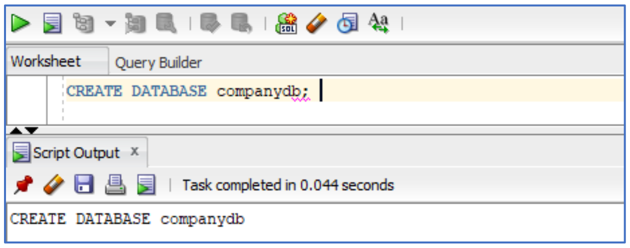
```
[{ 221, Mobile Phone, [{ 1002, Samsung Galaxy, 299.00 }, { 1001, iPhone 6S, 699.00 }] }, 2]
[{ 231, Fashion, [{ 1006, Belt, 9.90 }, { 1003, Designer Skirt, 49.00 }, { 1004, Jeans, 78.99 },
{ 1005, Scarf, 19.99 }] }, 4]
[{ 241, Home, [{ 1007, Sporinkler, 89.00 }] }, 1]
[{ 251, School, [{ 1008, Notebook, 9.00 }, { 1009, Pen, 4.99 }] }, 2]
[{ 261, Books, [{ 1011, Awful Anties, 3.99 }, { 1012, Timmy Failure, 4.99 }, { 1010, Diary of a
Wimpy Kid, 5.99 }] }, 3]
[{ 271, Groceries, [{ 1014, Orange, 1.29 }, { 1015, Lemons, 0.99 }, { 1013, Apple, 2.99 }] }, 3]
```

And if you scroll down to the very bottom, you'll find `Categories` and their count of `Products`. There are two `products` within the "Mobile Phones" `category`. You can see the two there. You can scroll down further. Let's take a look at the "Home" `category` here with exactly one `product`. And the "School" `category` with two `products`.

The criteria API is extremely powerful. And you can construct very complex queries. Our explanation of the criteria API here has simply scratched the surface. There is a lot more to the criteria API.

## Working With Persistence Callbacks

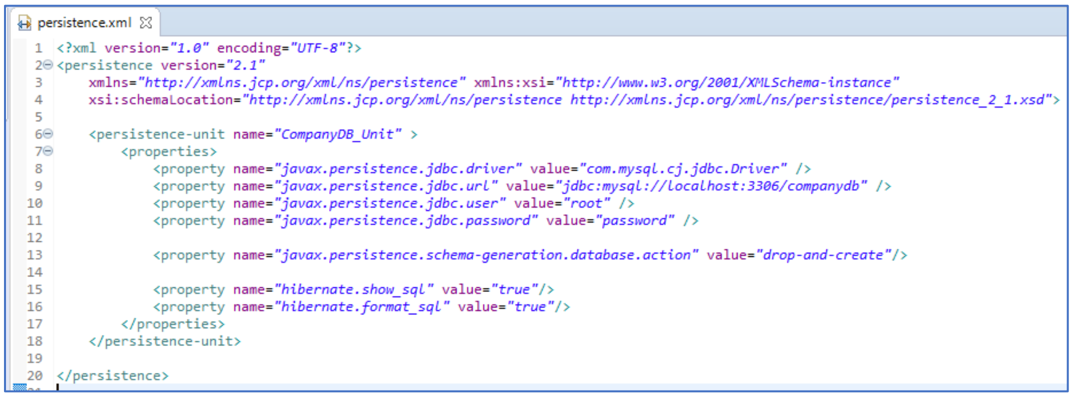
In this demo and in the next few demos, we'll understand how entity listeners and callback methods work in JPA. Now we'll work with a new database, the *companydb*. We've already created this database before for some of our earlier demos, but you can always recreate it if you need to.



```
CREATE DATABASE companydb;
```

```
CREATE DATABASE companydb
```

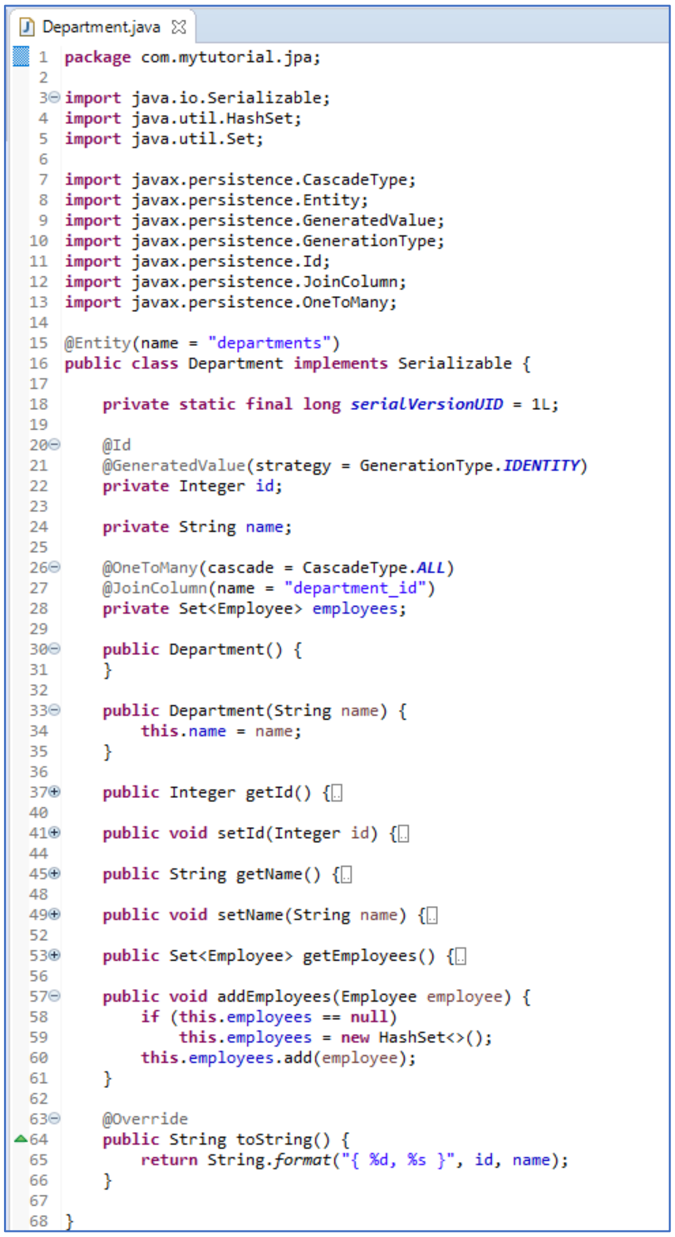
This is what our **persistence.xml** file will look like. The persistence-unit is the *CompanyDB_Unit*.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
    xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">

    <persistence-unit name="CompanyDB_Unit" >
        <properties>
            <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/companydb" />
            <property name="javax.persistence.jdbc.user" value="root" />
            <property name="javax.persistence.jdbc.password" value="password" />

            <property name="javax.persistence.schema-generation.database.action" value="drop-and-create"/>

            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
        </properties>
    </persistence-unit>

</persistence>
```

The database that we'll connect to his companydb.
We'll use the database action drop-and-create to recreate database tables each time we run our application.

The entities that we work with we worked with earlier in this learning path. Here is a **Department** which has a one to many relationship with employees.

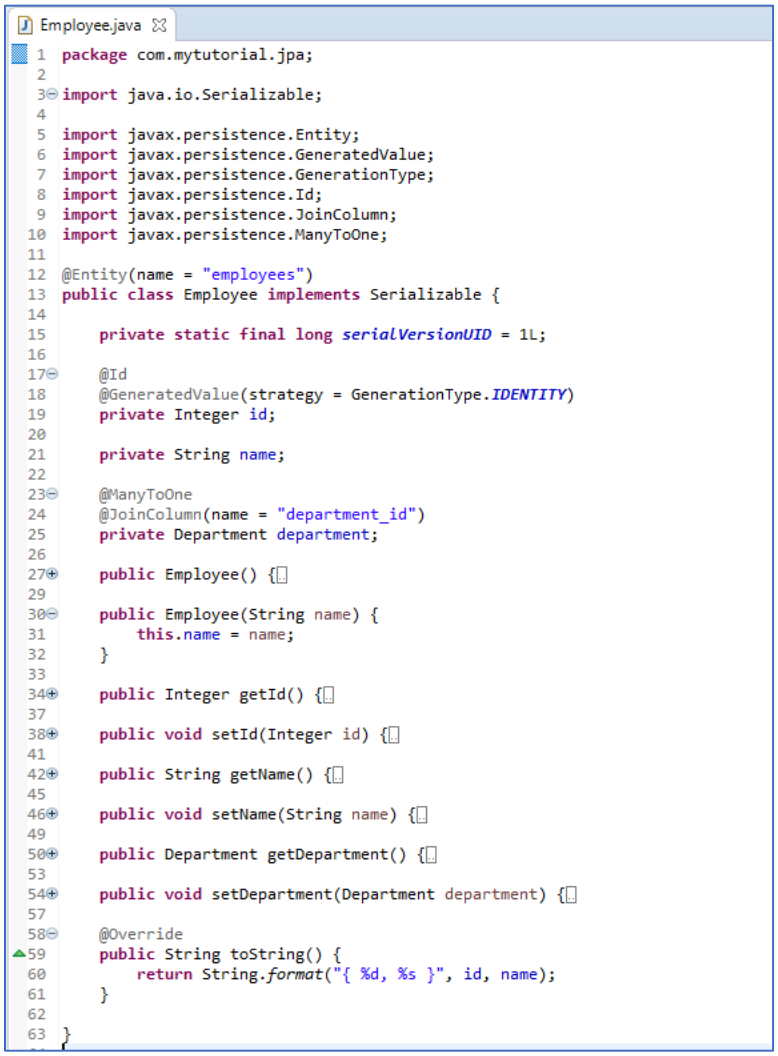


Department.java

```
package com.mytutorial.jpa;

import java.io.Serializable;
import java.util.HashSet;
import java.util.Set;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToMany;

@Entity(name = "departments")
public class Department implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String name;

    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "department_id")
    private Set<Employee> employees;

    public Department() {
    }

    public Department(String name) {
        this.name = name;
    }

    public Integer getId() {

    public void setId(Integer id) {

    public String getName() {

    public void setName(String name) {

    public Set<Employee> getEmployees() {

    public void addEmployees(Employee employee) {
        if (this.employees == null)
            this.employees = new HashSet<>();
        this.employees.add(employee);
    }

    @Override
    public String toString() {
        return String.format("{ %d, %s }", id, name);
    }

}
```

- The "departments" is an entity and department objects will be persisted in our underlying database tables.
- Every department record will contain an id that is the **primary key**, name, and a Department is mapped to a Set of employees.
  It's a one to many mapping, and we represent this mapping in the underlying database using a @JoinColumn. Every Employee has a department_id which is the **foreign key** into the Department's table.
- Set up the **getters** and **setters**, nothing really new there. All of this is code that we are familiar with.

Let's head over to the **Employee.java** file and set up the Employee entity.

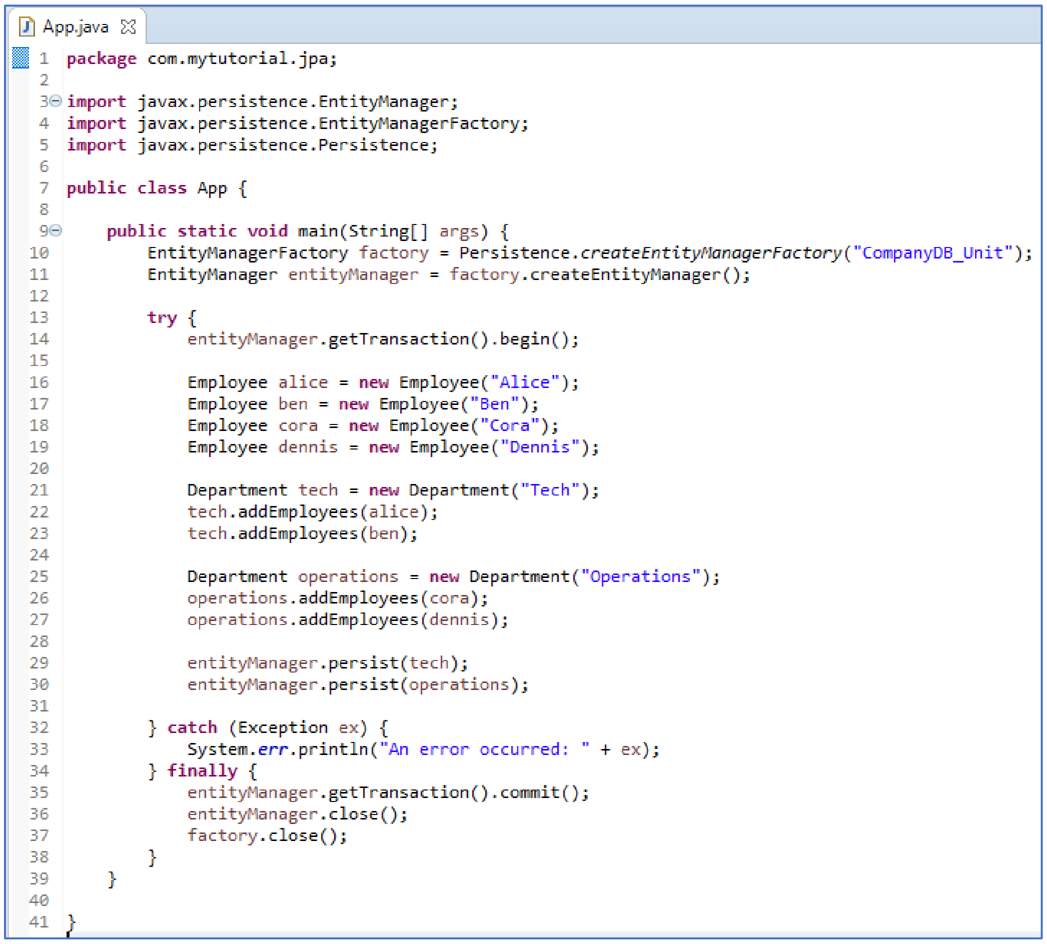


Employee.java

```java
package com.mytutorial.jpa;

import java.io.Serializable;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;

@Entity(name = "employees")
public class Employee implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String name;

    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;

    public Employee() {

    public Employee(String name) {
        this.name = name;
    }

    public Integer getId() {

    public void setId(Integer id) {

    public String getName() {

    public void setName(String name) {

    public Department getDepartment() {

    public void setDepartment(Department department) {

    @Override
    public String toString() {
        return String.format("{ %d, %s }", id, name);
    }

}
```

- Set up the **@Entity** annotation, make sure `Employee` `implements` the `Serializable` `interface`, and let's set up the member variables for the `Employee`.
- We'll have our table store the primary key that is the `id` of an `Employee`, the `name` of an `Employee`, and the `department` to which the `Employee` belongs. This is a many to one mapping because many `Employee` can belong to the same `Department`.
- The rest of the code include **`constructors`**, **`getters`**, **`setters`**, **`toString()`** representations, all stuff that you know.

Let's head over to **App.java** and create and persist a few entities.

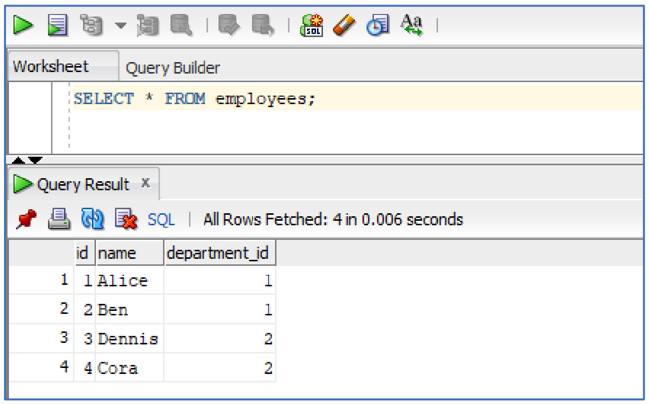


App.java

```
package com.mytutorial.jpa;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class App {

    public static void main(String[] args) {
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("CompanyDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {
            entityManager.getTransaction().begin();

            Employee alice = new Employee("Alice");
            Employee ben = new Employee("Ben");
            Employee cora = new Employee("Cora");
            Employee dennis = new Employee("Dennis");

            Department tech = new Department("Tech");
            tech.addEmployees(alice);
            tech.addEmployees(ben);

            Department operations = new Department("Operations");
            operations.addEmployees(cora);
            operations.addEmployees(dennis);

            entityManager.persist(tech);
            entityManager.persist(operations);

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.getTransaction().commit();
            entityManager.close();
            factory.close();
        }
    }

}
```

- I'm going to create four employees Alice, Ben, Cora, and Dennis, two departments, tech and operations.
- I'll have Alice and Ben be in tech, Cora and Dennis in operations.
- And I'll persist tech and operations and because of cascade type all the corresponding employees will also be persisted.

Let's run this code, head over to MySQL and make sure the databases are populated.
Do a *select * from employees*,



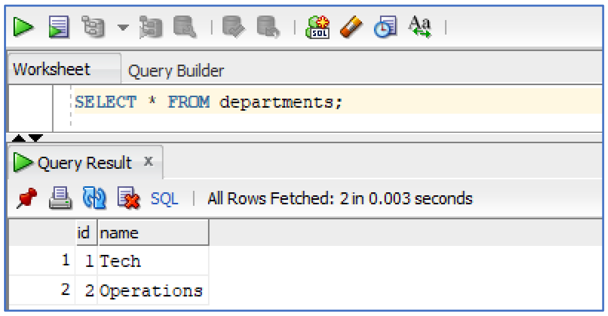
```
SELECT * FROM employees;
```

All Rows Fetched: 4 in 0.006 seconds

|   | id | name | department_id |
|---|---|---|---|
| 1 | 1 | Alice | 1 |
| 2 | 2 | Ben | 1 |
| 3 | 3 | Dennis | 2 |
| 4 | 4 | Cora | 2 |

you can see that four employee records are present here.

Let's do a *select * from* the *department*'s table,



```
SELECT * FROM departments;
```

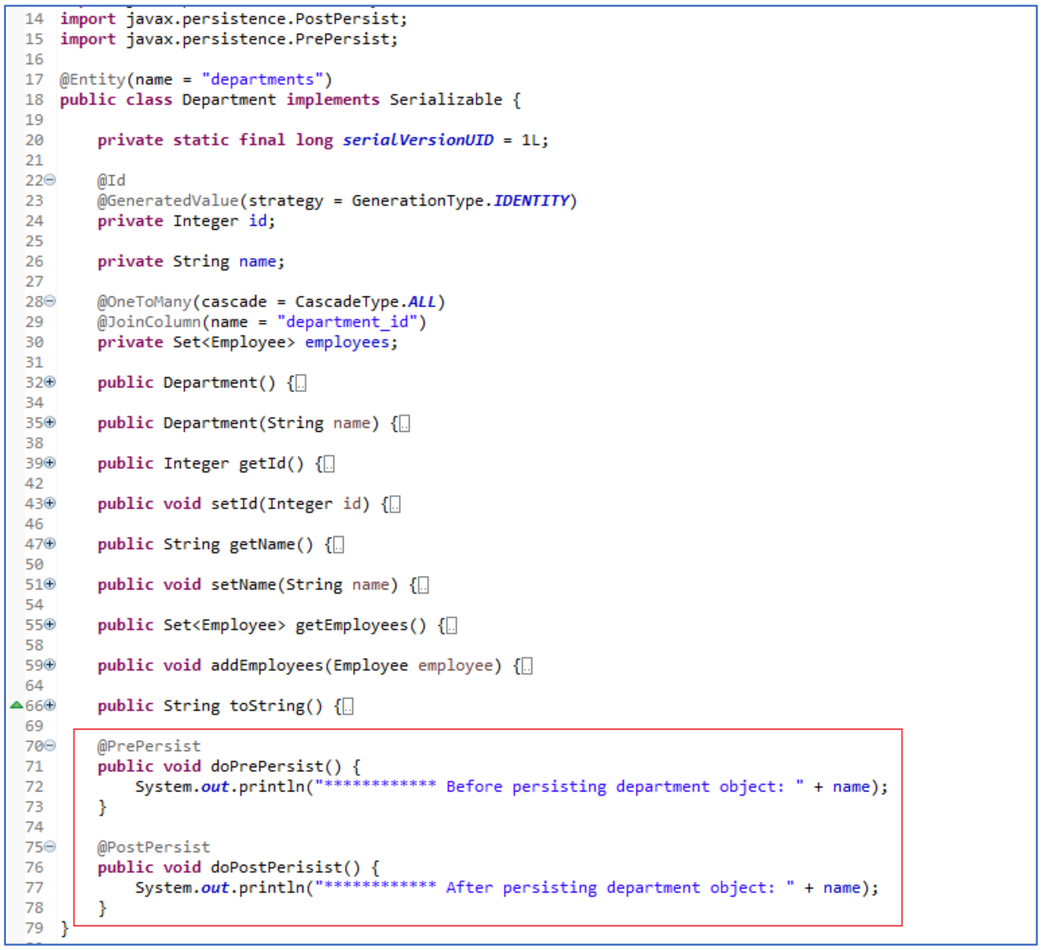
|   | id | name |
|---|---|---|
| 1 | 1 | Tech |
| 2 | 2 | Operations |

and you can see we have two departments, “Tech” and “Operations”.

When you're working with entities in JPA, it's quite possible that at certain points in your entitiy's life cycle you want some code to run. Maybe just before you persist an entity, or just after you persist an entity, maybe just after you've updated an already existing entity, or maybe just after you've removed an entity.

It's possible for you to **hook code** into the different parts of your entity lifecycle using **callback methods**. **Callback methods** are a way that we can be notified or called during certain events in the entities' life cycle. It's possible for you to specify these callback methods within an entity using annotations.

We'll explore the first two annotations here, **post persist** and **pre-persist**.



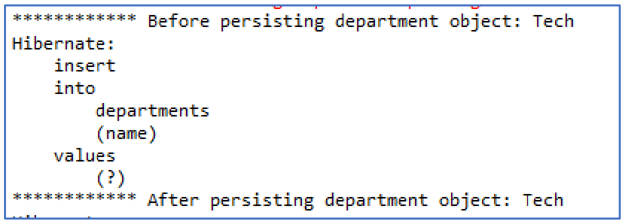
```java
import javax.persistence.PostPersist;
import javax.persistence.PrePersist;

@Entity(name = "departments")
public class Department implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String name;

    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "department_id")
    private Set<Employee> employees;

    public Department() {

    public Department(String name) {

    public Integer getId() {

    public void setId(Integer id) {

    public String getName() {

    public void setName(String name) {

    public Set<Employee> getEmployees() {

    public void addEmployees(Employee employee) {

    public String toString() {

    @PrePersist
    public void doPrePersist() {
        System.out.println("************ Before persisting department object: " + name);
    }

    @PostPersist
    public void doPostPerisist() {
        System.out.println("************ After persisting department object: " + name);
    }
}
```

- Callbacks that are invoked just before an object is persisted and just after an object is persisted. I've declared a method here within my department object `onPrePersist()`.
  This is code that will be called just before a department is persisted out to the underlying table.
- `onPostPersist()` will be called just after a department object is persisted.
- Notice the on **@prePersist** and on **@postPersist** annotations on these callbacks.

You can execute meaningful business logic, maybe perform some checks on your entities before and after they've been written out to the table. For the purposes of this demo, I've kept things simple and just printed out stuff to screen, the name of the department that we are persisting.

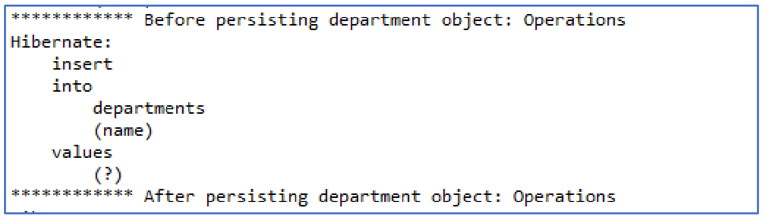
Now let's head over to App.java and run the same code as before where we had set up two departments tech and operations and persisted those out.

When you run this code, you'll see that our callbacks have been invoked at the right point in the object's lifecycle. Here is our print statement before the tech department entity has been persisted.

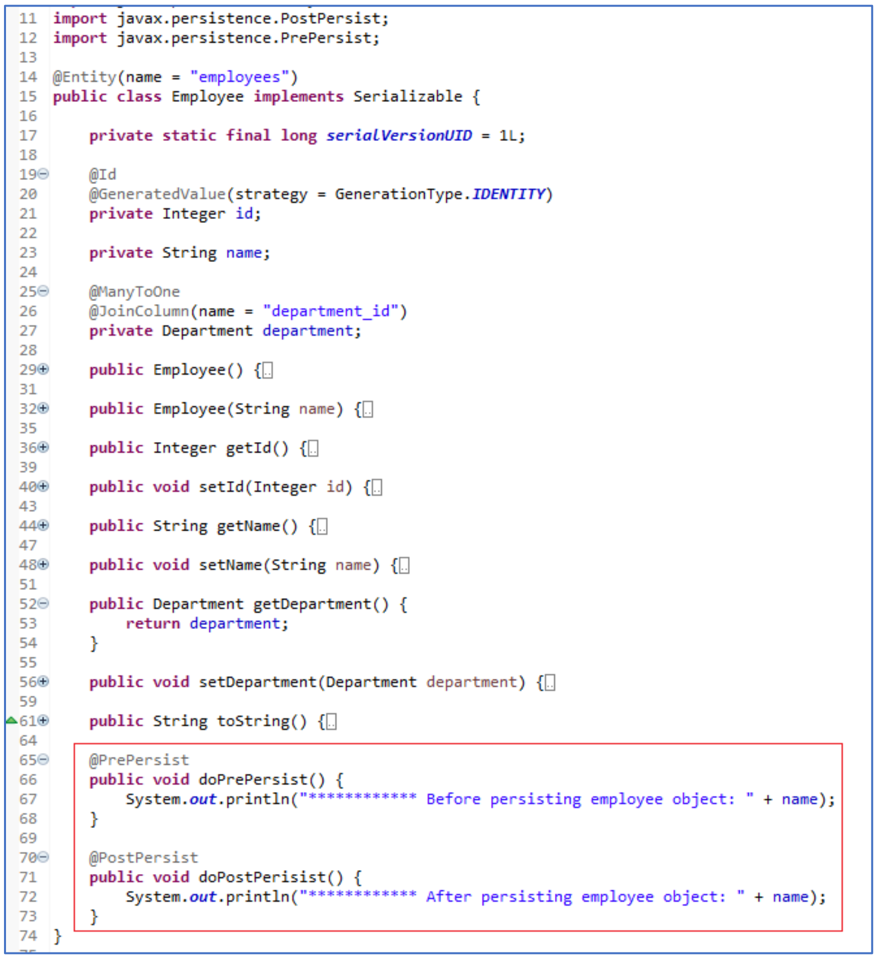

```
************ Before persisting department object: Tech
Hibernate:
    insert
    into
        departments
        (name)
    values
        (?)
************ After persisting department object: Tech
```

You can see immediately after that, we have an insert operation.
Just after the insert operation, our post persist code runs after persisting department object.

After tech, we persist the operations department. Here is the code invoke just before persistence and here is the code invoke just after persisting the operations entity.


```
************ Before persisting department object: Operations
Hibernate:
    insert
    into
        departments
        (name)
    values
        (?)
************ After persisting department object: Operations
```

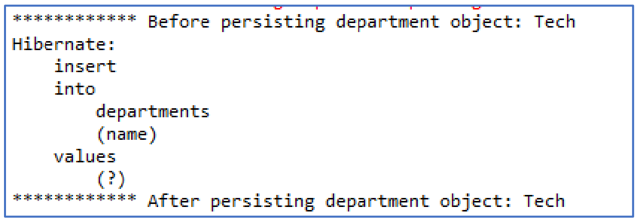
I'm now going to head over to **Employee.java** and add in callbacks for `PrePersist` and `PostPersist` to this entity as well.



```java
import javax.persistence.PostPersist;
import javax.persistence.PrePersist;

@Entity(name = "employees")
public class Employee implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String name;

    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;

    public Employee() {

    public Employee(String name) {

    public Integer getId() {

    public void setId(Integer id) {

    public String getName() {

    public void setName(String name) {

    public Department getDepartment() {
        return department;
    }

    public void setDepartment(Department department) {

    public String toString() {

    @PrePersist
    public void doPrePersist() {
        System.out.println("************ Before persisting employee object: " + name);
    }

    @PostPersist
    public void doPostPerisist() {
        System.out.println("************ After persisting employee object: " + name);
    }
}
```

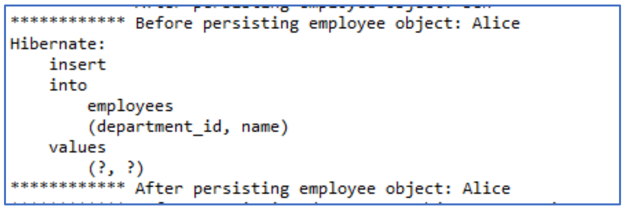
Once again, we'll keep the code simple just print statements. In each case, we'll print out the name of the employee before persisting the employee object and after persisting the employee object. Note the annotations `@PrePersist` and `@PostPersist`.

Let's switch over to App.java and run the same code as earlier. Now when we persist entities, we persist the parent entities. The department entity is persisted first.



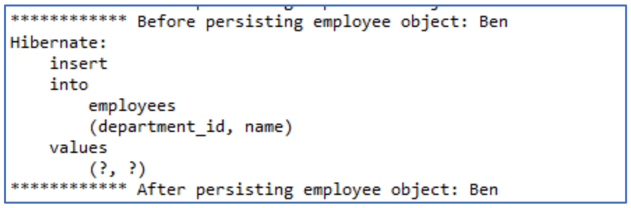
```
************ Before persisting department object: Tech
Hibernate:
    insert
    into
        departments
        (name)
    values
        (?)
************ After persisting department object: Tech
```

Here is the code for before we persist the department object, we scroll down below. Here is the code for after persisting the department object after which we persist the employee Alice, the first tech employee.



```
************ Before persisting employee object: Alice
Hibernate:
    insert
    into
        employees
        (department_id, name)
    values
        (?, ?)
************ After persisting employee object: Alice
```

We then persist the employee Ben, the second tech employee.

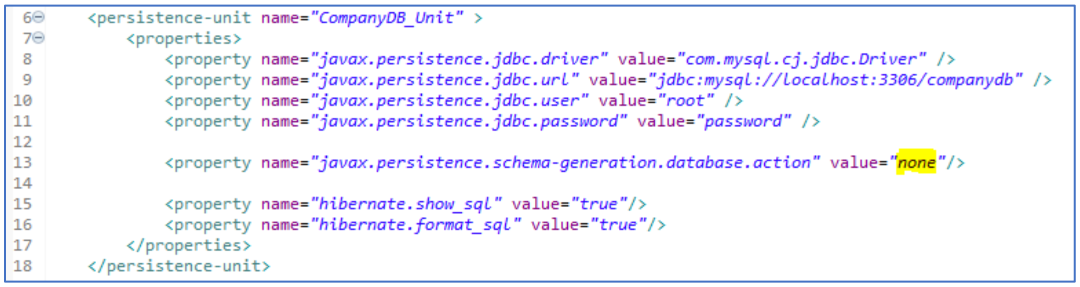


```
************ Before persisting employee object: Ben
Hibernate:
    insert
    into
        employees
        (department_id, name)
    values
        (?, ?)
************ After persisting employee object: Ben
```

Once Alice and Ben have been persisted, the post persist code runs. After persisting the employee object Alice and then after persisting the employee object Ben. These callbacks give us great insight into the order in which each of the objects are persisted.

## Working With Load Callbacks

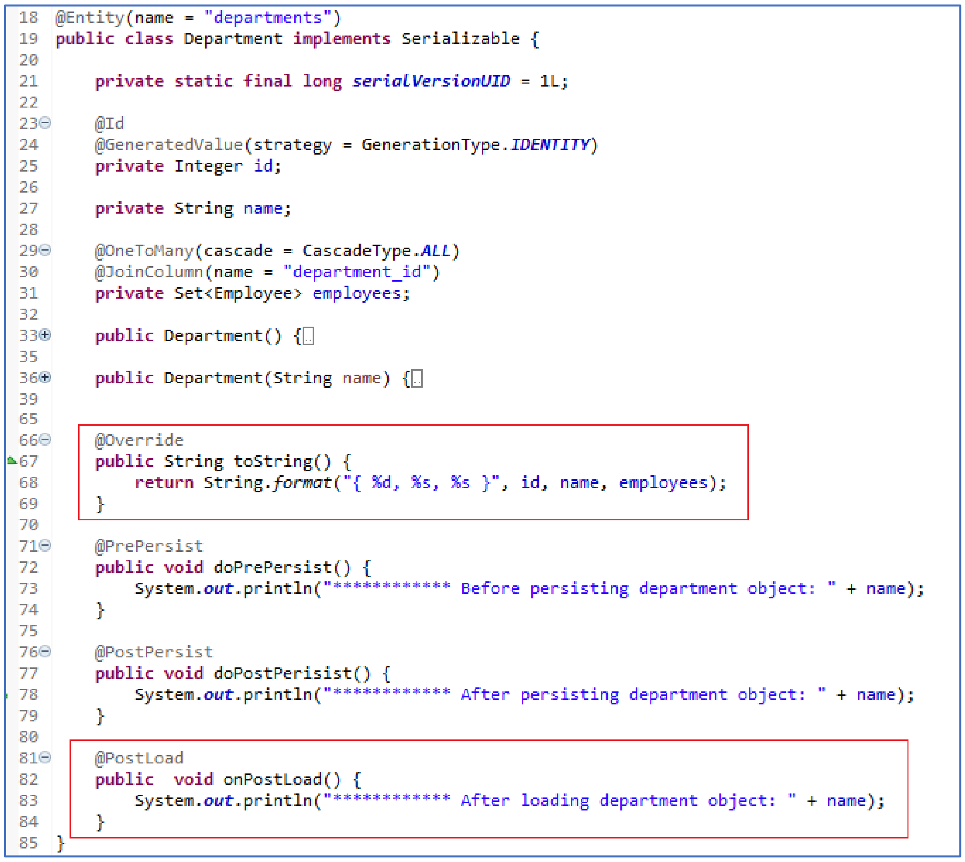
Here I am, back on the persistence.xml file. For this demo and for the next few demos, I'm going to change the database action to no longer be drop-and-create and instead be none. I don't want the database tables to be recreated each time I run my application.



```
    <persistence-unit name="CompanyDB_Unit" >
        <properties>
            <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/companydb" />
            <property name="javax.persistence.jdbc.user" value="root" />
            <property name="javax.persistence.jdbc.password" value="password" />

            <property name="javax.persistence.schema-generation.database.action" value="none"/>

            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
        </properties>
    </persistence-unit>
```

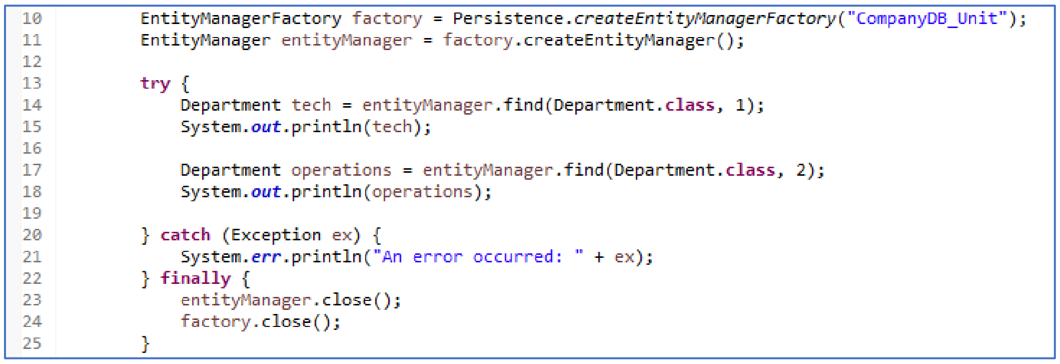
Now let's look at some of the other callback method that we can set up as a part of our entity life cycle. Now, each time you load an object, it's possible to **hook** some code in, and run that code each time you load an entity from the underlying table.

Make sure you annotate your method using **@PostLoad**. I have a method within my department entity called `onPostLoad()`, and I've annotated it with **@PostLoad**. And this code will run after you've loaded the `Department` object into your application, and will print out the name of the `Department`.



```
@Entity(name = "departments")
public class Department implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String name;

    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "department_id")
    private Set<Employee> employees;

    public Department() {

    public Department(String name) {


    @Override
    public String toString() {
        return String.format("{ %d, %s, %s }", id, name, employees);
    }

    @PrePersist
    public void doPrePersist() {
        System.out.println("************ Before persisting department object: " + name);
    }

    @PostPersist
    public void doPostPerisist() {
        System.out.println("************ After persisting department object: " + name);
    }

    @PostLoad
    public  void onPostLoad() {
        System.out.println("************ After loading department object: " + name);
    }
}
```

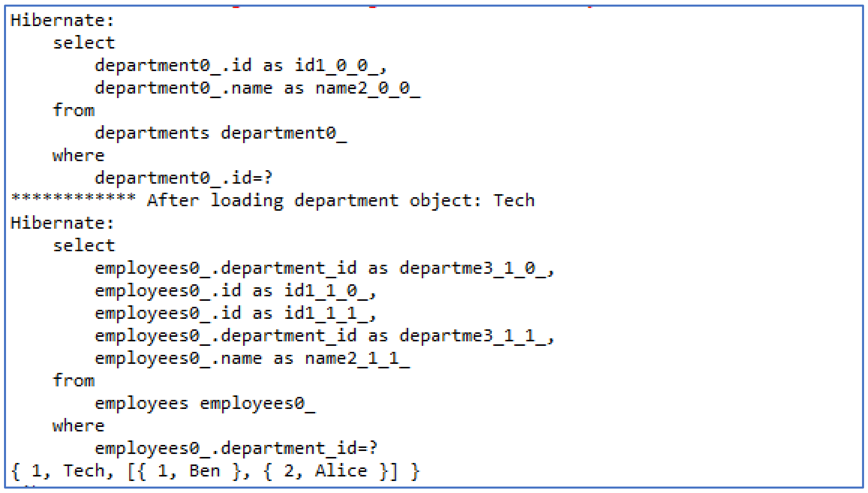
Let's head over to App.java and change the code that we have written here.



```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("CompanyDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {
            Department tech = entityManager.find(Department.class, 1);
            System.out.println(tech);

            Department operations = entityManager.find(Department.class, 2);
            System.out.println(operations);

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

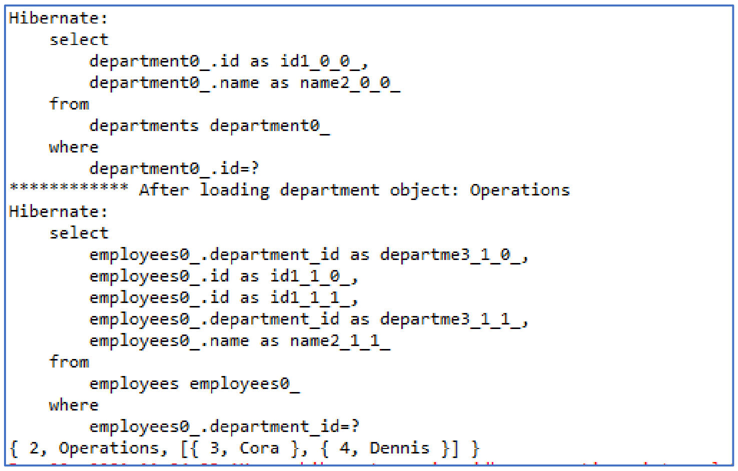
- I'll use the `find()` method to retrieve the two `Department` entities that we've persisted to the underlying tables. `entityManager.find()` the `Department` with ID 1, and `Department` with ID 2.
- And we'll print out the tech and operations `Department` out to screen.

Run this code and let's see whether our post load callback methods have been invoked.



```
Hibernate: 
    select
        department0_.id as id1_0_0_,
        department0_.name as name2_0_0_ 
    from
        departments department0_ 
    where
        department0_.id=?
************ After loading department object: Tech
Hibernate: 
    select
        employees0_.department_id as departme3_1_0_,
        employees0_.id as id1_1_0_,
        employees0_.id as id1_1_1_,
        employees0_.department_id as departme3_1_1_,
        employees0_.name as name2_1_1_ 
    from
        employees employees0_ 
    where
        employees0_.department_id=?
{ 1, Tech, [{ 1, Ben }, { 2, Alice }] }
```

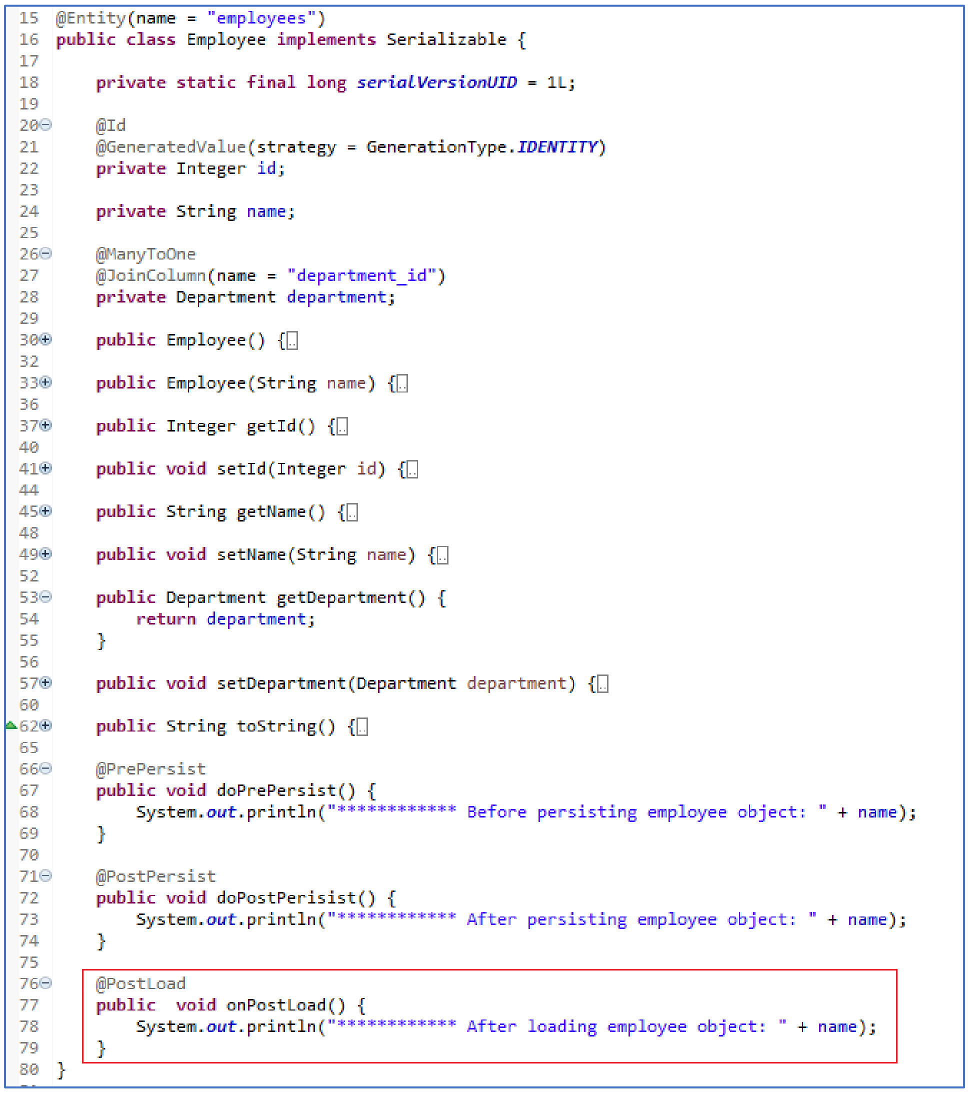
- Here's the select statement to select the first department.
- And after loading this department, which is the Tech department, our post load code has been invoked.
- Next is the select statement for the employees on the Tech department. Let's skip over those.
- Now the Tech department has been loaded and the employee names Alice and Ben have been printed out to screen. Let's scroll down further.



```
Hibernate: 
    select
        department0_.id as id1_0_0_,
        department0_.name as name2_0_0_ 
    from
        departments department0_ 
    where
        department0_.id=?
************ After loading department object: Operations
Hibernate: 
    select
        employees0_.department_id as departme3_1_0_,
        employees0_.id as id1_1_0_,
        employees0_.id as id1_1_1_,
        employees0_.department_id as departme3_1_1_,
        employees0_.name as name2_1_1_ 
    from
        employees employees0_ 
    where
        employees0_.department_id=?
{ 2, Operations, [{ 3, Cora }, { 4, Dennis }] }
```

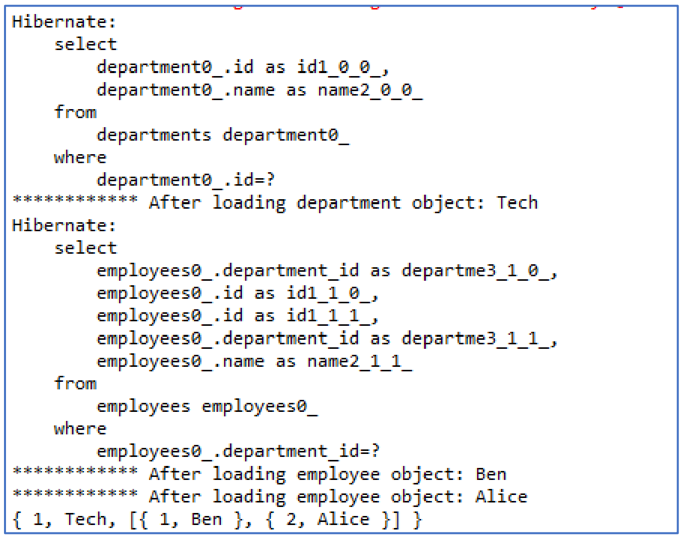
- And the next select statement is to load the operations department after which our callback method is invoked.
- And here at the bottom our operations employee names have been printed out to screen Dennis and Cora.

In order to get some more practice with this post load callback. I'm going to head over to **Employee.java**, and set up methods here as well annotated with **@PostLoad**.



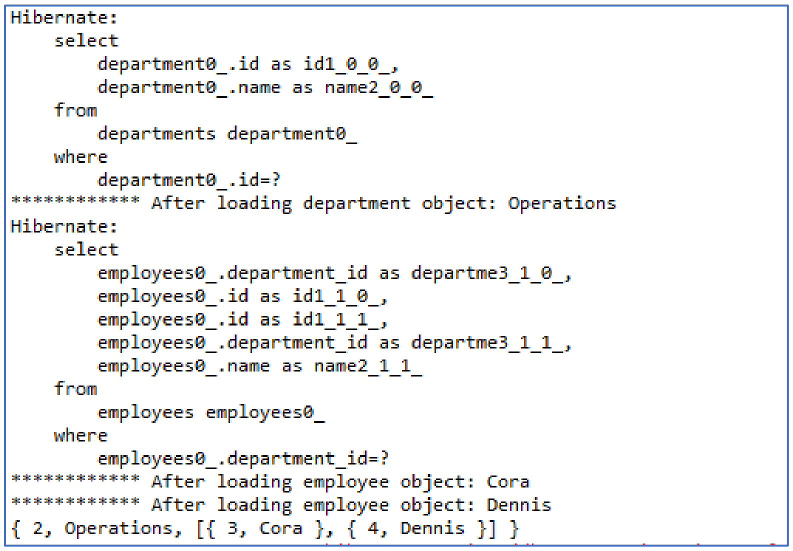
```java
@Entity(name = "employees")
public class Employee implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String name;

    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;

    public Employee() {

    public Employee(String name) {

    public Integer getId() {

    public void setId(Integer id) {

    public String getName() {

    public void setName(String name) {

    public Department getDepartment() {
        return department;
    }

    public void setDepartment(Department department) {

    public String toString() {

    @PrePersist
    public void doPrePersist() {
        System.out.println("************ Before persisting employee object: " + name);
    }

    @PostPersist
    public void doPostPerisist() {
        System.out.println("************ After persisting employee object: " + name);
    }

    @PostLoad
    public  void onPostLoad() {
        System.out.println("************ After loading employee object: " + name);
    }
}
```

After employ entities have been loaded, we'll print out the name of the employee to screen. Run our application and you'll see that the callback is called at the right point.



```
Hibernate: 
    select
        department0_.id as id1_0_0_,
        department0_.name as name2_0_0_ 
    from
        departments department0_ 
    where
        department0_.id=?
************ After loading department object: Tech
Hibernate: 
    select
        employees0_.department_id as departme3_1_0_,
        employees0_.id as id1_1_0_,
        employees0_.id as id1_1_1_,
        employees0_.department_id as departme3_1_1_,
        employees0_.name as name2_1_1_ 
    from
        employees employees0_ 
    where
        employees0_.department_id=?
************ After loading employee object: Ben
************ After loading employee object: Alice
{ 1, Tech, [{ 1, Ben }, { 2, Alice }] }
```

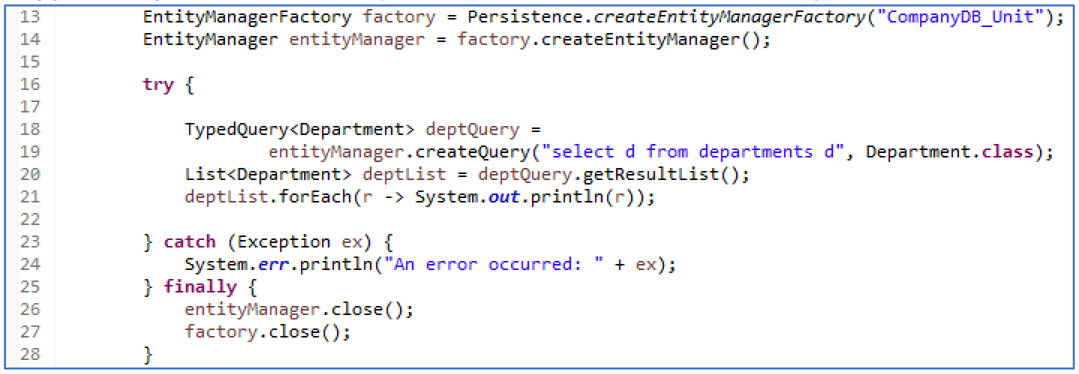
- Here we select the department,
- and notice the callback after the department object has been loaded.
- We scroll down, we select the employees from the department, after which the PostLoad callback methods for Alice and Ben are executed.



```
Hibernate: 
    select
        department0_.id as id1_0_0_,
        department0_.name as name2_0_0_ 
    from
        departments department0_ 
    where
        department0_.id=?
************ After loading department object: Operations
Hibernate: 
    select
        employees0_.department_id as departme3_1_0_,
        employees0_.id as id1_1_0_,
        employees0_.id as id1_1_1_,
        employees0_.department_id as departme3_1_1_,
        employees0_.name as name2_1_1_ 
    from
        employees employees0_ 
    where
        employees0_.department_id=?
************ After loading employee object: Cora
************ After loading employee object: Dennis
{ 2, Operations, [{ 3, Cora }, { 4, Dennis }] }
```

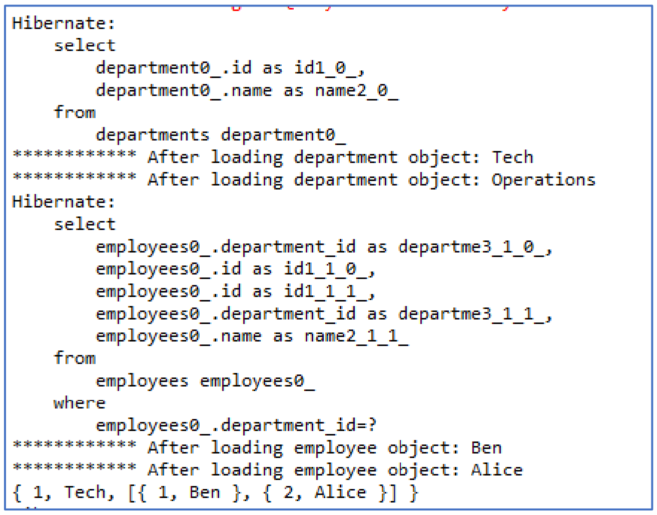
- Next when we load the operations department the PostLoad callback for operations is executed.
- And then the PostLoad callback for the employees who belong to the operations department, Dennis and Cora.

`PostLoad` callback methods are invoked, even when you retrieve entities using queries, whether they are simply JPQL queries or queries, which use the criteria API. Here I'm going to show you an example of a `TypedQuery` that I create using JPQL queries. `"select d from departments d"`.



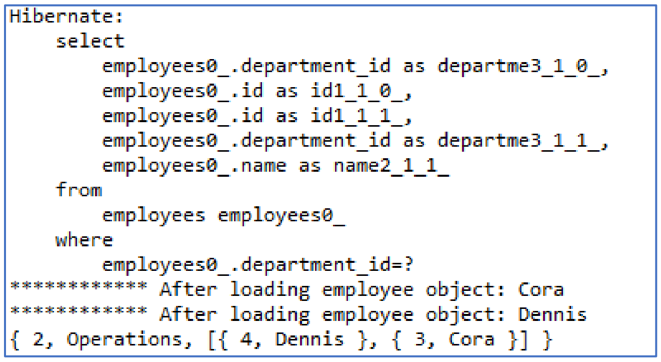
```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("CompanyDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {

            TypedQuery<Department> deptQuery =
                    entityManager.createQuery("select d from departments d", Department.class);
            List<Department> deptList = deptQuery.getResultList();
            deptList.forEach(r -> System.out.println(r));

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

We'll run this query load all departments into our application.



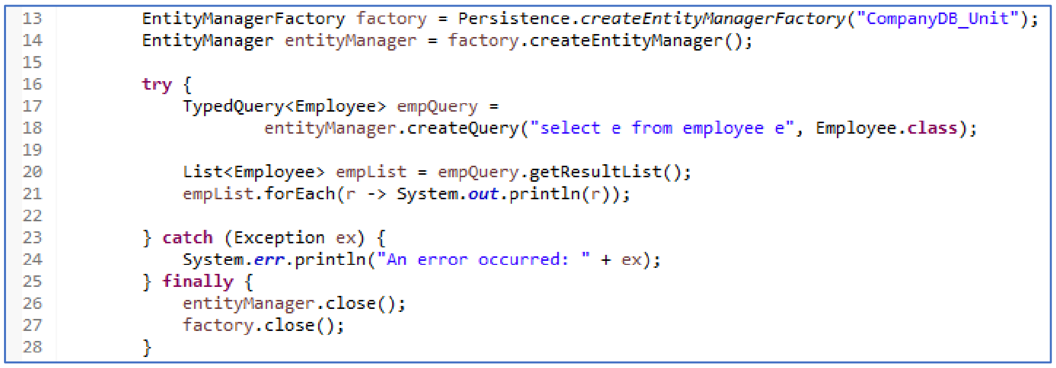
```
Hibernate: 
    select
        department0_.id as id1_0_,
        department0_.name as name2_0_ 
    from
        departments department0_
************ After loading department object: Tech
************ After loading department object: Operations
Hibernate: 
    select
        employees0_.department_id as departme3_1_0_,
        employees0_.id as id1_1_0_,
        employees0_.id as id1_1_1_,
        employees0_.department_id as departme3_1_1_,
        employees0_.name as name2_1_1_ 
    from
        employees employees0_ 
    where
        employees0_.department_id=?
************ After loading employee object: Ben
************ After loading employee object: Alice
{ 1, Tech, [{ 1, Ben }, { 2, Alice }] }
```

- And for each department, the PostLoad callback will be invoked.
- Both Tech and Operations call backs have been invoked.
- And if you scroll down further, you'll see that employees are loaded as well because when we retrieve the department, we retrieve corresponding employees.
- Here's the PostLoad callback for Alice, and Ben after which you can see the PostLoad callback for employees, Dennis and Cora.

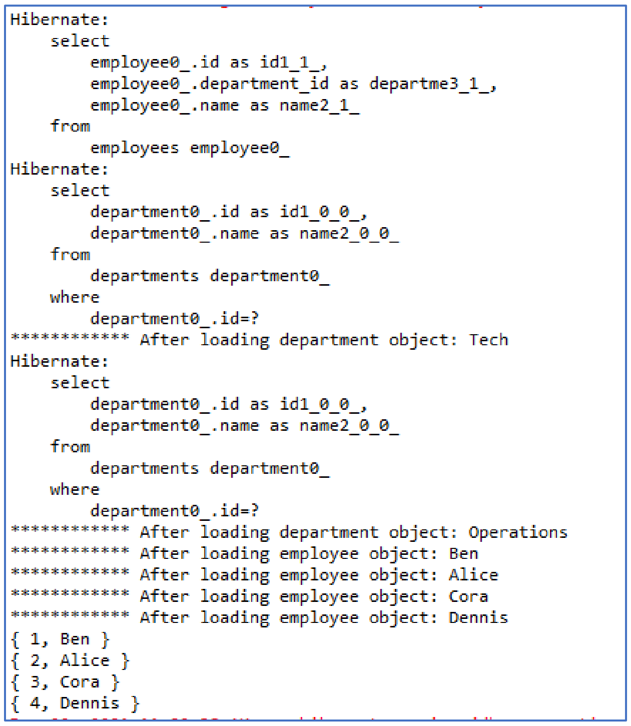


```
Hibernate: 
    select
        employees0_.department_id as departme3_1_0_,
        employees0_.id as id1_1_0_,
        employees0_.id as id1_1_1_,
        employees0_.department_id as departme3_1_1_,
        employees0_.name as name2_1_1_ 
    from
        employees employees0_ 
    where
        employees0_.department_id=?
************ After loading employee object: Cora
************ After loading employee object: Dennis
{ 2, Operations, [{ 4, Dennis }, { 3, Cora }] }
```

I'm going to change the query that I executed, and select all employee entities from the underlying table.



```java
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("CompanyDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {
            TypedQuery<Employee> empQuery =
                    entityManager.createQuery("select e from employee e", Employee.class);

            List<Employee> empList = empQuery.getResultList();
            empList.forEach(r -> System.out.println(r));

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.close();
            factory.close();
        }
```

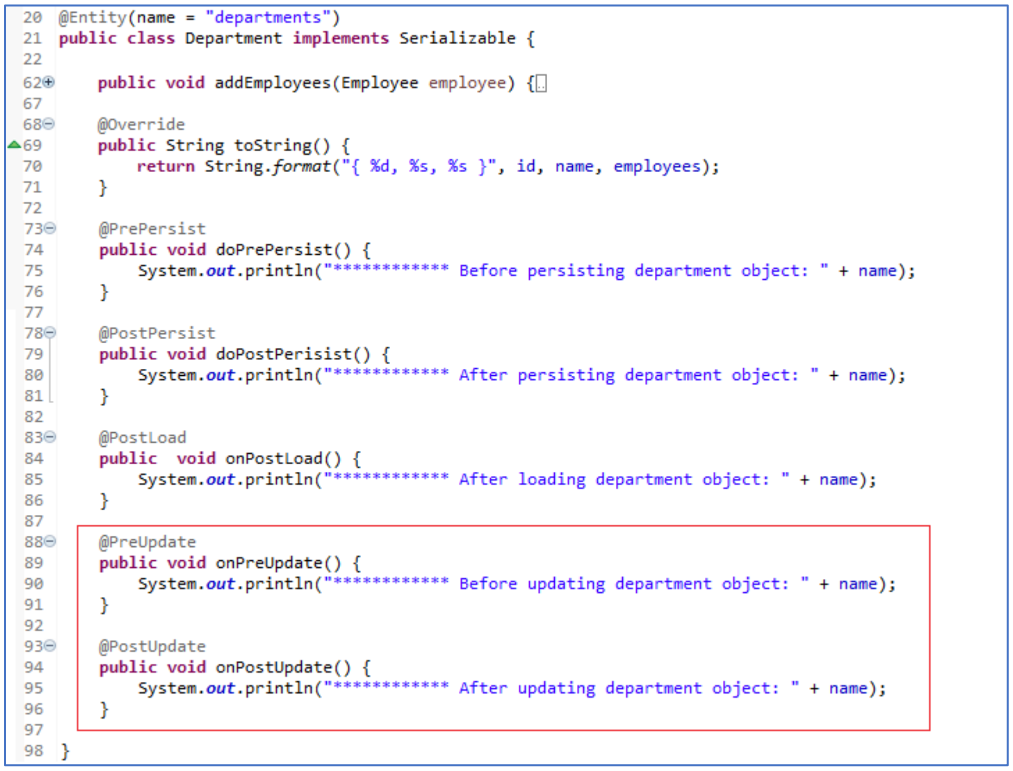


```
Hibernate:
    select
        employee0_.id as id1_1_,
        employee0_.department_id as departme3_1_,
        employee0_.name as name2_1_
    from
        employees employee0_
Hibernate:
    select
        department0_.id as id1_0_0_,
        department0_.name as name2_0_0_
    from
        departments department0_
    where
        department0_.id=?
************ After loading department object: Tech
Hibernate:
    select
        department0_.id as id1_0_0_,
        department0_.name as name2_0_0_
    from
        departments department0_
    where
        department0_.id=?
************ After loading department object: Operations
************ After loading employee object: Ben
************ After loading employee object: Alice
************ After loading employee object: Cora
************ After loading employee object: Dennis
{ 1, Ben }
{ 2, Alice }
{ 3, Cora }
{ 4, Dennis }
```

- PostLoad call backs for employee entities will be invoked.
- Retrieving employees will also retrieve the departments associated with each employee.
- Here is the PostLoad callback for the Tech department.
- And if you scroll down below you'll see the PostLoad callback for Operations.
- Then for employees who are part of Tech department, Alice and Ben, and employees who are part of Operations, Dennis and Cora.

## Working With Update Callbacks

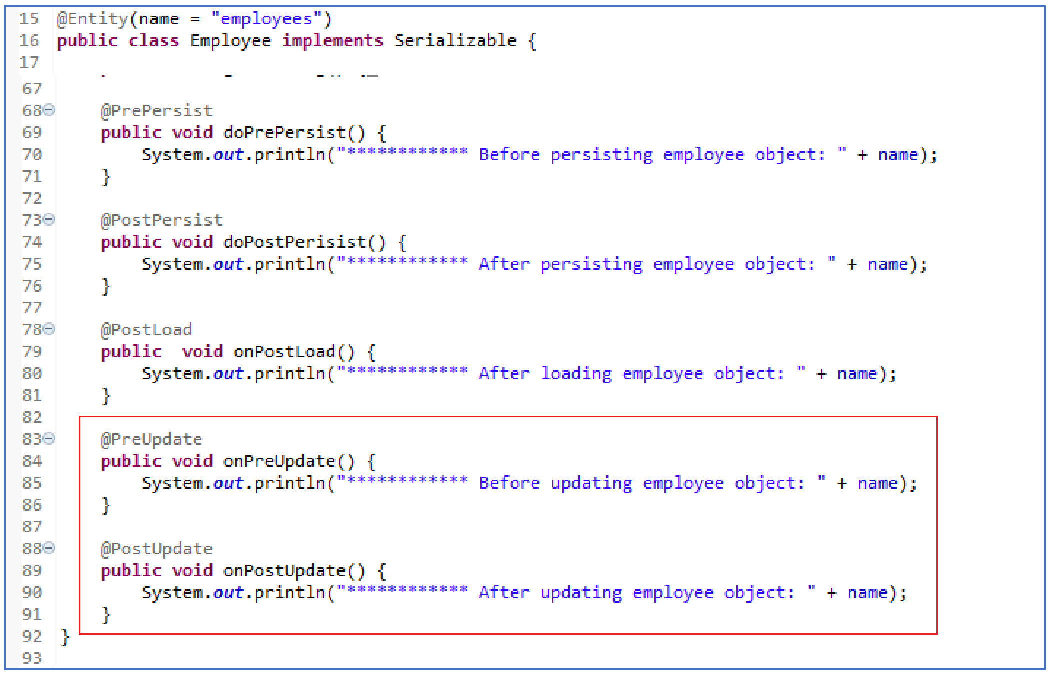
You can have callback methods that are invoked just before you update an entity and just after you update an entity using the annotations, **@PreUpdate** and **@PostUpdate**. Notice the two methods that I have setup here.



```java
@Entity(name = "departments")
public class Department implements Serializable {

    public void addEmployees(Employee employee) {

    @Override
    public String toString() {
        return String.format("{ %d, %s, %s }", id, name, employees);
    }

    @PrePersist
    public void doPrePersist() {
        System.out.println("************ Before persisting department object: " + name);
    }

    @PostPersist
    public void doPostPerisist() {
        System.out.println("************ After persisting department object: " + name);
    }

    @PostLoad
    public  void onPostLoad() {
        System.out.println("************ After loading department object: " + name);
    }

    @PreUpdate
    public void onPreUpdate() {
        System.out.println("************ Before updating department object: " + name);
    }

    @PostUpdate
    public void onPostUpdate() {
        System.out.println("************ After updating department object: " + name);
    }

}
```

- The first one with the annotation, **@PreUpdate**, will be invoked just before updating the department object we have within the department entity.
- The second method will be called just after updating the department object on **@PostUpdate**.

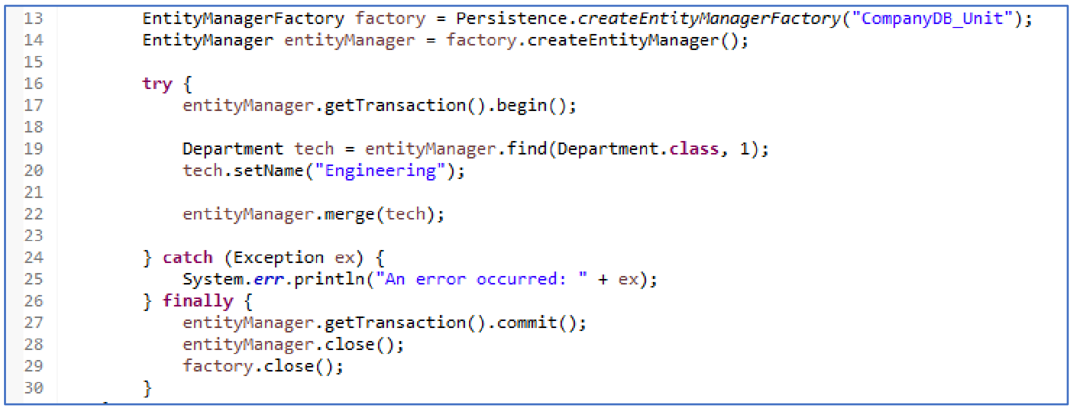
Now let's head over to Employee.java and set up PreUpdate and PostUpdate callbacks here as well.



```
@Entity(name = "employees")
public class Employee implements Serializable {


    @PrePersist
    public void doPrePersist() {
        System.out.println("************ Before persisting employee object: " + name);
    }

    @PostPersist
    public void doPostPerisist() {
        System.out.println("************ After persisting employee object: " + name);
    }

    @PostLoad
    public  void onPostLoad() {
        System.out.println("************ After loading employee object: " + name);
    }

    @PreUpdate
    public void onPreUpdate() {
        System.out.println("************ Before updating employee object: " + name);
    }

    @PostUpdate
    public void onPostUpdate() {
        System.out.println("************ After updating employee object: " + name);
    }
}
```

- In both cases, I have a simple `System.out.println()` statement, but you can have any code in here within these callback methods.
- Make sure you have the right annotations, **@PreUpdate** and **@PostUpdate**, so that these methods are called at the right point in your entity lifecycle.

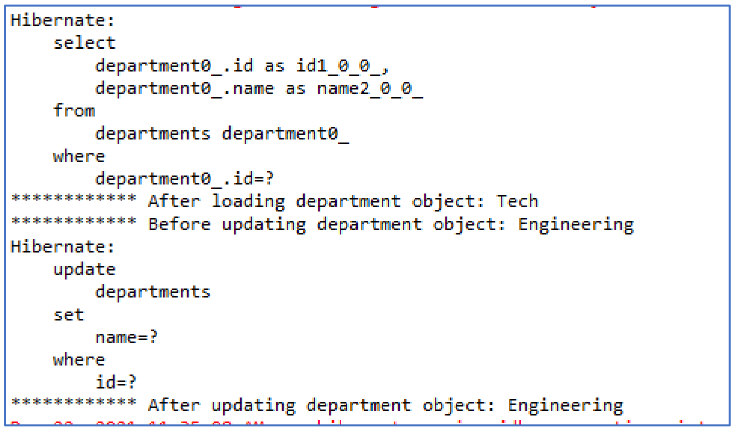
Now we'll head over to **App.java**, and perform a few updates so we can see when exactly these callbacks are invoked.



```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("CompanyDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {
            entityManager.getTransaction().begin();

            Department tech = entityManager.find(Department.class, 1);
            tech.setName("Engineering");

            entityManager.merge(tech);

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.getTransaction().commit();
            entityManager.close();
            factory.close();
        }
```

- I'm going to look for the department entity with ID 1.
- This is the tech department and I'm going to update it to set its name to be `"Engineering"`.
- I then call, `entityManager.merge(tech)`.

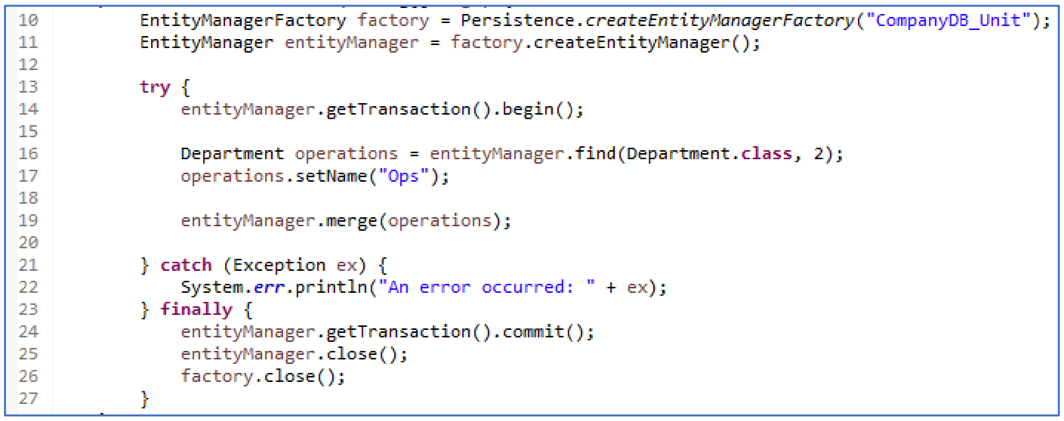
Let's run this code and see when the Pre and PostUpdate callbacks are called.



```
Hibernate:
    select
        department0_.id as id1_0_0_,
        department0_.name as name2_0_0_
    from
        departments department0_
    where
        department0_.id=?
************ After loading department object: Tech
************ Before updating department object: Engineering
Hibernate:
    update
        departments
    set
        name=?
    where
        id=?
************ After updating department object: Engineering
```

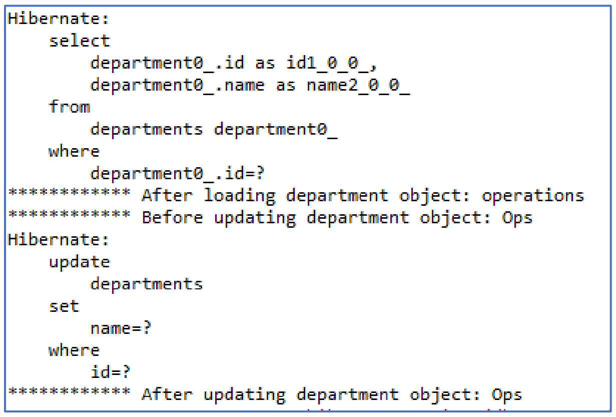
- This is where we load the department object.
- After that, the **postLoad** callback is called, "After loading department object: Tech"
- and the **PreUpdate** callback is also invoked. "Before updating department object: Engineering"
- You can then see the update query to actually update the department object, from `"tech"` to `"Engineering"`.
- And then the **PostUpdate** callback, `"After updating the department object"`, we've changed its name.

Let's try this once again. We'll update the second department as well. Instead of `"operations"`, we'll call it `"Ops"`.



```java
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("CompanyDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {
            entityManager.getTransaction().begin();

            Department operations = entityManager.find(Department.class, 2);
            operations.setName("Ops");

            entityManager.merge(operations);

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.getTransaction().commit();
            entityManager.close();
            factory.close();
        }
```

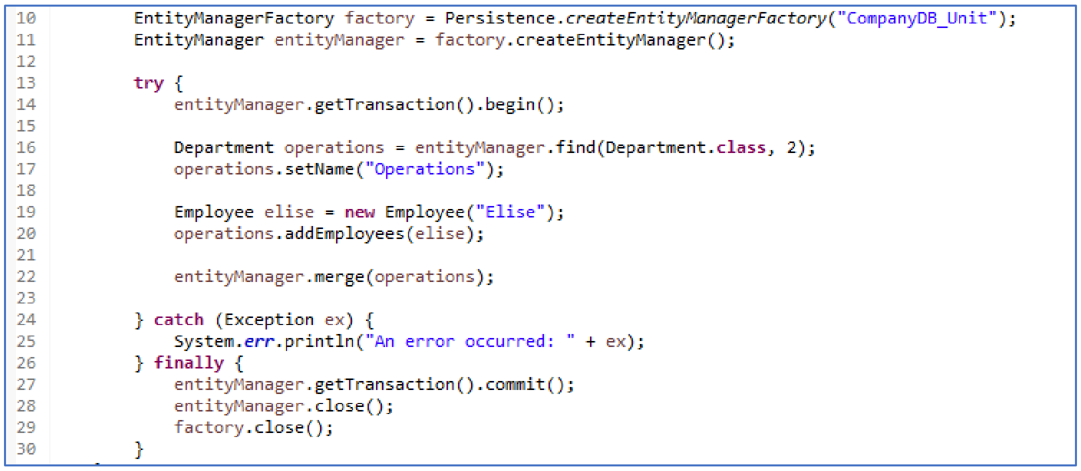
Go ahead and run this code.



```
Hibernate: 
    select
        department0_.id as id1_0_0_,
        department0_.name as name2_0_0_ 
    from
        departments department0_ 
    where
        department0_.id=?
************ After loading department object: operations
************ Before updating department object: Ops
Hibernate: 
    update
        departments 
    set
        name=? 
    where
        id=?
************ After updating department object: Ops
```

- Here is the select statement for the department after which the post load callback will be invoked and after that, the PreUpdate callback for Ops.
- We then have the update SQL statement and finally, the PostUpdate callback.

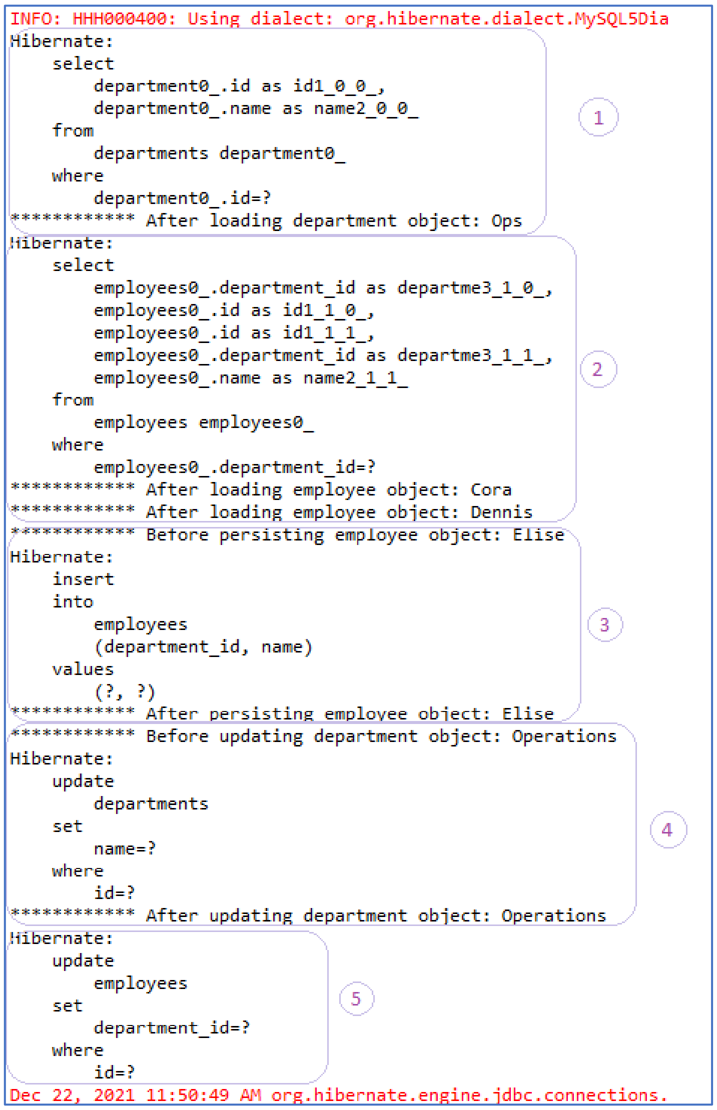
I'll now perform a slightly more interesting update.



```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("CompanyDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {
            entityManager.getTransaction().begin();

            Department operations = entityManager.find(Department.class, 2);
            operations.setName("Operations");

            Employee elise = new Employee("Elise");
            operations.addEmployees(elise);

            entityManager.merge(operations);

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.getTransaction().commit();
            entityManager.close();
            factory.close();
        }
```

- I'm going to retrieve the operations department,
- update its name
- and add an employee to that department, the employee Elise.

And this results the invocation of a number of callback methods.



```
INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQL5Dia
Hibernate:
    select
        department0_.id as id1_0_0_,
        department0_.name as name2_0_0_
    from
        departments department0_
    where
        department0_.id=?
************ After loading department object: Ops
Hibernate:
    select
        employees0_.department_id as departme3_1_0_,
        employees0_.id as id1_1_0_,
        employees0_.id as id1_1_1_,
        employees0_.department_id as departme3_1_1_,
        employees0_.name as name2_1_1_
    from
        employees employees0_
    where
        employees0_.department_id=?
************ After loading employee object: Cora
************ After loading employee object: Dennis
************ Before persisting employee object: Elise
Hibernate:
    insert
    into
        employees
        (department_id, name)
    values
        (?, ?)
************ After persisting employee object: Elise
************ Before updating department object: Operations
Hibernate:
    update
        departments
    set
        name=?
    where
        id=?
************ After updating department object: Operations
Hibernate:
    update
        employees
    set
        department_id=?
    where
        id=?
Dec 22, 2021 11:50:49 AM org.hibernate.engine.jdbc.connections.
```

1. First, the Ops department is loaded.

2. After which if you scroll down further, you'll find the employees associated with the “Ops” department are also loaded, “Dennis” and “Cora”.

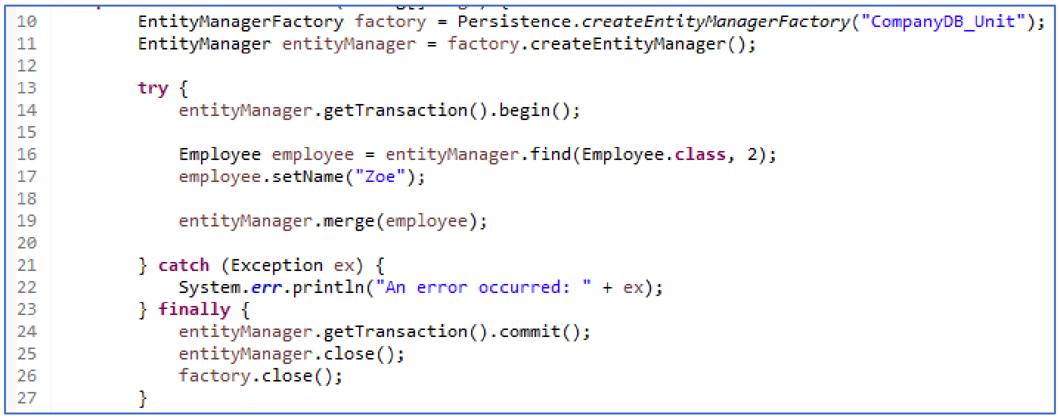
3. The “Elise” entity has been created. We are just about to persist Elise.

4. We are also updating the “operations” department to which Elise belongs. So here is the callback before updating the department object

5. and here is the callback after persisting the employee, Elise.

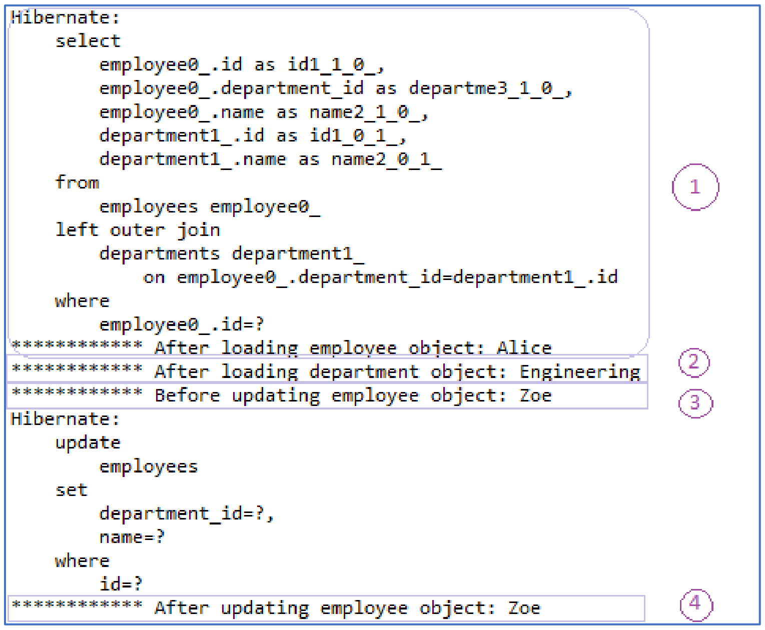
If we scroll down further, we've completed updating the department object. Elise belongs to the operations department.

Now let's try and update an employee object. We load the employee with ID two, that is “Alice”, and change the name to “Zoe”.



```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("CompanyDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {
            entityManager.getTransaction().begin();

            Employee employee = entityManager.find(Employee.class, 2);
            employee.setName("Zoe");

            entityManager.merge(employee);

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.getTransaction().commit();
            entityManager.close();
            factory.close();
        }
```

Let's run this code and see the lifecycle of this entity.



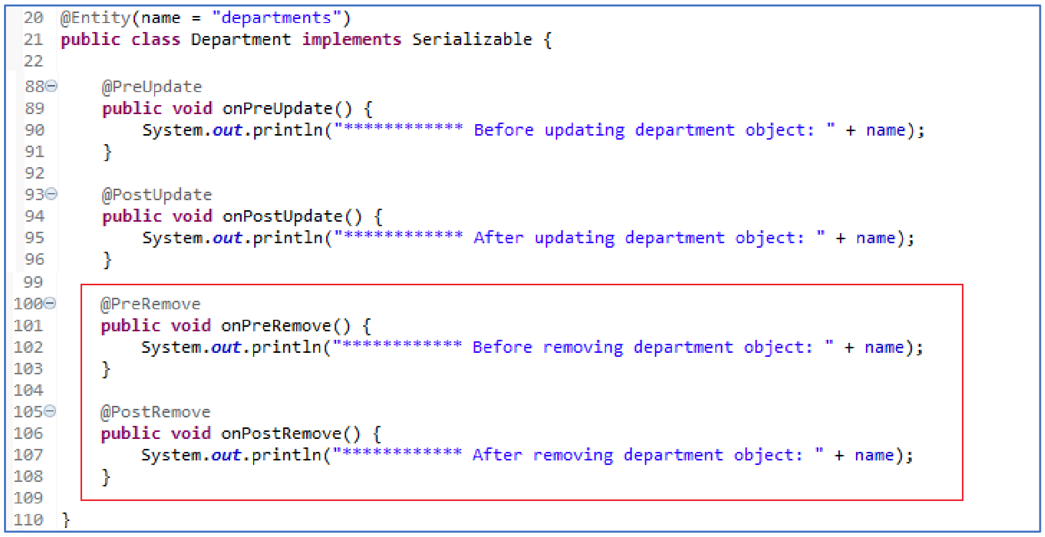
```
Hibernate: 
    select
        employee0_.id as id1_1_0_,
        employee0_.department_id as departme3_1_0_,
        employee0_.name as name2_1_0_,
        department1_.id as id1_0_1_,
        department1_.name as name2_0_1_ 
    from
        employees employee0_ 
    left outer join
        departments department1_ 
            on employee0_.department_id=department1_.id 
    where
        employee0_.id=?
************ After loading employee object: Alice
************ After loading department object: Engineering
************ Before updating employee object: Zoe
Hibernate: 
    update
        employees 
    set
        department_id=?,
        name=? 
    where
        id=?
************ After updating employee object: Zoe
```

1. First, we load the employee object to be updated, that is “Alice”.
2. We also load the corresponding department, that is ”Engineering”.
3. The **PreUpdate** callback is invoked before updating the employee object, “Zoe”,
4. and then the **PostUpdate** callback is invoked after “Zoe” has been updated.

## Working With Remove Callbacks

And finally we'll explore the last set of callback methods that allows us to plug into our entity lifecycle, **PreRemove** and **PostRemove**.

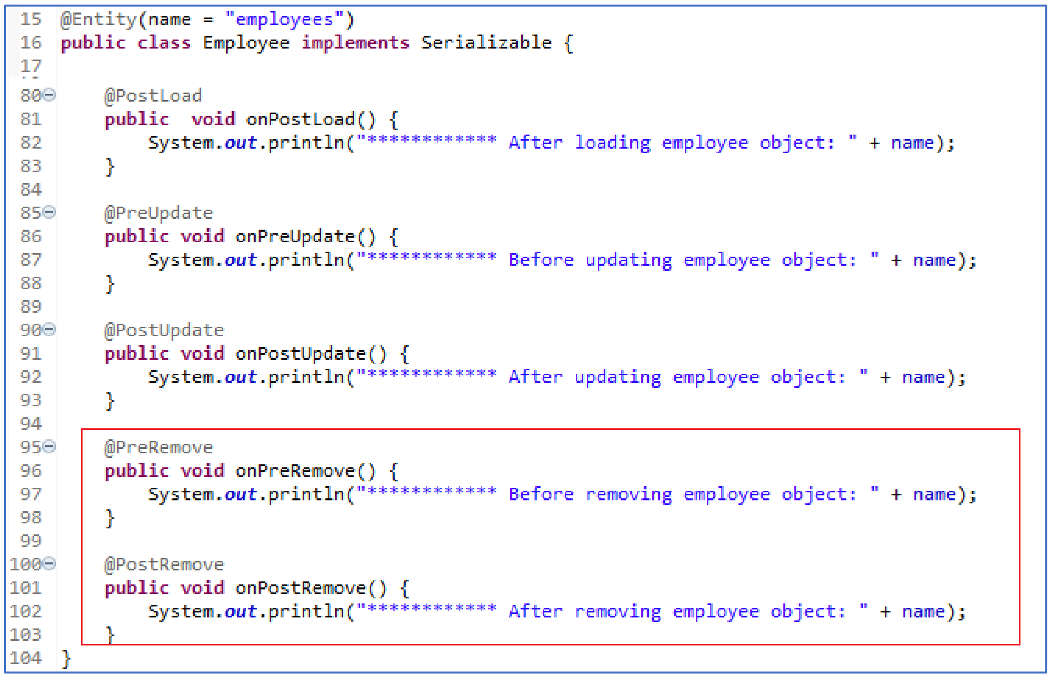
Methods in your entity class that have been annotated using these annotations will be invoked. Just before you remove an entity from your database table, and just after that entity has been removed.



```java
@Entity(name = "departments")
public class Department implements Serializable {

    @PreUpdate
    public void onPreUpdate() {
        System.out.println("************ Before updating department object: " + name);
    }

    @PostUpdate
    public void onPostUpdate() {
        System.out.println("************ After updating department object: " + name);
    }

    @PreRemove
    public void onPreRemove() {
        System.out.println("************ Before removing department object: " + name);
    }

    @PostRemove
    public void onPostRemove() {
        System.out.println("************ After removing department object: " + name);
    }

}
```

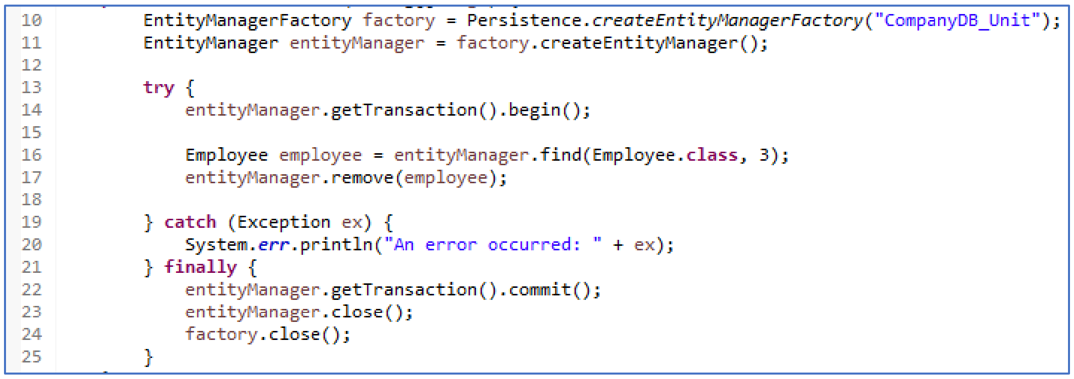
I've set up two methods here **onPreRemove** and **onPostRemove**, and annotated them using **@PreRemove** and **@PostRemove** respectively.

They contain simple `System.out.println()` statements, but you can of course execute any code that you want to. This is in **Department.java**, head over to **Employee.java** where I'll do the same thing.



```java
@Entity(name = "employees")
public class Employee implements Serializable {

    @PostLoad
    public  void onPostLoad() {
        System.out.println("************ After loading employee object: " + name);
    }

    @PreUpdate
    public void onPreUpdate() {
        System.out.println("************ Before updating employee object: " + name);
    }

    @PostUpdate
    public void onPostUpdate() {
        System.out.println("************ After updating employee object: " + name);
    }

    @PreRemove
    public void onPreRemove() {
        System.out.println("************ Before removing employee object: " + name);
    }

    @PostRemove
    public void onPostRemove() {
        System.out.println("************ After removing employee object: " + name);
    }
}
```

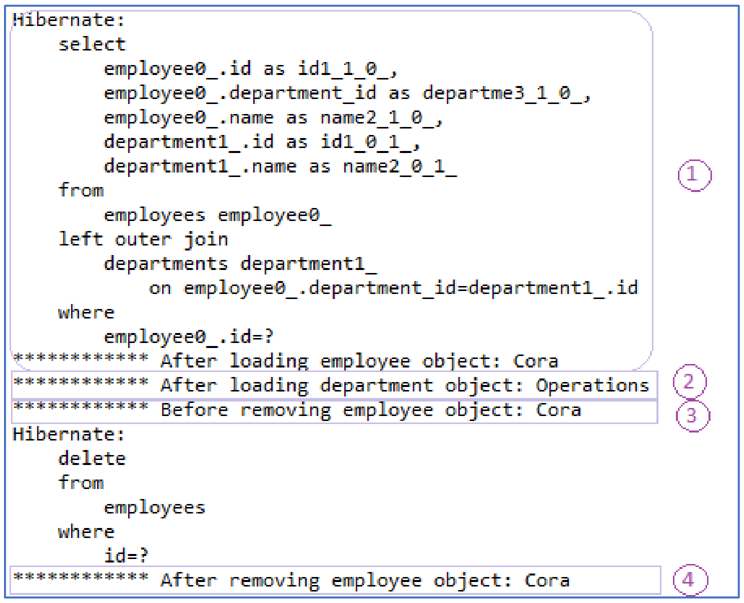
Now let's head over to App.java and perform some deletions and see when these callbacks are invoked.



```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("CompanyDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {
            entityManager.getTransaction().begin();

            Employee employee = entityManager.find(Employee.class, 3);
            entityManager.remove(employee);

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.getTransaction().commit();
            entityManager.close();
            factory.close();
        }
```

I have code here to access the `employee` with id 3, and then I remove that `employee` using `entityManager.remove()`.

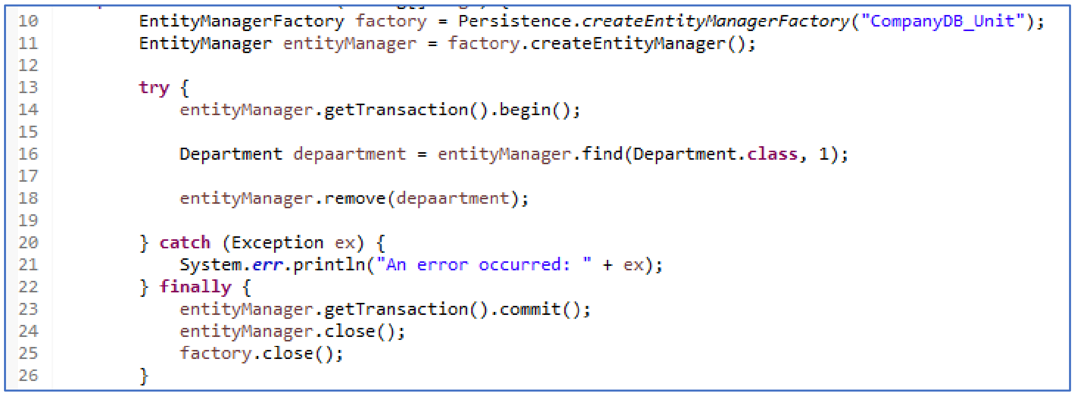
Let's run this code and let's see when precisely our **PreRemove** and **PostRemoved** callbacks are invoked.



```
Hibernate:
    select
        employee0_.id as id1_1_0_,
        employee0_.department_id as departme3_1_0_,
        employee0_.name as name2_1_0_,
        department1_.id as id1_0_1_,
        department1_.name as name2_0_1_
    from
        employees employee0_
    left outer join
        departments department1_
            on employee0_.department_id=department1_.id
    where
        employee0_.id=?
************ After loading employee object: Cora
************ After loading department object: Operations
************ Before removing employee object: Cora
Hibernate:
    delete
    from
        employees
    where
        id=?
************ After removing employee object: Cora
```

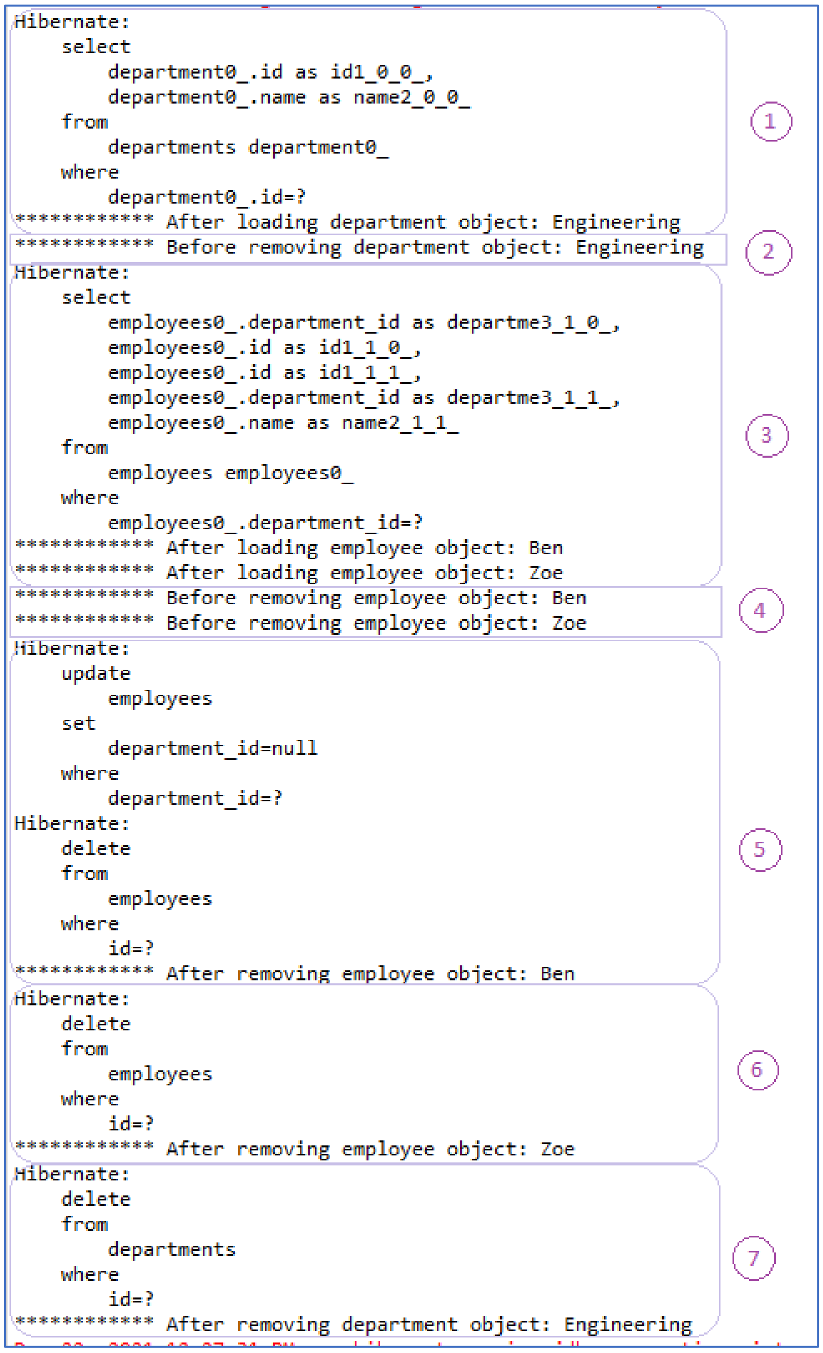
1) First the “Cora” object is loaded. Here is the `postLoad` code.
   "After loading employee object: Cora "
2) And then we load the corresponding department that “Cora” belongs to, that is the `postLoad` callback as well. "After loading department object: Operations"
3) Then is our pre remove callback before removing the employee object Cora.
   "Before removing employee object: Cora"
4) Scroll down below, Cora has been deleted, and then we have the post remove callback method invoked.
   "After removing employee object: Cora"

After removing the employee object Cora, let's change our code to remove a department entity. Removing a department will also remove the employees who belong to that department, thanks to CascadeType all.



```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("CompanyDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {
            entityManager.getTransaction().begin();

            Department depaartment = entityManager.find(Department.class, 1);

            entityManager.remove(depaartment);

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.getTransaction().commit();
            entityManager.close();
            factory.close();
        }
```

Let's run this code.

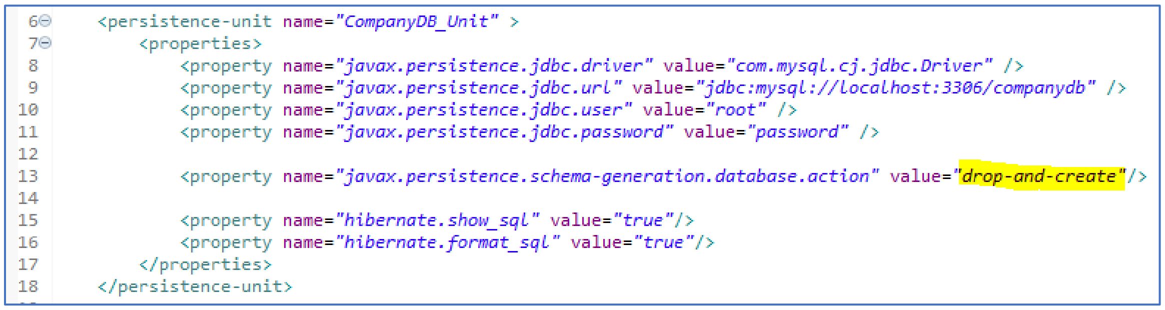


```
Hibernate:
    select
        department0_.id as id1_0_0_,
        department0_.name as name2_0_0_
    from
        departments department0_
    where
        department0_.id=?
************ After loading department object: Engineering
************ Before removing department object: Engineering
Hibernate:
    select
        employees0_.department_id as departme3_1_0_,
        employees0_.id as id1_1_0_,
        employees0_.id as id1_1_1_,
        employees0_.department_id as departme3_1_1_,
        employees0_.name as name2_1_1_
    from
        employees employees0_
    where
        employees0_.department_id=?
************ After loading employee object: Ben
************ After loading employee object: Zoe
************ Before removing employee object: Ben
************ Before removing employee object: Zoe
Hibernate:
    update
        employees
    set
        department_id=null
    where
        department_id=?
Hibernate:
    delete
    from
        employees
    where
        id=?
************ After removing employee object: Ben
Hibernate:
    delete
    from
        employees
    where
        id=?
************ After removing employee object: Zoe
Hibernate:
    delete
    from
        departments
    where
        id=?
************ After removing department object: Engineering
```

1. Notice that first the “Engineering” department is loaded into memory.
2. Then the **`preRemove()`** code is called before removing the department “Engineering”.
3. The employees who belong to the Engineering department are also loaded in “Ben” and “Zoe”.
4. And then the **`preRemove()`** code for these employees are also called **`preRemove()`** for “Ben” and “Zoe”.
5. The employees are deleted first. Here is the **`postRemove()`** code for “Ben” executed,
6. and then the post remove code for the employee “Zoe”.
7. And finally the **`postRemove()`** code for the “Engineering” department.

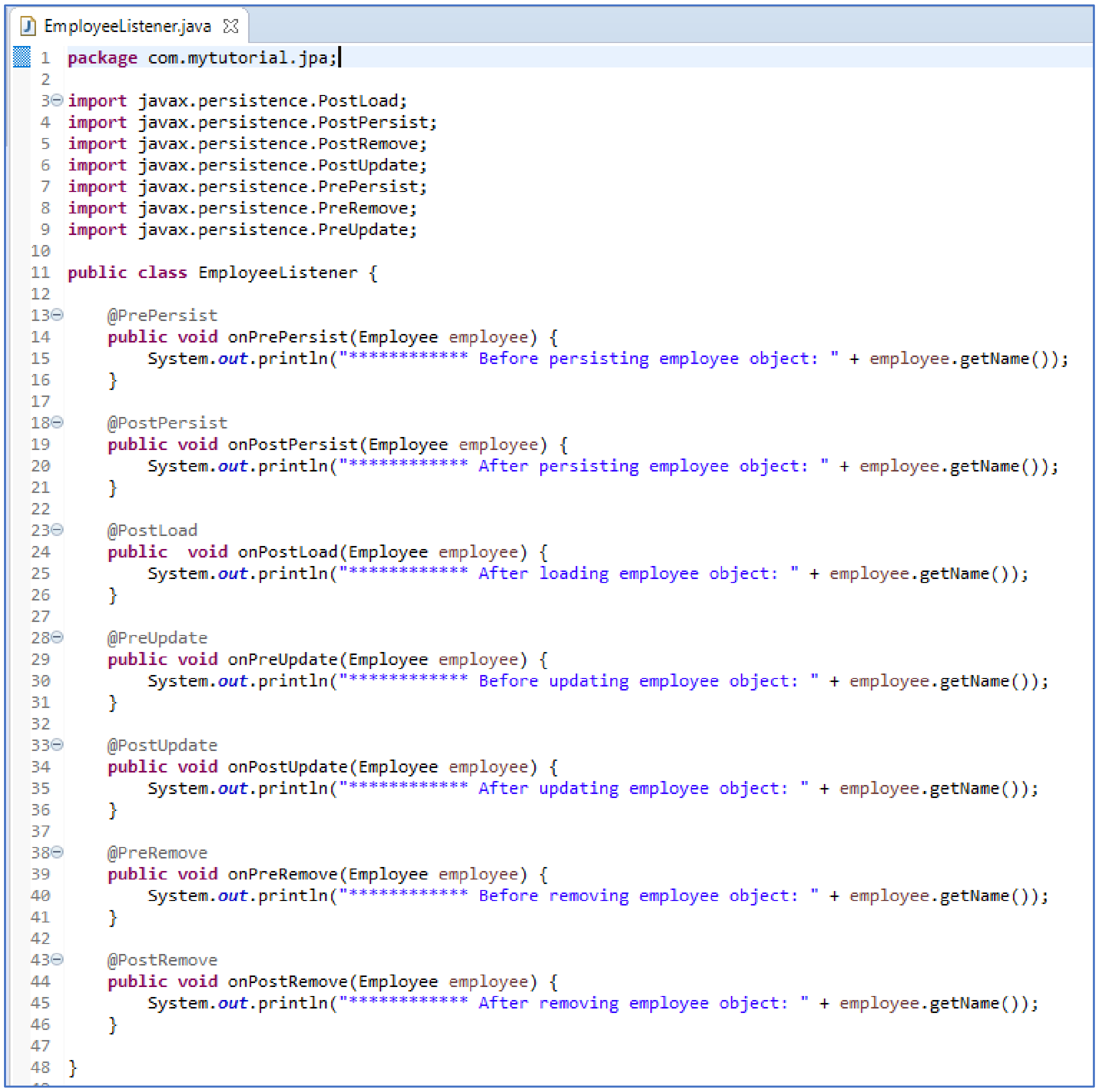
## Configuring Entity Listeners for Callback Methods

Instead of having all of your entity lifecycle callback methods defined within your entity class, you can extract all of these methods and put them in a different class, that becomes your entity listener. That's exactly what we'll see now, update the database action in **persistence.xml** to be drop-and-create, that new tables are created.



```xml
    <persistence-unit name="CompanyDB_Unit" >
        <properties>
            <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/companydb" />
            <property name="javax.persistence.jdbc.user" value="root" />
            <property name="javax.persistence.jdbc.password" value="password" />

            <property name="javax.persistence.schema-generation.database.action" value="drop-and-create"/>

            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
        </properties>
    </persistence-unit>
```

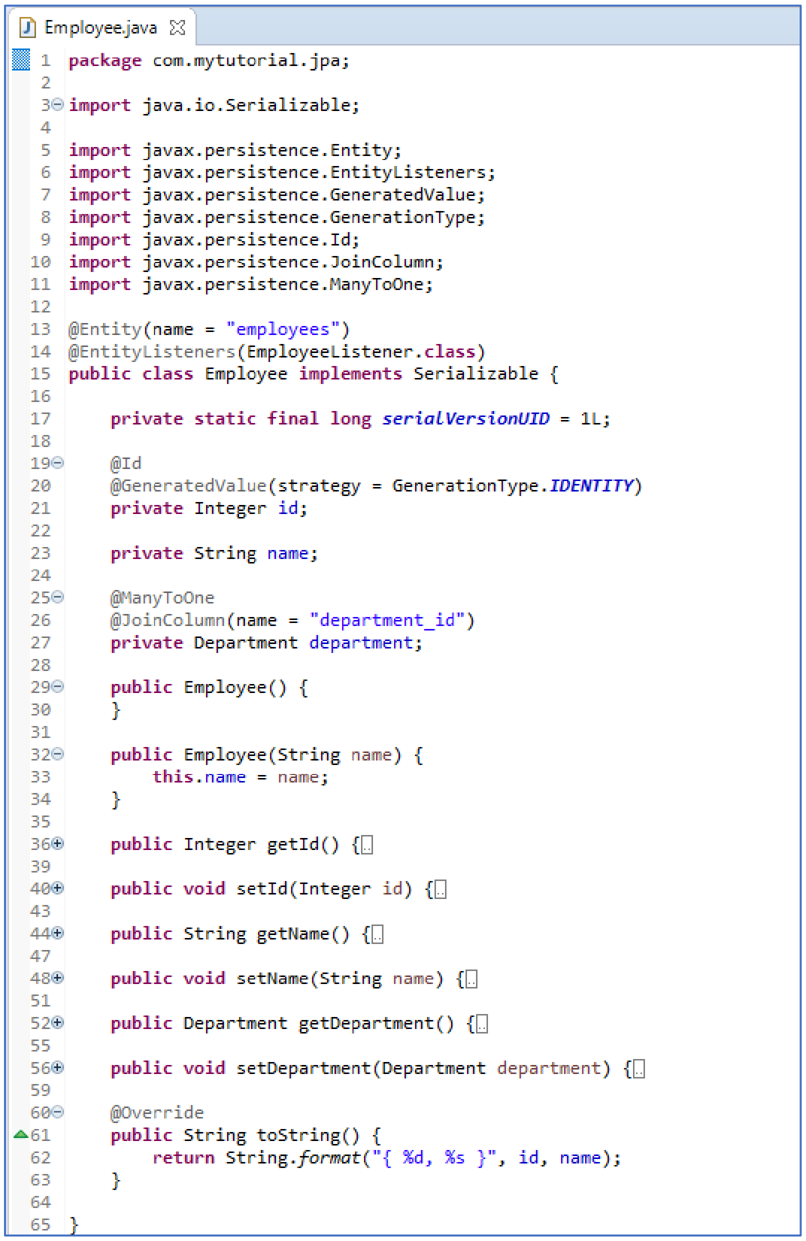
Here is my first entity listener, a listener on the employee entity. I've called it **EmployeeListener**,



```java
package com.mytutorial.jpa;

import javax.persistence.PostLoad;
import javax.persistence.PostPersist;
import javax.persistence.PostRemove;
import javax.persistence.PostUpdate;
import javax.persistence.PrePersist;
import javax.persistence.PreRemove;
import javax.persistence.PreUpdate;

public class EmployeeListener {

    @PrePersist
    public void onPrePersist(Employee employee) {
        System.out.println("************ Before persisting employee object: " + employee.getName());
    }

    @PostPersist
    public void onPostPersist(Employee employee) {
        System.out.println("************ After persisting employee object: " + employee.getName());
    }

    @PostLoad
    public  void onPostLoad(Employee employee) {
        System.out.println("************ After loading employee object: " + employee.getName());
    }

    @PreUpdate
    public void onPreUpdate(Employee employee) {
        System.out.println("************ Before updating employee object: " + employee.getName());
    }

    @PostUpdate
    public void onPostUpdate(Employee employee) {
        System.out.println("************ After updating employee object: " + employee.getName());
    }

    @PreRemove
    public void onPreRemove(Employee employee) {
        System.out.println("************ Before removing employee object: " + employee.getName());
    }

    @PostRemove
    public void onPostRemove(Employee employee) {
        System.out.println("************ After removing employee object: " + employee.getName());
    }

}
```

- Set up the import statements for all of the callback annotations.
- Instead of defining the callback methods within the entity class, we'll define it separately in this entity listener class. This is the EmployeeListener, notice that we have all life cycle method call backs defined here, **`onPrePersist, onPostPersist`**, and each of these methods have the right annotations **@PrePersist** and **@PostPersist**.
- The one difference here is that you have to explicitly pass in an input argument for the employee object, that is the employee entity that this is listening on.
- JPA will automatically pass this employee object when these callbacks are invoked.
- Lets scroll down below and take a look at the remaining callbacks, **`onPostLoad`**, **`onPreUpdate`**, **`onPostUpdate`**, **`onPreRemove`**, and finally, **`onPostRemove`**. For all of these callbacks, the employee object is passed in as an input argument.
- JPA will take care of that when the callbacks are actually executed.

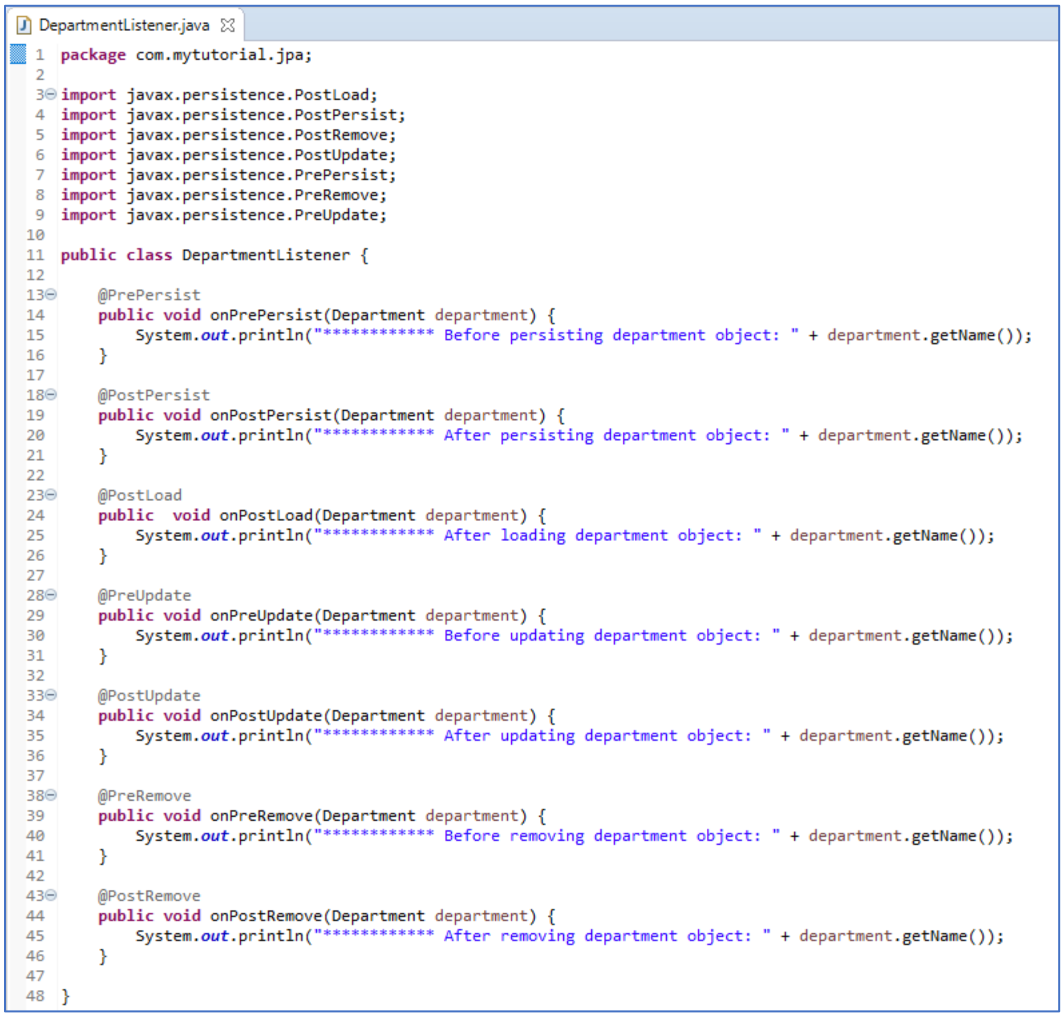
Now let's head over to **Employee.java** and configure the entity listener class on the employee entity.



```java
package com.mytutorial.jpa;

import java.io.Serializable;

import javax.persistence.Entity;
import javax.persistence.EntityListeners;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;

@Entity(name = "employees")
@EntityListeners(EmployeeListener.class)
public class Employee implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String name;

    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;

    public Employee() {
    }

    public Employee(String name) {
        this.name = name;
    }

    public Integer getId() {

    public void setId(Integer id) {

    public String getName() {

    public void setName(String name) {

    public Department getDepartment() {

    public void setDepartment(Department department) {

    @Override
    public String toString() {
        return String.format("{ %d, %s }", id, name);
    }

}
```

- Use the **@EntityListener** annotation and pass in the class of the entity listener, `EmployeeListener.class`.
- This is the only change that we make to the employee object. Rest of the employee object remains exactly the same, there is no change in the actual entity.

I'm going to get rid of all of the callback methods in my `Department` class as well. And pull them all into this **DepartmentListener**,

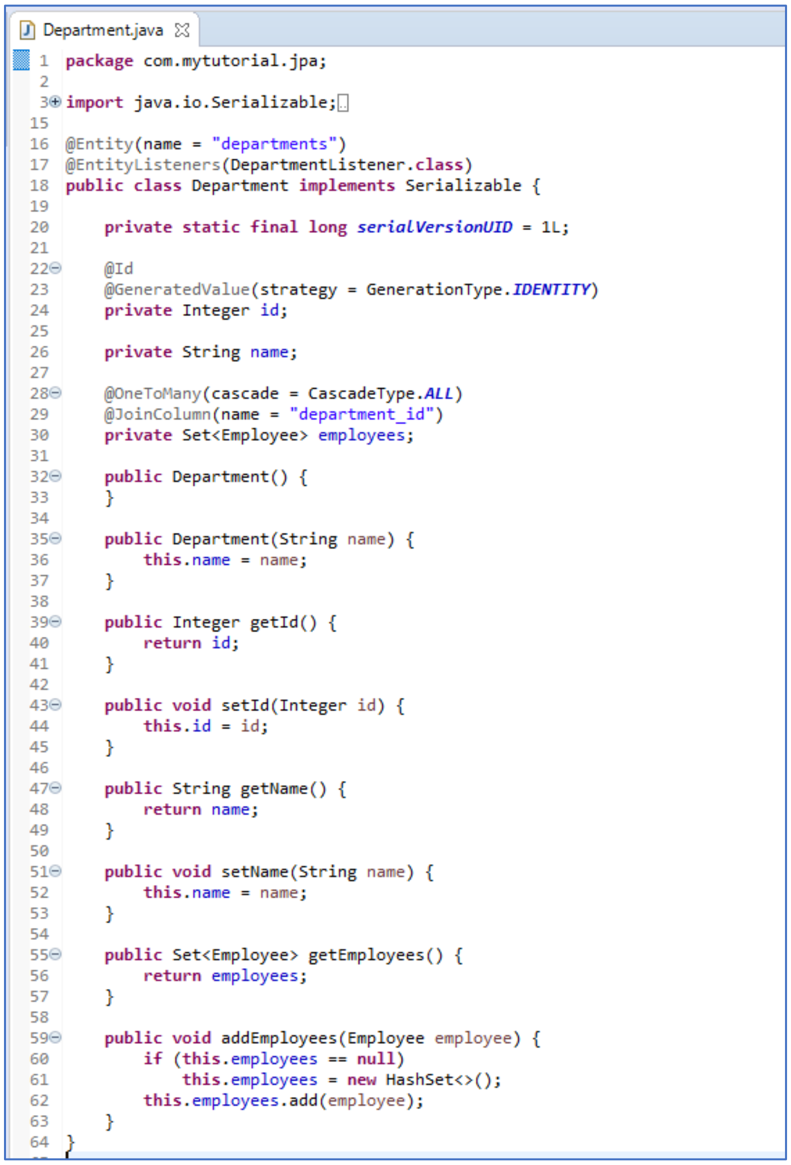


```java
package com.mytutorial.jpa;

import javax.persistence.PostLoad;
import javax.persistence.PostPersist;
import javax.persistence.PostRemove;
import javax.persistence.PostUpdate;
import javax.persistence.PrePersist;
import javax.persistence.PreRemove;
import javax.persistence.PreUpdate;

public class DepartmentListener {

    @PrePersist
    public void onPrePersist(Department department) {
        System.out.println("************ Before persisting department object: " + department.getName());
    }

    @PostPersist
    public void onPostPersist(Department department) {
        System.out.println("************ After persisting department object: " + department.getName());
    }

    @PostLoad
    public  void onPostLoad(Department department) {
        System.out.println("************ After loading department object: " + department.getName());
    }

    @PreUpdate
    public void onPreUpdate(Department department) {
        System.out.println("************ Before updating department object: " + department.getName());
    }

    @PostUpdate
    public void onPostUpdate(Department department) {
        System.out.println("************ After updating department object: " + department.getName());
    }

    @PreRemove
    public void onPreRemove(Department department) {
        System.out.println("************ Before removing department object: " + department.getName());
    }

    @PostRemove
    public void onPostRemove(Department department) {
        System.out.println("************ After removing department object: " + department.getName());
    }

}
```

- Set up the import statements for the annotations for all of the entity life cycle callback methods.
- And within this `DepartmentListener`, set up methods for each callback.
- Because these are callbacks on `Department` entities, each callback method accepts as an input argument, the `Department` object.

Make sure you pass this input argument in for every callback, and make sure you annotate all of the callback methods correctly.

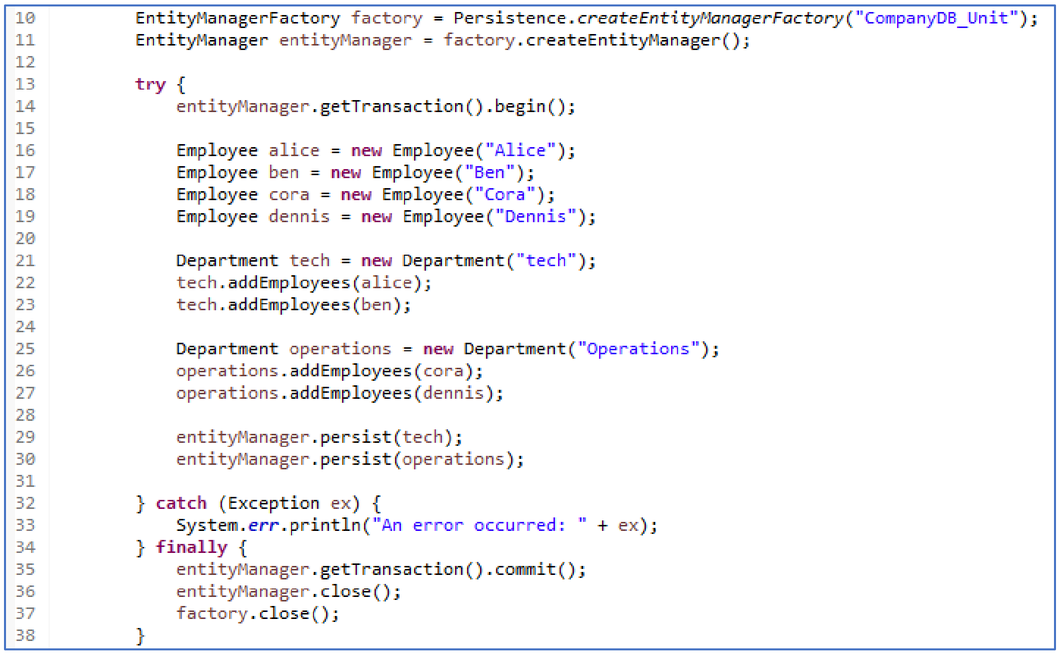
Now let's head over to **Department.java** and configure the `DepartmentListener` as an entity listener.



```java
package com.mytutorial.jpa;

import java.io.Serializable;

@Entity(name = "departments")
@EntityListeners(DepartmentListener.class)
public class Department implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String name;

    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "department_id")
    private Set<Employee> employees;

    public Department() {
    }

    public Department(String name) {
        this.name = name;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Set<Employee> getEmployees() {
        return employees;
    }

    public void addEmployees(Employee employee) {
        if (this.employees == null)
            this.employees = new HashSet<>();
        this.employees.add(employee);
    }
}
```

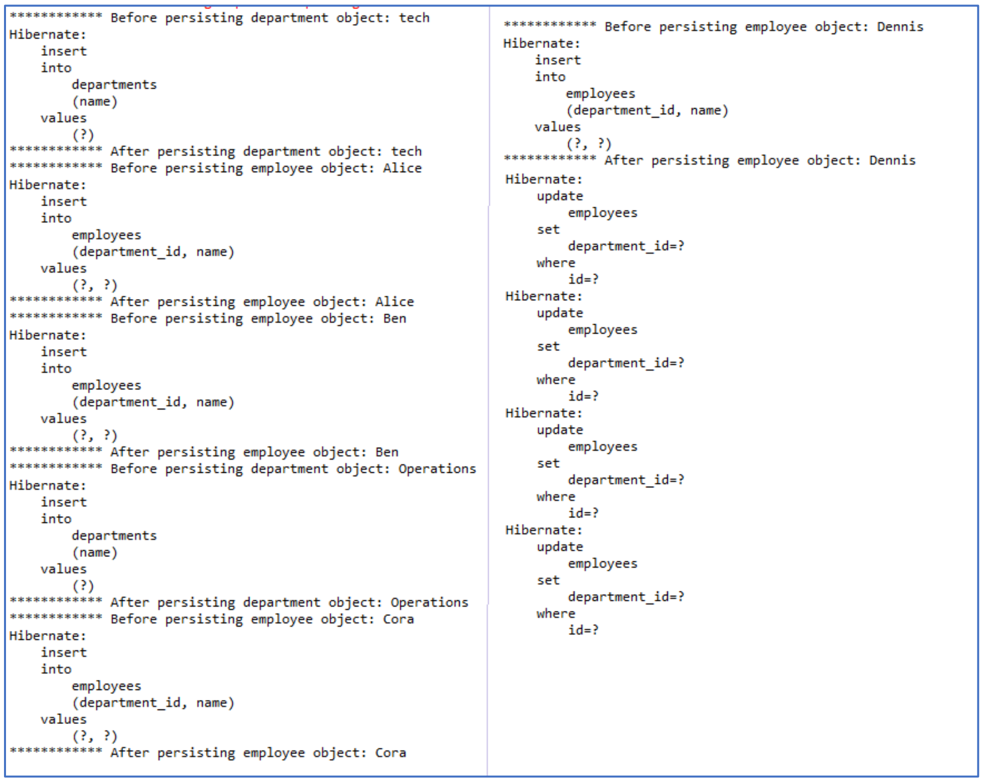
- Use the @EntityListeners annotation and add in the `DepartmentListener.class`.
- This is the only change that we'll make, the rest of the code for this department entity remains the same as in the previous demos.

We're now ready to head over to App.java. You can now perform any kind of operation that you want using Department and Employee entities, and you'll see that the right callbacks will be called at the right time.



```java
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("CompanyDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {
            entityManager.getTransaction().begin();

            Employee alice = new Employee("Alice");
            Employee ben = new Employee("Ben");
            Employee cora = new Employee("Cora");
            Employee dennis = new Employee("Dennis");

            Department tech = new Department("tech");
            tech.addEmployees(alice);
            tech.addEmployees(ben);

            Department operations = new Department("Operations");
            operations.addEmployees(cora);
            operations.addEmployees(dennis);

            entityManager.persist(tech);
            entityManager.persist(operations);

        } catch (Exception ex) {
            System.err.println("An error occurred: " + ex);
        } finally {
            entityManager.getTransaction().commit();
            entityManager.close();
            factory.close();
        }
```

Now, I leave it to you as an exercise. Here I'm persisting four Employees and two Department entities, run this code, you will scroll down, and you'll see that our various callback methods are invoked at the right time. These are the call backs that we configured using entity listeners.

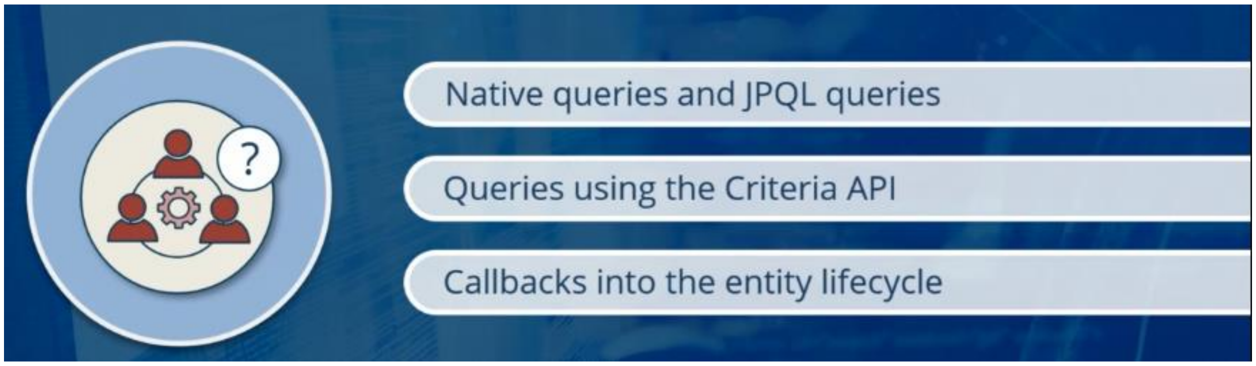


```
************ Before persisting department object: tech
Hibernate:
    insert
    into
        departments
        (name)
    values
        (?)
************ After persisting department object: tech
************ Before persisting employee object: Alice
Hibernate:
    insert
    into
        employees
        (department_id, name)
    values
        (?, ?)
************ After persisting employee object: Alice
************ Before persisting employee object: Ben
Hibernate:
    insert
    into
        employees
        (department_id, name)
    values
        (?, ?)
************ After persisting employee object: Ben
************ Before persisting department object: Operations
Hibernate:
    insert
    into
        departments
        (name)
    values
        (?)
************ After persisting department object: Operations
************ Before persisting employee object: Cora
Hibernate:
    insert
    into
        employees
        (department_id, name)
    values
        (?, ?)
************ After persisting employee object: Cora
```

```
************ Before persisting employee object: Dennis
Hibernate:
    insert
    into
        employees
        (department_id, name)
    values
        (?, ?)
************ After persisting employee object: Dennis
Hibernate:
    update
        employees
    set
        department_id=?
    where
        id=?
Hibernate:
    update
        employees
    set
        department_id=?
    where
        id=?
Hibernate:
    update
        employees
    set
        department_id=?
    where
        id=?
Hibernate:
    update
        employees
    set
        department_id=?
    where
        id=?
```

## Course Summary

Executing Queries and Managing the Entity Lifecycle


Native queries and JPQL queries
Queries using the Criteria API
Callbacks into the entity lifecycle

In this course we explored a number of different ways to execute queries using JPA. And saw the use of callbacks to manage our Entity Lifecycle. We saw how JPA supports native Queries that are dependent on the SQL syntax of the underlying database implementation.

- We then discussed how it's better to have our queries be platform independent by using JPQL, the Java Persistence Query Language. JPQL supports dynamic parameterized queries which can be configured using named parameters as well as positional parameters. We saw how common queries can be reused using named query.
- We then got a quick overview of the CriteriaAPI, which allows us to programmatically construct dynamic queries. Thus avoiding syntax errors that might occur in our query specification. The CriteriaAPI is extremely powerful. But tends to be verbose and requires a lot of boilerplate code.
- We rounded off this course by seeing how we can use callbacks to plug into the entity lifecycle. We explored callbacks that are invoked before and after entity persistence, after entity load, before and after entity update, and before and after entity delete.