Contents

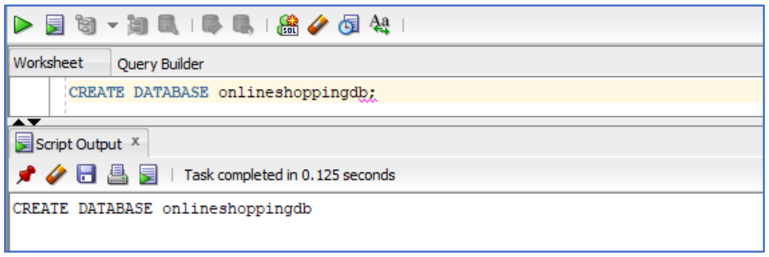
Course Overview

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 in high level programming languages to tables, which are the fundamental units of databases. JPA's module 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 are 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 the various kinds of relationships that you might want to express using JPA annotations and how these relationships map to relational table design. We will specifically look at unidirectional as well as bidirectional, one to one, one to many, many to one, and many to many relationships. Once you're done with this course, you will be able to model and express real world relationships between your entities using JPA annotations. And finally, configure how you want these relationships set up using underlying database tables.

One-to-one Unidirectional Mapping

Here we are on our MySQL Workbench. I'm going to create a new database that I'll use for this demo and for some other demos that follow.

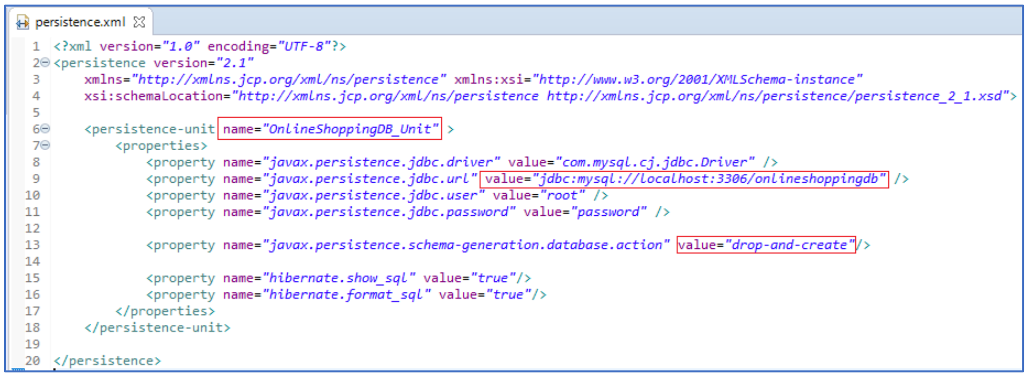


```
CREATE DATABASE onlineshoppingdb;
```

Task completed in 0.125 seconds

```
CREATE DATABASE onlineshoppingdb
```

This database is called *onlineshoppingdb*, and we'll imagine that this is a database which contains the tables for an e-commerce site. Database has been successfully created, we can switch over to our **persistence.xml** file.

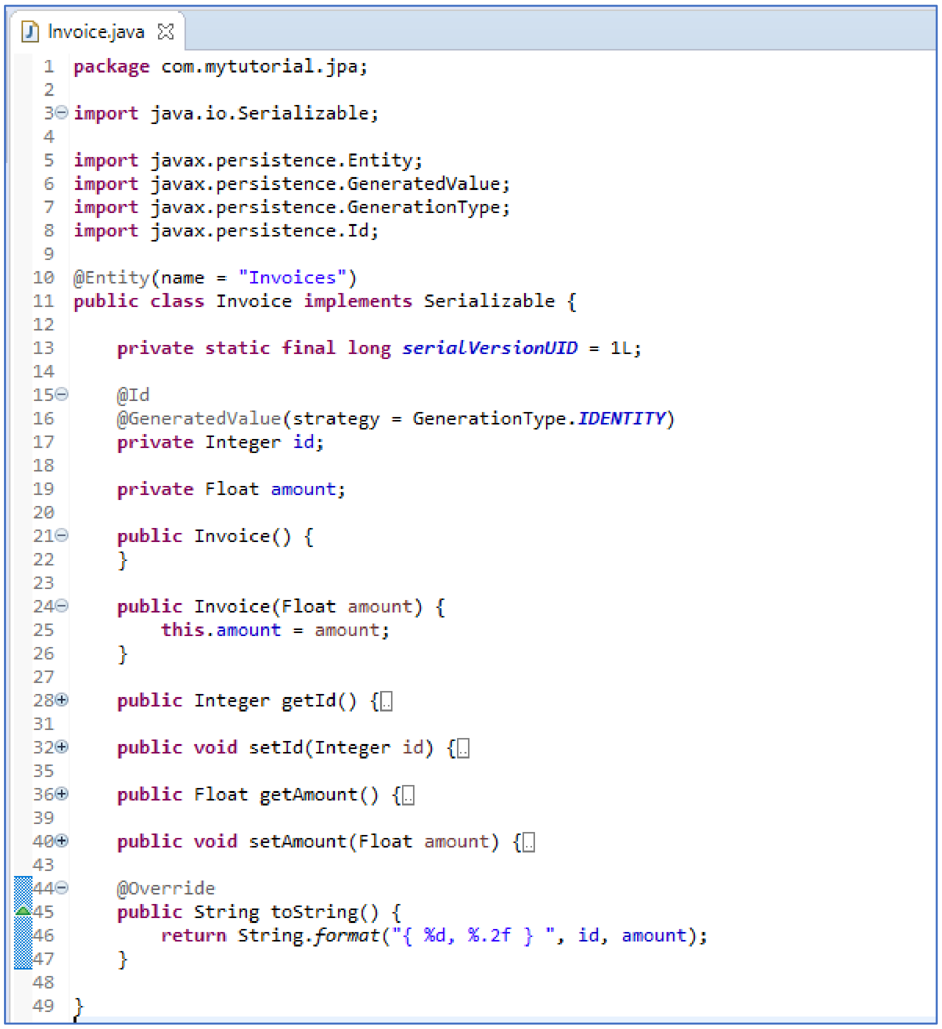


```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
    xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">

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

</persistence>
```

- Because we have a new database that we are working with, that is our new persistence-unit, the persistence-unit name is OnlineShoppingDB_Unit.
- Observe the value of the **JDBC URL**, it's *localhost:3306/onlineshoppingdb*.
- Make sure you specify the name and password of the user that you're connecting as. We're connecting as the root user, our password happens to be password.
- The database action property value is drop-and-create, which means each time we run this application, we'll drop all tables in our onlineshoppingdb database, and recreate those tables, and re-perform our create, read, update, delete operations.

We are now ready to set up the entity classes that correspond to our database tables.

The first class that I set up is an Invoice. You can imagine that when you place an order on an e-commerce site, you get an invoice for that order. That is what this class will represent, **Invoice.java**.

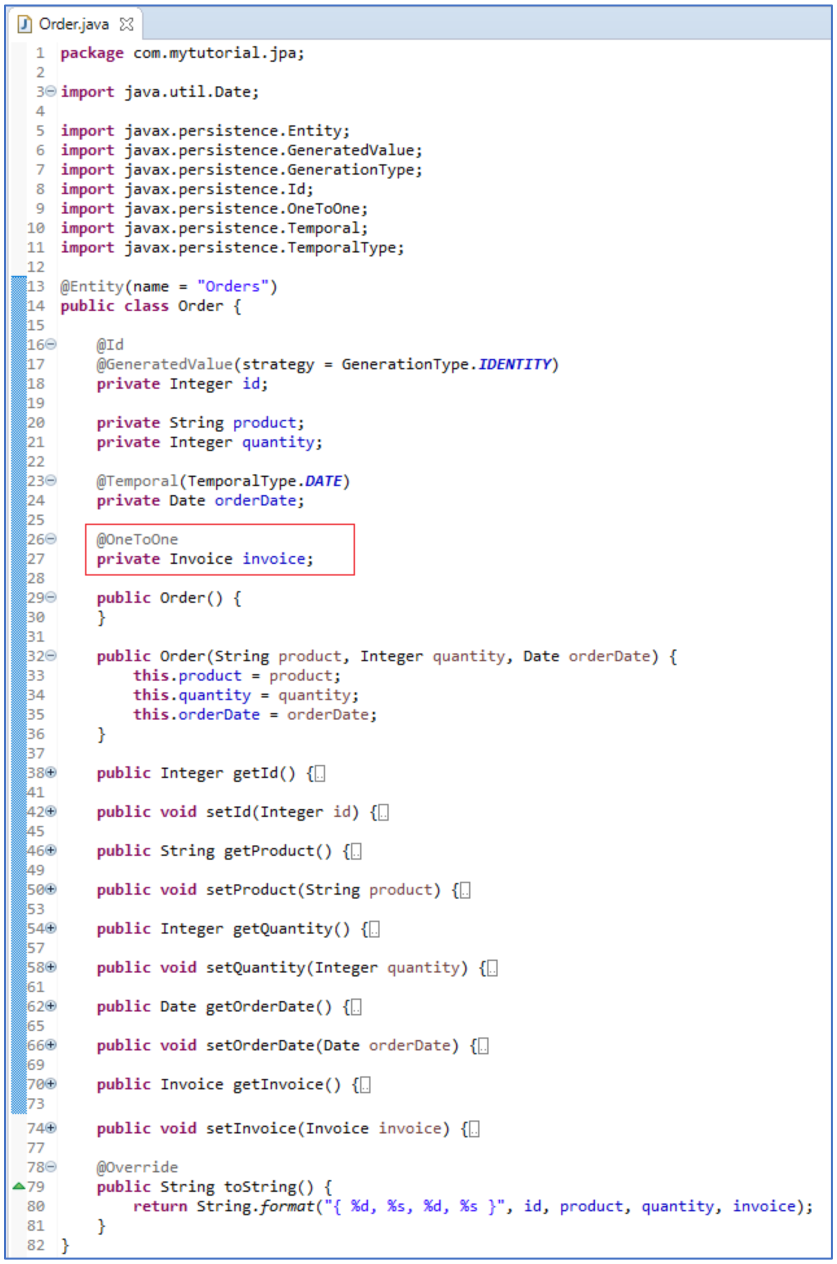


```
package com.mytutorial.jpa;

import java.io.Serializable;

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

@Entity(name = "Invoices")
public class Invoice implements Serializable {

    private static final long serialVersionUID = 1L;

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

    private Float amount;

    public Invoice() {
    }

    public Invoice(Float amount) {
        this.amount = amount;
    }

    public Integer getId() {[]

    public void setId(Integer id) {[]

    public Float getAmount() {[]

    public void setAmount(Float amount) {[]

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

}
```

- Go ahead and set up the import statements for all of the jpa annotations,
- And specify the **@Entity** annotation for this invoice class.
  Now I've specified an additional argument which we haven't seen before for this entity annotation, and that is the name property. The name property allows you to configure the name of the database table where you want these invoice objects to be stored. This is an alternative way to configure the name of the database table. We've already seen the **@Table** annotation before.
  Now because my underlying table will store Invoices, that is invoice objects, I have specified the plural form of Invoice as the name of my table, Invoices.
- I'll also have this Invoice class **implement Serializable**. As far as possible, you should have all of your Entity objects implement the Serializable interface, because these will be used in a larger context. Maybe you're using it within a Spring application where these entities map to JavaBeans and need to be serializable.

- Now, let's add the member variables, that is the columns of our database table.
  We have the id column with the identity generation strategy. We have the amount of the invoice as well.
- Specify the default no argument **constructor**, and also specify any additional constructors that you want.
- I'm also going to set up **getters** and **setters** for all of the individual member variables,
- And also override the **toString()** method of the object base class giving us a string representation of the invoice.

I'll now create a new entity class called Order which corresponds to the Order's table in my underlying database. Create this new class **Order.java**.


Order.java

```java
package com.mytutorial.jpa;

import java.util.Date;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity(name = "Orders")
public class Order {

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

    private String product;
    private Integer quantity;

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    @OneToOne
    private Invoice invoice;

    public Order() {
    }

    public Order(String product, Integer quantity, Date orderDate) {
        this.product = product;
        this.quantity = quantity;
        this.orderDate = orderDate;
    }

    public Integer getId() {

    public void setId(Integer id) {

    public String getProduct() {

    public void setProduct(String product) {

    public Integer getQuantity() {

    public void setQuantity(Integer quantity) {

    public Date getOrderDate() {

    public void setOrderDate(Date orderDate) {

    public Invoice getInvoice() {

    public void setInvoice(Invoice invoice) {

    @Override
    public String toString() {
        return String.format("{ %d, %s, %d, %s }", id, product, quantity, invoice);
    }
}
```

- As usual, we set up the import statements for all of the JPA annotations that we'll use within this Order class.
- And we'll also tag the Order class with the **@Entity** annotation.
- Once again, I specify a value for the **name** property, the name of the table will be Orders.
- Set up the member variables for this Order table.

And here within the member variables, you'll see something interesting. First, let's take a look at what we already know.

- The id column is the primary key for the orders table. It has a generated value using `GenerationType.`***`IDENTITY`***.
- We have a `product` and `quantity` column as well for every order. The `product` is simply the name of the product.
- We have the `orderDate` column using the **@Temporal** annotation, TemporalType is **Date**. That is, only the date will be persisted, not the time.

Every order that is placed on this e-commerce site is associated with an invoice, and I want to represent this relationship. And the way I do this is by using a one is to one relation map.

- This is specified using an annotation. I have a reference to the invoice that corresponds to this order, and I have tagged it with the annotation **@OneToOne**.

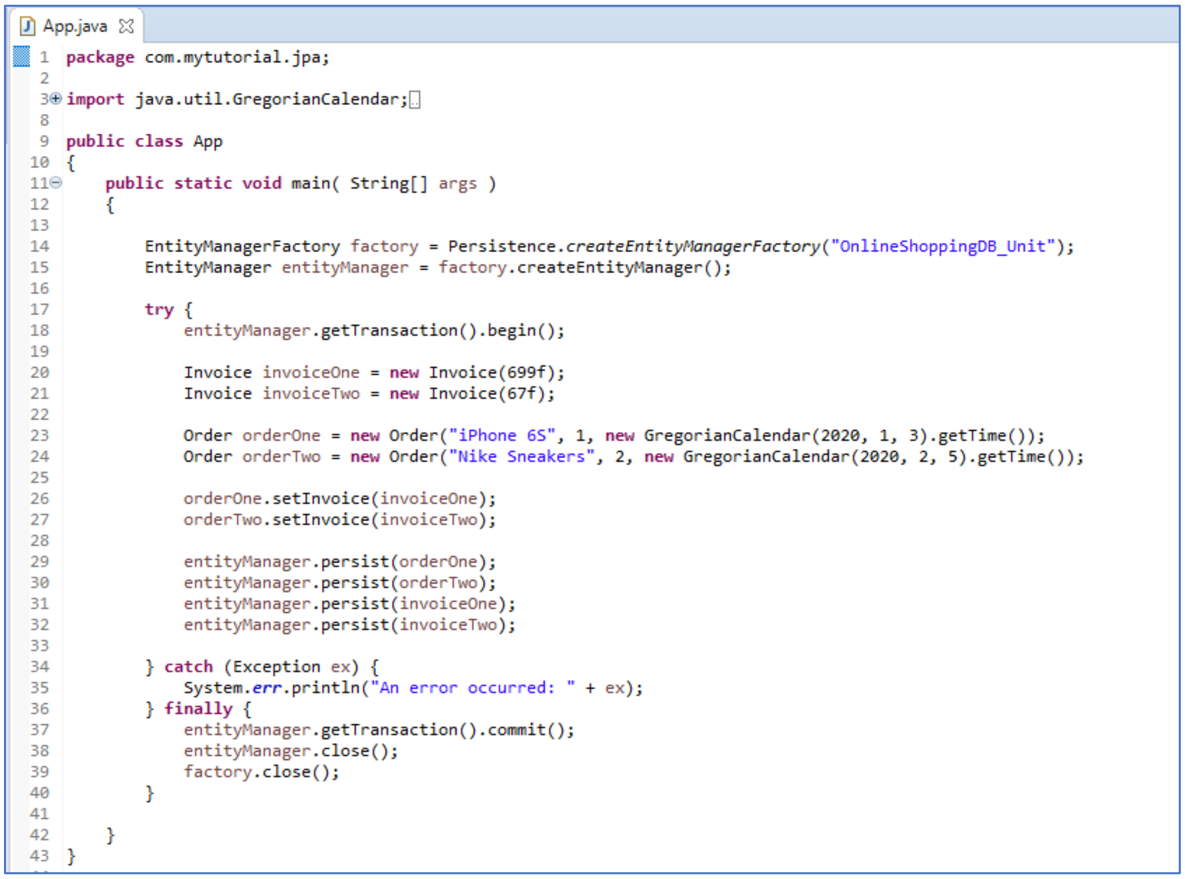
Now I have a reference from the *order* to the *invoice* and this makes this order entity **the owning entity in this relationship**. Also observe that because I refer the *invoice* entity from the *order* entity, this relationship is **unidirectional**, it goes from the order to the invoice. There is no relationship in the reverse direction from the invoice to the order.

This is what the one is to one relationship indicates, every *order* is associated with exactly one *invoice*, and every *invoice* corresponds to just one *order*. Setting up a **@OneToOne** mapping in this order basically means that when we access the order entity, we should be able to reference the invoice that comes along with the order.

- The remaining bits of code here are familiar. We have the default no argument **constructor**, and the Order constructor which takes as an input argument the `product`, the `quantity`, and the `orderDate`. We haven't specified the invoice here, we'll set the invoice corresponding to the order using a setup.
- As usual, we set up **getters** and **setters** for all of our member variables, `id`, and `product` getters and setters are seen here.
- We'll also set up **getters** and **setters** for `quantity`, `orderDate`, and also the `invoice` entity that has a one is to one relationship with this Order entity.
- I've also overridden the **toString()** method to represent an order. And when we print out the contents of an order **toString()**, we'll also print out the details of the corresponding invoice.

Now that we've set up this one is to one relationship between our *Order* and *Invoice* entities, this is a **unidirectional** relationship right now.

Let's head over to **App.java**, and write some code to create some order objects and invoice objects in our database table.

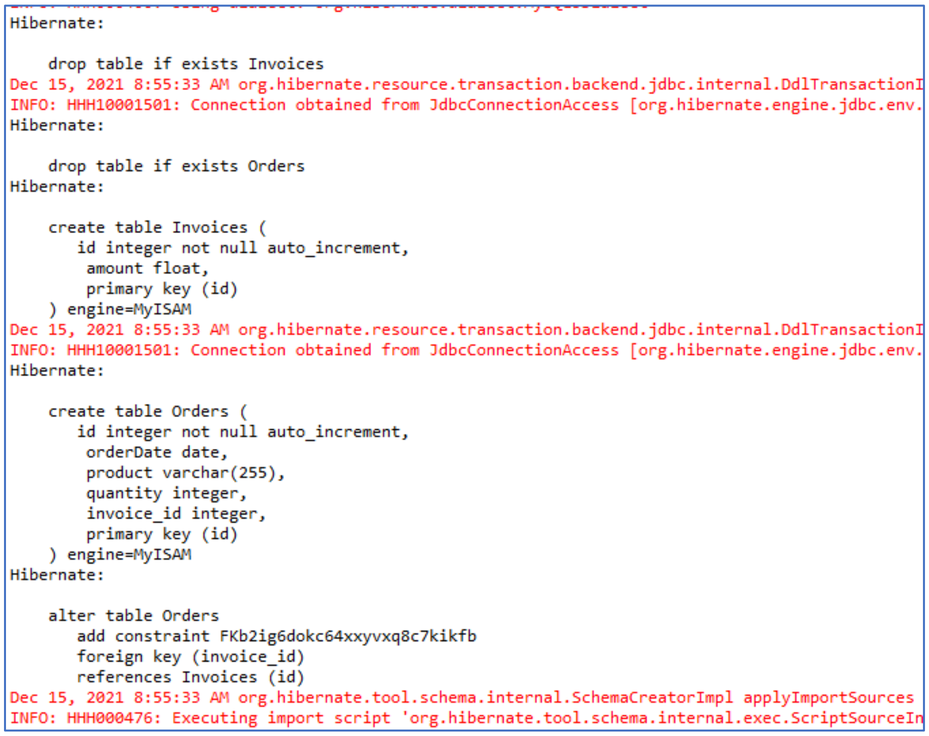


App.java

```java
 1  package com.mytutorial.jpa;
 2
 3  import java.util.GregorianCalendar;
 8
 9  public class App
10  {
11      public static void main( String[] args )
12      {
13
14          EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
15          EntityManager entityManager = factory.createEntityManager();
16
17          try {
18              entityManager.getTransaction().begin();
19
20              Invoice invoiceOne = new Invoice(699f);
21              Invoice invoiceTwo = new Invoice(67f);
22
23              Order orderOne = new Order("iPhone 6S", 1, new GregorianCalendar(2020, 1, 3).getTime());
24              Order orderTwo = new Order("Nike Sneakers", 2, new GregorianCalendar(2020, 2, 5).getTime());
25
26              orderOne.setInvoice(invoiceOne);
27              orderTwo.setInvoice(invoiceTwo);
28
29              entityManager.persist(orderOne);
30              entityManager.persist(orderTwo);
31              entityManager.persist(invoiceOne);
32              entityManager.persist(invoiceTwo);
33
34          } catch (Exception ex) {
35              System.err.println("An error occurred: " + ex);
36          } finally {
37              entityManager.getTransaction().commit();
38              entityManager.close();
39              factory.close();
40          }
41
42      }
43  }
```

- We'll use the EntityManagerFactory and the EntityManager to manage the persistence life cycle of our objects.
- Within a transaction, make sure that you call entityManager.getTransaction.begin().
- I'll instantiate two invoice objects, this I do on lines 21 and 20.
- I'll instantiate two order objects as well on lines 23 and 24.
- I then set a reference to the invoice objects on the order object.
  Remember, this is a unidirectional relationship. I call orderOne.setInvoice and setInvoiceOne.
  And I call orderTwo.setInvoice and set the second invoice.
  This is what allows the Hibernate implementation to correctly set up the one is to one relationship that exists between order and invoice.
  Order here being the owning entity, which is why we call order.setInvoice() and not the reverse.
- We then have to persist all of the entities that we want to have saved in our table. We call persist() on both the orders as well as both the invoices.
- Once you perform the create operation, make sure you commit the transaction, close the entityManager as well as the EntityManagerFactory.

Let's go ahead and run this code and take a look at the console statements.



```
Hibernate:

    drop table if exists Invoices
Dec 15, 2021 8:55:33 AM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionI
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.
Hibernate:

    drop table if exists Orders
Hibernate:

    create table Invoices (
       id integer not null auto_increment,
        amount float,
        primary key (id)
    ) engine=MyISAM
Dec 15, 2021 8:55:33 AM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionI
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.
Hibernate:

    create table Orders (
       id integer not null auto_increment,
        orderDate date,
        product varchar(255),
        quantity integer,
        invoice_id integer,
        primary key (id)
    ) engine=MyISAM
Hibernate:

    alter table Orders
       add constraint FKb2ig6dokc64xxyvxq8c7kikfb
       foreign key (invoice_id)
       references Invoices (id)
Dec 15, 2021 8:55:33 AM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceIn
```

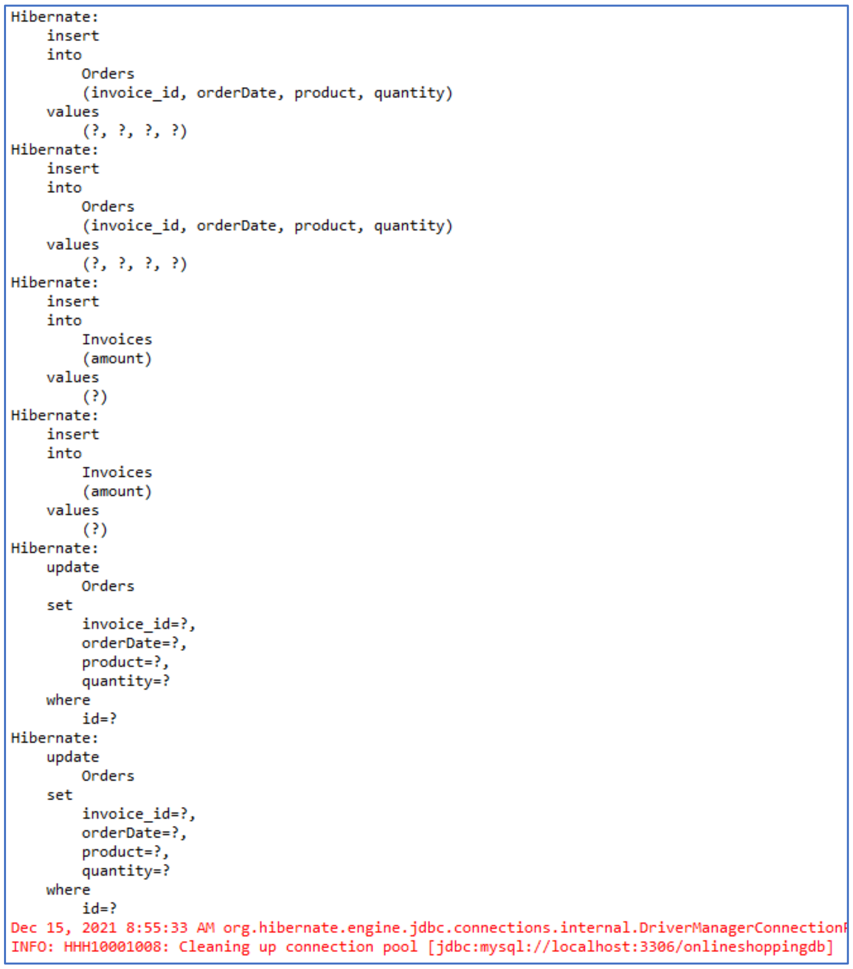
That'll tell us what tables have been created and how the one is to one mapping constraints have been set up.

- Observe here the *Invoices* table where the columns are id and amount, the primary key is the **id**.
- After which the orders tables has been created, we have the **id**, **orderDate**, the product ordered, the **quantity**, and the **invoice_id** that is an integer.
- Observe this additional column that is the `invoice_id`. Remember, the *Order* entity has a reference to the *Invoice*.
- The primary key of this table is the **id**.

Now if you scroll down below, you'll see that Hibernate has added a **foreign key** constraint to the *Orders* table.

- This constraint specifies that the Invoice id column in the orders table references the id column in the invoices table. So this is a foreign key that references the primary key of the invoices table.
  And this is how the one is to one mapping between orders and invoices is set up.
  We know what invoice a particular order is associated with using this invoice_id foreign key.

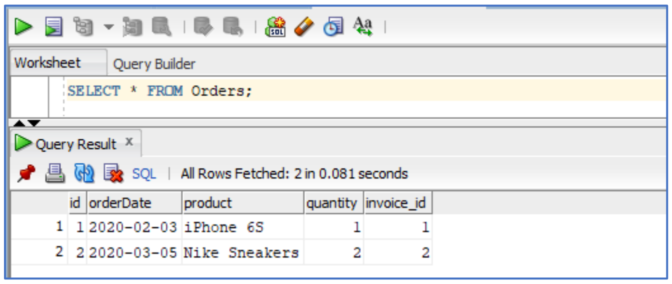
Let's scroll down further.



```
Hibernate: 
    insert 
    into
        Orders
        (invoice_id, orderDate, product, quantity) 
    values
        (?, ?, ?, ?)
Hibernate: 
    insert 
    into
        Orders
        (invoice_id, orderDate, product, quantity) 
    values
        (?, ?, ?, ?)
Hibernate: 
    insert 
    into
        Invoices
        (amount) 
    values
        (?)
Hibernate: 
    insert 
    into
        Invoices
        (amount) 
    values
        (?)
Hibernate: 
    update
        Orders 
    set
        invoice_id=?,
        orderDate=?,
        product=?,
        quantity=? 
    where
        id=?
Hibernate: 
    update
        Orders 
    set
        invoice_id=?,
        orderDate=?,
        product=?,
        quantity=? 
    where
        id=?
Dec 15, 2021 8:55:33 AM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionP
INFO: HHH10001008: Cleaning up connection pool [jdbc:mysql://localhost:3306/onlineshoppingdb]
```

And here you'll see the remaining insert statements inserting data into our two tables, orders and invoices.

Let's switch over to MySQL Workbench and run a *select* * on the *Orders* table to see entries here.



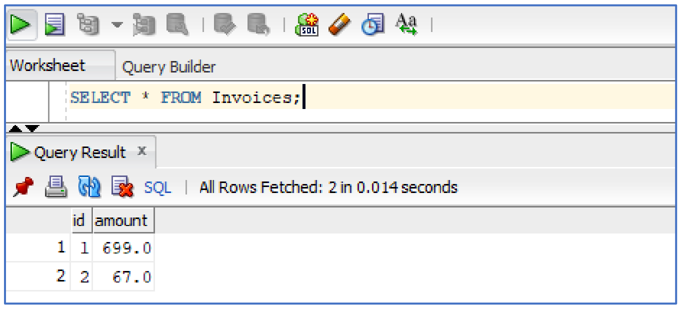
```
SELECT * FROM Orders;
```

All Rows Fetched: 2 in 0.081 seconds

	id	orderDate	product	quantity	invoice_id
1	1	2020-02-03	iPhone 6S	1	1
2	2	2020-03-05	Nike Sneakers	2	2

- Observe that the first order here where we bought the "iPhone 6S" is associated with the `invoice_id` 1, that is the *Invoice* with `id` 1.
- And where we bought Nike Sneakers, that's associated with `invoice_id` 2.

Let's take a look at the *Invoices* table here within our onlineshoppingdb.



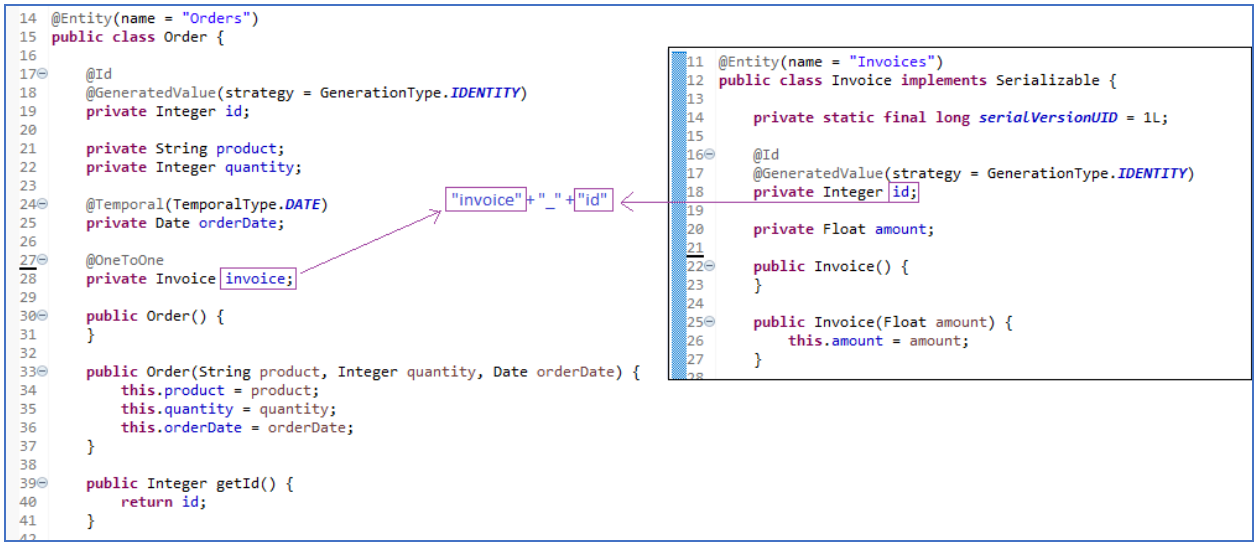
```
SELECT * FROM Invoices;
```

All Rows Fetched: 2 in 0.014 seconds

	id	amount
1	1	699.0
2	2	67.0

- And here are the two invoices that we have persisted, one with `id` 1, that is 699 for the "iPhone 6S",
- and one with `id` 2. $67 was the total cost of the "Nike Sneakers" that we bought.

This is how we set up a one is to one mapping using a foreign key constraint.



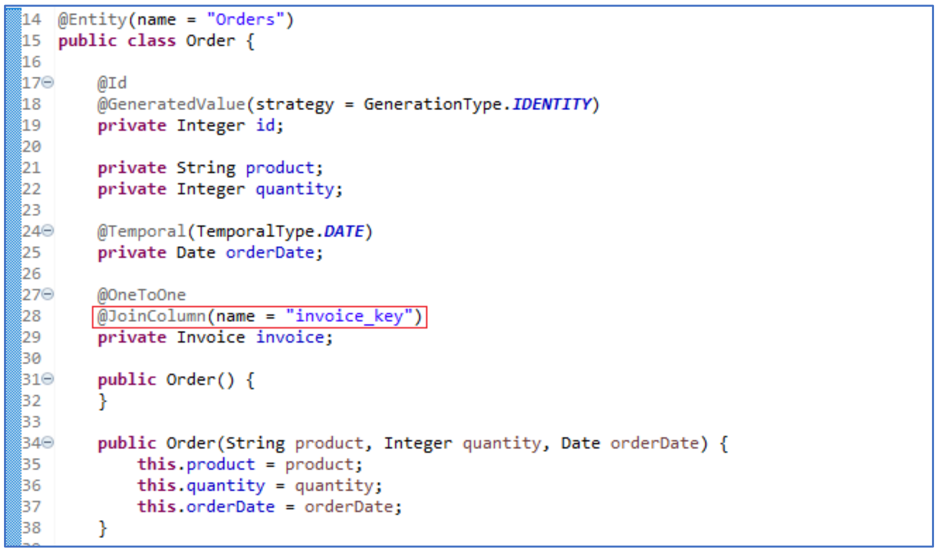
```
@Entity(name = "Orders")
public class Order {

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

    private String product;
    private Integer quantity;

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    @OneToOne
    private Invoice invoice;

    public Order() {
    }

    public Order(String product, Integer quantity, Date orderDate) {
        this.product = product;
        this.quantity = quantity;
        this.orderDate = orderDate;
    }

    public Integer getId() {
        return id;
    }
```

```
@Entity(name = "Invoices")
public class Invoice implements Serializable {

    private static final long serialVersionUID = 1L;

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

    private Float amount;

    public Invoice() {
    }

    public Invoice(Float amount) {
        this.amount = amount;
    }
```

One-to-one Mapping with Join Columns

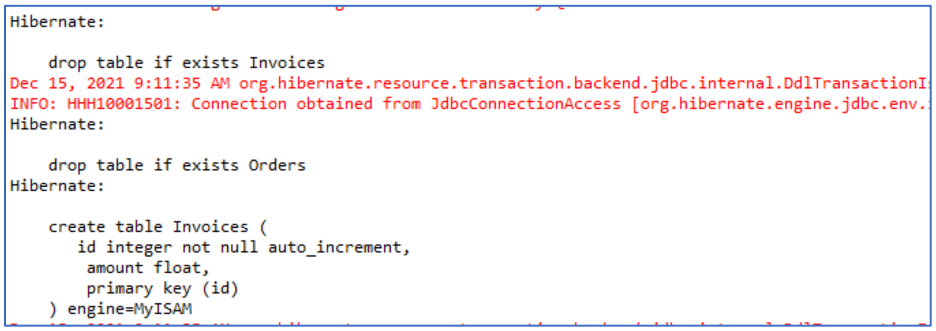
We've seen earlier that a one is to one mapping has been set up between the order entity and the invoice entity using the **@OneToOne** annotation in jpa.

What we have is a **unidirectional** mapping. There is a reference from the order entity to the invoice entity, but no reference in the reverse direction. We saw that the one is to one mapping was achieved by having a **foreign key** in the orders table, which references the invoice id that the order is associated with. Once we have this **foreign key reference** within the orders table, retrieving orders and invoices together is simply a join operation. And we can finally configure what column we want the orders table and the invoices table to be joined on using the **@JoinColumn** annotation.


```
@Entity(name = "Orders")
public class Order {

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

    private String product;
    private Integer quantity;

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    @OneToOne
    @JoinColumn(name = "invoice_key")
    private Invoice invoice;

    public Order() {
    }

    public Order(String product, Integer quantity, Date orderDate) {
        this.product = product;
        this.quantity = quantity;
        this.orderDate = orderDate;
    }
```

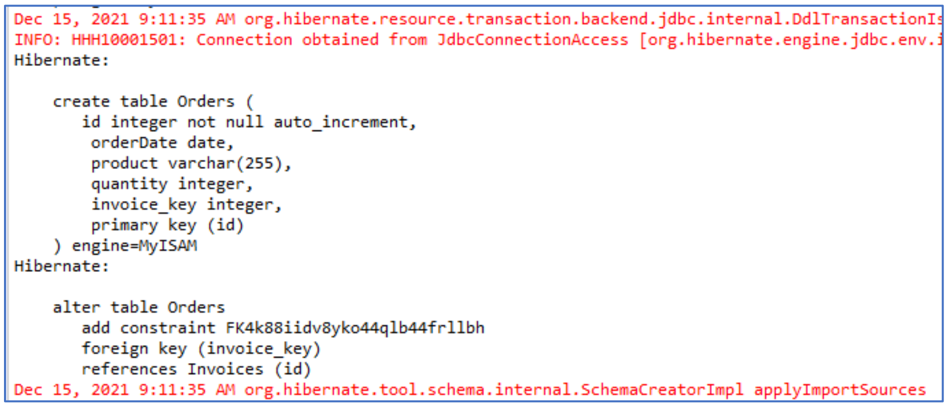
- Along with the **@OneToOne** annotation, I've added a **@JoinColumn** annotation to our reference to the *Invoice* corresponding to this *Order*.
- We'll need to add a specific import statement for this **@JoinColumn** annotation. The JoinColumn annotation is a part of jpa and you can see the `import javax.persistence.JoinColumn;`
- The *name* property of the **@JoinColumn** annotation allows you to specify the name of the column within your orders table that will hold the **foreign key reference** to the *Invoice* `id`. Here, the name of the column is `"invoice_key"`.

This is the only change that I'm going to make in the application. I'm now going to switch over to App.java and run the same code.



```
Hibernate: 

    drop table if exists Invoices
Dec 15, 2021 9:11:35 AM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionI
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.
Hibernate: 

    drop table if exists Orders
Hibernate: 

    create table Invoices (
       id integer not null auto_increment,
        amount float,
        primary key (id)
    ) engine=MyISAM
```

Here's the invoices table that has been created, no change there, we have the `id`, and the `amount` of the invoice. If you scroll down a little, you'll see that Hibernate has created the orders table.

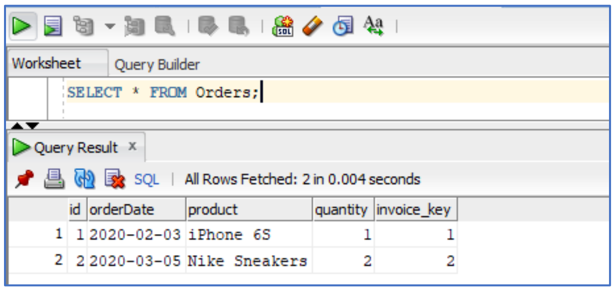


```
Dec 15, 2021 9:11:35 AM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIs
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.i
Hibernate: 

    create table Orders (
       id integer not null auto_increment,
        orderDate date,
        product varchar(255),
        quantity integer,
        invoice_key integer,
        primary key (id)
    ) engine=MyISAM
Hibernate: 

    alter table Orders 
       add constraint FK4k88iidv8yko44qlb44frllbh 
       foreign key (invoice_key) 
       references Invoices (id)
Dec 15, 2021 9:11:35 AM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
```

Observe that we no longer have an `invoice_id` column here, we have a column called `invoice_key` that is of an **Integer** type.

This is thanks to our **@JoinColumn** annotation. We've configured the name of the column to be `invoice_key`. And you can see a **foreign key** constraint has been added to the *Orders* table as well. The `invoice_key` *references* the `id` column in the *Invoices* table, which is the **primary key** of *Invoices*.

We can quickly verify this in MySQL Workbench.
Let's run a *select ** on the *Orders* table here.



Worksheet Query Builder

```
SELECT * FROM Orders;
```

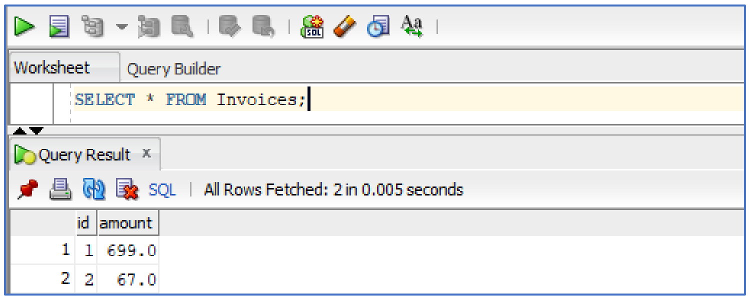
Query Result

SQL | All Rows Fetched: 2 in 0.004 seconds

```
   id orderDate  product      quantity invoice_key
1   1 2020-02-03 iPhone 6S           1           1
2   2 2020-03-05 Nike Sneakers       2           2
```

You'll see that orders has an `invoice_key` column where the first order has the `invoice_key` 1, the second order has the `invoice_key` 2.

If we quickly run a *select ** on the *Invoices* table, you'll see that we have two entries here corresponding to *Invoice* `id` 1 and *Invoice* `id` 2.



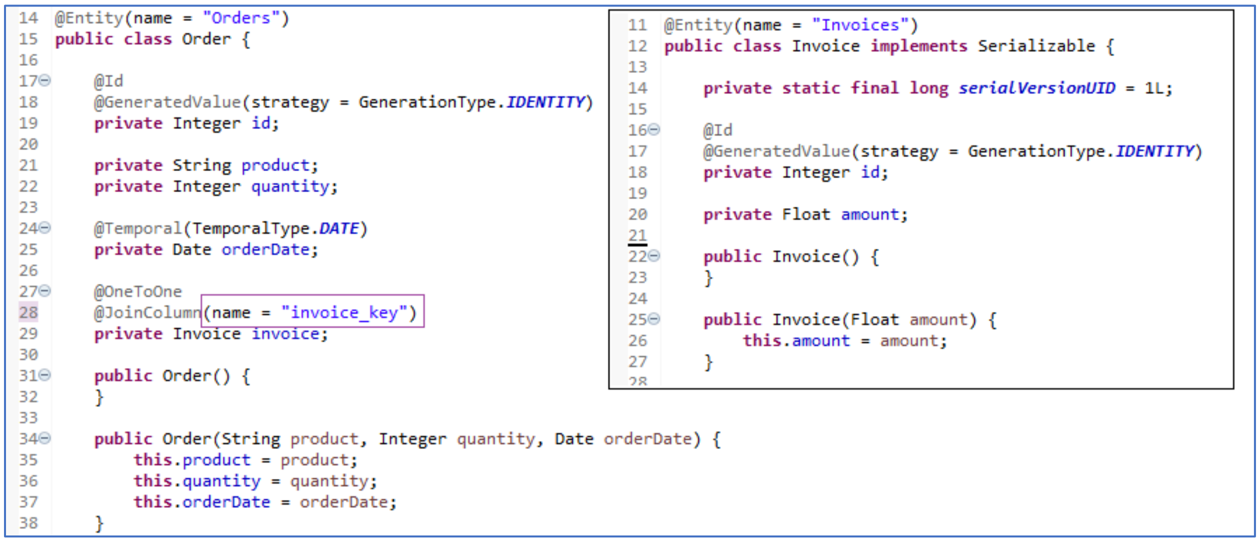
Worksheet Query Builder

```
SELECT * FROM Invoices;
```

Query Result

SQL | All Rows Fetched: 2 in 0.005 seconds

```
   id amount
1   1  699.0
2   2   67.0
```

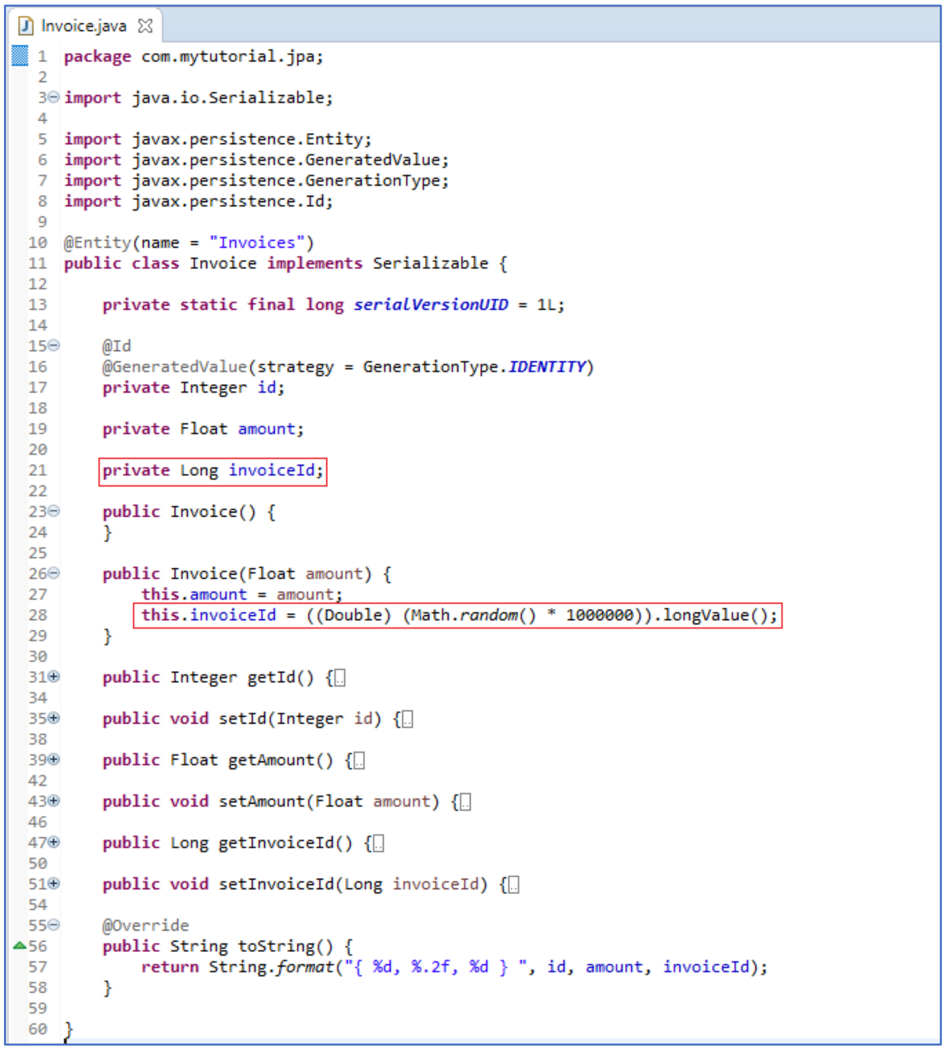


```
@Entity(name = "Orders")
public class Order {

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

    private String product;
    private Integer quantity;

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    @OneToOne
    @JoinColumn(name = "invoice_key")
    private Invoice invoice;

    public Order() {
    }

    public Order(String product, Integer quantity, Date orderDate) {
        this.product = product;
        this.quantity = quantity;
        this.orderDate = orderDate;
    }
```

```
@Entity(name = "Invoices")
public class Invoice implements Serializable {

    private static final long serialVersionUID = 1L;

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

    private Float amount;

    public Invoice() {
    }

    public Invoice(Float amount) {
        this.amount = amount;
    }
```

Back to our Eclipse IDE. So far when we specify the one-to-one mapping between the *Order* entity and the *Invoice* entity, the **primary key** of the invoices table was automatically added as a foreign key constraint on the *Orders* table. But what if we want our **foreign key** in the *Orders* table to reference some other member variable within the *Invoice* table?

This is exactly what we'll configure next. Now our invoices entity, **Invoice.java** is the file that we are on right now, already has a **primary key**.



Invoice.java

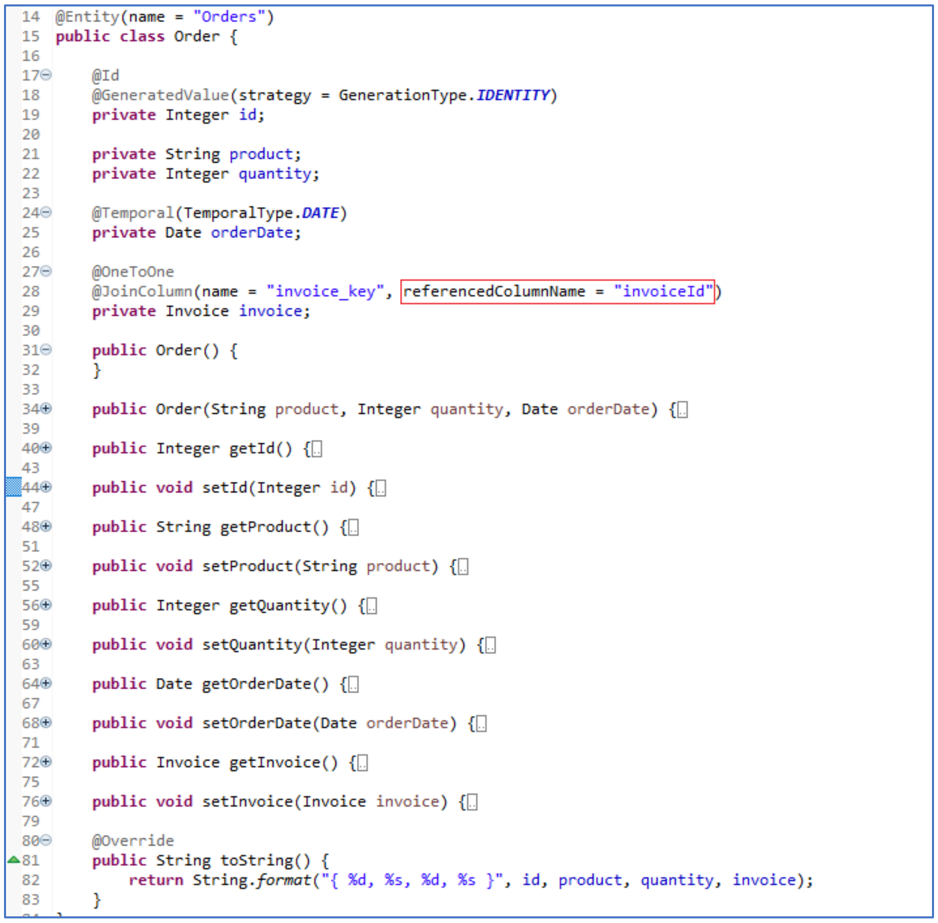
```
package com.mytutorial.jpa;

import java.io.Serializable;

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

@Entity(name = "Invoices")
public class Invoice implements Serializable {

    private static final long serialVersionUID = 1L;

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

    private Float amount;

    private Long invoiceId;

    public Invoice() {
    }

    public Invoice(Float amount) {
        this.amount = amount;
        this.invoiceId = ((Double) (Math.random() * 1000000)).longValue();
    }

    public Integer getId() {

    public void setId(Integer id) {

    public Float getAmount() {

    public void setAmount(Float amount) {

    public Long getInvoiceId() {

    public void setInvoiceId(Long invoiceId) {

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

}
```

- Now what if this primary key is only for the purposes of storing invoice data in our relational database table? But within our e-commerce site and application, every invoice has a **uniquely generated id** which is some very long number. And this is what *Invoices* are referenced by in the rest of the app. Now this `invoiceId` may be stored in a different column in the invoices table, called `invoiceId` not just plain `Id`, which is the **primary key**. This `invoiceId` is unique for every invoice generated by our system, and this is what we want the *Orders* table to reference in its foreign key. This is what we'll configure.
- I've set up the `invoiceId` here, I'm going to update the constructor of this invoice entity to randomly generate a unique `invoiceId` for every invoice.

Now when we use `Math.random()`, this `invoiceId` is of course not guaranteed to be unique, but for the purposes of the demo, this will suffice. Imagine that this is the `invoiceId` that the rest of our e-commerce system uses to reference an invoice. And we want to set up our orders table to reference this `invoiceId` as well.

- I'm going to change the **toString()** representation of an invoice to include this `invoiceId`.

Now let's head back to the **Order.java** file, where I'll configure the **@JoinColumn** attribute to reference a particular column from the invoices table.

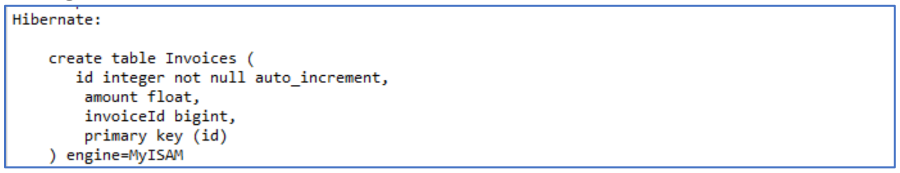


```java
@Entity(name = "Orders")
public class Order {

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

    private String product;
    private Integer quantity;

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    @OneToOne
    @JoinColumn(name = "invoice_key", referencedColumnName = "invoiceId")
    private Invoice invoice;

    public Order() {
    }

    public Order(String product, Integer quantity, Date orderDate) {

    public Integer getId() {

    public void setId(Integer id) {

    public String getProduct() {

    public void setProduct(String product) {

    public Integer getQuantity() {

    public void setQuantity(Integer quantity) {

    public Date getOrderDate() {

    public void setOrderDate(Date orderDate) {

    public Invoice getInvoice() {

    public void setInvoice(Invoice invoice) {

    @Override
    public String toString() {
        return String.format("{ %d, %s, %d, %s }", id, product, quantity, invoice);
    }
```

- I've chained **@JoinColumn** to take in an additional input argument, that is the referencedColumnName. This referencedColumnName refers to the name of a column in the *Invoices* table that is at the other end of this one is to one mapping. The referencedColumnName that we have specified this `"invoiceId"`.

This **@JoinColumn** annotation along with the referencedColumnName tells Hibernate that the **foreign key** constraint should be set up using this `invoiceId` column, rather than simply using the **primary key** of the invoices table. So the **foreign key** constraint should be on the `invoiceId` column of the invoices table.

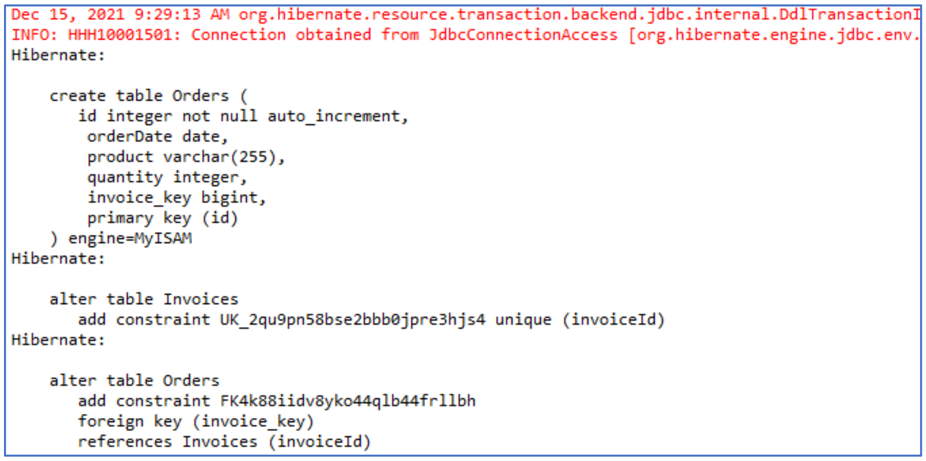
Let's go ahead and run this code.



```
Hibernate:

    create table Invoices (
       id integer not null auto_increment,
        amount float,
        invoiceId bigint,
        primary key (id)
    ) engine=MyISAM
```

If you look within the Console window, you'll see the creation of the invoices table. We have the primary key `Id`, the `amount` and an additional key called `invoiceId`, which is of type big integer.

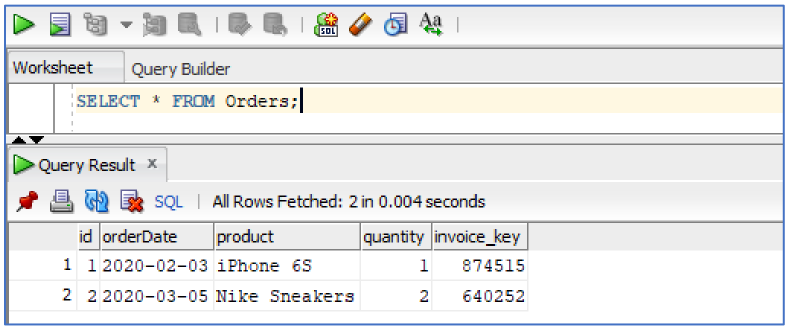
If you scroll down further, you'll see the SQL statement creating the *Orders* table.



```
Dec 15, 2021 9:29:13 AM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionI
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.
Hibernate:

    create table Orders (
       id integer not null auto_increment,
        orderDate date,
        product varchar(255),
        quantity integer,
        invoice_key bigint,
        primary key (id)
    ) engine=MyISAM
Hibernate:

    alter table Invoices
       add constraint UK_2qu9pn58bse2bbb0jpre3hjs4 unique (invoiceId)
Hibernate:

    alter table Orders
       add constraint FK4k88iidv8yko44qlb44frllbh
       foreign key (invoice_key)
       references Invoices (invoiceId)
```

- Notice the *Orders* table has an `invoice_key` column, which now is of type **bigint**. That's because this `invoice_key` references the `invoiceId`. And this reference is set up using the **foreign key constraint** that you see here.
- Before setting up the **foreign key constraint**, notice that Hibernate has correctly altered the invoices table to add a **uniqueness constraint** on the **invoiceId** column.
- And once this uniqueness constraint has been specified, we add a **foreign key reference** from the `invoice_key` column in the *Orders* table to the `invoiceId` column in the Invoices table. This foreign key explicitly references the `invoiceId` column, and not the primary key of the *Invoices* table. That's the change here.

Let's head over to MySQL Workbench and run a *select* * on the *Orders* table.



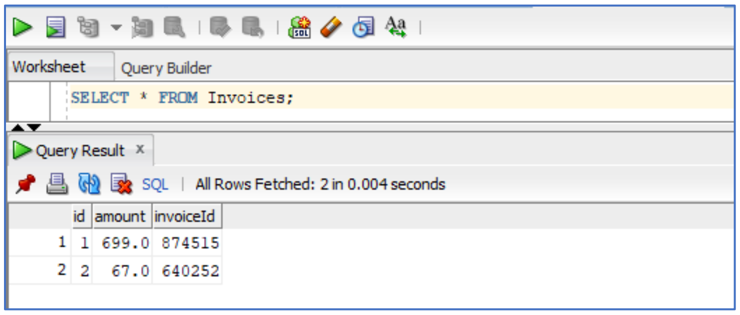
```
SELECT * FROM Orders;
```

All Rows Fetched: 2 in 0.004 seconds

	id	orderDate	product	quantity	invoice_key
1	1	2020-02-03	iPhone 6S	1	874515
2	2	2020-03-05	Nike Sneakers	2	640252

Observe that the `invoice_key` column contains entries that references the `invoiceId`. These are large numbers, not the auto generated primary keys for the invoices.

To further verify this, let's run a quick select * on the *Invoices* table.



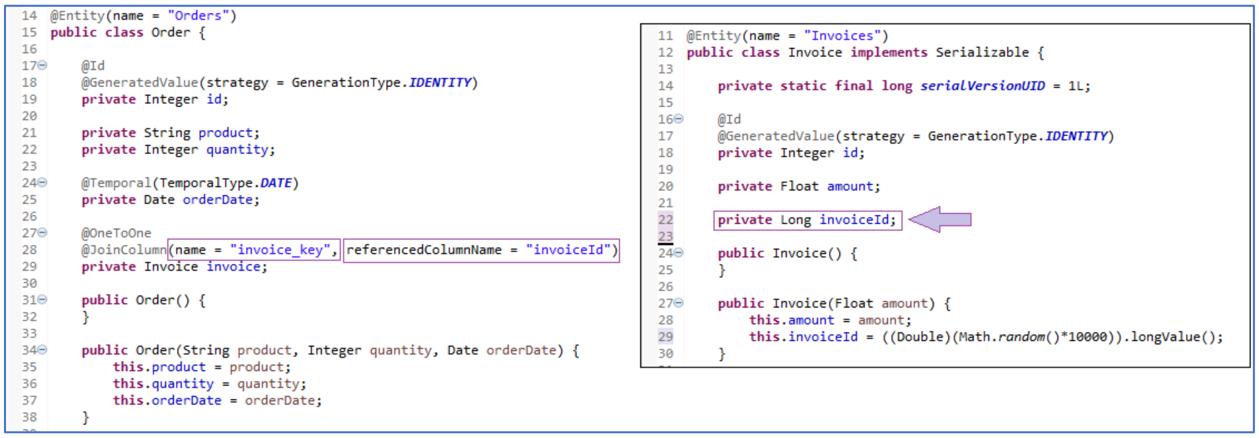
```
SELECT * FROM Invoices;
```

All Rows Fetched: 2 in 0.004 seconds

	id	amount	invoiceId
1	1	699.0	874515
2	2	67.0	640252

And you can see that every invoice now has three columns,

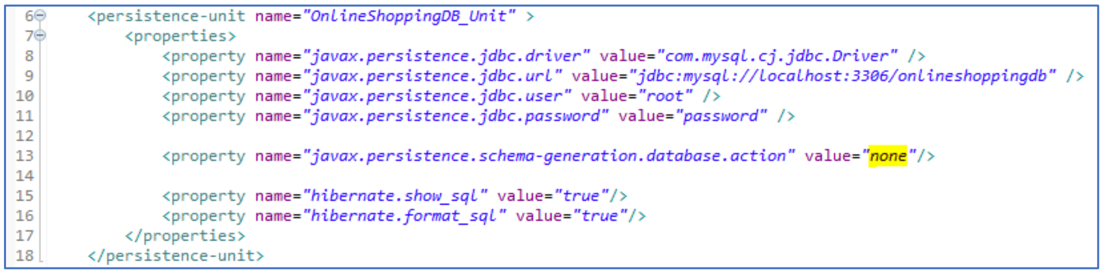
- the `id` that is the primary key for this relational database table,
- the `amount` of the invoice, and
- the `invoiceId` that is unique for every invoice in our system. And this is what is referenced by the *Orders* table.



```java
@Entity(name = "Orders")
public class Order {

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

    private String product;
    private Integer quantity;

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    @OneToOne
    @JoinColumn(name = "invoice_key", referencedColumnName = "invoiceId")
    private Invoice invoice;

    public Order() {
    }

    public Order(String product, Integer quantity, Date orderDate) {
        this.product = product;
        this.quantity = quantity;
        this.orderDate = orderDate;
    }
```

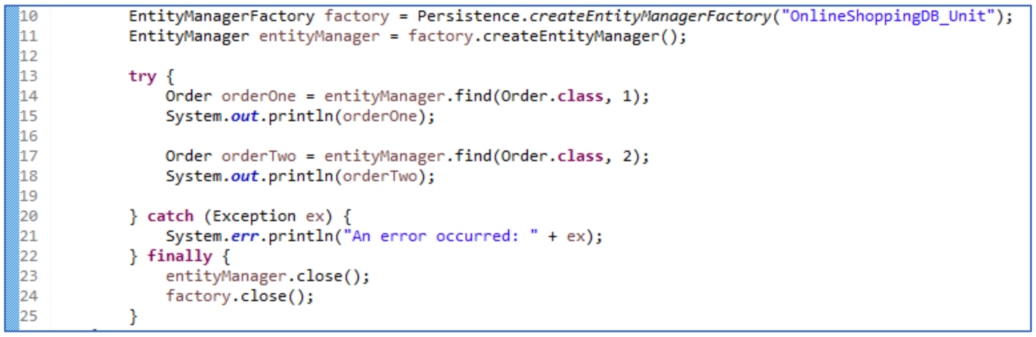
```java
@Entity(name = "Invoices")
public class Invoice implements Serializable {

    private static final long serialVersionUID = 1L;

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

    private Float amount;

    private Long invoiceId;

    public Invoice() {
    }

    public Invoice(Float amount) {
        this.amount = amount;
        this.invoiceId = ((Double)(Math.random()*10000)).longValue();
    }
```

Now that we know how to create entities which have a one is to one relationship mapping, let's see how we can **query** these entities. Here we are in **persistence.xml**, and I'm going to change this database action here, going from drop-and-create to none. I set it to none because I want to work with the existing onlineshoppingdb, and the tables that we have set up under onlineshoppingdb.



```
    <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="none"/>

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

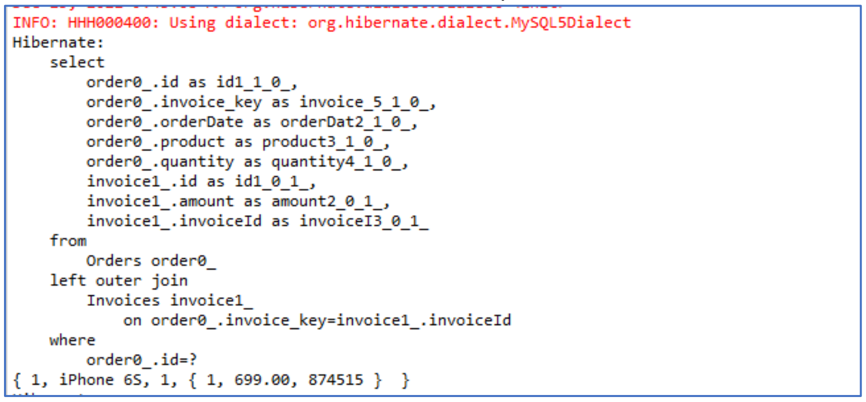
I'm now going to perform read operations on the records that we have in the orders and invoices table. Let's head over to **App.java**, where I've used the find method to perform read operations on the two order entries in our table.



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

        try {
            Order orderOne = entityManager.find(Order.class, 1);
            System.out.println(orderOne);

            Order orderTwo = entityManager.find(Order.class, 2);
            System.out.println(orderTwo);

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

- I've used `entityManager.find()` to retrieve the order entity with primary key 1, and then the order entry with primary key 2.
- And I've printed out `orderOne` and `orderTwo` out to screen.
  Now retrieving the order entities should retrieve the corresponding invoice entities as well.

Let's run this code and take a look at the queries that Hibernate runs behind the scenes.

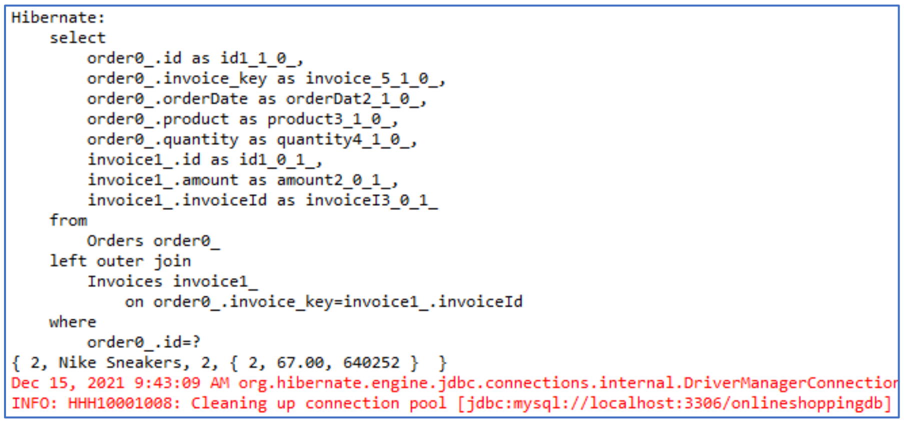


```
INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQL5Dialect
Hibernate: 
    select
        order0_.id as id1_1_0_,
        order0_.invoice_key as invoice_5_1_0_,
        order0_.orderDate as orderDat2_1_0_,
        order0_.product as product3_1_0_,
        order0_.quantity as quantity4_1_0_,
        invoice1_.id as id1_0_1_,
        invoice1_.amount as amount2_0_1_,
        invoice1_.invoiceId as invoiceI3_0_1_ 
    from
        Orders order0_ 
    left outer join
        Invoices invoice1_ 
            on order0_.invoice_key=invoice1_.invoiceId 
    where
        order0_.id=?
{ 1, iPhone 6S, 1, { 1, 699.00, 874515 }  }
```

- Here is a select query retrieving the first entity, the first order.
- In addition to all of the order details, you can see that fields from the invoice table are also queried using this select query.

- And if you scroll down a little bit, you'll see that the select operation actually performs a **left outer join**, a join operation between the *Orders* table and the *Invoices* table to retrieve the invoices corresponding to each order.
  Thanks to the one-to-one mapping that we had specified between order and invoice, retrieving an order also retrieves the corresponding invoice information.
- And here are the order details printed out to screen, the “iPhone 6S”, for $699.

Corresponding to the second invocation of the find method, here is the second select operation that Hibernate performs.



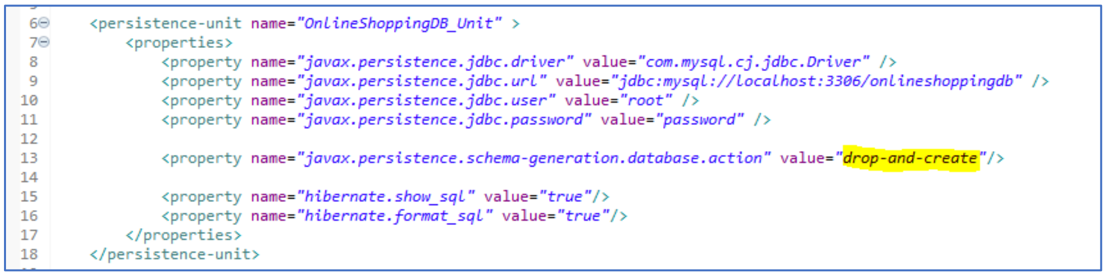
```
Hibernate:
    select
        order0_.id as id1_1_0_,
        order0_.invoice_key as invoice_5_1_0_,
        order0_.orderDate as orderDat2_1_0_,
        order0_.product as product3_1_0_,
        order0_.quantity as quantity4_1_0_,
        invoice1_.id as id1_0_1_,
        invoice1_.amount as amount2_0_1_,
        invoice1_.invoiceId as invoiceI3_0_1_
    from
        Orders order0_
    left outer join
        Invoices invoice1_
            on order0_.invoice_key=invoice1_.invoiceId
    where
        order0_.id=?
{ 2, Nike Sneakers, 2, { 2, 67.00, 640252 }  }
Dec 15, 2021 9:43:09 AM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectio
INFO: HHH10001008: Cleaning up connection pool [jdbc:mysql://localhost:3306/onlineshoppingdb]
```

- Once again, this involves a join operation between the *Orders* table and the *Invoices* table using the **invoiceId** foreign key.
- Here are the details of the second order printed out to screen, “Nike Sneakers” for $67.
- Both order and invoice information is available with a single retrieval.

One-to-one Mapping Bidirectional

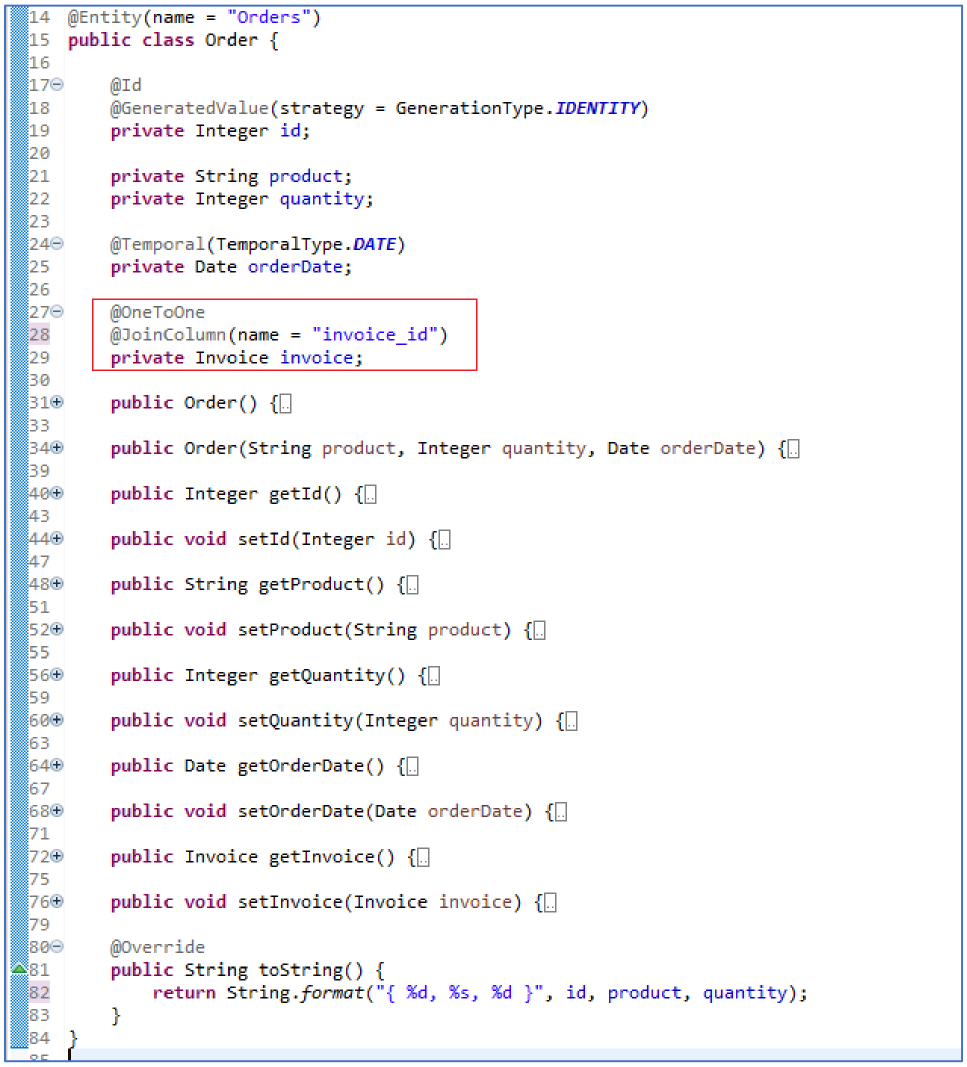
In this demo, we'll see how we can set up a **bidirectional** one-to-one mapping between the order entity and the invoice entity.

Back to **persistence.xml** and I'm going to change the database action to be drop-and-create.



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

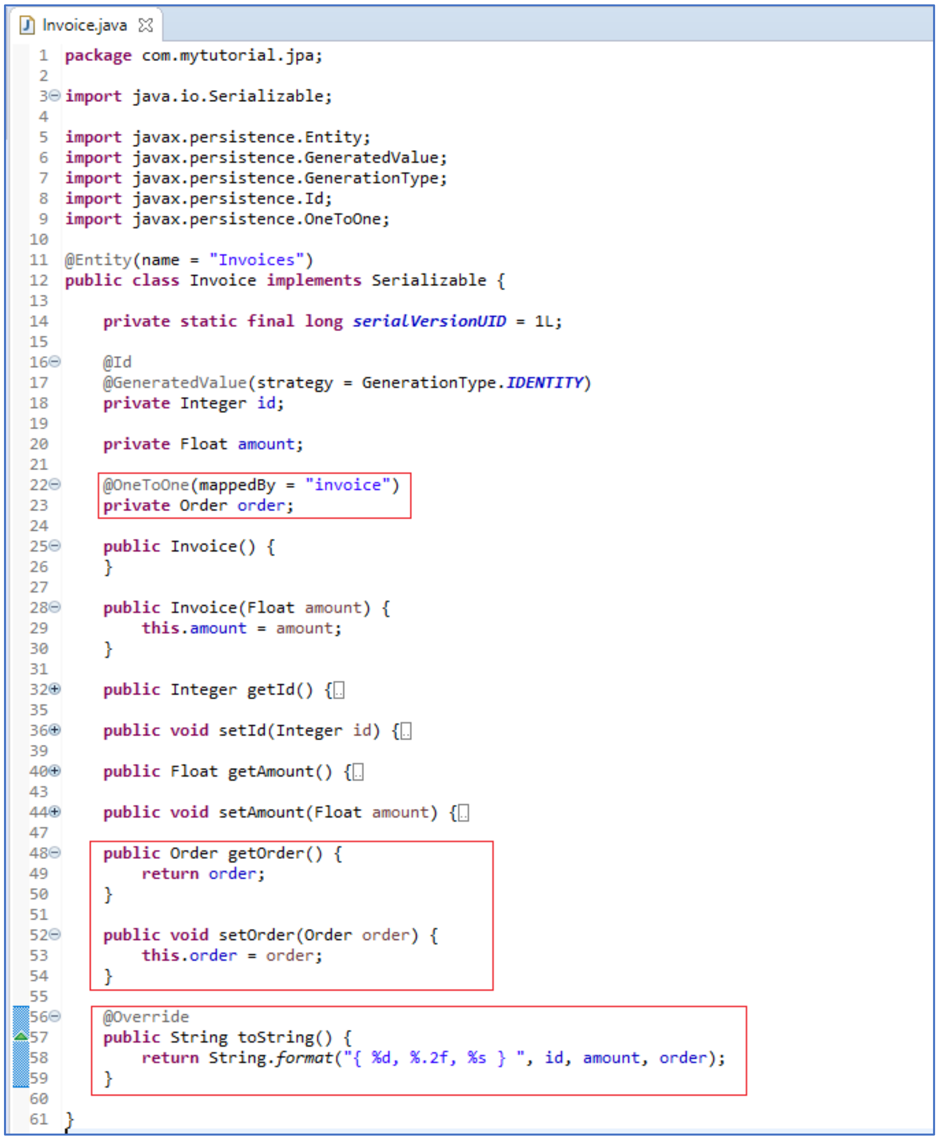
Then this application runs, I want the database tables in the online shopping DB to be recreated and repopulated. Now let's head over to **Order.java**.



```java
@Entity(name = "Orders")
public class Order {

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

    private String product;
    private Integer quantity;

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    @OneToOne
    @JoinColumn(name = "invoice_id")
    private Invoice invoice;

    public Order() {

    public Order(String product, Integer quantity, Date orderDate) {

    public Integer getId() {

    public void setId(Integer id) {

    public String getProduct() {

    public void setProduct(String product) {

    public Integer getQuantity() {

    public void setQuantity(Integer quantity) {

    public Date getOrderDate() {

    public void setOrderDate(Date orderDate) {

    public Invoice getInvoice() {

    public void setInvoice(Invoice invoice) {

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

- This is **the owning entity** in our one-to-one relationship.
- Order has a reference to the invoice entity, and we've used the **@OneToOne** annotation here.
- I'm going to tweak this **@JoinColumn** and only specify the name of the JoinColumn within this orders table. The name of the **@JoinColumn** is `"invoice_id"`.
  This is the foreign key in the orders table that references the primary key of the invoices table.
- I'll also scroll down and update the **toString()** representation of an order to only display the `id`, `product` and `quantity` and not reference the invoice.

In order to set up a bidirectional one-to-one mapping, I'm now going to head over to **Invoice.java** and have the invoice entity reference the order entity as well.



```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.OneToOne;

@Entity(name = "Invoices")
public class Invoice implements Serializable {

    private static final long serialVersionUID = 1L;

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

    private Float amount;

    @OneToOne(mappedBy = "invoice")
    private Order order;

    public Invoice() {
    }

    public Invoice(Float amount) {
        this.amount = amount;
    }

    public Integer getId() {

    public void setId(Integer id) {

    public Float getAmount() {

    public void setAmount(Float amount) {

    public Order getOrder() {
        return order;
    }

    public void setOrder(Order order) {
        this.order = order;
    }

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

}
```

- Go ahead and set up the import statement for the annotation OneToOne.
  `import javax.persistence.OneToOne;`
- The bidirectional OneToOne relationship is established by having this invoice entity reference the `order` entity. My *Invoice* class now contains a reference to the *Order* that this *Invoice* is associated with.

```
@OneToOne(mappedBy = "invoice")
private Order order;
```

You can see this on line 22 and 23. I've also added the **@OneToOne** annotation to the `order` reference. And within the **@OneToOne**, I've specified the **mappedBy** property.

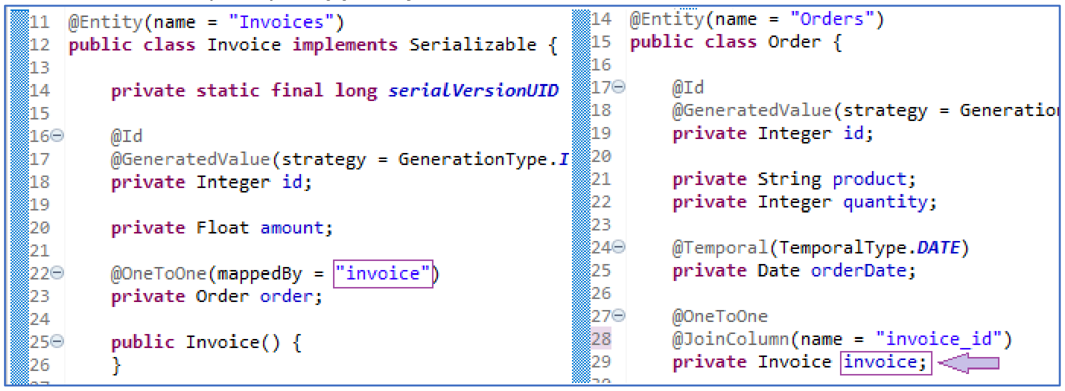
The presence of the **mappedBy** property indicates that **this entity is not the owning entity**. It is the inverse relationship in this bidirectional one is to one mapping.

In any one-to-one bidirectional relationship mapping, the owning side defines the mapping. And the referencing side simply references the mapping that has already been defined.

In our example here, the owning side is *Order.java*. That's the one which has the **@OneToOne** annotation and the **@JoinColumn**, the referencing side is the invoice side. And we know this is the inverse side because of the **mappedBy** property that we specify to the OneToOne mapping.

The value of the **mappedBy** property references the name of the variable in the *Order.java* file which references this invoice entity.

If you remember the name of the variable, you can see that in the inset here, the variable is `invoice`. So we specify **mappedBy** "`invoice`".



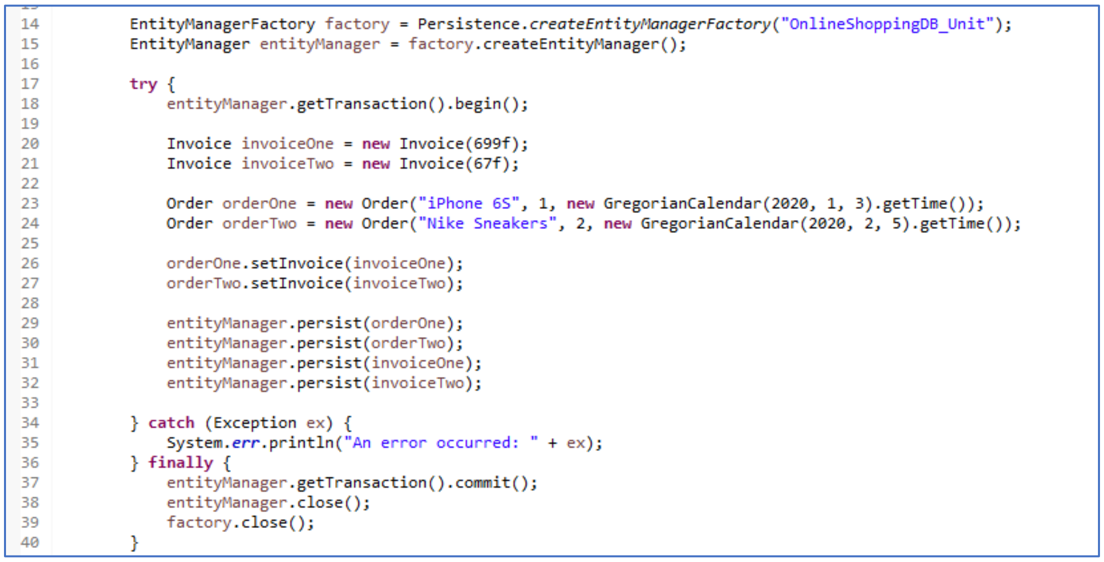
```
11 @Entity(name = "Invoices")
12 public class Invoice implements Serializable {
13
14     private static final long serialVersionUID
15
16     @Id
17     @GeneratedValue(strategy = GenerationType.I
18     private Integer id;
19
20     private Float amount;
21
22     @OneToOne(mappedBy = "invoice")
23     private Order order;
24
25     public Invoice() {
26     }
```

```
14 @Entity(name = "Orders")
15 public class Order {
16
17     @Id
18     @GeneratedValue(strategy = Generatio
19     private Integer id;
20
21     private String product;
22     private Integer quantity;
23
24     @Temporal(TemporalType.DATE)
25     private Date orderDate;
26
27     @OneToOne
28     @JoinColumn(name = "invoice_id")
29     private Invoice invoice;
```

This is how we set up the second leg of the bidirectional relationship.

- We'd gotten rid of the `invoiceId` column that used to exist.
- I'll also update the code in the **toString()** method here to no longer reference the `invoiceId`. Instead, for every *Invoice*, we'll also print out the details of the corresponding `order`.

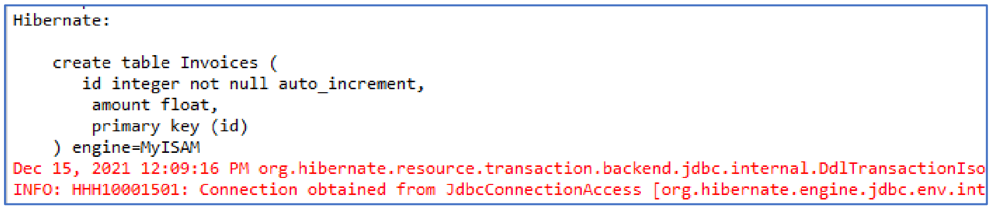
With this bidirectional mapping setup, we are now ready to run the code that exists in **App.java**.



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

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

            Invoice invoiceOne = new Invoice(699f);
            Invoice invoiceTwo = new Invoice(67f);

            Order orderOne = new Order("iPhone 6S", 1, new GregorianCalendar(2020, 1, 3).getTime());
            Order orderTwo = new Order("Nike Sneakers", 2, new GregorianCalendar(2020, 2, 5).getTime());

            orderOne.setInvoice(invoiceOne);
            orderTwo.setInvoice(invoiceTwo);

            entityManager.persist(orderOne);
            entityManager.persist(orderTwo);
            entityManager.persist(invoiceOne);
            entityManager.persist(invoiceTwo);

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

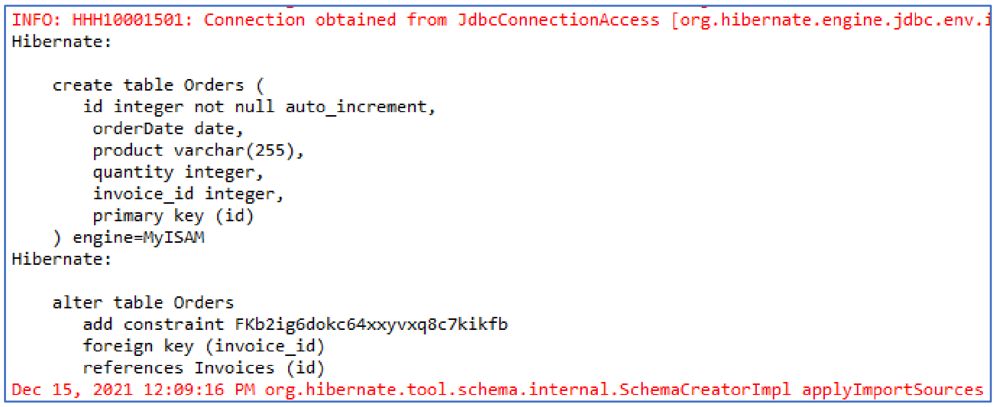
- We'll create two invoices for different amounts, `invoiceOne` and `invoiceTwo`.
- And we'll create two orders as well, one for the iPhone 6S and one for Nike sneakers.
- Observe that I have set the references for the invoices from the owning side of the relationship, `orderOne.setInvoice()`and `orderTwo.setInvoice()`.
  I haven't set up the references in the reverse direction. I've also persisted all four entities, two orders, two invoices.

Go ahead and run this code and let's take a look at how Hibernate sets up this bidirectional mapping.



```
Hibernate:

    create table Invoices (
       id integer not null auto_increment,
        amount float,
        primary key (id)
    ) engine=MyISAM
Dec 15, 2021 12:09:16 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIso
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.int
```

Here is the *Invoices* table. Notice that it has just two columns as it did earlier, the id that is the primary key and the amount which is a floating point.

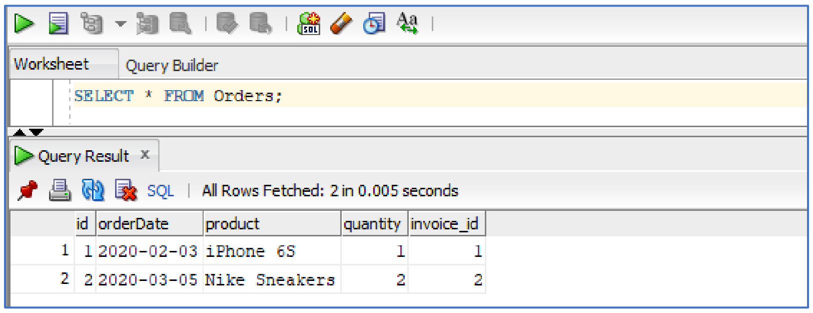


```
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.i
Hibernate:

    create table Orders (
       id integer not null auto_increment,
        orderDate date,
        product varchar(255),
        quantity integer,
        invoice_id integer,
        primary key (id)
    ) engine=MyISAM
Hibernate:

    alter table Orders
       add constraint FKb2ig6dokc64xxyvxq8c7kikfb
       foreign key (invoice_id)
       references Invoices (id)
Dec 15, 2021 12:09:16 PM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
```

Because the order entity is the owning entity, it's the *Orders* table that the **foreign key** reference to the `invoice_id`.

The invoice_id column is the foreign key, which points to the **primary key** in the *Invoices* table.

Even though this is a bidirectional relationship, you can see that the foreign key reference has been set up from the *Orders* table to the *Invoices* table.



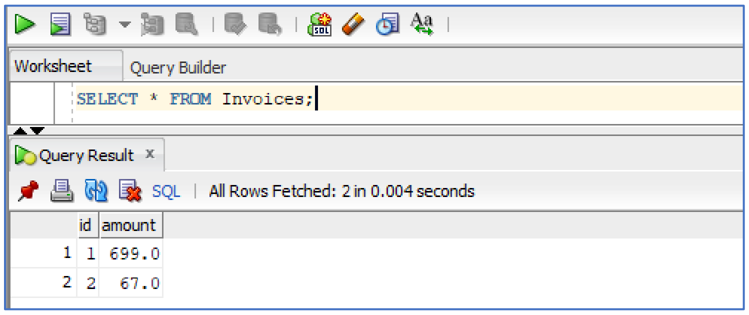
```
SELECT * FROM Orders;
```

All Rows Fetched: 2 in 0.005 seconds

	id	orderDate	product	quantity	invoice_id
1	1	2020-02-03	iPhone 6S	1	1
2	2	2020-03-05	Nike Sneakers	2	2

There is an invoice_id column in our *Orders* table here.

If you run a *select ** on the *Invoices* table here,

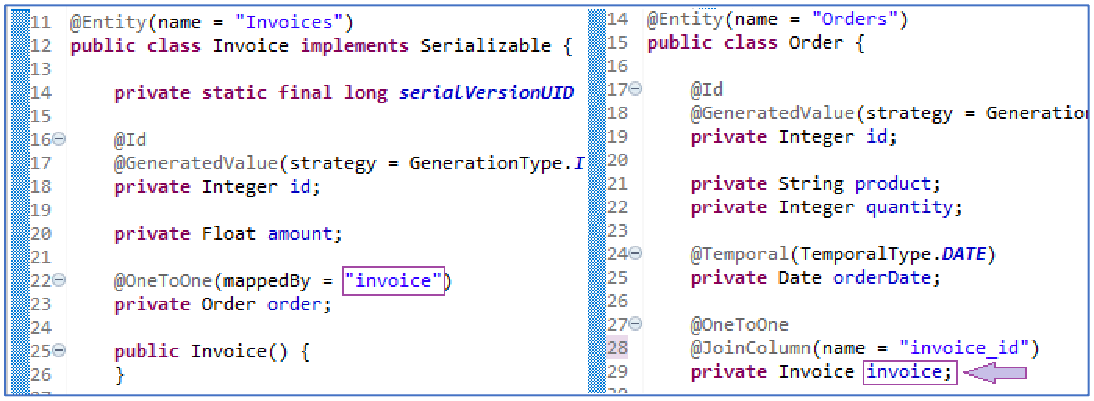


```
SELECT * FROM Invoices;
```

All Rows Fetched: 2 in 0.004 seconds

	id	amount
1	1	699.0
2	2	67.0

you will see that the columns remain the same. We have the id and the amount. There is no additional column that references the *Orders* table.

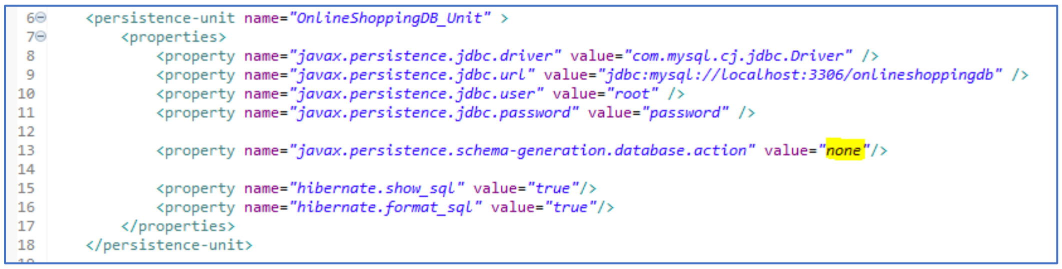


```
@Entity(name = "Invoices")
public class Invoice implements Serializable {

    private static final long serialVersionUID

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

    private Float amount;

    @OneToOne(mappedBy = "invoice")
    private Order order;

    public Invoice() {
    }
```

```
@Entity(name = "Orders")
public class Order {

    @Id
    @GeneratedValue(strategy = Generatio
    private Integer id;

    private String product;
    private Integer quantity;

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    @OneToOne
    @JoinColumn(name = "invoice_id")
    private Invoice invoice;
```

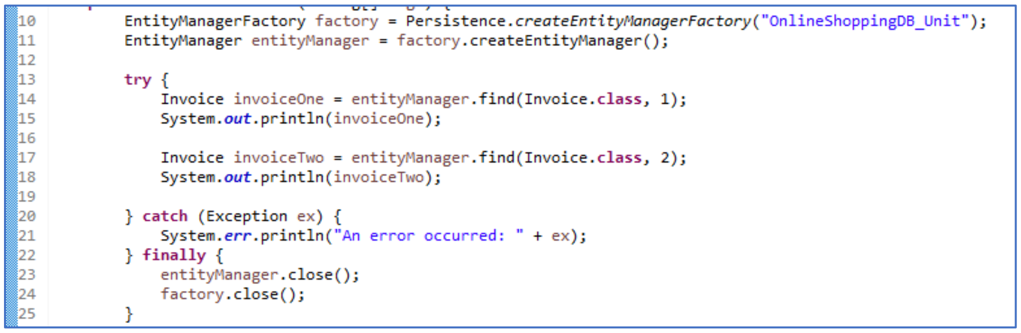
Setting up this bidirectional relationship though will allow us to retrieve the order corresponding to the invoice when we just retrieve the invoice information.

I'm going to set my database action to none so that we'll work with the database tables that we've already created.



```
<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="none"/>

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

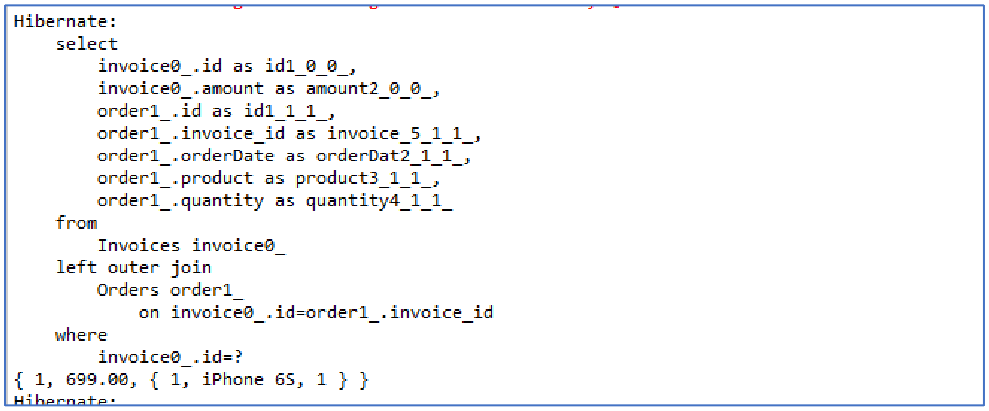
I'll now update the code that I'll run in **App.java**.



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

        try {
            Invoice invoiceOne = entityManager.find(Invoice.class, 1);
            System.out.println(invoiceOne);

            Invoice invoiceTwo = entityManager.find(Invoice.class, 2);
            System.out.println(invoiceTwo);

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

Within the try block, I'm going to use the find method to retrieve invoice information. I'm going to retrieve the invoice with primary key 1 and primary key 2.

Since the invoice contains a reference to the corresponding order and we have set up a bidirectional mapping, retrieving the invoice should also retrieve order details.

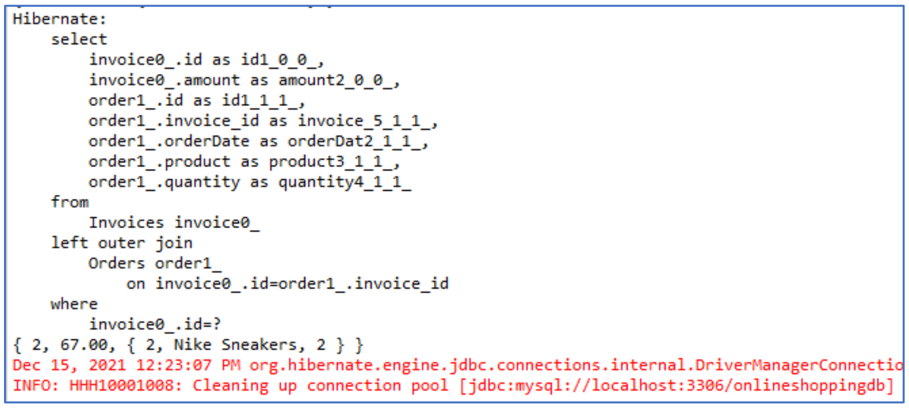


```
Hibernate:
    select
        invoice0_.id as id1_0_0_,
        invoice0_.amount as amount2_0_0_,
        order1_.id as id1_1_1_,
        order1_.invoice_id as invoice_5_1_1_,
        order1_.orderDate as orderDat2_1_1_,
        order1_.product as product3_1_1_,
        order1_.quantity as quantity4_1_1_
    from
        Invoices invoice0_
    left outer join
        Orders order1_
            on invoice0_.id=order1_.invoice_id
    where
        invoice0_.id=?
{ 1, 699.00, { 1, iPhone 6S, 1 } }
Hibernate:
```

Take a look at the select statement that Hibernate runs under the hood. You can see that we perform select on invoice fields as well as the corresponding order fields. And if we scroll down a little bit, you'll see that Hibernate has performed a **join operation** where we join the `invoice_id` with the `invoice_id` field in the *Orders* table.

At the bottom, we have the details for invoice 1 printed out, $699. And the order that it's associated with is the "iPhone 6S", so retrieving the invoice give us the order as well.

If you scroll down, you'll see the second select statement in our Console window. This corresponds to the find on line 17.



```
Hibernate:
    select
        invoice0_.id as id1_0_0_,
        invoice0_.amount as amount2_0_0_,
        order1_.id as id1_1_1_,
        order1_.invoice_id as invoice_5_1_1_,
        order1_.orderDate as orderDat2_1_1_,
        order1_.product as product3_1_1_,
        order1_.quantity as quantity4_1_1_
    from
        Invoices invoice0_
    left outer join
        Orders order1_
            on invoice0_.id=order1_.invoice_id
    where
        invoice0_.id=?
{ 2, 67.00, { 2, Nike Sneakers, 2 } }
Dec 15, 2021 12:23:07 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectio
INFO: HHH10001008: Cleaning up connection pool [jdbc:mysql://localhost:3306/onlineshoppingdb]
```

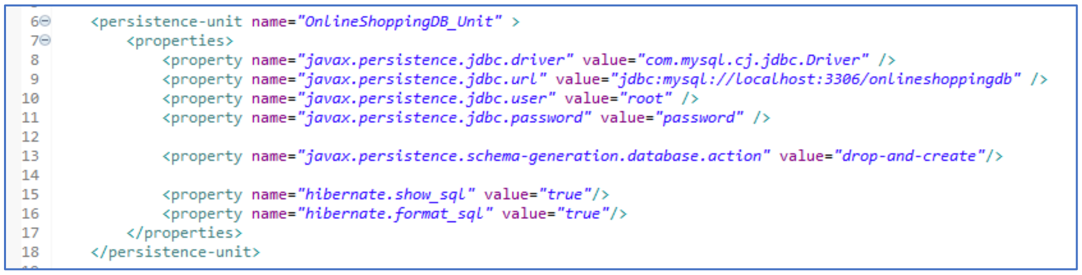
In addition to the invoice details, we also retrieve the order details by performing a join operation.

The *Orders* and *Invoices* tables are joined **using the foreign key reference** from *Orders* to *Invoices*. The invoice with `id` 2 is for $67 and for the product “Nike sneakers”. That information is present in the order.

One-to-one Mapping with Shared Primary Keys

Now if you have a one to one mapping between order and invoice. There's really no reason to have a separate primary key for an order and an invoice. They can share a primary key, and that's exactly what we'll configure here in this demo.

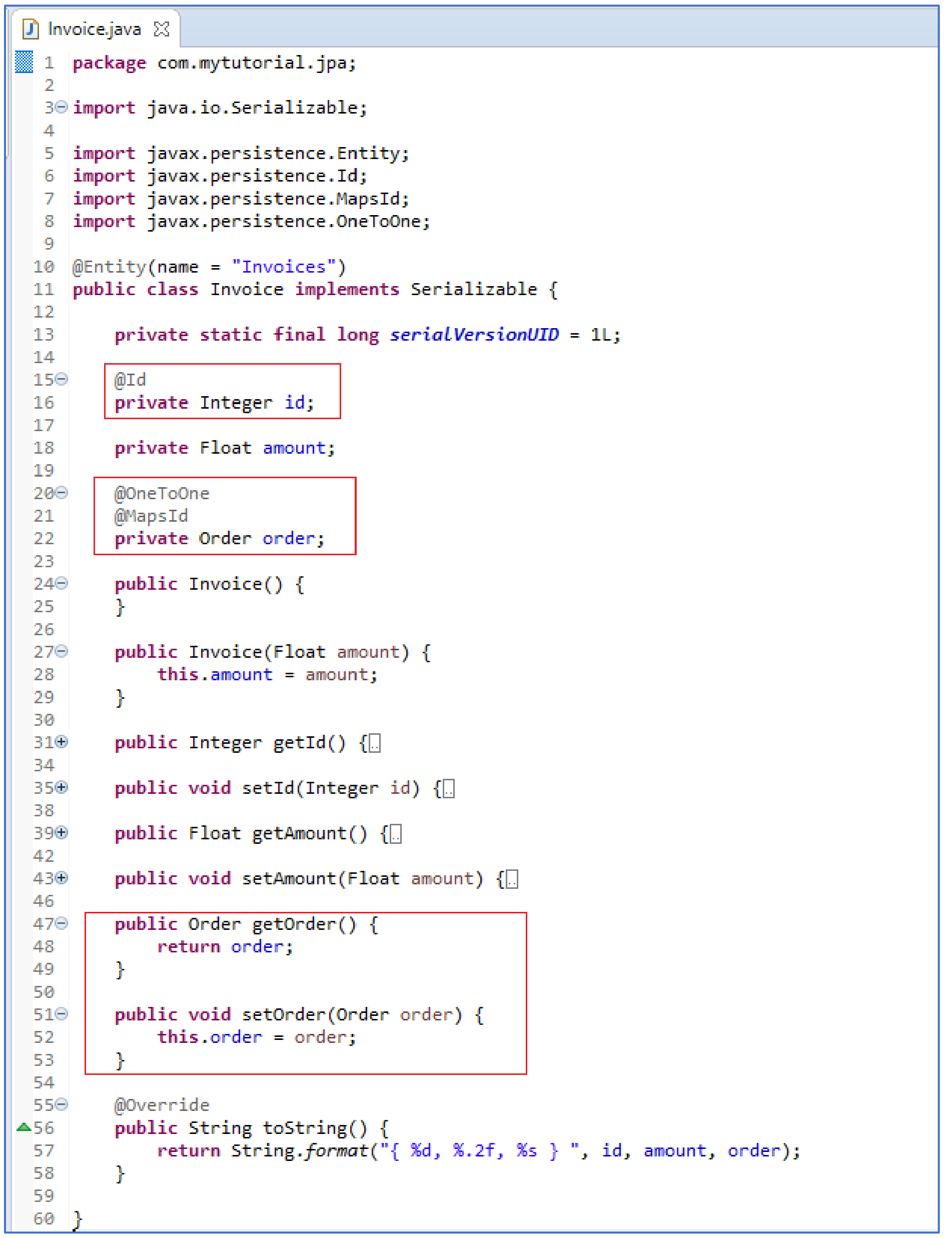
Change **persistence.xml** back to drop-and-create for the database action.



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

When we run our JPA hibernate app, I want all of the database tables to be recreated and repopulated.

Head over to **Invoice.java**.



```java
package com.mytutorial.jpa;

import java.io.Serializable;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.MapsId;
import javax.persistence.OneToOne;

@Entity(name = "Invoices")
public class Invoice implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    private Integer id;

    private Float amount;

    @OneToOne
    @MapsId
    private Order order;

    public Invoice() {
    }

    public Invoice(Float amount) {
        this.amount = amount;
    }

    public Integer getId() {

    public void setId(Integer id) {

    public Float getAmount() {

    public void setAmount(Float amount) {

    public Order getOrder() {
        return order;
    }

    public void setOrder(Order order) {
        this.order = order;
    }

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

}
```

- And let's get rid of this **@GeneratedValue** annotation that we have on the primary key `id`. We want the `id` variable to continue to hold the primary key for an `Invoice`. But we want this **primary key to be shared** with the *Orders* table. That is the order ID will be the primary key for an invoice.
- So we'll have exactly one `Order` ID which will double up as the primary key for an `Invoice`. And to configure this, we'll need to change the way we specify this **@OneToOne** annotation. Instead of using the *mappedBy* attribute, we'll specify an additional **@MapsId** attribute.
- Now this import statement has to be included, as follows

```java
import javax.persistence.MapsId;
import javax.persistence.OneToOne;
```

```
@OneToOne
@MapsId
private Order order;
```

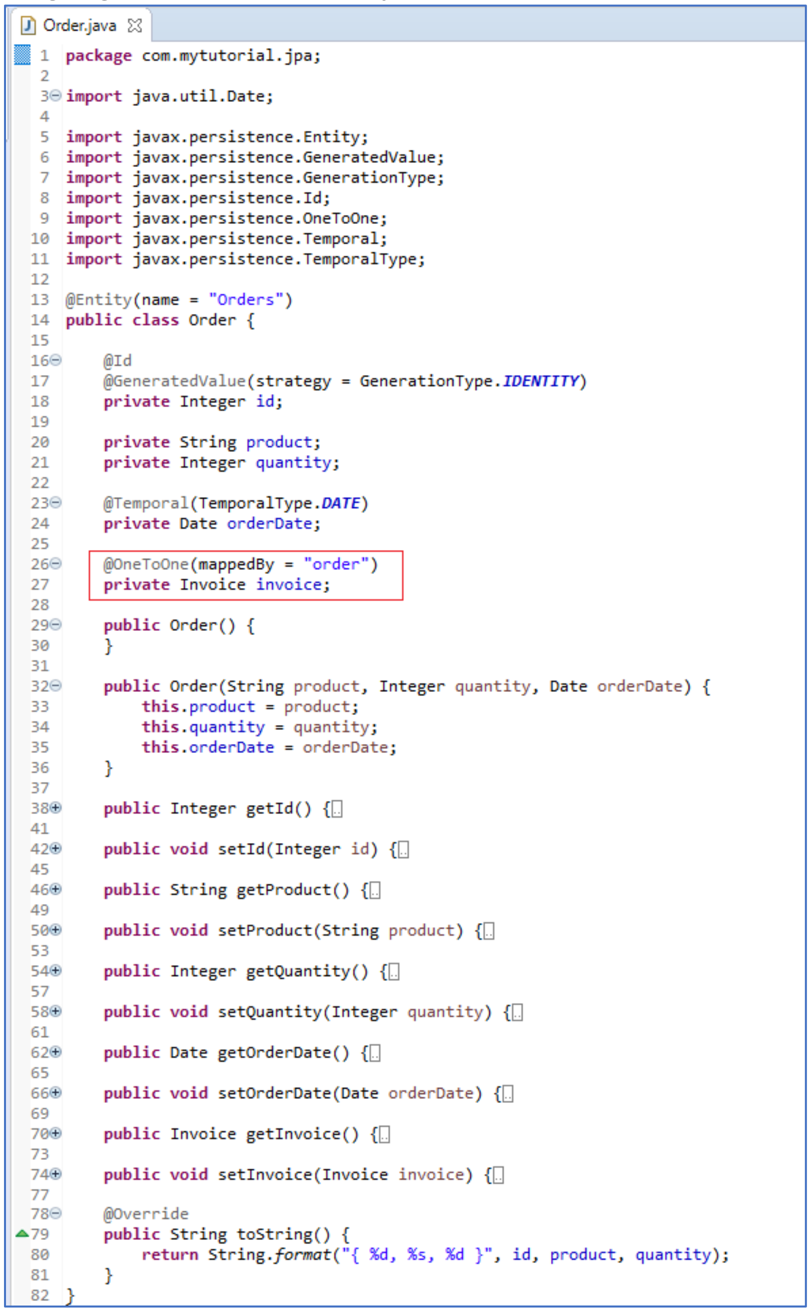
Let's understand the annotations that we have here.

- We have the **@OneToOne** annotation, indicating a one-to-one relationship between the `Invoice` and the `Order` entity.
- And we have the **@MapsId**. Indicating that the **primary key** of the `Order` entity in the *Orders* table **will be shared** by this `Invoice` entity in the *Invoices* table.

This **@MapsId** in the `Invoice` class makes the `Invoice` entity *the owning entity*. Notice the `import` for `javax.persistence.MapsId` that allows us to create a **shared primary key**.

- I'm going to go ahead and add a **getter** and **setter** for the `order` reference here within this *invoice.java* class, **getOrder()** and **setOrder()**.

In order to complete mapping this relationship, where orders and invoices will use a shared primary key, I'm going to head over to **order.java**.



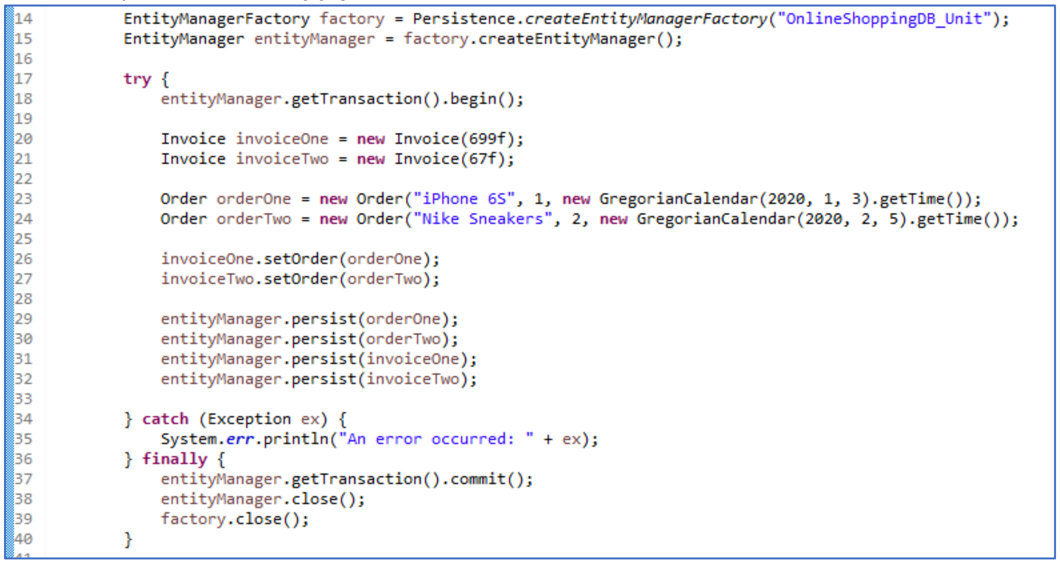
Order.java

```java
package com.mytutorial.jpa;

import java.util.Date;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity(name = "Orders")
public class Order {

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

    private String product;
    private Integer quantity;

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    @OneToOne(mappedBy = "order")
    private Invoice invoice;

    public Order() {
    }

    public Order(String product, Integer quantity, Date orderDate) {
        this.product = product;
        this.quantity = quantity;
        this.orderDate = orderDate;
    }

    public Integer getId() {

    public void setId(Integer id) {

    public String getProduct() {

    public void setProduct(String product) {

    public Integer getQuantity() {

    public void setQuantity(Integer quantity) {

    public Date getOrderDate() {

    public void setOrderDate(Date orderDate) {

    public Invoice getInvoice() {

    public void setInvoice(Invoice invoice) {

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

- I'm going to get rid of the existing annotations that exist here, the OneToOne and the JoinColumn annotation. And I'm going to add a single **@OneToOne** annotation with a value for the **mappedBy** property.
  The existence of this **mappedBy** property in this **@OnetoOne** annotation makes the `order` entity part of the inverse mapping.
  `Order` is no longer the owning entity, `Invoice` *is now the owning entity*.
  The value of this **mappedBy** property is set to `"order"`, which is the name of the member variable within the **invoice.java** class.

The member variable that references the `order` associated with every `invoice`. Having set up this OneToOne mapping to **share a primary key**, that is the `Order` ID will be the primary key for our *Invoices* table.

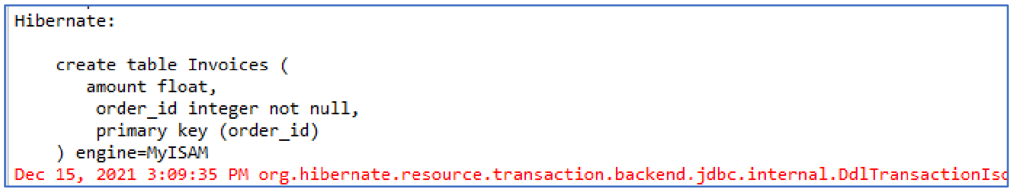
Let's set up our code in **App.java**.



```
14         EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
15         EntityManager entityManager = factory.createEntityManager();
16 
17         try {
18             entityManager.getTransaction().begin();
19 
20             Invoice invoiceOne = new Invoice(699f);
21             Invoice invoiceTwo = new Invoice(67f);
22 
23             Order orderOne = new Order("iPhone 6S", 1, new GregorianCalendar(2020, 1, 3).getTime());
24             Order orderTwo = new Order("Nike Sneakers", 2, new GregorianCalendar(2020, 2, 5).getTime());
25 
26             invoiceOne.setOrder(orderOne);
27             invoiceTwo.setOrder(orderTwo);
28 
29             entityManager.persist(orderOne);
30             entityManager.persist(orderTwo);
31             entityManager.persist(invoiceOne);
32             entityManager.persist(invoiceTwo);
33 
34         } catch (Exception ex) {
35             System.err.println("An error occurred: " + ex);
36         } finally {
37             entityManager.getTransaction().commit();
38             entityManager.close();
39             factory.close();
40         }
```

- We create two invoices, invoice one and two, and two orders, orders one and two.
- An important thing to notice here in this code is that on line 26 and 27, we call `invoiceOne.setOrder(orderOne)` and `invoiceOne.setOrder(orderTwo)`.
  We set up the references from the owning entity, that is the `Invoice`, to the owned entity which is the `Order`.
- While persisting entities, the references always have to be from the owning entity to the owned entity. Go ahead and call persist on all four entities and commit the transaction.

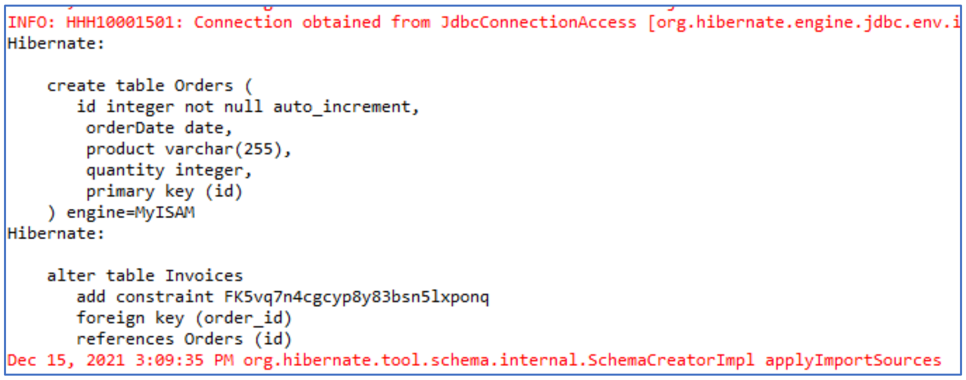
Once this is done, let's run this code and take a look at the tables created.



```
Hibernate: 
    
    create table Invoices (
       amount float,
       order_id integer not null,
        primary key (order_id)
    ) engine=MyISAM
Dec 15, 2021 3:09:35 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIs
```

The *Invoices* table here has two columns, the `amount`, and the `order_id`. This is the shared primary key. The primary key of the orders table is the primary key of the invoices table as well.

Let's scroll down and here is the *Orders* table,



```
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.i
Hibernate: 

    create table Orders (
       id integer not null auto_increment,
        orderDate date,
        product varchar(255),
        quantity integer,
        primary key (id)
    ) engine=MyISAM
Hibernate: 

    alter table Invoices 
       add constraint FK5vq7n4cgcyp8y83bsn5lxponq 
       foreign key (order_id) 
       references Orders (id)
Dec 15, 2021 3:09:35 PM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
```

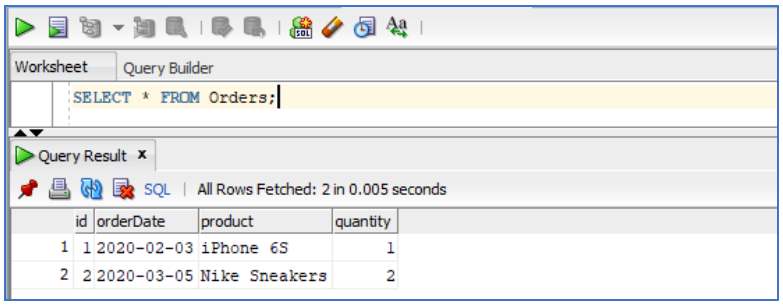
no change here, `id` is the **primary key**. And this `id` is also shared by the *Invoices* table.

If you scroll down further, you'll see the foreign key constraint. The *Invoices* table, which is the owning entity, has a **foreign key reference** to the *Orders* table.

The `order_id`, which is the primary key of the *Invoices* table, references the `id` in the *Orders* table, also the *Orders* table primary key.

Using **@MapsId**, we've configured the two tables to **share a primary key**.

Let's verify this using the mySQL workbench. I'm going to run a *select ** on the *Orders* table.



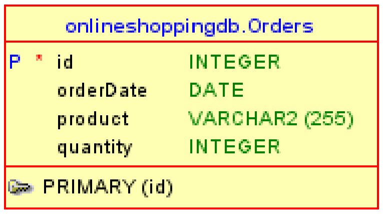
Worksheet Query Builder

```
SELECT * FROM Orders;
```

Query Result

All Rows Fetched: 2 in 0.005 seconds

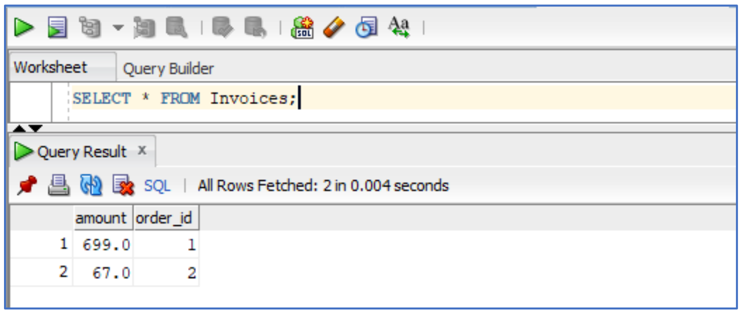
```
   id orderDate  product        quantity
1  1  2020-02-03 iPhone 6S             1
2  2  2020-03-05 Nike Sneakers         2
```



```
onlineshoppingdb.Orders
P * id          INTEGER
    orderDate   DATE
    product     VARCHAR2 (255)
    quantity    INTEGER
PRIMARY (id)
```

Observe that we have an id column with values 1 and 2.

I'll now run a *select ** on the *Invoices* table,



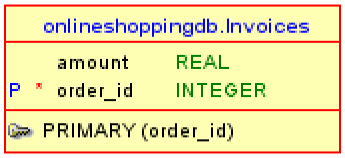
Worksheet Query Builder

```
SELECT * FROM Invoices;
```

Query Result

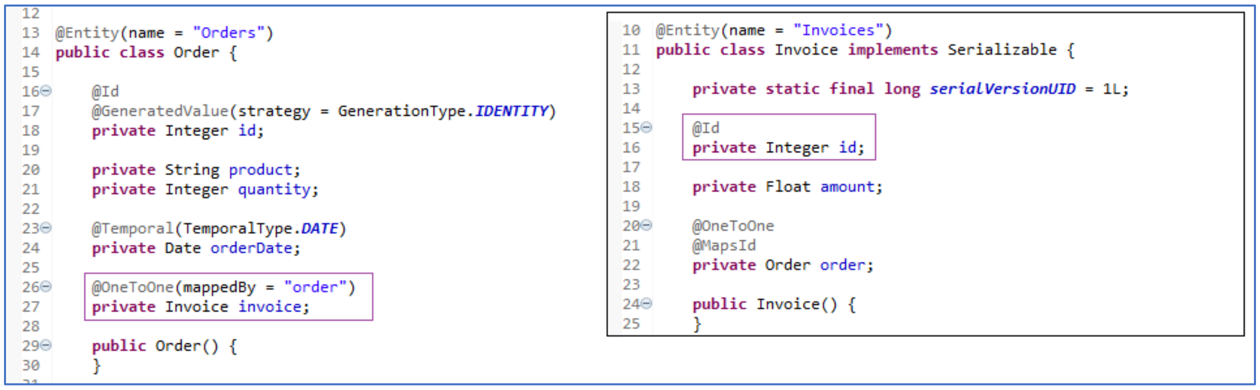
All Rows Fetched: 2 in 0.004 seconds

```
   amount order_id
1  699.0         1
2   67.0         2
```



```
onlineshoppingdb.Invoices
    amount      REAL
P * order_id    INTEGER
PRIMARY (order_id)
```

and you'll see that we have two entries in the *Invoices* table corresponding to two *Invoices*. And the primary key for these *Invoices* is `order_id`, the **shared primary key**.



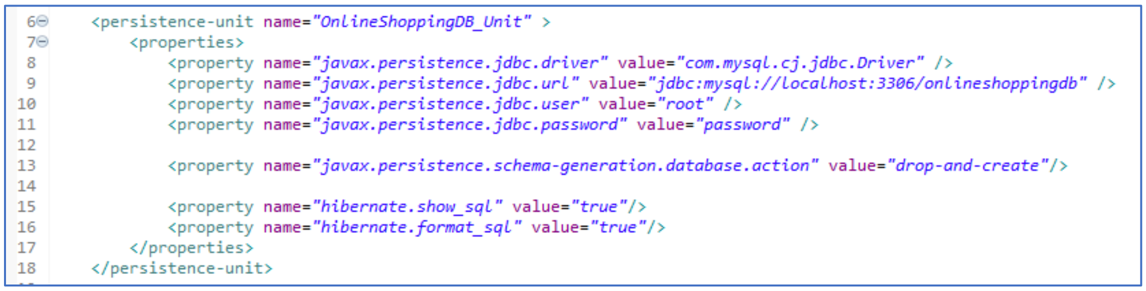
```java
@Entity(name = "Orders")
public class Order {

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

    private String product;
    private Integer quantity;

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    @OneToOne(mappedBy = "order")
    private Invoice invoice;

    public Order() {
    }
```

```java
@Entity(name = "Invoices")
public class Invoice implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    private Integer id;

    private Float amount;

    @OneToOne
    @MapsId
    private Order order;

    public Invoice() {
    }
```

One-to-one Mapping with Join Tables

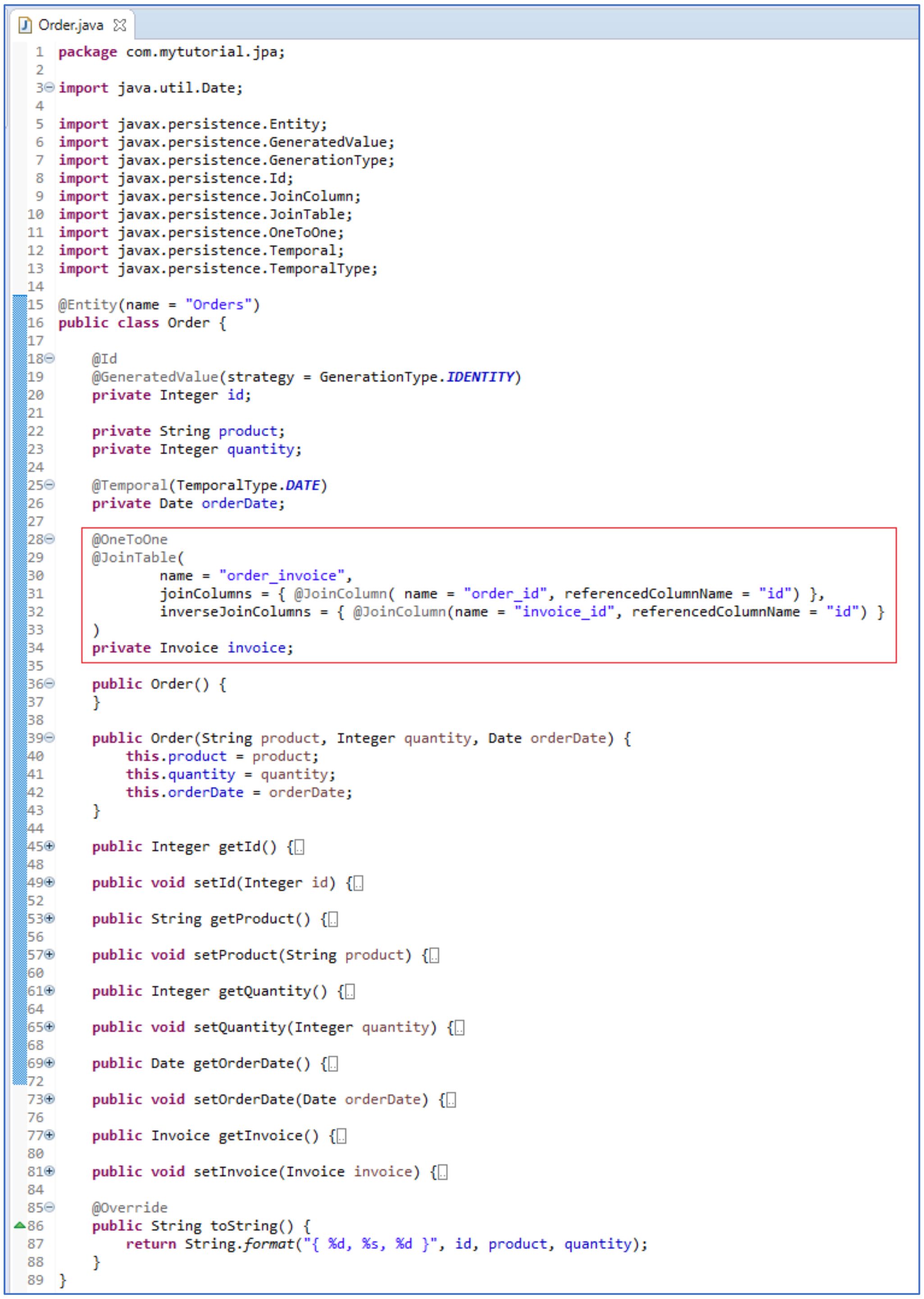
So far, when we've set up a one is to one relationship mapping between two entities, Hibernate has modeled this mapping using foreign key constraints. So we set up a foreign key column in the Owning Entity which references either the primary key or some other column in the Non Owning Entity. This time around, we'll do something a little different. We'll set up a one is to one mapping using a separate table, that is the join table.

For this demo, I've reset the **persistence.xml** file so that the database.action property is once again set to drop-and-create. We'll recreate the tables each time we run this application.



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

Now let's switch over to **Order.java**.



Order.java

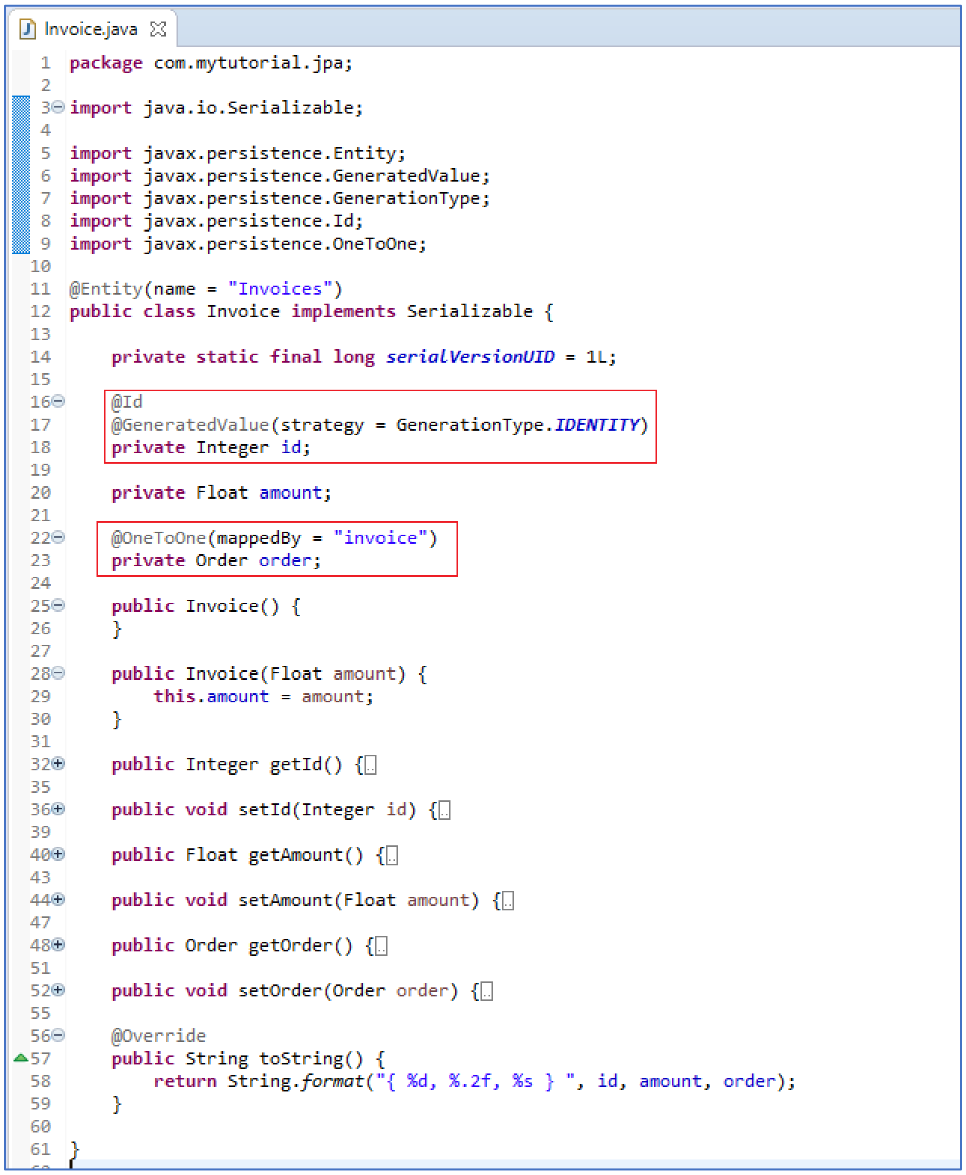
```java
package com.mytutorial.jpa;

import java.util.Date;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.OneToOne;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity(name = "Orders")
public class Order {

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

    private String product;
    private Integer quantity;

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    @OneToOne
    @JoinTable(
            name = "order_invoice",
            joinColumns = { @JoinColumn( name = "order_id", referencedColumnName = "id") },
            inverseJoinColumns = { @JoinColumn(name = "invoice_id", referencedColumnName = "id") }
    )
    private Invoice invoice;

    public Order() {
    }

    public Order(String product, Integer quantity, Date orderDate) {
        this.product = product;
        this.quantity = quantity;
        this.orderDate = orderDate;
    }

    public Integer getId() {

    public void setId(Integer id) {

    public String getProduct() {

    public void setProduct(String product) {

    public Integer getQuantity() {

    public void setQuantity(Integer quantity) {

    public Date getOrderDate() {

    public void setOrderDate(Date orderDate) {

    public Invoice getInvoice() {

    public void setInvoice(Invoice invoice) {

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

- This Order entity is once again going to be my owning entity.
- I've set up the `import` statements for `javax.persistence.JoinTable` and `javax.persistence.JoinColumn`.
- This **@JoinTable** is a separate table that will hold the mapping of entities from `Order` to `Invoice` and vice versa.

```
@OneToOne
@JoinTable(
        name = "order_invoice",
        joinColumns = {
                @JoinColumn( name = "order_id", referencedColumnName = "id") },
        inverseJoinColumns = {
                @JoinColumn(name = "invoice_id", referencedColumnName = "id") })
private Invoice invoice;
```

- I'm going to update my **@OneToOne** annotation on the `invoice` member variable here. This is where the `Order` entity references the `Invoice` that it's associated with.
- Once again, I have **@OneToOne** mapping, the mapping relationship remains the same, but I have an additional annotation for join table **@JoinTable**.
- And I use the **name** property to specify the table that will hold the mapping relationship. I've called this table `"order_invoice"` because it maps from an `Order` ID to an `Invoice` ID. I've also specified what columns this table should map on. That is in the **joinColumns** attribute.
- The **joinColumns** property allows us to specify what column in this table that is the *Orders* table is referenced in the **@JoinTable** `order_invoice`. So the reference column here is the `id` column, which is the primary key and the name of the join column in the `order_invoice` table is `invoice_id`. That's what the first **joinColumns** property configures.
- The **@JoinTable** annotation takes in an additional property that is the **inverseJoinColumns**. This property specifies what column in the inverse table which is the `Invoice` table is mapped as a foreign key in the join table `order_invoice`. You can see that the column `id` in the `Invoice` table is mapped with the name `invoice_id`, in this `order_invoice` join table.

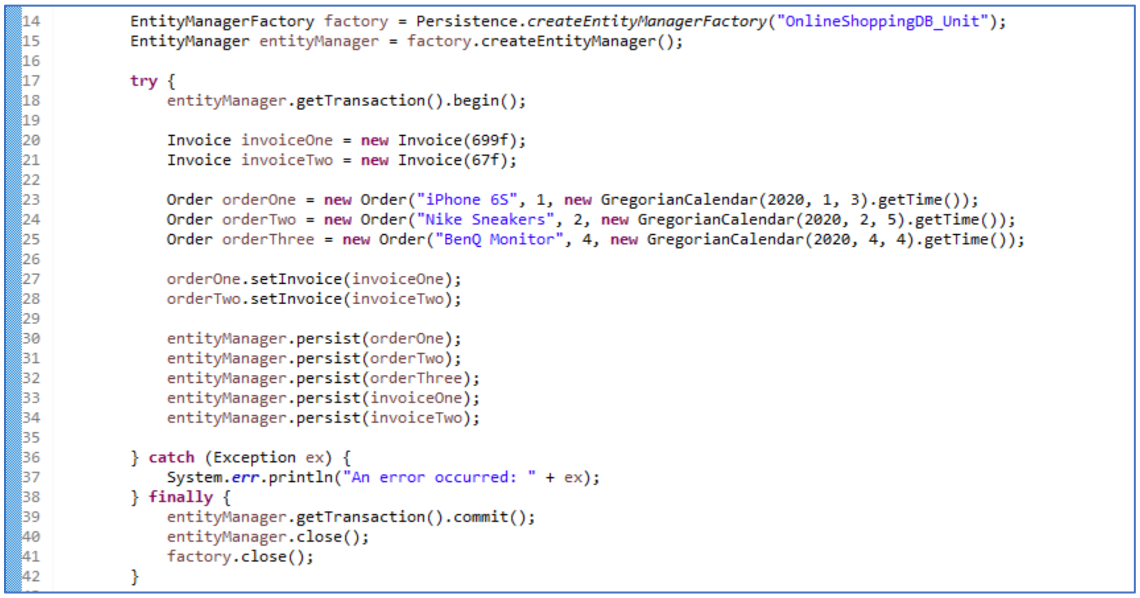
In order to complete the configuration of one is to one mapping between order and invoice using the join table, let's switch over to **Invoice.java**.



```
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.OneToOne;

@Entity(name = "Invoices")
public class Invoice implements Serializable {

    private static final long serialVersionUID = 1L;

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

    private Float amount;

    @OneToOne(mappedBy = "invoice")
    private Order order;

    public Invoice() {
    }

    public Invoice(Float amount) {
        this.amount = amount;
    }

    public Integer getId() {

    public void setId(Integer id) {

    public Float getAmount() {

    public void setAmount(Float amount) {

    public Order getOrder() {

    public void setOrder(Order order) {

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

}
```

- I'm going to add in the **import** for `javax.persistence.GeneratedValue` and `javax.persistence.GenerationType` and remove the import for MapsId. MapsId was for the shared primary key, which we're not using anymore.
- I've added in the **@GeneratedValue** annotation on the primary key for this `Invoice` table so that the primary key is automatically generated.
- And I'm going to update the `order` reference from the `Invoice` to have the **@OneToOne** mapping annotation, and the **mappedBy** property. This **mappedBy** property will tell Hibernate that invoice references order in the inverse relationship.
  The value in the **mappedBy** property tells us what member variable in the `order` entity references `"invoice"`, the `Invoice` member variable.

Now let's head over to **App.java** and see how we can persist entities for this one is to one relationship expressed using a join table.

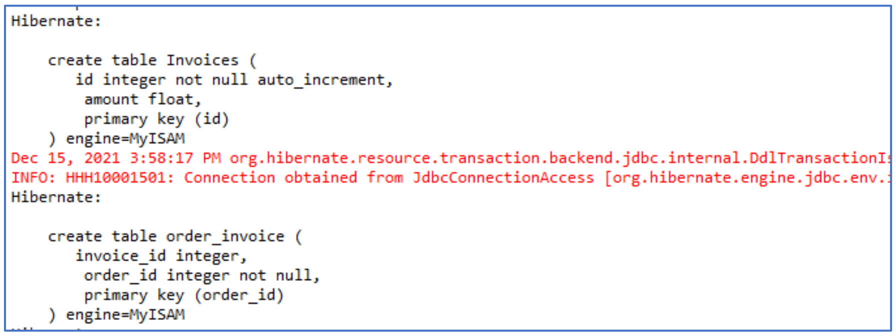


```java
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

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

            Invoice invoiceOne = new Invoice(699f);
            Invoice invoiceTwo = new Invoice(67f);

            Order orderOne = new Order("iPhone 6S", 1, new GregorianCalendar(2020, 1, 3).getTime());
            Order orderTwo = new Order("Nike Sneakers", 2, new GregorianCalendar(2020, 2, 5).getTime());
            Order orderThree = new Order("BenQ Monitor", 4, new GregorianCalendar(2020, 4, 4).getTime());

            orderOne.setInvoice(invoiceOne);
            orderTwo.setInvoice(invoiceTwo);

            entityManager.persist(orderOne);
            entityManager.persist(orderTwo);
            entityManager.persist(orderThree);
            entityManager.persist(invoiceOne);
            entityManager.persist(invoiceTwo);

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

- Now I have two `Invoice`, `invoiceOne` and `invoiceTwo`, and I have three orders. `orderOne`, `orderTwo` and `orderThree` for the “iPhone 6S”, “Nike sneakers” and the “BenQ monitor”.
- Now two of these orders have corresponding invoices. So `orderOne` corresponds to `invoiceOne`, `orderTwo` corresponds to `invoiceTwo`, but `orderThree` has no associated invoice.
- Next, I persist all 5 entities that I have created, three orders, two invoices.

Note that in this one is to one relationship, it's not necessary that every order references an invoice. We have `orderThree` with no associated invoice.

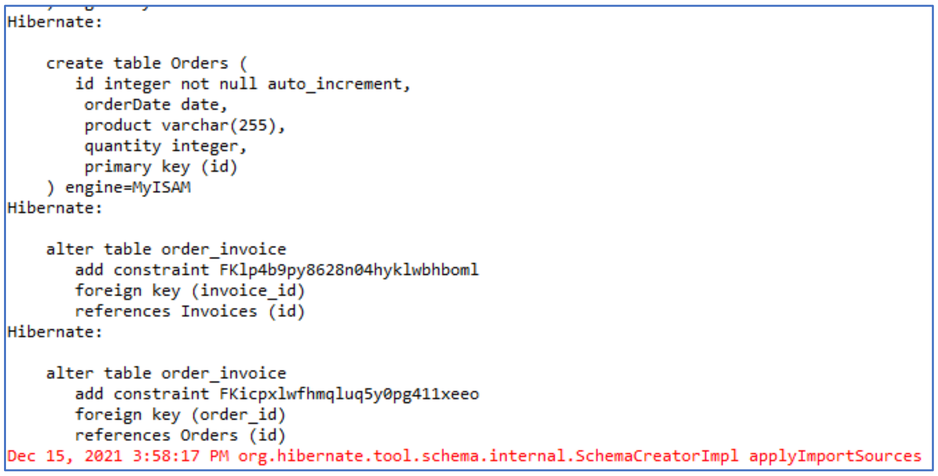
Let's go ahead and run this code and take a look at the tables that were created.



```
Hibernate:

    create table Invoices (
       id integer not null auto_increment,
        amount float,
        primary key (id)
    ) engine=MyISAM
Dec 15, 2021 3:58:17 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionI
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.
Hibernate:

    create table order_invoice (
       invoice_id integer,
        order_id integer not null,
        primary key (order_id)
    ) engine=MyISAM
```

- Notice that we have the *Invoices* table with `id` and `amount`, the primary key is the `id`
- and we have this join table called *`order_invoice`*.
  This join table that holds the mapping relationship has two columns, `invoice_id` and `order_id`.
  The primary key is the `order_id`.
- This will hold the mapping from `order_id` to `invoice_id`.

If you scroll down below, you'll see that the *Orders* table has been created as well where `id` is the primary key.

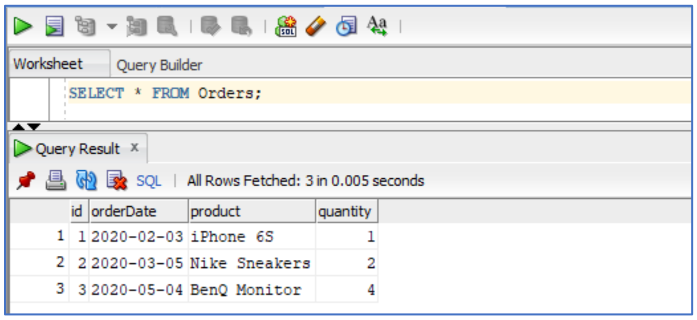


```
Hibernate:

    create table Orders (
       id integer not null auto_increment,
        orderDate date,
        product varchar(255),
        quantity integer,
        primary key (id)
    ) engine=MyISAM
Hibernate:

    alter table order_invoice
       add constraint FKlp4b9py8628n04hyklwbhboml
       foreign key (invoice_id)
       references Invoices (id)
Hibernate:

    alter table order_invoice
       add constraint FKicpxlwfhmqluq5y0pg411xeeo
       foreign key (order_id)
       references Orders (id)
Dec 15, 2021 3:58:17 PM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
```

And the remaining details of an order make up the other columns.

- Notice that Hibernate has added a foreign key constraint where the `invoice_id` column in the order invoice table references the ID column in the invoices table.
- And the order ID column in the `order_invoice` join table references the `id` column in the order table.

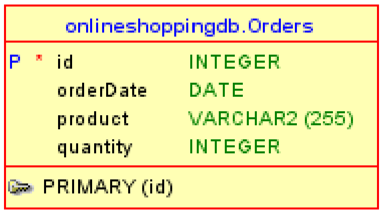
Let's switch over to the MySQL Workbench, and we'll see how the entities that we've persisted in our tables have been represented using the join table. I'll do a *select ** from the *Orders* table.



```
SELECT * FROM Orders;
```

All Rows Fetched: 3 in 0.005 seconds

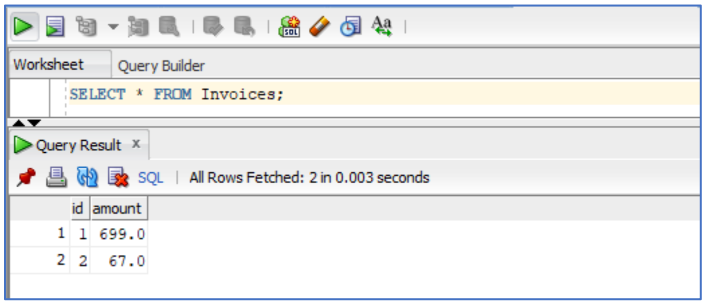
	id	orderDate	product	quantity
1	1	2020-02-03	iPhone 6S	1
2	2	2020-03-05	Nike Sneakers	2
3	3	2020-05-04	BenQ Monitor	4



```
onlineshoppingdb.Orders
P * id          INTEGER
    orderDate   DATE
    product     VARCHAR2 (255)
    quantity    INTEGER
PRIMARY (id)
```

This will give us three order entries with id 1, 2, and 3. If you remember one of these orders, the order with id 3, one for the BenQ monitor does not have a corresponding invoice.

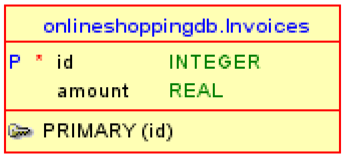
Let's do a *select ** from the *Invoices* table.



```
SELECT * FROM Invoices;
```

All Rows Fetched: 2 in 0.003 seconds

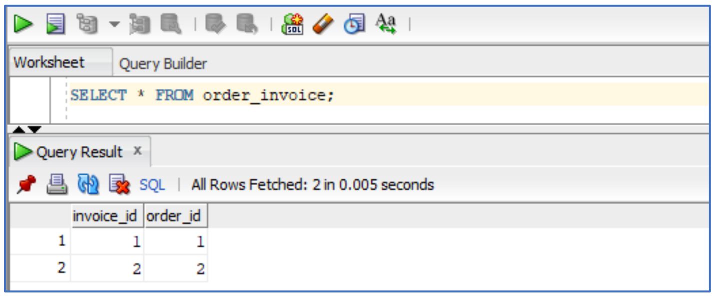
	id	amount
1	1	699.0
2	2	67.0



```
onlineshoppingdb.Invoices
P * id       INTEGER
    amount   REAL
PRIMARY (id)
```

Notice that we have entries for two invoices with id 1 and 2. These correspond to the “iPhone 6S” and “Nike sneakers”.

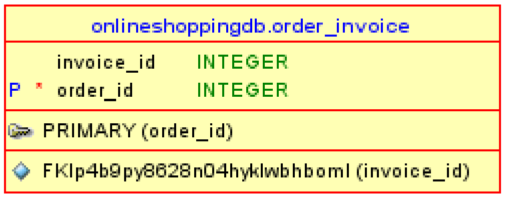
We'll now look at the mapping in the *order_invoice* table. Run a *select ** in this table,



```
SELECT * FROM order_invoice;
```

All Rows Fetched: 2 in 0.005 seconds

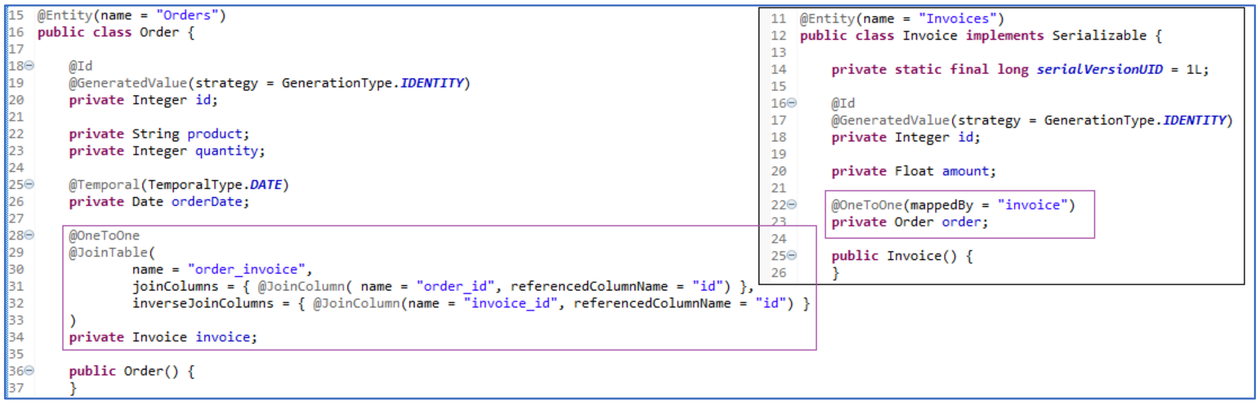
	invoice_id	order_id
1	1	1
2	2	2



```
onlineshoppingdb.order_invoice
    invoice_id   INTEGER
P * order_id     INTEGER
PRIMARY (order_id)
FKlp4b9py8628n04hyklwbhboml (invoice_id)
```

and you'll see there are two entries mapping the order id to the corresponding invoice id.

The order with id 3 has no entry in this table because it's not associated with an invoice.



```java
@Entity(name = "Orders")
public class Order {

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

    private String product;
    private Integer quantity;

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    @OneToOne
    @JoinTable(
            name = "order_invoice",
            joinColumns = { @JoinColumn( name = "order_id", referencedColumnName = "id") },
            inverseJoinColumns = { @JoinColumn(name = "invoice_id", referencedColumnName = "id") }
    )
    private Invoice invoice;

    public Order() {
    }
```

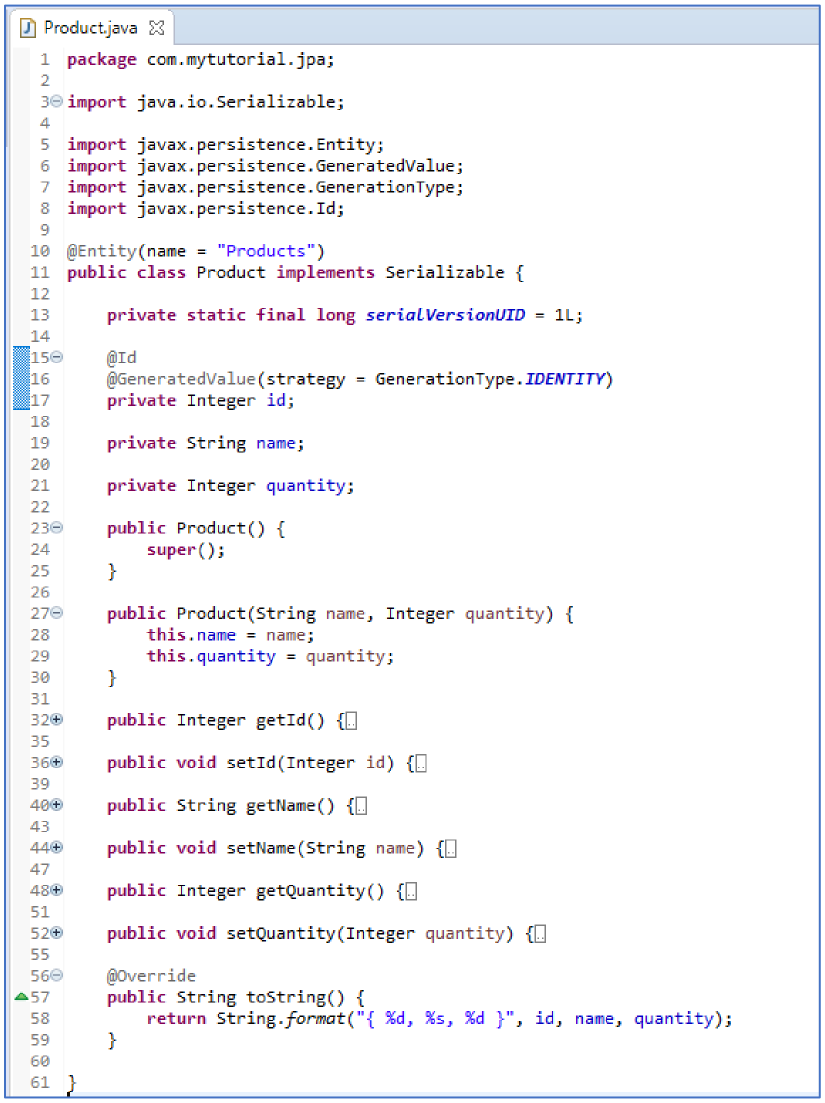
```java
@Entity(name = "Invoices")
public class Invoice implements Serializable {

    private static final long serialVersionUID = 1L;

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

    private Float amount;

    @OneToOne(mappedBy = "invoice")
    private Order order;

    public Invoice() {
    }
```

One-to-many Unidirectional Mapping

In this demo, we`ll see how we can set up a one-to-many relationship mapping between the entities that we use to populate our database tables. A one-to-many mapping in database tables means that one row in one table can be mapped to many rows in another table.

Here we are going to set up this one to many mapping using the relationship that exists between orders and products. Set up a new class here to represent a product entity that will be stored in the products table.

Go ahead and create **Product.java**.



Product.java

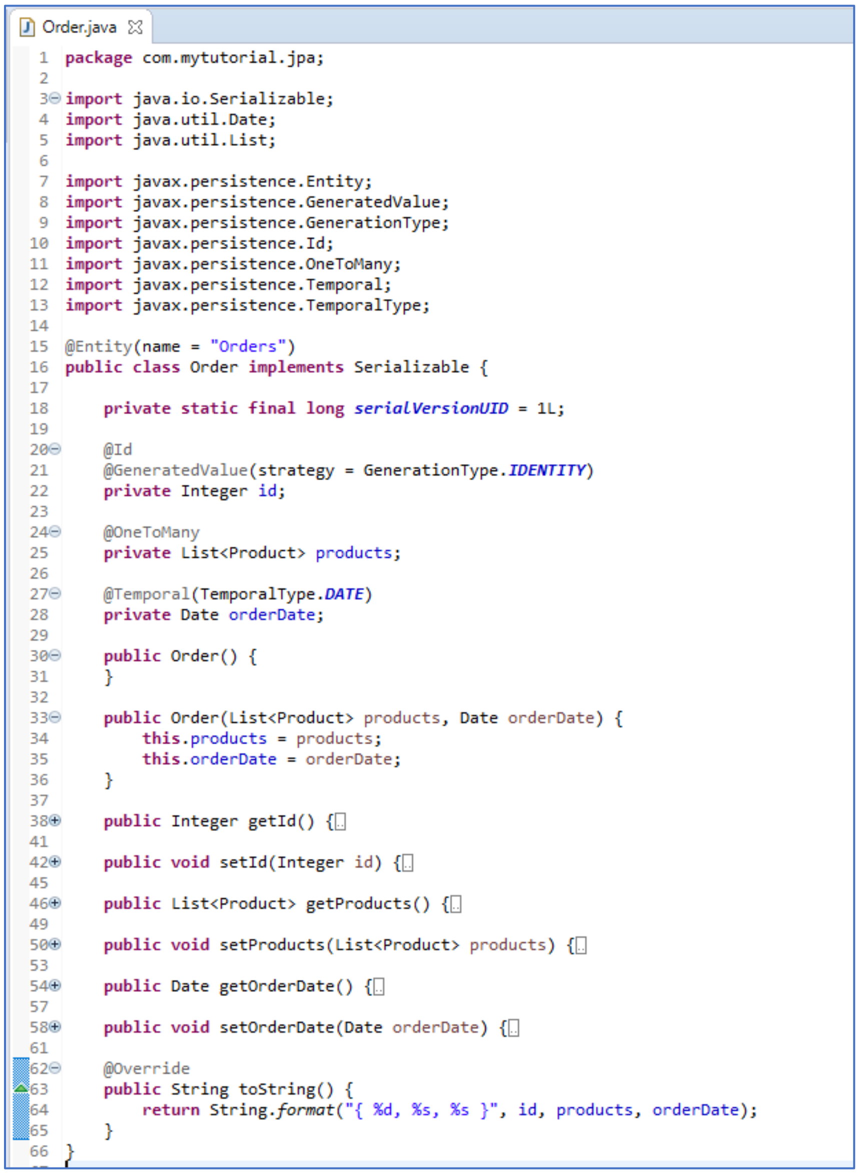
```java
package com.mytutorial.jpa;

import java.io.Serializable;

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

@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 Integer quantity;

    public Product() {
        super();
    }

    public Product(String name, Integer quantity) {
        this.name = name;
        this.quantity = quantity;
    }

    public Integer getId() {

    public void setId(Integer id) {

    public String getName() {

    public void setName(String name) {

    public Integer getQuantity() {

    public void setQuantity(Integer quantity) {

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

}
```

Product.java here is an entity that represents a product that we may buy on an e-commerce site.

- Set up the import statements for the entity annotation and other primary key id related annotations that we'll use in this product file.

- To start off with, I apply the **@Entity** annotation to this product class with the name property indicating that product objects that is product entities will be stored in a table called `"Products"`. The table name uses the plural form of the entity which is a more natural way to name database tables.
- So we have the **@Entity** annotations. Next step is to set up the member variables that make up the columns in the product table. But make sure that you have all of your classes **implement** the **Serializable** interface. That in general is a best practice.
- We have the **@Id** that is the **primary key**. We generate a value for `id`, and we have the `name` of the product and the `quantity` of the product that we have bought in an order.
- As usual, we'll set up the default no argument **constructor** and any other **constructors** that we want, which take in input arguments.
- Set up **getters** and **setters** for all of our member variables, `id`, `name`, `quantity`.
- I've also overridden the **toString()** method of the object base class to display the details of a product. `id` and `name` will be the details that will display in toString.

Now let's head over to **Order.java** and set this class up. There is not much change in this class, so I'm going to go through this quickly.



Order.java

```java
package com.mytutorial.jpa;

import java.io.Serializable;
import java.util.Date;
import java.util.List;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity(name = "Orders")
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;

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

    @OneToMany
    private List<Product> products;

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    public Order() {
    }

    public Order(List<Product> products, Date orderDate) {
        this.products = products;
        this.orderDate = orderDate;
    }

    public Integer getId() {

    public void setId(Integer id) {

    public List<Product> getProducts() {

    public void setProducts(List<Product> products) {

    public Date getOrderDate() {

    public void setOrderDate(Date orderDate) {

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

- The Order class implements the Serializable interface.
- We use the **@Entity** annotation and store order objects in the Orders table.
- The primary key is the `id` member variable and we tag it using **@Id** and the **@GeneratedValue** annotations.
- Now comes the interesting bit. Here is where we set up our unit directional **OneToMany mapping**. In a single order, it's possible for our customers to buy multiple products. This means that one record or one order entry in our Orders table, maps to multiple products in the products table. So this I'm going to represent using an **@OneToMany** annotation.

```java
@OneToMany
private List<Product> products;
```

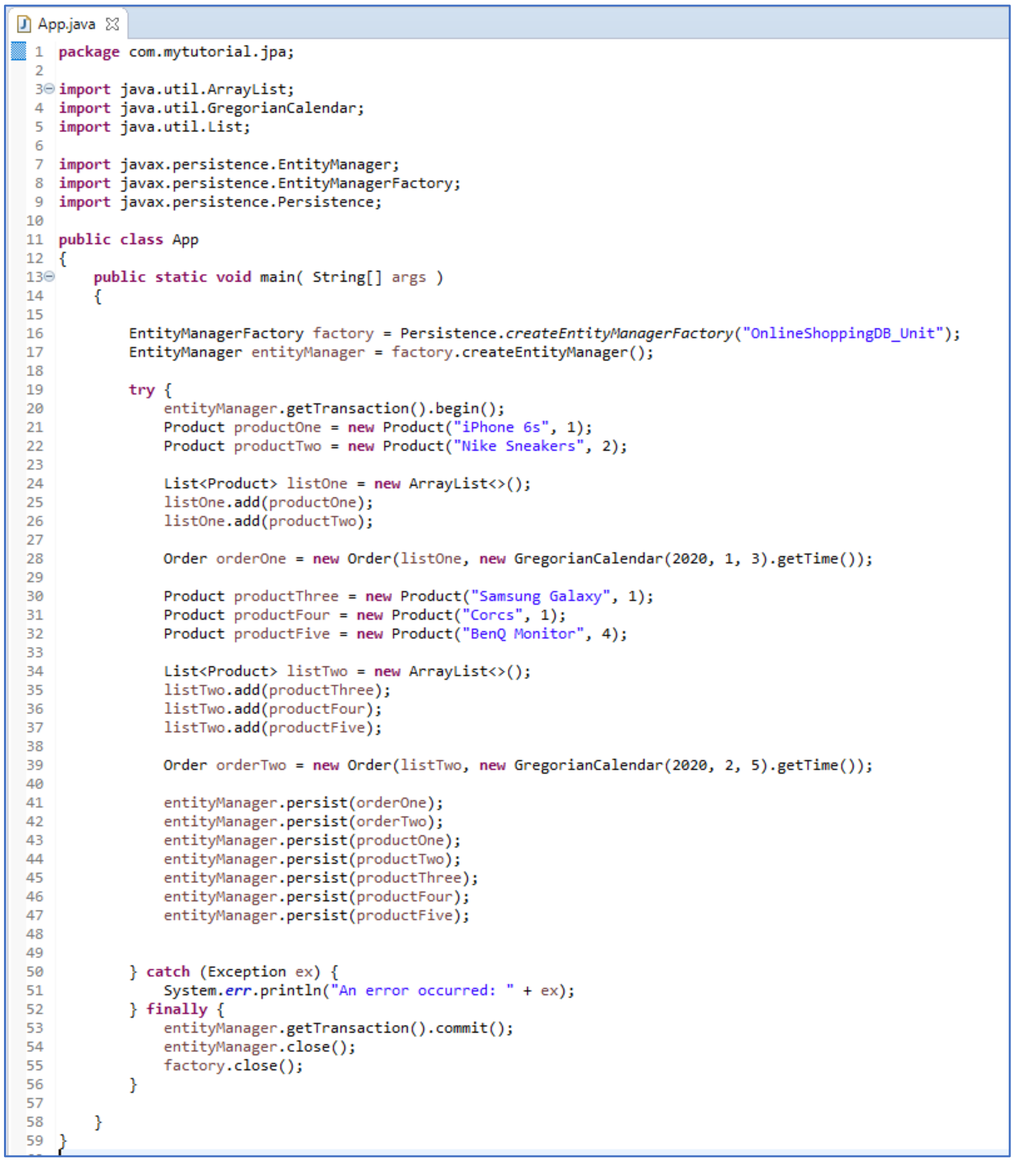
Notice that I've set up a **java.util.List** which references all of the products that are associated with this single order. I have a List of `Product` in the `products` member variable. It is this reference to the multiple `Products` associated with a single order that we tag using the **@OneToMany** annotation. This makes the `Order` entity the owning entity.

Also because only the `Order` entity has a reference to the `Products` that it contains, and there is **no reverse reference**, this is a **unidirectional mapping**.

It's important that you keep track of which side of the relationship is the owning site so that you set up your references correctly while persisting entities.

- In addition, I have the date of the order tagged with the **@Temporal** `(TemporalType.DATE).`

Time for us to head over to **App.java** and persist records using this one-to-many mapping relationship.

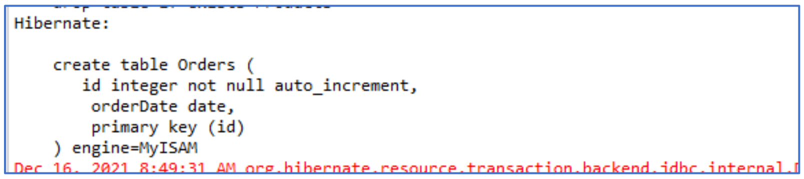

App.java

```java
package com.mytutorial.jpa;

import java.util.ArrayList;
import java.util.GregorianCalendar;
import java.util.List;

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("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {
            entityManager.getTransaction().begin();
            Product productOne = new Product("iPhone 6s", 1);
            Product productTwo = new Product("Nike Sneakers", 2);

            List<Product> listOne = new ArrayList<>();
            listOne.add(productOne);
            listOne.add(productTwo);

            Order orderOne = new Order(listOne, new GregorianCalendar(2020, 1, 3).getTime());

            Product productThree = new Product("Samsung Galaxy", 1);
            Product productFour = new Product("Corcs", 1);
            Product productFive = new Product("BenQ Monitor", 4);

            List<Product> listTwo = new ArrayList<>();
            listTwo.add(productThree);
            listTwo.add(productFour);
            listTwo.add(productFive);

            Order orderTwo = new Order(listTwo, new GregorianCalendar(2020, 2, 5).getTime());

            entityManager.persist(orderOne);
            entityManager.persist(orderTwo);
            entityManager.persist(productOne);
            entityManager.persist(productTwo);
            entityManager.persist(productThree);
            entityManager.persist(productFour);
            entityManager.persist(productFive);


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

    }
}
```

- Here in App.java, much of the boilerplate code remains the same, we use the `EntityManager`, the `EntityManagerFactory`, the `Persistence` **`class`**.
- Create the `EntityManager`, and then set up the entities that you want persisted to our underlying database.
- Within a transaction, I've created two product entities `productOne` and `productTwo` and added them together to a list called `listOne`.
- This `listOne` is associated with `orderOne`.
- Note that the owning entity that is the order references the `List` of `Products` in the mapping relationship. Then I have three more `Products`, `productThree`, `productFour`, `productFive`,
- and I set up `listTwo` with these three `Products`.
- If you scroll down below, you'll see I've defined another order entity that is `orderTwo` that references `listTwo` with three products.

- I then go ahead and `persist()` all of the entities that I've created, two `Orders` and five `Products`, a total of seven entities.

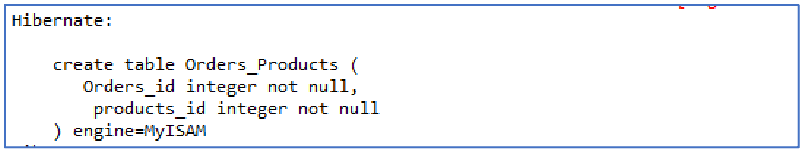
We are now ready to run this JPA Hibernate application and see how this one-to-many mapping relationship is set up by Hibernate.



```
Hibernate: 

    create table Orders (
       id integer not null auto_increment,
        orderDate date,
        primary key (id)
    ) engine=MyISAM
Dec 16, 2021 8:49:31 AM org.hibernate.resource.transaction.backend.jdbc.internal.
```

Notice that we have an `Orders` table with `id` and `orderDate`, `id` is the **primary key**.

Scrolling down further, you'll see that by default, a one-to-many mapping relationship is represented using a join table.

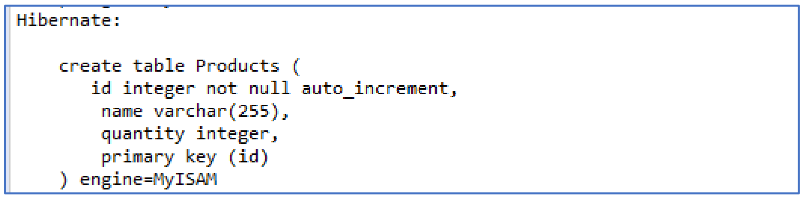


```
Hibernate: 

    create table Orders_Products (
       Orders_id integer not null,
        products_id integer not null
    ) engine=MyISAM
```

And the *default name of this join table is the name of the two entities that exist in this mapping relationship*, `Orders_Products`.

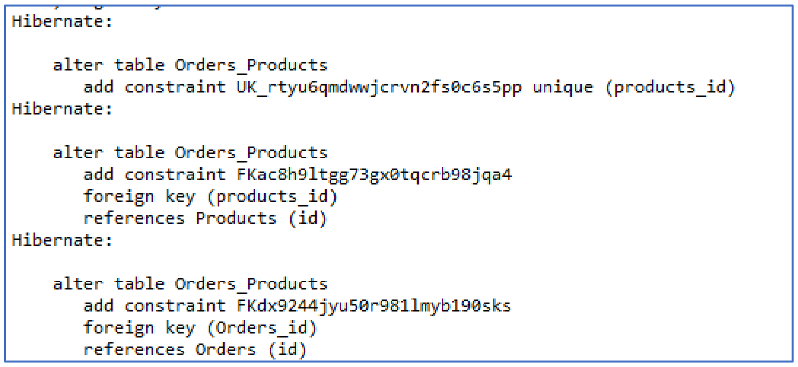
The two columns in the join table represent the `Orders_id` and the `product_id`. And these entries will set up the mapping between one order to multiple `Product`'s. Both of these columns have **the not null** constraint, the `"_id"`s **cannot be null**.

Scrolling further down, we have the products table with the primary key that is the id.



```
Hibernate: 

    create table Products (
       id integer not null auto_increment,
        name varchar(255),
        quantity integer,
        primary key (id)
    ) engine=MyISAM
```

And then you'll see the SQL statement setting up the foreign key constraints on our join table.

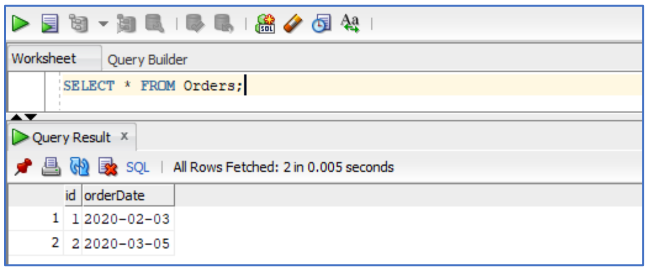


```
Hibernate: 

    alter table Orders_Products 
       add constraint UK_rtyu6qmdwwjcrvn2fs0c6s5pp unique (products_id)
Hibernate: 

    alter table Orders_Products 
       add constraint FKac8h9ltgg73gx0tqcrb98jqa4 
       foreign key (products_id) 
       references Products (id)
Hibernate: 

    alter table Orders_Products 
       add constraint FKdx9244jyu50r981lmyb190sks 
       foreign key (Orders_id) 
       references Orders (id)
```

- Notice that the `Orders_Products` table has an additional **uniqueness constraint** that has been added by Hibernate. `products_id` should be **unique**. This is a requirement in the one-to-many mapping. On the many side, the `Id` should be **unique**.

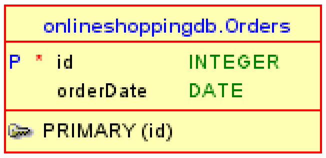
- Notice that the order id that is on the one side of the relationship, that cannot be unique. There'll be multiple entries in the `Orders_Products` table with the same order `Id`.
- In addition, there is a **foreign key** constraint from the join table that references the primary key in the `Products` table.
- Scrolling further down, notice that there is an additional **foreign key constraint** from the join table which references the `id` of the `order` in the `Orders` table.

Time to head over to MySQL Workbench to see how the relationship has been set up. Do a *select * from* the *Orders* table.



```
SELECT * FROM Orders;
```

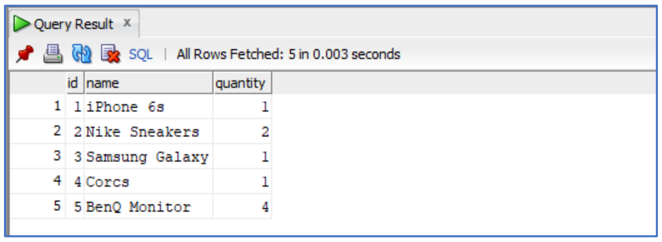
```
All Rows Fetched: 2 in 0.005 seconds

   id orderDate
1   1 2020-02-03
2   2 2020-03-05
```



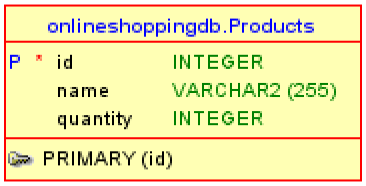
```
onlineshoppingdb.Orders
P * id          INTEGER
    orderDate   DATE
PRIMARY (id)
```

Here are the two order entities with `id` 1 and `id` 2 that we had inserted into the *Orders* table.

Lets do a *select ** on the *Products* table.



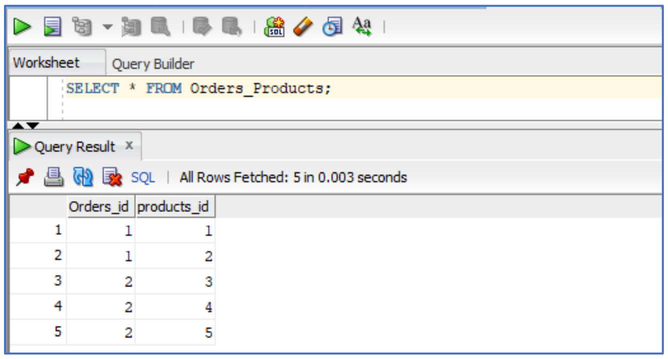
```
All Rows Fetched: 5 in 0.003 seconds

   id name            quantity
1   1 iPhone 6s              1
2   2 Nike Sneakers          2
3   3 Samsung Galaxy         1
4   4 Corcs                  1
5   5 BenQ Monitor           4
```



```
onlineshoppingdb.Products
P * id          INTEGER
    name        VARCHAR2 (255)
    quantity    INTEGER
PRIMARY (id)
```

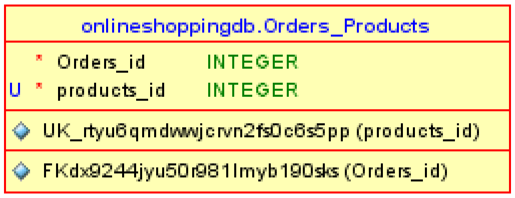
And you can see that all five product entities are present here with `id`'s ranging from 1 to 5.

Orders and products have been persisted correctly. Let's take a look at the join table which holds the mapping relationship one-to-many *Orders_Products*.



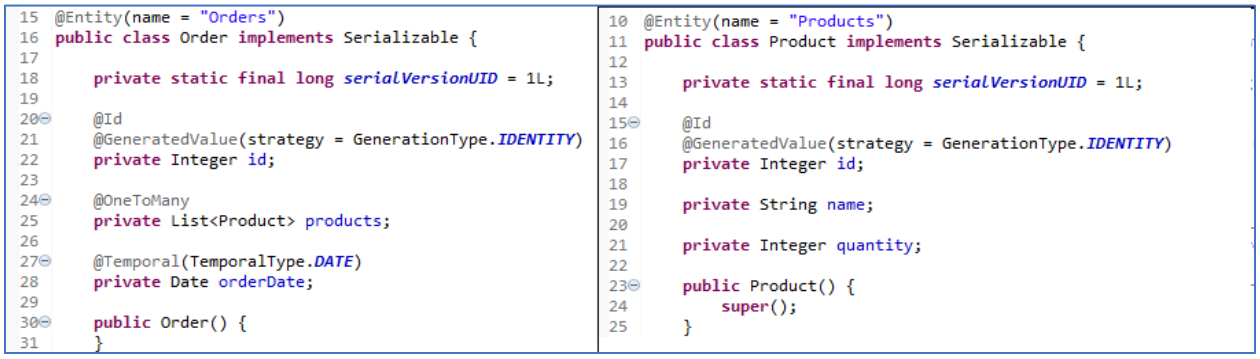
```
SELECT * FROM Orders_Products;
```

```
All Rows Fetched: 5 in 0.003 seconds

   Orders_id products_id
1          1           1
2          1           2
3          2           3
4          2           4
5          2           5
```



```
onlineshoppingdb.Orders_Products
  * Orders_id     INTEGER
U * products_id   INTEGER
UK_rtyu6qmdwwjcrvn2fs0c6s5pp (products_id)
FKdx9244jyu50r981lmyb190sks (Orders_id)
```

And notice here, that *Orders* `id` 1 is associated with two products. And `order_id` 2 is associated with three *Products*, ids 3, 4 and 5.

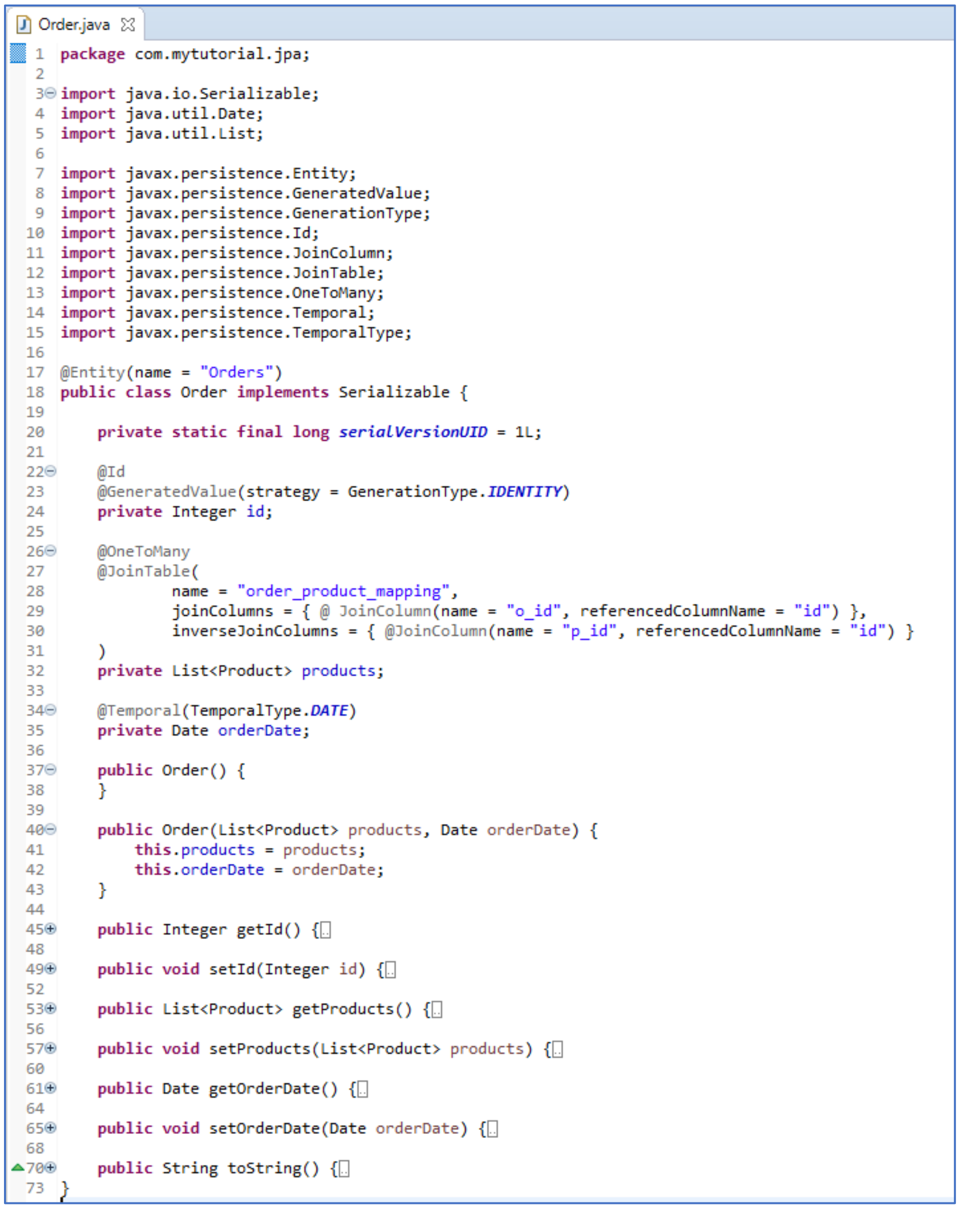


```java
@Entity(name = "Orders")
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;

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

    @OneToMany
    private List<Product> products;

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    public Order() {
    }
```

```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 Integer quantity;

    public Product() {
        super();
    }
```

One-to-many Mapping Using Join Tables

In the last demo we saw that the default manner in which Hibernate sets up the one to many mapping between tables is to set up a separate join table. It's possible for us to explicitly configure this join table as well. For this, we'll head over to the owning entity which is the `Order` entity, and make a few changes.


Order.java

```java
package com.mytutorial.jpa;

import java.io.Serializable;
import java.util.Date;
import java.util.List;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.OneToMany;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity(name = "Orders")
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;

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

    @OneToMany
    @JoinTable(
            name = "order_product_mapping",
            joinColumns = { @ JoinColumn(name = "o_id", referencedColumnName = "id") },
            inverseJoinColumns = { @JoinColumn(name = "p_id", referencedColumnName = "id") }
    )
    private List<Product> products;

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    public Order() {
    }

    public Order(List<Product> products, Date orderDate) {
        this.products = products;
        this.orderDate = orderDate;
    }

    public Integer getId() {

    public void setId(Integer id) {

    public List<Product> getProducts() {

    public void setProducts(List<Product> products) {

    public Date getOrderDate() {

    public void setOrderDate(Date orderDate) {

    public String toString() {
}
```

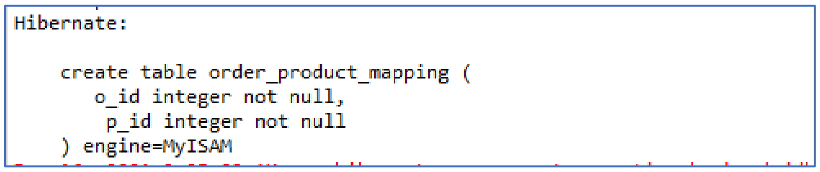
- Set up the **import** statements for `javax.persistence.JoinColumn` and `javax.persistence.JoinTable`. Because these are the JPA annotations that we'll be using.
- The `Order` entity has a `List` of references to the `Products` that are associated with each order. I'm going to set up the one to many relationship mapping as before.

```
@OneToMany
@JoinTable(
        name = "order_product_mapping",
        joinColumns =
                { @JoinColumn(name = "o_id", referencedColumnName = "id") },
        inverseJoinColumns =
                { @JoinColumn(name = "p_id", referencedColumnName = "id") }
        )
private List<Product> products;
```

- You can see the **@OneToMany** annotation is present on line 26.
- In addition, I specify the **@JoinTable** annotation, allowing me to configure the join table that holds this one to many mapping.
- The name of the join table, I want this to be "order_product_mapping". I specify this using the name property in the join table. The joinColumns property indicates what column in the Order entity will be referenced as a **foreign key** in our Order Product mapping join table.
- The joinColumns says name equal = "o_id". This is the name of the column in our join table. And the referencedColumnName from the current Orders table is the id column.
  The joinColumns property allows us to configure the constraint that exists between the owning entity which is the order entity. And the @JoinTable column, which references the Order entity.
- Similarly, we have an inverseJoinColumns property that we can specify.
  This sets up the relationship between the **foreign key reference** in the join table, which refers to the **primary key** of the Products table. This is the many side of the mapping. So the join column name is "p_id". This is the name of the column in the order_product_mapping join table. And this references the id column in the Products table. That is the primary key of the Products table.

We'll head over to App.java. We won't make any changes to the entities and the mappings that we set up from the previous demo. orderOne maps to product one and two. orderTwo maps to product three, four, and five.

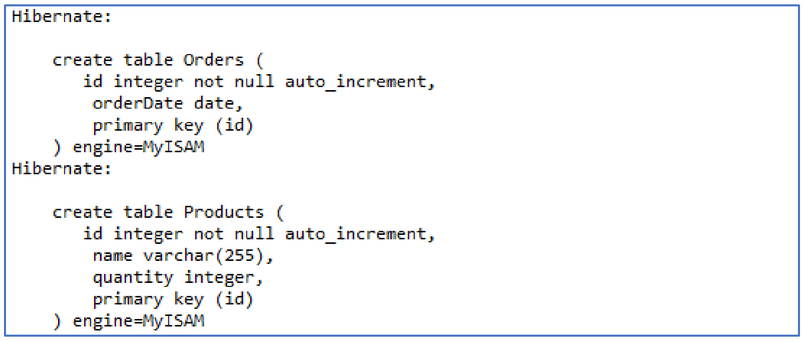
Go ahead and run this code. And let's take a look at how the join table is set up to hold this one to many relationship mapping.



```
Hibernate: 

    create table order_product_mapping (
       o_id integer not null,
        p_id integer not null
    ) engine=MyISAM
```

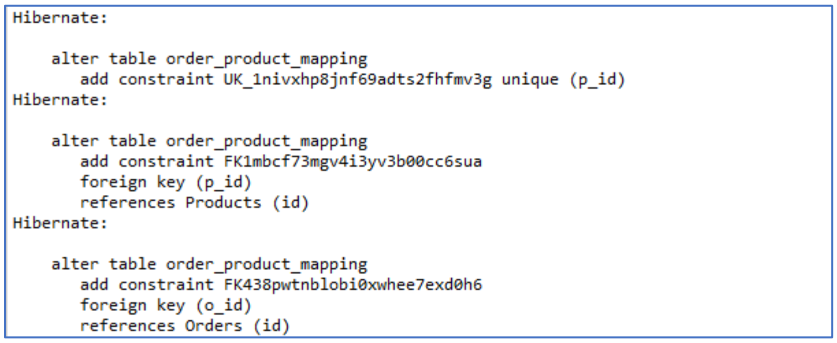
- Notice that the name of the join table is order_product_mapping, as we had specified in the **@JoinTable** annotation.
- The o_id is the column that references the Order id.
- The p_id is the column that references the Product id.

The remaining tables are set up as before. We have the Orders table containing information about an order. And the Products table containing information about products that we buy.



```
Hibernate: 

    create table Orders (
       id integer not null auto_increment,
        orderDate date,
        primary key (id)
    ) engine=MyISAM
Hibernate: 

    create table Products (
       id integer not null auto_increment,
        name varchar(255),
        quantity integer,
        primary key (id)
    ) engine=MyISAM
```

Notice that we alter the order_product_mapping table to add a uniqueness constraint to the p_id column.

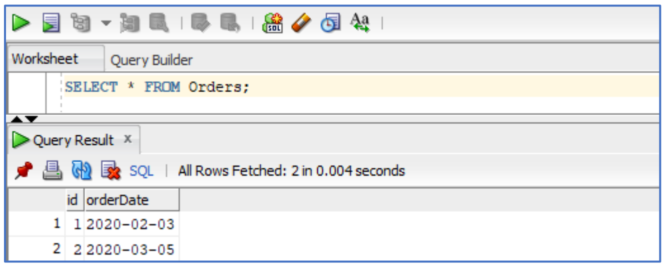


```
Hibernate: 

    alter table order_product_mapping 
       add constraint UK_1nivxhp8jnf69adts2fhfmv3g unique (p_id)
Hibernate: 

    alter table order_product_mapping 
       add constraint FK1mbcf73mgv4i3yv3b00cc6sua 
       foreign key (p_id) 
       references Products (id)
Hibernate: 

    alter table order_product_mapping 
       add constraint FK438pwtnblobi0xwhee7exd0h6 
       foreign key (o_id) 
       references Orders (id)
```

We also set up the foreign key references where p_id references the primary key of the Products table. And o_id references the primary key of the Orders table.

So the setup is the same, but we've been able to configure the join table in a very granular manner.

Let's head over to SQL Workbench. And I'm going to run a *select ** from the *Orders* table.



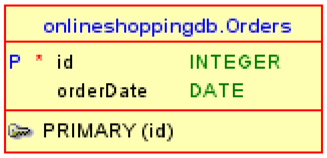
Worksheet | Query Builder

```
SELECT * FROM Orders;
```

Query Result ×

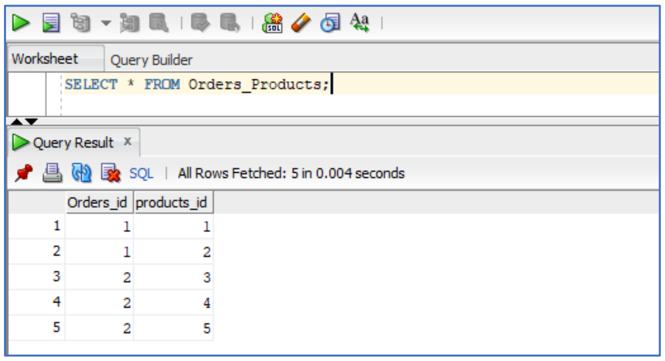
SQL | All Rows Fetched: 2 in 0.004 seconds

	id	orderDate
1	1	2020-02-03
2	2	2020-03-05



```
onlineshoppingdb.Orders
P * id          INTEGER
    orderDate   DATE
PRIMARY (id)
```

Here are the two `Order` entities with `id` 1 and 2 that we have persisted.
Similarly let's run a *select ** on the *Products* table.



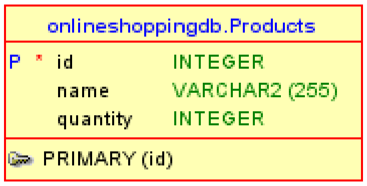
Worksheet | Query Builder

```
SELECT * FROM Orders_Products;
```

Query Result ×

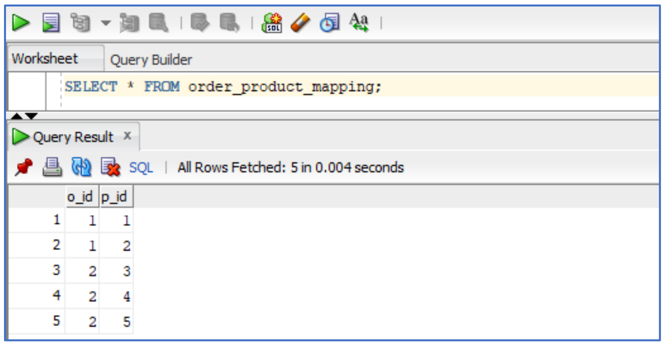
SQL | All Rows Fetched: 5 in 0.004 seconds

	Orders_id	products_id
1	1	1
2	1	2
3	2	3
4	2	4
5	2	5



```
onlineshoppingdb.Products
P * id          INTEGER
    name        VARCHAR2 (255)
    quantity    INTEGER
PRIMARY (id)
```

And this gives us the five product entities that we have persisted in here.
These tables don't hold the mapping information. That is present in a separate join table called *order_product_mapping*. Let's run a *select ** here.



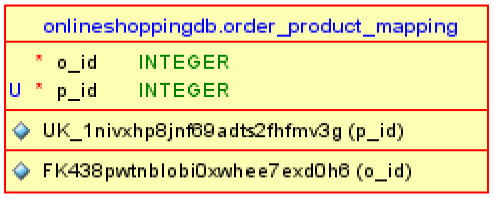
Worksheet | Query Builder

```
SELECT * FROM order_product_mapping;
```

Query Result ×

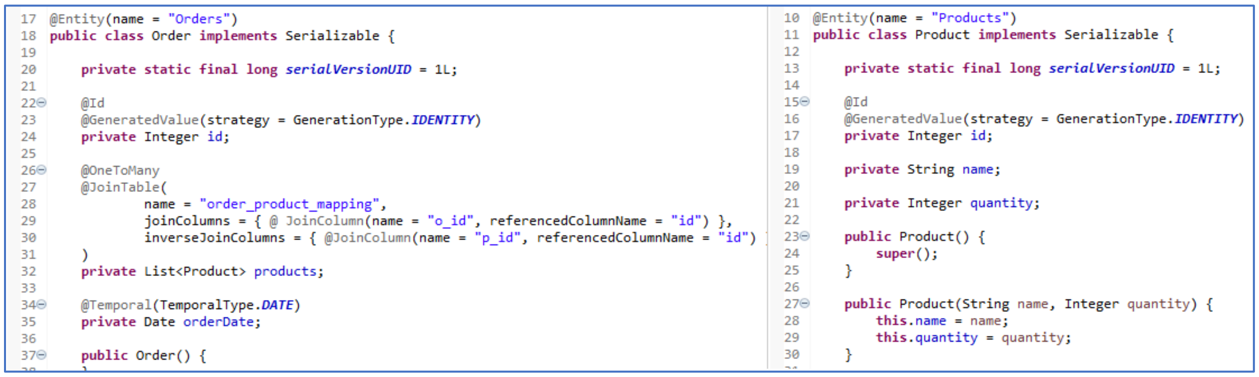
SQL | All Rows Fetched: 5 in 0.004 seconds

	o_id	p_id
1	1	1
2	1	2
3	2	3
4	2	4
5	2	5



```
onlineshoppingdb.order_product_mapping
  * o_id    INTEGER
U * p_id    INTEGER
UK_1nivxhp8jnf69adts2fhfmv3g (p_id)
FK438pwtnblobi0xwhee7exd0h6 (o_id)
```

Notice that `Order id` 1 is associated with two `Products`. And `Order id` 2 is associated with three `Products`.



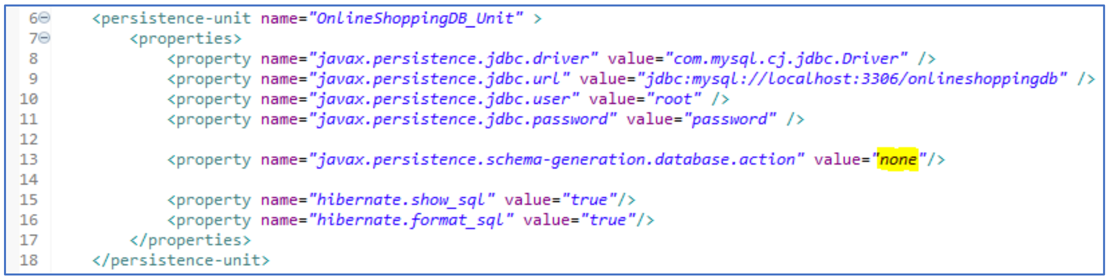
```java
@Entity(name = "Orders")
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;

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

    @OneToMany
    @JoinTable(
            name = "order_product_mapping",
            joinColumns = { @ JoinColumn(name = "o_id", referencedColumnName = "id") },
            inverseJoinColumns = { @JoinColumn(name = "p_id", referencedColumnName = "id")
    )
    private List<Product> products;

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    public Order() {
```

```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 Integer quantity;

    public Product() {
        super();
    }

    public Product(String name, Integer quantity) {
        this.name = name;
        this.quantity = quantity;
    }
```

One-to-many Mapping Eager Loading

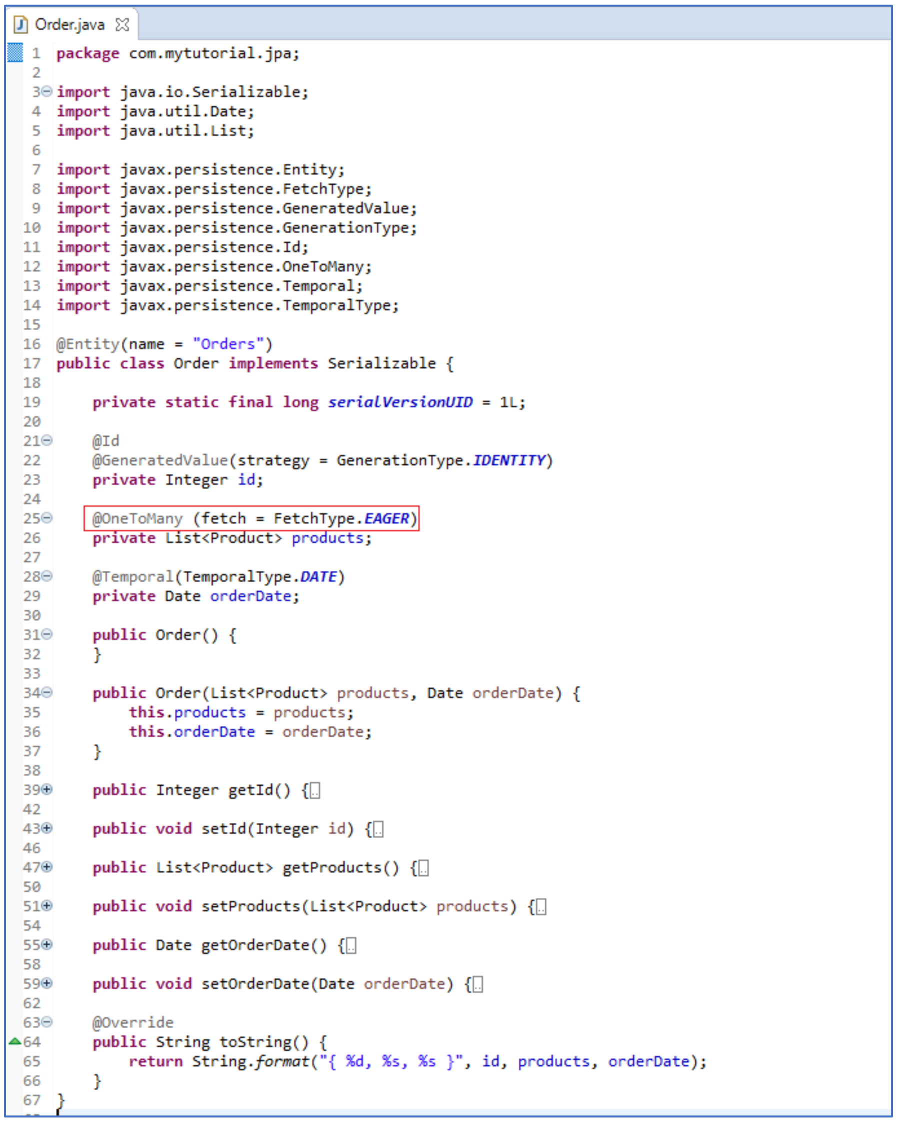
Now when you set up a mapping relationship between two entities, whether it's one-to-one, one-to-many, many-to-one or many to many. You have several ways in which your data can be loaded when you fetch one entity. We'll explore those two techniques, eager loading and lazy loading.

In this first demo here, we'll see how eager loading of related entities work. In my persistence.xml file, I'm going to update the database.action property to be set to none so that new tables are not created. Instead, we'll query the entities that are already present in our underlying database tables.



```
<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="none"/>

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

I'll now head over to the owning entity in this unidirectional one-to-many relationship, which is the `Order` entity. And I'm going to specify a fetch type for related `Products`.



Order.java

```java
package com.mytutorial.jpa;

import java.io.Serializable;
import java.util.Date;
import java.util.List;

import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity(name = "Orders")
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;

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

    @OneToMany (fetch = FetchType.EAGER)
    private List<Product> products;

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    public Order() {
    }

    public Order(List<Product> products, Date orderDate) {
        this.products = products;
        this.orderDate = orderDate;
    }

    public Integer getId() {

    public void setId(Integer id) {

    public List<Product> getProducts() {

    public void setProducts(List<Product> products) {

    public Date getOrderDate() {

    public void setOrderDate(Date orderDate) {

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

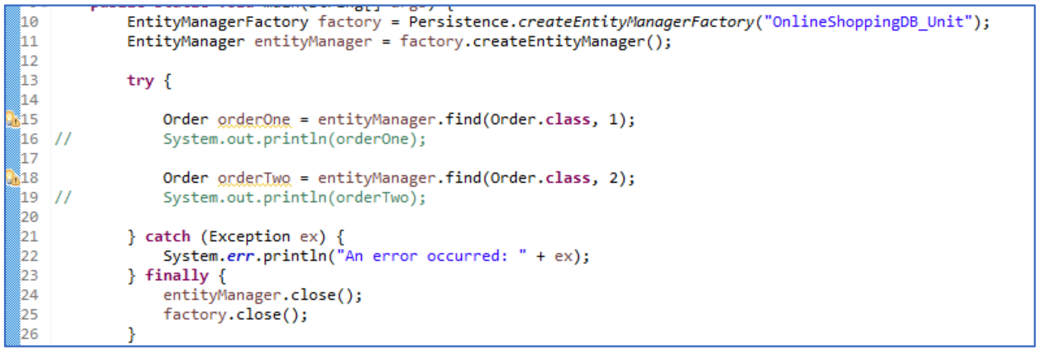
- Now in this OneToMany annotation, I'll update this, I'll get rid of the JoinTable annotation.
- And I'm going to specify a fetch type associated with the `fetch` property. `FetchType` is equal to ***EAGER***.

Specifying a `fetch` type of ***EAGER*** means that you eagerly load mapped entities. That is, whenever you're retrieving an `Order`, go ahead and fetch all of the related products as well.

So for every order retrieval, you'll retrieve associated products even if you don't explicitly access those products within your code. Eager loading is something that you'll specify if you're fairly sure that each time you retrieve an order, you're going to access products.

So in that case, it's better to prefetch these products. But if you know that you'll generally access an order entity by itself without accessing its product. This can be wasteful because it will go ahead and perform additional join operations to load in the related products. It'll also cause extra data transfer between the database and your application. When you specify eager loading, make sure you think through why exactly you need it.

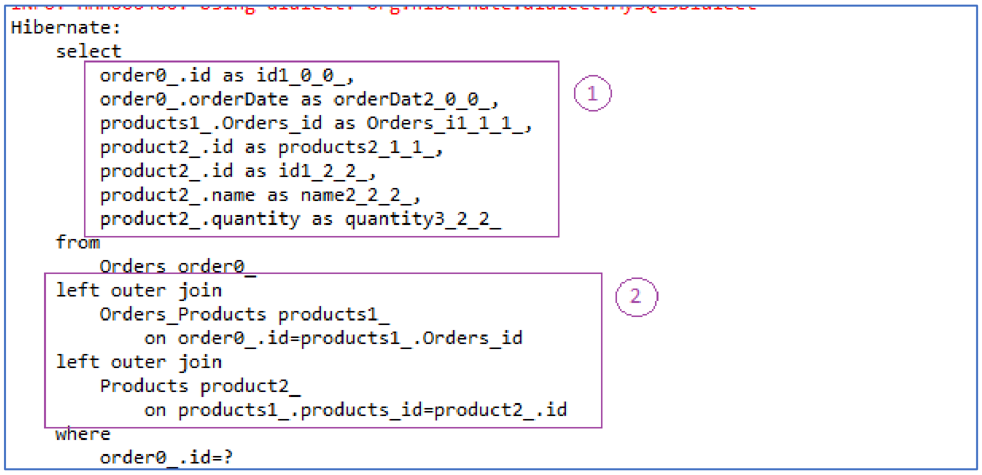
Head over to **App.java**, and change the code already exists here. I'm going to change the code to perform retrieve operations on order entities.



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

        try {

            Order orderOne = entityManager.find(Order.class, 1);
//          System.out.println(orderOne);

            Order orderTwo = entityManager.find(Order.class, 2);
//          System.out.println(orderTwo);

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

- Notice that I invoke the `entityManager.find()` operation to retrieve both `Orders`, the one with ID 1 and the one with ID 2.
- However, after retrieving the orders, I'm not actually accessing the orders or the related products within an order. I have commented out the code which prints out the individual orders. When we print out an order, the way I've defined the **toString()** operation on the order class is to print out the related products as well. So we're not accessing the products, we're only retrieving the orders.

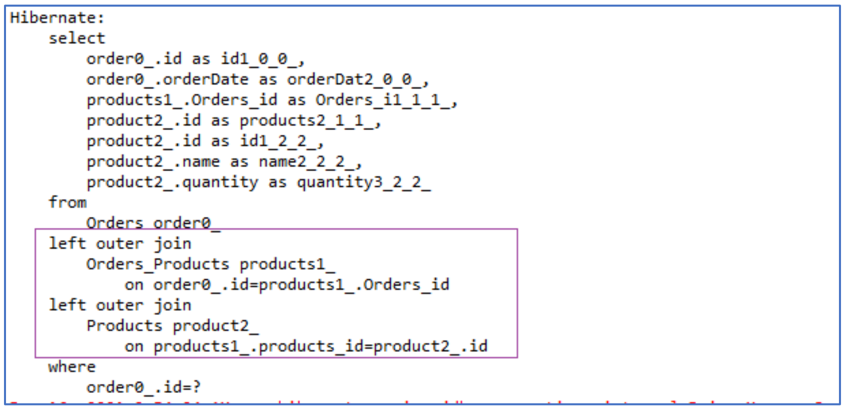
Let's go ahead and run this code and take a look at the select queries that are run under the hood.



```
Hibernate: 
    select
        order0_.id as id1_0_0_,
        order0_.orderDate as orderDat2_0_0_,
        products1_.Orders_id as Orders_i1_1_1_,
        product2_.id as products2_1_1_,
        product2_.id as id1_2_2_,
        product2_.name as name2_2_2_,
        product2_.quantity as quantity3_2_2_ 
    from
        Orders order0_ 
    left outer join
        Orders_Products products1_ 
            on order0_.id=products1_.Orders_id 
    left outer join
        Products product2_ 
            on products1_.products_id=product2_.id 
    where
        order0_.id=?
```

1) Because we specified ***EAGERLY*** loading of the `Products` associated with an `Order`. When we perform a select query on the `Orders` table, we retrieve the associated `Product` information as well. You can see that the select includes the `Product` fields.
2) In order to be able to retrieve the products associated with an order, notice that hibernate performs a join operation.
   It first joins with the mapping table, that is `orders_products`. And then it performs a further join with the Products table to retrieve the actual product information.

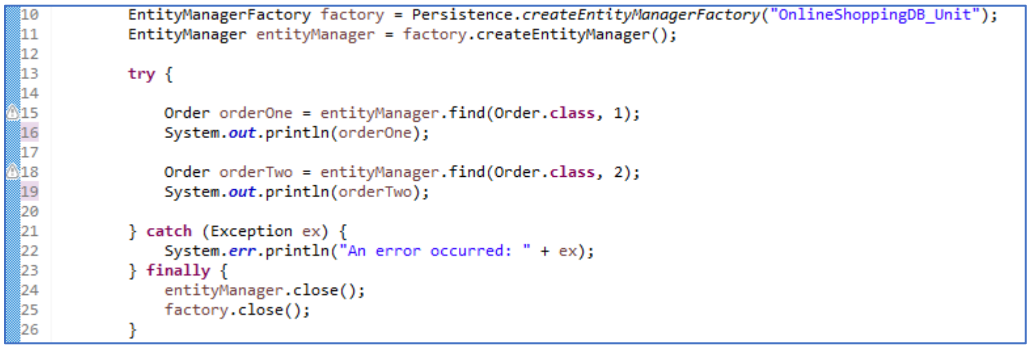
And the same thing is true for the second select query to retrieve the second order with ID 2.



```
Hibernate: 
    select
        order0_.id as id1_0_0_,
        order0_.orderDate as orderDat2_0_0_,
        products1_.Orders_id as Orders_i1_1_1_,
        product2_.id as products2_1_1_,
        product2_.id as id1_2_2_,
        product2_.name as name2_2_2_,
        product2_.quantity as quantity3_2_2_ 
    from
        Orders order0_ 
    left outer join
        Orders_Products products1_ 
            on order0_.id=products1_.Orders_id 
    left outer join
        Products product2_ 
            on products1_.products_id=product2_.id 
    where
        order0_.id=?
```

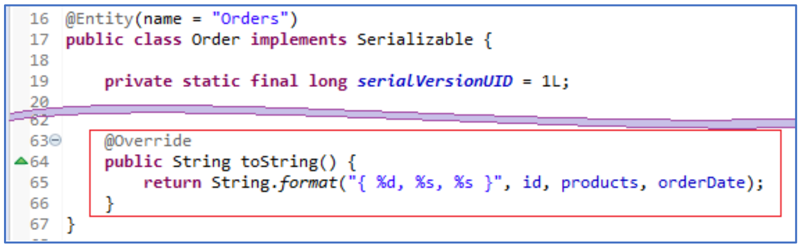
So retrieving an order will retrieve the associated products by performing join operations under the hood. And this is wasteful if you don't need the products information.

So use eager loading with care. I'll now uncomment the two lines of code that prints out the contents of `orderOne` and `orderTwo`.



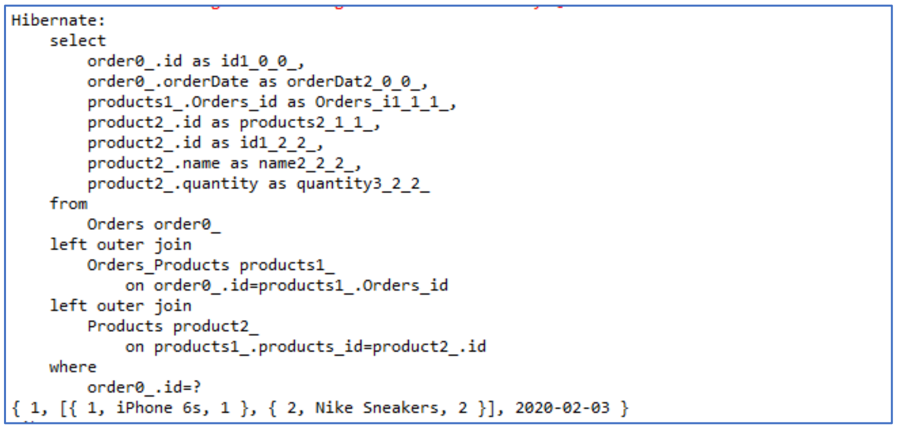
```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {

            Order orderOne = entityManager.find(Order.class, 1);
            System.out.println(orderOne);

            Order orderTwo = entityManager.find(Order.class, 2);
            System.out.println(orderTwo);

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

Our **toString()** implementation in the order class which will be invoked in these print statements references the products associated with an order.



```
@Entity(name = "Orders")
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;

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

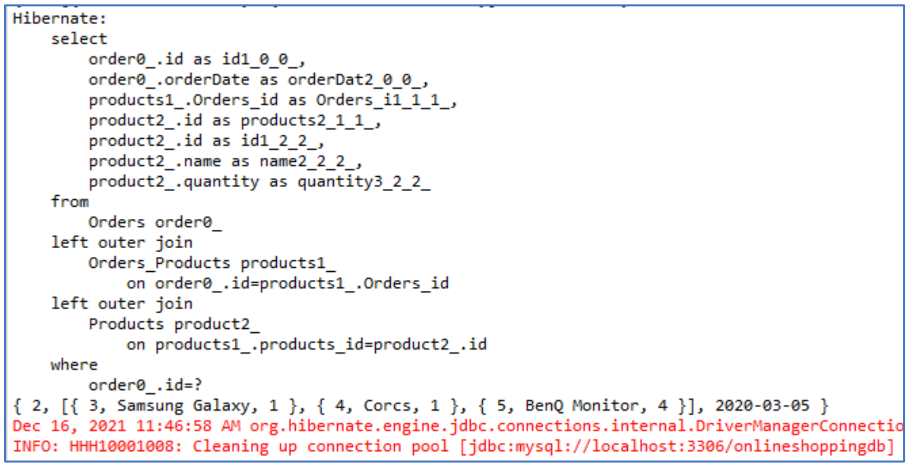
So we are actually referencing the products now, not just retrieving the order. Now if you run this code and if you take a look at the select statements, nothing has changed.



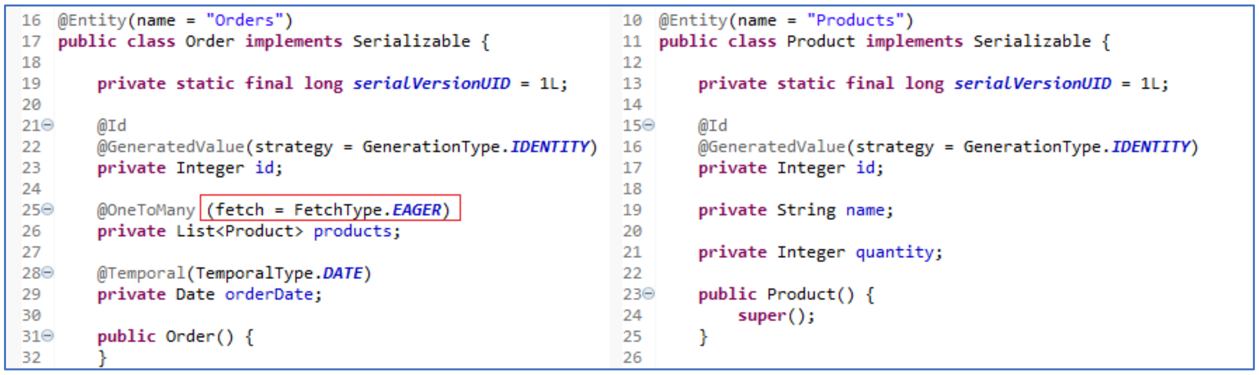
```
Hibernate:
    select
        order0_.id as id1_0_0_,
        order0_.orderDate as orderDat2_0_0_,
        products1_.Orders_id as Orders_i1_1_1_,
        product2_.id as products2_1_1_,
        product2_.id as id1_2_2_,
        product2_.name as name2_2_2_,
        product2_.quantity as quantity3_2_2_
    from
        Orders order0_
    left outer join
        Orders_Products products1_
            on order0_.id=products1_.Orders_id
    left outer join
        Products product2_
            on products1_.products_id=product2_.id
    where
        order0_.id=?
{ 1, [{ 1, iPhone 6s, 1 }, { 2, Nike Sneakers, 2 }], 2020-02-03 }
```

- So we retrieve the products using join operations when we reference the products as well.
- So whether or not we reference the products, the products information is eagerly fetched from the database.
- Here on screen we've printed out the contents of order 1 and the two products in that order, iPhone and Nike sneakers.

And if you scroll down below, you'll see the contents of order 2 and the three products associated with that order printed out to screen as well.



```
Hibernate:
    select
        order0_.id as id1_0_0_,
        order0_.orderDate as orderDat2_0_0_,
        products1_.Orders_id as Orders_i1_1_1_,
        product2_.id as products2_1_1_,
        product2_.id as id1_2_2_,
        product2_.name as name2_2_2_,
        product2_.quantity as quantity3_2_2_
    from
        Orders order0_
    left outer join
        Orders_Products products1_
            on order0_.id=products1_.Orders_id
    left outer join
        Products product2_
            on products1_.products_id=product2_.id
    where
        order0_.id=?
{ 2, [{ 3, Samsung Galaxy, 1 }, { 4, Corcs, 1 }, { 5, BenQ Monitor, 4 }], 2020-03-05 }
Dec 16, 2021 11:46:58 AM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectio
INFO: HHH10001008: Cleaning up connection pool [jdbc:mysql://localhost:3306/onlineshoppingdb]
```

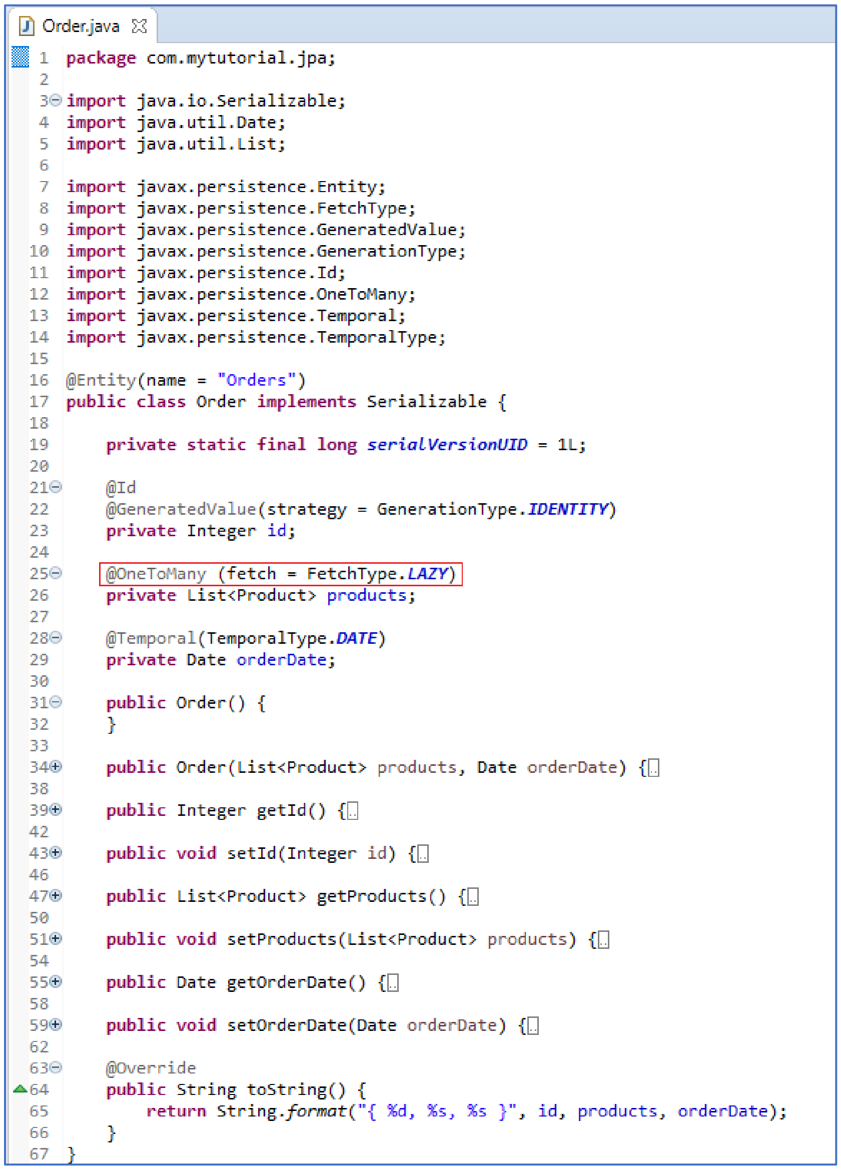


```
@Entity(name = "Orders")
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;

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

    @OneToMany(fetch = FetchType.EAGER)
    private List<Product> products;

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    public Order() {
    }
```

```
@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 Integer quantity;

    public Product() {
        super();
    }

```

One-to-many Mapping Lazy Loading

In order to make your app execution a little more performant. If you're not always going to access the products associated with an `Order` while retrieving an order. It's good practice to change the FetchType for your related products to be LAZY.



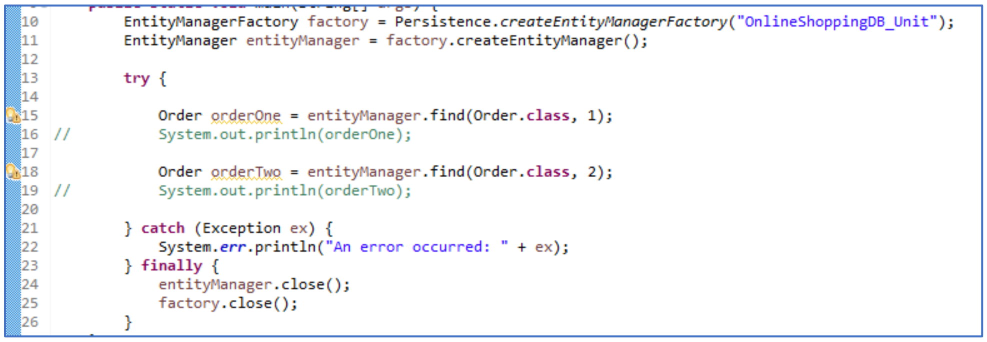
Order.java

```java
package com.mytutorial.jpa;

import java.io.Serializable;
import java.util.Date;
import java.util.List;

import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity(name = "Orders")
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;

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

    @OneToMany (fetch = FetchType.LAZY)
    private List<Product> products;

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    public Order() {
    }

    public Order(List<Product> products, Date orderDate) {

    public Integer getId() {

    public void setId(Integer id) {

    public List<Product> getProducts() {

    public void setProducts(List<Product> products) {

    public Date getOrderDate() {

    public void setOrderDate(Date orderDate) {

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

So in the OneToMany annotation here, I'm going to change the `FetchType` of the related `Products` for an `Order` to be `FetchType.`***`LAZY`***.
With lazy loading, only if we explicitly access the `Products` associated with an `Order` will the `Product` information be retrieved. Otherwise by default when we retrieve an `Order`, the `Products` information will not be retrieved, it will only be fetched on demand.

Based on how you plan to retrieve and use your order information, **lazy loading is a great performance optimization**. You'll fetch the related products for an order only when you explicitly access this products information from your order.

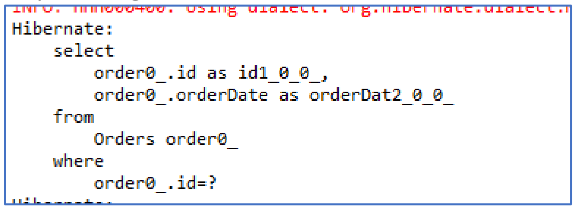
Let's now see how lazy loading works. We'll see this in action. Head over to **App.java** and I'm going to retrieve the two orders that we have persisted in our underlying database table.



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

        try {

            Order orderOne = entityManager.find(Order.class, 1);
//          System.out.println(orderOne);

            Order orderTwo = entityManager.find(Order.class, 2);
//          System.out.println(orderTwo);

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

- However, I'm not going to print out the order information. So I'm going to comment out the code for the `System.out.println()` statements.
  Remember, when we print out the order information, we access the related `Products` to print those out to screen as well.
- So I'm only retrieving the `Orders`, I'm not accessing `Products`.

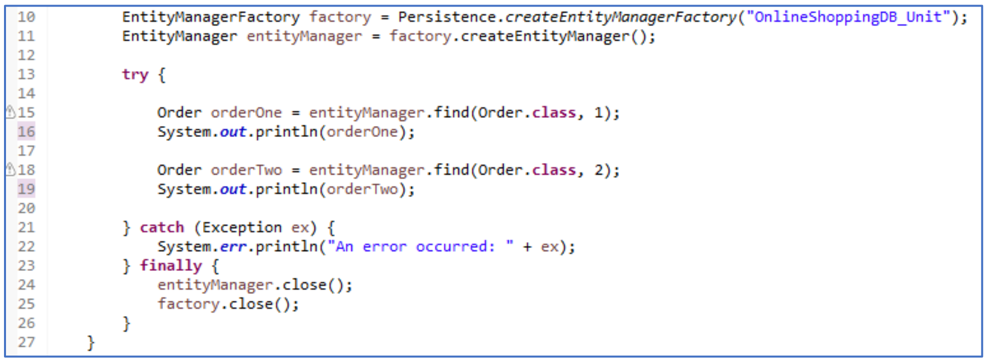
Once you run this code, you'll see the SQL statements run by Hibernate under the hood when you use lazy loading.



```
Hibernate: 
    select
        order0_.id as id1_0_0_,
        order0_.orderDate as orderDat2_0_0_ 
    from
        Orders order0_ 
    where
        order0_.id=?
```

- Observe that the select statement here references the `Orders` table and retrieves `Orders` information. But it **does not retrieve** the associated `Products` information. We only retrieve the `Orders` information for both of our find method invocations.
- Observe that there is **no join** performed with the `Products` table.

With lazy loading, your map entities will be retrieved if you explicitly access those entities from within your owning entity.

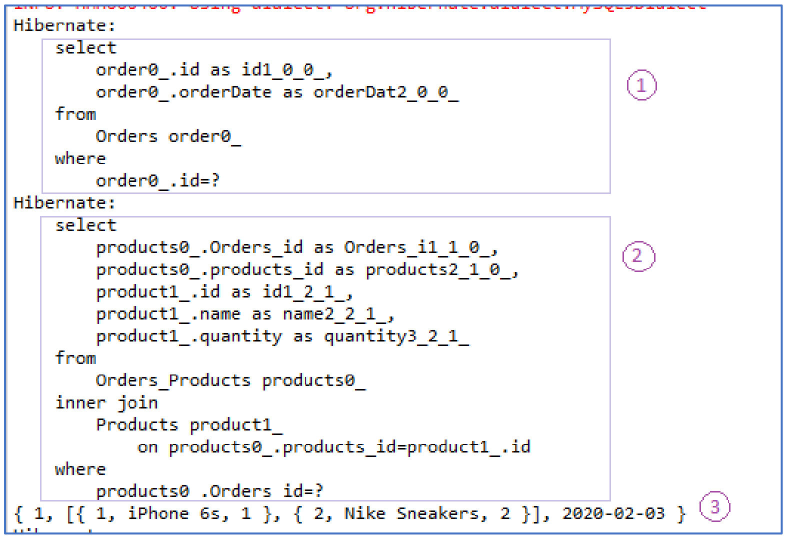


```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {

            Order orderOne = entityManager.find(Order.class, 1);
            System.out.println(orderOne);

            Order orderTwo = entityManager.find(Order.class, 2);
            System.out.println(orderTwo);

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

- I've uncommented out the two `System.out.println()` statements here in my code.
- When I print out `orderOne` and `orderTwo`, the toString methods of these objects will be invoked. And the **toString()** implementation of our order class references the products associated with each order.
  With these print statements, I'm explicitly referencing the products associated with each `Order`. Which means `Product` information should be retrieved from the underlying database tables.

Let's see if it is. I'm going to go ahead and run my code. And in the console window we'll see the SQL queries that are run.

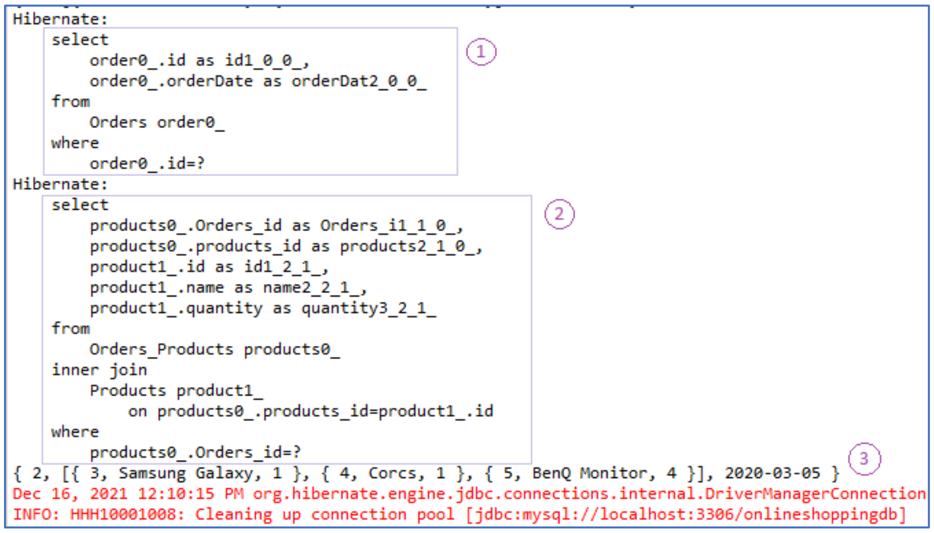
Here's a select query that retrieves the first `Order` with `id` 1. And just below that, you'll see another select query, which retrieves the products associated with this order.



```
Hibernate:
    select
        order0_.id as id1_0_0_,
        order0_.orderDate as orderDat2_0_0_
    from
        Orders order0_
    where
        order0_.id=?
Hibernate:
    select
        products0_.Orders_id as Orders_i1_1_0_,
        products0_.products_id as products2_1_0_,
        product1_.id as id1_2_1_,
        product1_.name as name2_2_1_,
        product1_.quantity as quantity3_2_1_
    from
        Orders_Products products0_
    inner join
        Products product1_
            on products0_.products_id=product1_.id
    where
        products0_.Orders_id=?
{ 1, [{ 1, iPhone 6s, 1 }, { 2, Nike Sneakers, 2 }], 2020-02-03 }
```

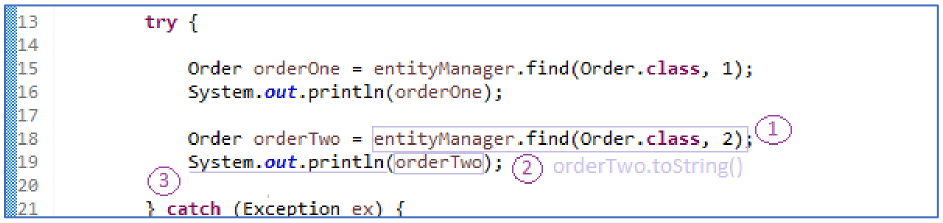
1) Observe that with lazy loading, Hibernate runs two separate queries. One to retrieve the `Order` information, the second query to retrieve the `Products` information.
2) The second query is run only if we explicitly reference the `Products` list. If you scroll a little further, you'll see the output of the `System.out.println()` command.
3) And lastly, here is the `Order` with id 1 and the two `Products` associated with this order are the “iPhone 6S” and “Nike sneakers”.

Let's scroll down further,



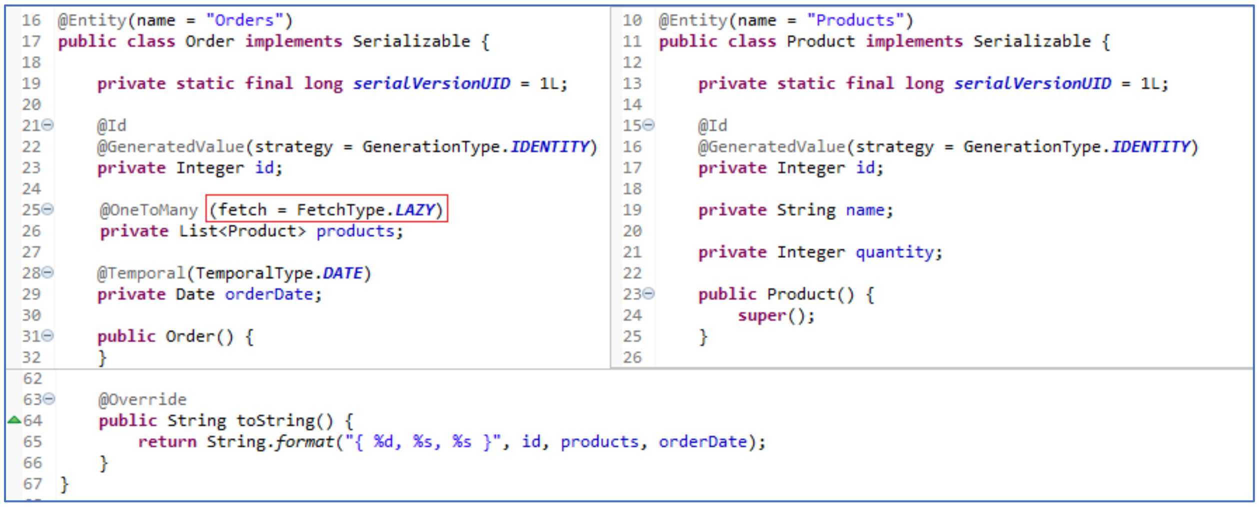
```
Hibernate: 
    select
        order0_.id as id1_0_0_,
        order0_.orderDate as orderDat2_0_0_ 
    from
        Orders order0_ 
    where
        order0_.id=?
Hibernate: 
    select
        products0_.Orders_id as Orders_i1_1_0_,
        products0_.products_id as products2_1_0_,
        product1_.id as id1_2_1_,
        product1_.name as name2_2_1_,
        product1_.quantity as quantity3_2_1_ 
    from
        Orders_Products products0_ 
    inner join
        Products product1_ 
            on products0_.products_id=product1_.id 
    where
        products0_.Orders_id=?
{ 2, [{ 3, Samsung Galaxy, 1 }, { 4, Corcs, 1 }, { 5, BenQ Monitor, 4 }], 2020-03-05 }
Dec 16, 2021 12:10:15 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnection
INFO: HHH10001008: Cleaning up connection pool [jdbc:mysql://localhost:3306/onlineshoppingdb]
```

1) Here is the second select statement for the second `Order`.



```
        try {

            Order orderOne = entityManager.find(Order.class, 1);
            System.out.println(orderOne);

            Order orderTwo = entityManager.find(Order.class, 2);
            System.out.println(orderTwo);

        } catch (Exception ex) {
```

2) And after that we have another select statement to retrieve the `Products` associated with the second `Order`.
   Retrieving the `Products` associated with an `Order` is performed only when we access the `Products` and this involves a **join operation**.
3) And here at the bottom we can see the output of the print statement. Where we print out the `Products` associated with `Order` `id` 2, the “Samsung Galaxy”, “Crocs“ and “BenQ Monitors”.



```
@Entity(name = "Orders")
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;

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

    @OneToMany (fetch = FetchType.LAZY)
    private List<Product> products;

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    public Order() {
    }
```

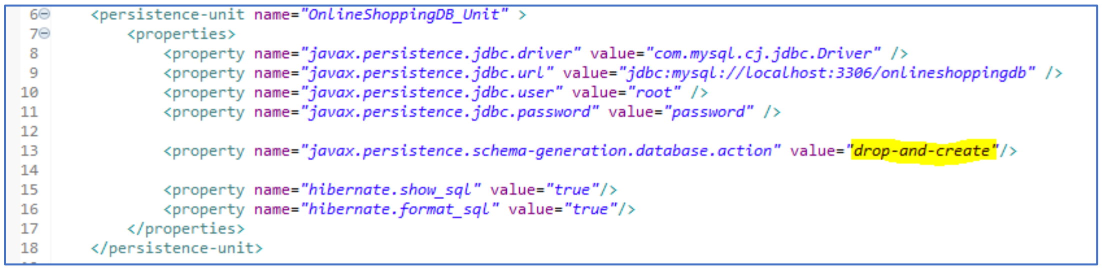
```
@Entity(name = "Products")
public class Product implements Serializable {

    private static final long serialVersionUID = 1L;

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

    private String name;

    private Integer quantity;

    public Product() {
        super();
    }

```

```

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

One-to-many Mapping With Join Columns

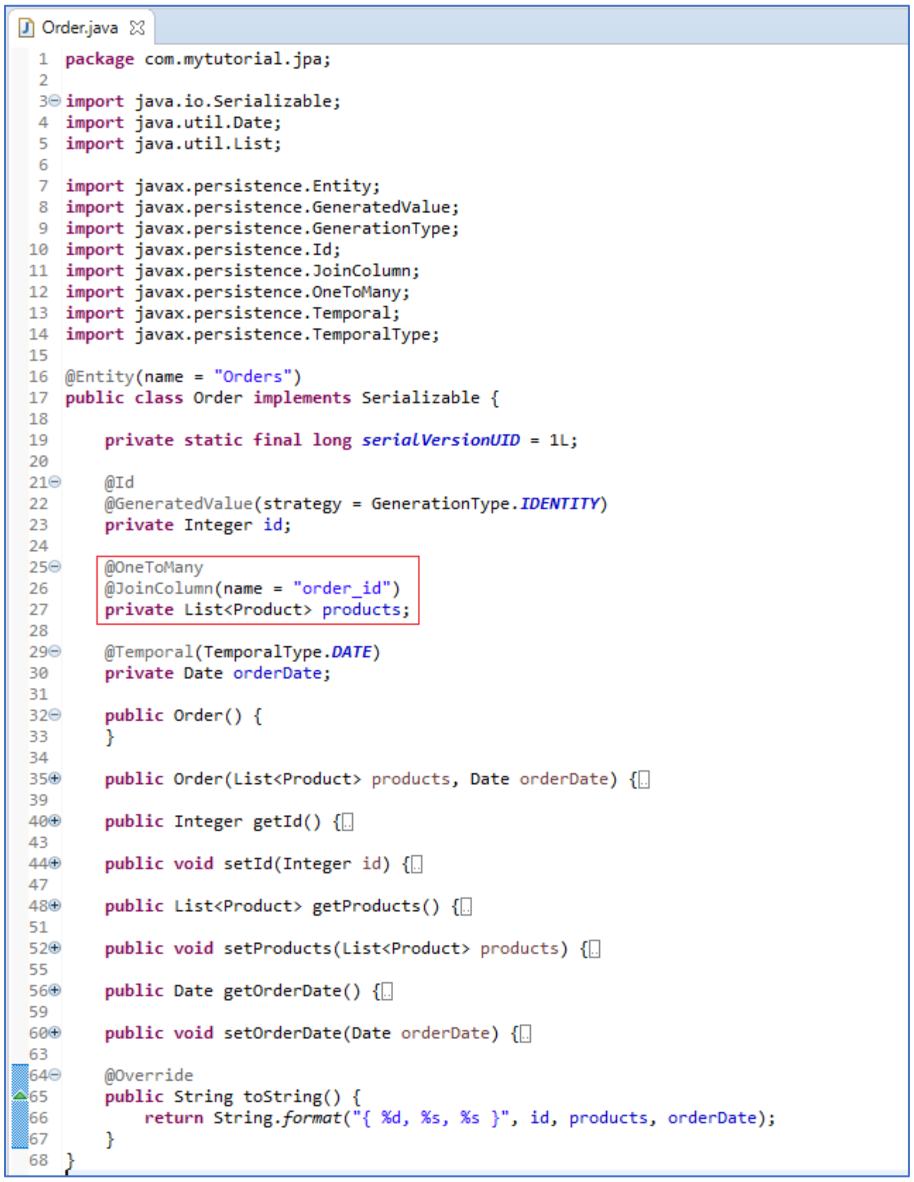
We've seen that by default when we set up a one-to-many mapping relationship between entities. This relationship is represented using a join table, a *separate table* that contains the entries representing the one-to-many mapping. Now it's also possible for you to specify how exactly you want to model this relationship using the underlying database tables. You can create a **foreign key constraint to represent the relationship** rather than use a joint table. And that's what we'll see here in this demo.

Here in **persistence.xml**, I'm going to go back to the drop and create database action to recreate my database tables each time I run my code.



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

I'll now switch over to **Order.java**, which is my owning entity, and which has the one-to-many mapping from order to products.

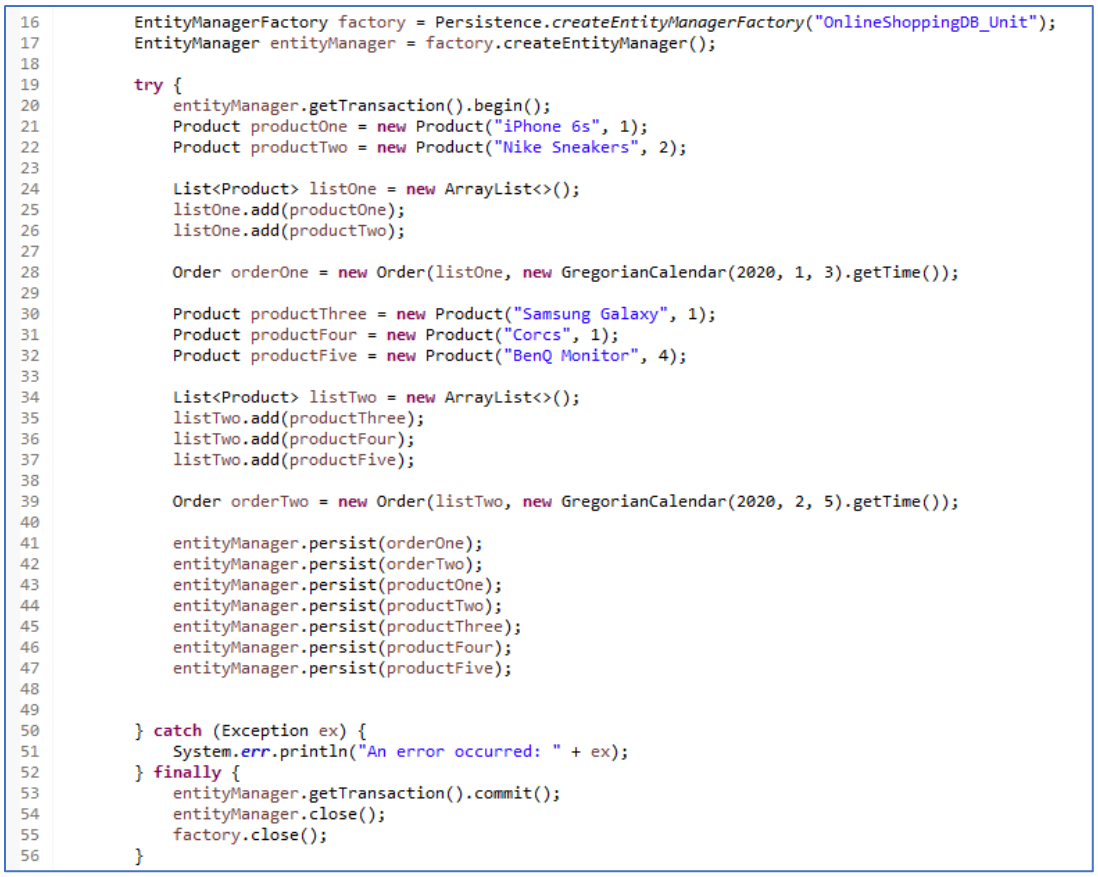


```java
package com.mytutorial.jpa;

import java.io.Serializable;
import java.util.Date;
import java.util.List;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToMany;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity(name = "Orders")
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;

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

    @OneToMany
    @JoinColumn(name = "order_id")
    private List<Product> products;

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    public Order() {
    }

    public Order(List<Product> products, Date orderDate) {

    public Integer getId() {

    public void setId(Integer id) {

    public List<Product> getProducts() {

    public void setProducts(List<Product> products) {

    public Date getOrderDate() {

    public void setOrderDate(Date orderDate) {

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

- I'm going to set up an `import` for the `javax.persistence.JoinColumn` here, which is what I'll use to set up my foreign key constraint.
- So rather than have just the **@OneToMany** annotation, I'm going to specify an additional **@JoinColumn** annotation.
- This **@JoinColumn** annotation takes the *name* property. The name I've specified is `"order_id"`. And this *name* property specifies the column in the `Products` table that has a **foreign key reference** to the `Order` entity that every `Product` is associated with.

Remember, one order entity is associated with many product entities that is the relationship mapping. This means we **cannot have a foreign key reference** from `Order` to `Products` because we can have just one entry for an `Order` in the `Orders` table. But we **can have a foreign key mapping** from `Products` into the `Orders` table, and that's what we've specified here using this **@JoinColumn**.

And I'll now head over to **App.java**. I'll update the code here in App.java to create and persist `Order` and `Product` entities.

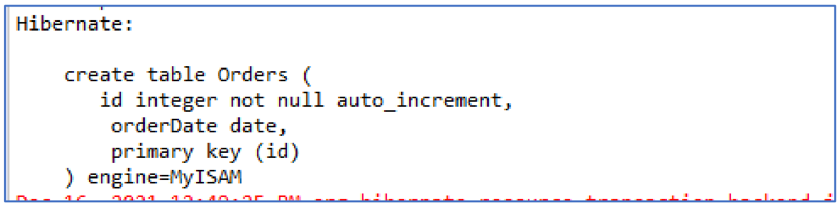


```java
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {
            entityManager.getTransaction().begin();
            Product productOne = new Product("iPhone 6s", 1);
            Product productTwo = new Product("Nike Sneakers", 2);

            List<Product> listOne = new ArrayList<>();
            listOne.add(productOne);
            listOne.add(productTwo);

            Order orderOne = new Order(listOne, new GregorianCalendar(2020, 1, 3).getTime());

            Product productThree = new Product("Samsung Galaxy", 1);
            Product productFour = new Product("Corcs", 1);
            Product productFive = new Product("BenQ Monitor", 4);

            List<Product> listTwo = new ArrayList<>();
            listTwo.add(productThree);
            listTwo.add(productFour);
            listTwo.add(productFive);

            Order orderTwo = new Order(listTwo, new GregorianCalendar(2020, 2, 5).getTime());

            entityManager.persist(orderOne);
            entityManager.persist(orderTwo);
            entityManager.persist(productOne);
            entityManager.persist(productTwo);
            entityManager.persist(productThree);
            entityManager.persist(productFour);
            entityManager.persist(productFive);


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

- I have two `Order`, `orderOne` and `orderTwo`.
- `productOne` and `productTwo` are associated with `orderOne`.
- `productThree`, `productFour`, `productFive` are associated with `orderTwo`.

`Order` is still the owning entity here, which is the entity which has the OneToMany annotation. This is still a unidirectional relationship. So make sure you set up the right references to the `Product` associated with each order on the `Order` entity.

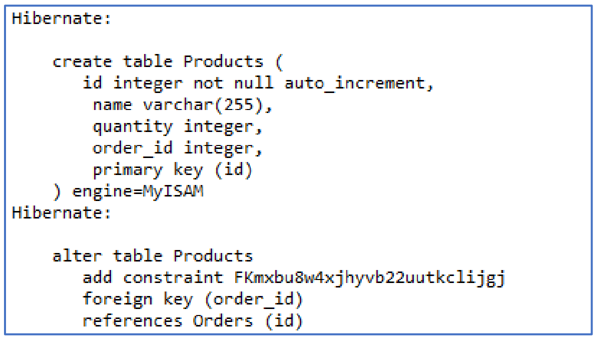
Let's run this code and see how this OneToMany relationship is modeled in our database table.



```
Hibernate: 

    create table Orders (
       id integer not null auto_increment,
        orderDate date,
        primary key (id)
    ) engine=MyISAM
```

Let's take a look at the `Orders` table here, no change. We have the `id` column, the `orderDate` column, the `id` is the **primary key**.

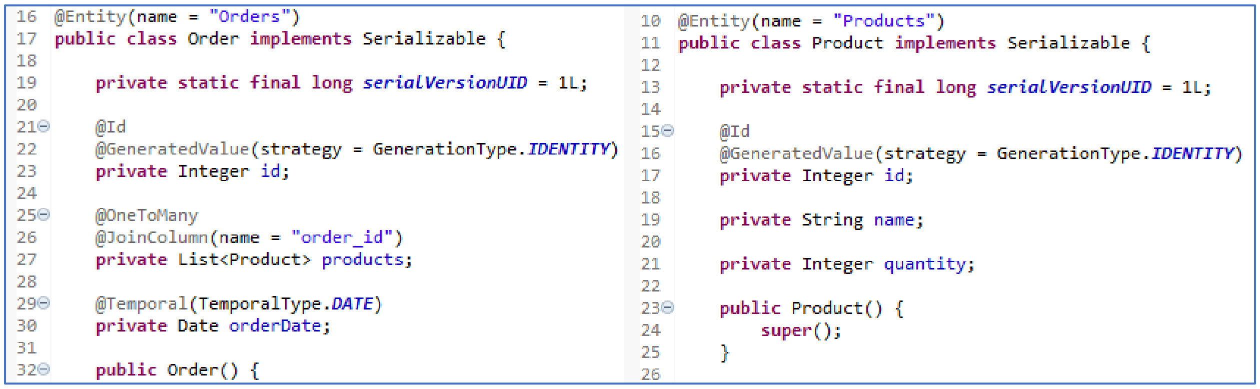
The real change is in the `Products` table. Observe how the products table has been created.



```
Hibernate:

    create table Products (
       id integer not null auto_increment,
        name varchar(255),
        quantity integer,
        order_id integer,
        primary key (id)
    ) engine=MyISAM
Hibernate:

    alter table Products
       add constraint FKmxbu8w4xjhyvb22uutkclijgj
       foreign key (order_id)
       references Orders (id)
```

- We have the `Product` `id`, `name`, and `quantity`.
- We have an additional column here called `order_id`.

  This is the name property that we had specified in the **`@JoinColumn`** attribute, which we have specified in *Order.java*.



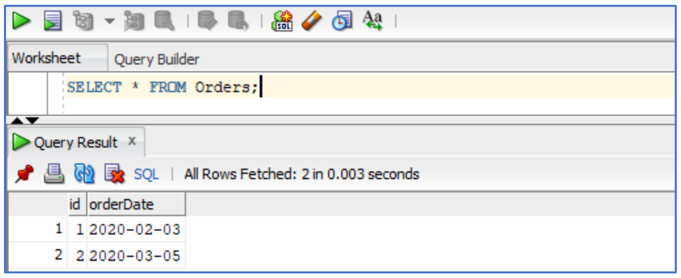
```java
@Entity(name = "Orders")
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;

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

    @OneToMany
    @JoinColumn(name = "order_id")
    private List<Product> products;

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    public Order() {
```

```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 Integer quantity;

    public Product() {
        super();
    }

```

  We know that every `Product` is associated with an `order_id`. Multiple `products` may be associated with the same `order_id`, and this is modeled as a **foreign key**.

- Observe that the `Products` table references the `id` column in the orders table using the foreign key `order_id`.

When we specify the join column, no additional join table is created to hold this mapping. We'll take a look at the actual entries in the database table.

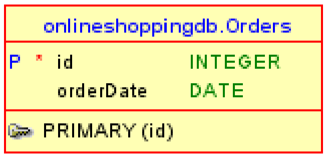


```
SELECT * FROM Orders;
```

Query Result

All Rows Fetched: 2 in 0.003 seconds

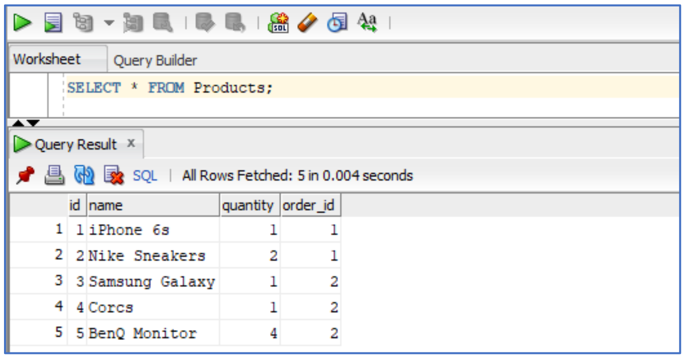
	id	orderDate
1	1	2020-02-03
2	2	2020-03-05



```
onlineshoppingdb.Orders
P * id          INTEGER
    orderDate   DATE
PRIMARY (id)
```

Here are the contents of the `Orders` table, every `Order` is represented.

And let's take a look at the contents of the `Products` table, which is what is interesting.

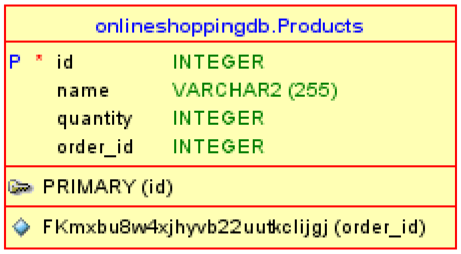


```
SELECT * FROM Products;
```

Query Result

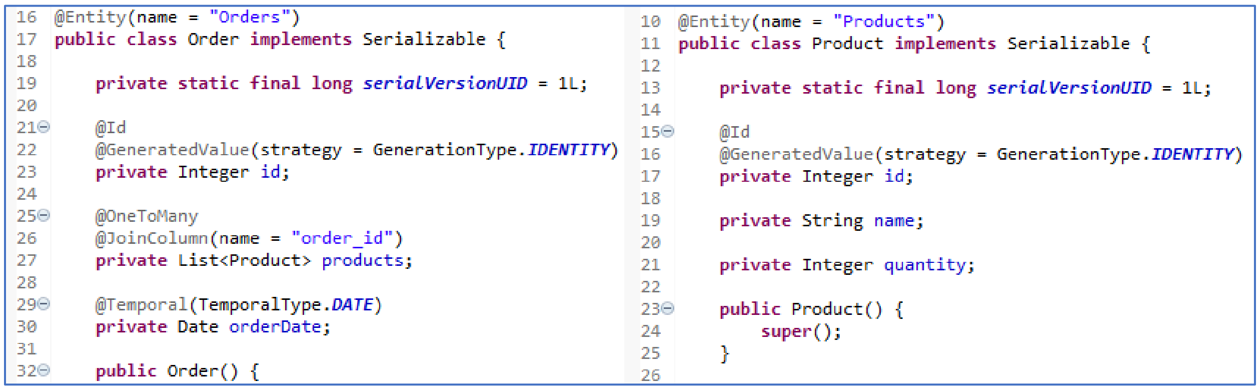
All Rows Fetched: 5 in 0.004 seconds

	id	name	quantity	order_id
1	1	iPhone 6s	1	1
2	2	Nike Sneakers	2	1
3	3	Samsung Galaxy	1	2
4	4	Corcs	1	2
5	5	BenQ Monitor	4	2



```
onlineshoppingdb.Products
P * id          INTEGER
    name        VARCHAR2 (255)
    quantity    INTEGER
    order_id    INTEGER
PRIMARY (id)
FKmxbu8w4xjhyvb22uutkclijgj (order_id)
```

Observe that we have five `Products` here, and each `Product` is associated with an `order_id`. The first two `Products` have `order_id` 1. The remaining three `Products` have `order_id` 2.

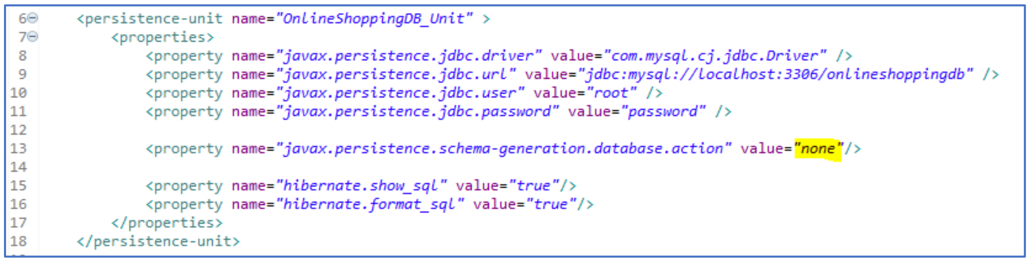


```java
@Entity(name = "Orders")
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;

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

    @OneToMany
    @JoinColumn(name = "order_id")
    private List<Product> products;

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    public Order() {
```

```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 Integer quantity;

    public Product() {
        super();
    }

```

Retrieve Many Entities in Order

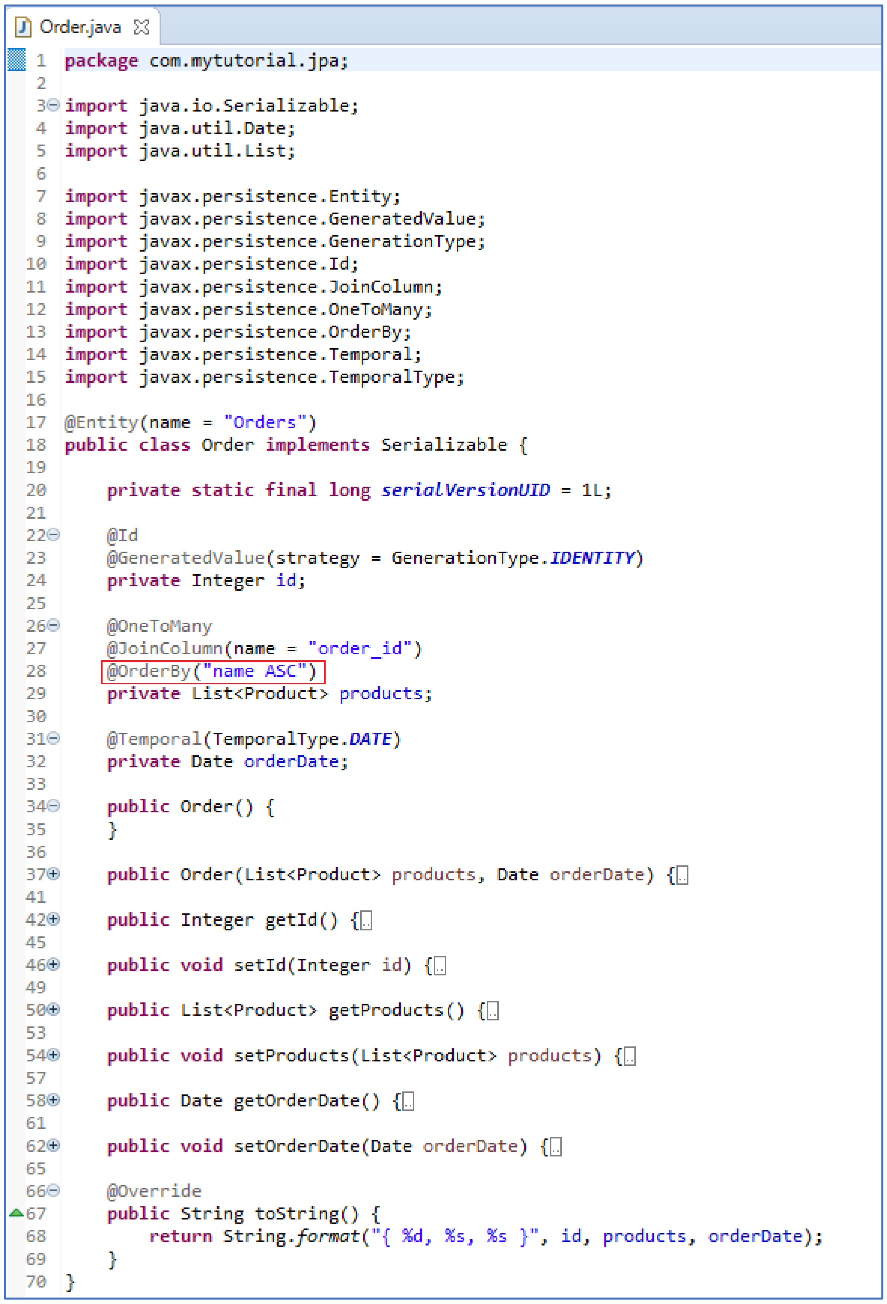
When you have a one-to-many mapping. On the many side of the relationship, you may want to return the related entities in some kind of order. It's possible to do this without writing queries. Let's see how in this demo. Here I am in **persistence.xml**. I'm going to change this drop-and-create database action and set it to none. We'll work with the database that we've previously created.



```
    <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="none"/>

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

This is where order two products had a one-to-many relationship set up using a foreign key constraint.

In this demo, we'll use and understand the **@OrderBy** annotation available in JPA.



Order.java

```java
package com.mytutorial.jpa;

import java.io.Serializable;
import java.util.Date;
import java.util.List;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToMany;
import javax.persistence.OrderBy;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity(name = "Orders")
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;

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

    @OneToMany
    @JoinColumn(name = "order_id")
    @OrderBy("name ASC")
    private List<Product> products;

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    public Order() {
    }

    public Order(List<Product> products, Date orderDate) {

    public Integer getId() {

    public void setId(Integer id) {

    public List<Product> getProducts() {

    public void setProducts(List<Product> products) {

    public Date getOrderDate() {

    public void setOrderDate(Date orderDate) {

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

- `import javax.persistence.OrderBy` and apply this **@OrderBy** annotation to the list of `products` that we'll retrieve along with each `Order`.

  This **@OrderBy** annotation allows you to specify the order in which the `products` will be retrieved.

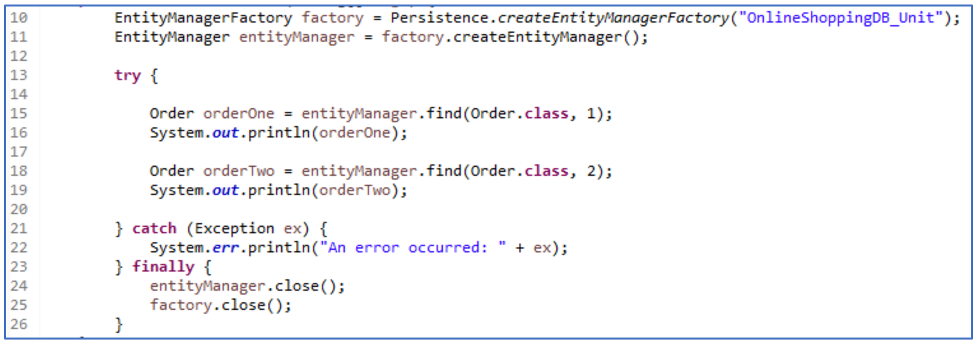
  Now the way the `products` will be saved in the database, that's irrelevant. **OrderBy** has to do with how the `products` are retrieved and presented to you within code.

- Here I've indicated that when the list of `products` associated with each order is retrieved. I want the products to be in the **ascending** order of the `name` of the `products`, `@OrderBy("name ASC")`, **ASC** for ascending.

  If you want to specify the descending order, you will use **DESC**. You can order by any attribute of the `Product`. Specify the name of the member variable in the related entity which you want to use to order the `products`.

  If you don't specify anything, the default ordering is by the primary key of the related entity. If you don't explicitly specify a sort order ascending or descending, the default is ascending.

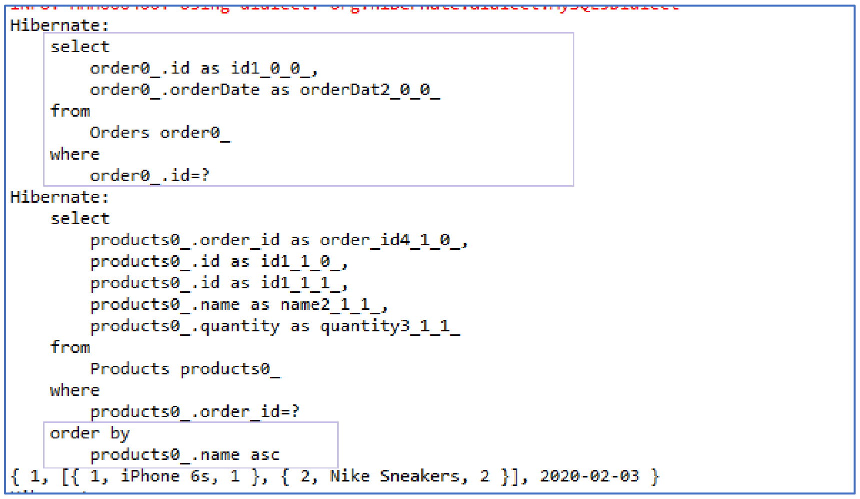
Now that we've made this change, all we have to do is switch over to **App.java** and retrieve the order entity. And see the order in which the products are retrieved.



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

        try {

            Order orderOne = entityManager.find(Order.class, 1);
            System.out.println(orderOne);

            Order orderTwo = entityManager.find(Order.class, 2);
            System.out.println(orderTwo);

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

- Here I have two find method invocations for the `Order` with `ID` 1 and 2.
- And I've printed out the `Orders` to screen once I've retrieved the `Order` information.

Let's run this code and let's take a look at our select queries.

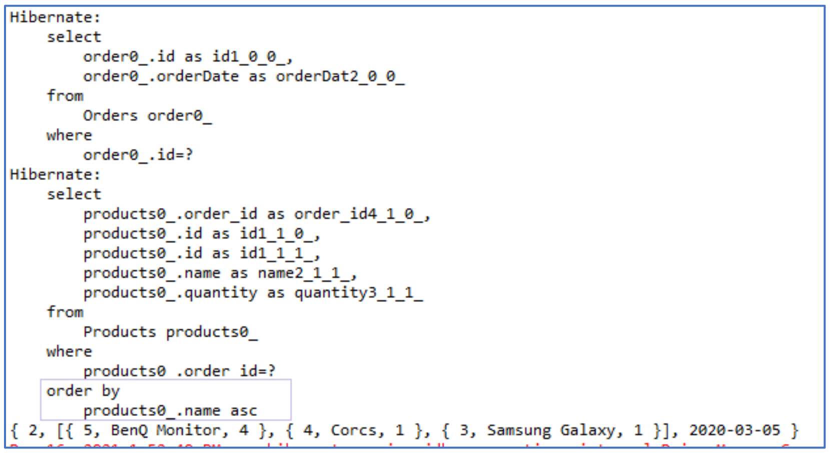


```
Hibernate:
    select
        order0_.id as id1_0_0_,
        order0_.orderDate as orderDat2_0_0_
    from
        Orders order0_
    where
        order0_.id=?
Hibernate:
    select
        products0_.order_id as order_id4_1_0_,
        products0_.id as id1_1_0_,
        products0_.id as id1_1_1_,
        products0_.name as name2_1_1_,
        products0_.quantity as quantity3_1_1_
    from
        Products products0_
    where
        products0_.order_id=?
    order by
        products0_.name asc
{ 1, [{ 1, iPhone 6s, 1 }, { 2, Nike Sneakers, 2 }], 2020-02-03 }
```

- You can see the order information is retrieved first, then the product information, that is the default, is lazy loading.
- Because we've specified that we want the products in a certain order. Notice that the Hibernate query specifies an **order by** clause for the **select statement** for `Products`.

- Observe that when we retrieve the `Products`, the `Products` are ordered on the name column.
- And if you scroll down further, you'll see that the products have been retrieved in the order of their name. The “iPhone 6s” first and then “Nike Sneakers”.

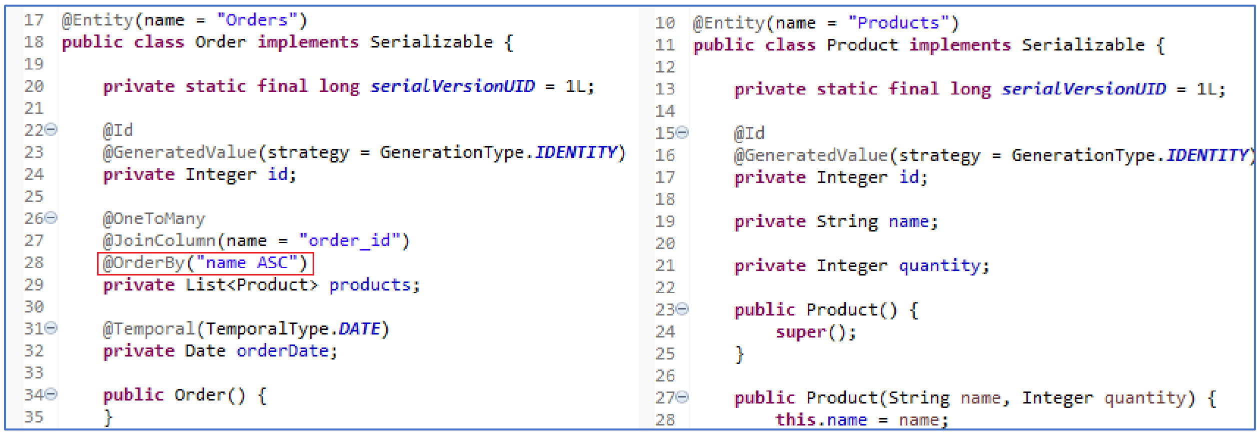
Now in order to further confirm that the `Products` are retrieved in the `Order` of their names. Let's scroll down further and take a look at the second select statement for the products associated with the second order, `orderTwo`.



```
Hibernate: 
    select
        order0_.id as id1_0_0_,
        order0_.orderDate as orderDat2_0_0_ 
    from
        Orders order0_ 
    where
        order0_.id=?
Hibernate: 
    select
        products0_.order_id as order_id4_1_0_,
        products0_.id as id1_1_0_,
        products0_.id as id1_1_1_,
        products0_.name as name2_1_1_,
        products0_.quantity as quantity3_1_1_ 
    from
        Products products0_ 
    where
        products0_.order_id=? 
    order by
        products0_.name asc
{ 2, [{ 5, BenQ Monitor, 4 }, { 4, Corcs, 1 }, { 3, Samsung Galaxy, 1 }], 2020-03-05 }
```

- Once again we see that there is an **order by** clause in the SQL **select statement** while retrieving `Orders`.
- There is an **order by** clause here which sorts the `Products` by `name` at the time of retrieval.
- And if you scroll further down, you'll see that the `Products` associated with `orderTwo` are retrieved based on the sort order of their names. “BenQ Monitors” first, then “Crocs” and then the “Samsung Galaxy”.

This is not the order in which these `products` were persisted in the underlying database table. You can also see that this is not the primary key order. The primary key for BenQ Monitors is 5. For Crocs, it's 4, and for the Samsung Galaxy, it's 3. The `products` have been retrieved sorted by the `name`’s of the `products`.



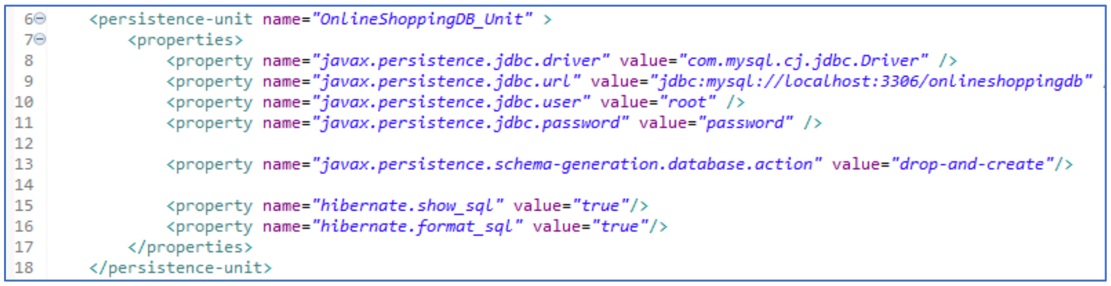
```java
@Entity(name = "Orders")
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;

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

    @OneToMany
    @JoinColumn(name = "order_id")
    @OrderBy("name ASC")
    private List<Product> products;

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    public Order() {
    }
```

```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 Integer quantity;

    public Product() {
        super();
    }

    public Product(String name, Integer quantity) {
        this.name = name;
```

Persist Many Entities in Order

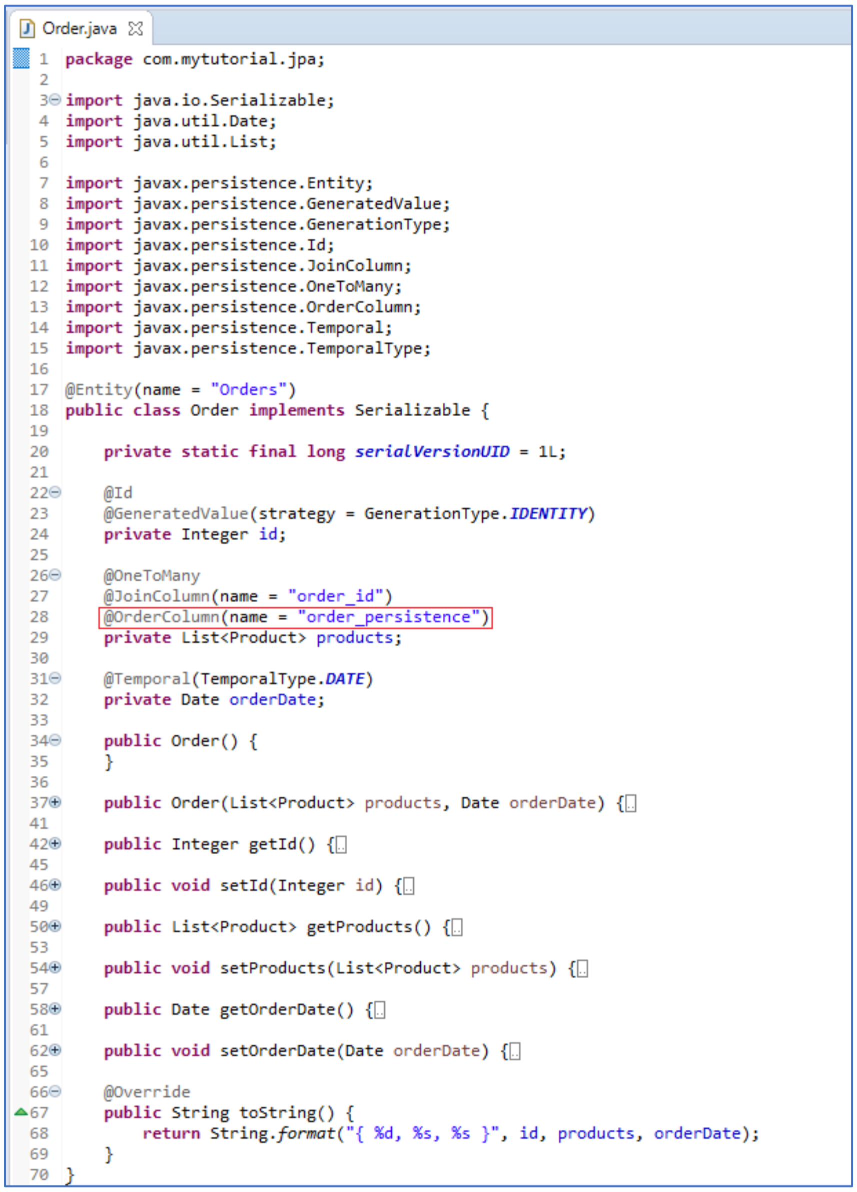
In the previous demo, we used the **@OrderBy** annotation to retrieve the list of products associated with an order in a particular sort order. The ascending order of name. In this demo, we are going to persist the products associated with an order in a particular sort order in the database table. So we'll change the order of persistence. We won't worry about the order in which products are retrieved.

Here we are in the **persistence.xml** file. I'm going to change the database action from none to drop-and-create so that my database tables are recreated when we run our application.



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

We can configure the order in which the `products` associated with an `Order` are persisted. By heading over to the owning entity in this unidirectional relationship. This is the `Order` entity.



Order.java

```java
package com.mytutorial.jpa;

import java.io.Serializable;
import java.util.Date;
import java.util.List;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToMany;
import javax.persistence.OrderColumn;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity(name = "Orders")
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;

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

    @OneToMany
    @JoinColumn(name = "order_id")
    @OrderColumn(name = "order_persistence")
    private List<Product> products;

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    public Order() {
    }

    public Order(List<Product> products, Date orderDate) {

    public Integer getId() {

    public void setId(Integer id) {

    public List<Product> getProducts() {

    public void setProducts(List<Product> products) {

    public Date getOrderDate() {

    public void setOrderDate(Date orderDate) {

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

- Setup the import statements for the OrderColumn annotation
  `import javax.persistence.OrderColumn;`
- and let's update the annotation that we apply to `products`. Instead of using **@OrderBy**, I'm going to use the **@OrderColumn**. In this **@OrderColumn** annotation I have specified the name property, and this name property is set to `"order_persistence"`.
  This property specifies the name of the column in the `Products` table. Which will hold the order in which the `products` associated with an order have been persisted.

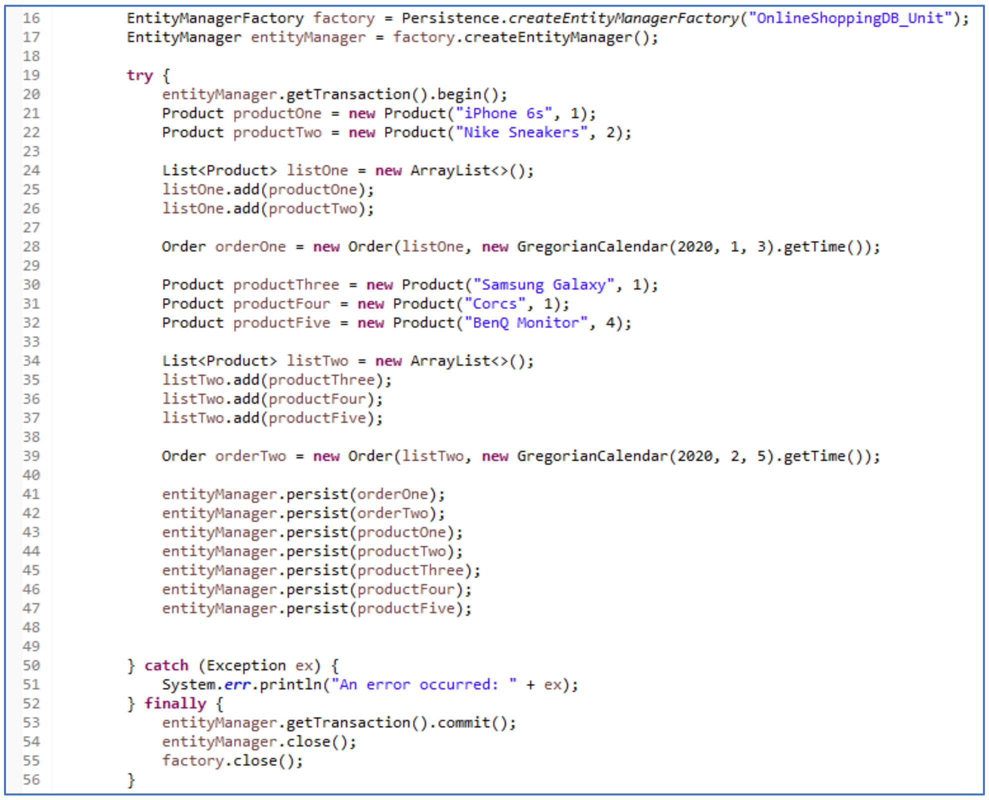
When you use the **@OrderColumn** annotation, it's the responsibility of the persistence provider, which in our case happens to be Hibernate.

To retrieve the products in the right persistence order. Hibernate is also responsible for maintaining updated persistence information. So if you update a `Product` or if you delete a `Product` associated with an `Order`, it's Hibernate's responsibility to maintain this persistence order.

When you specify the **@OrderColumn** annotation, you do not use the **@OrderBy** annotation as well. Because **@OrderBy** explicitly is used for how you want to retrieve your products. With the **@OrderColumn**, the retrieval is based on the order in which you persisted products.

Now it can be hard to wrap your head around how exactly @OrderColumn works.

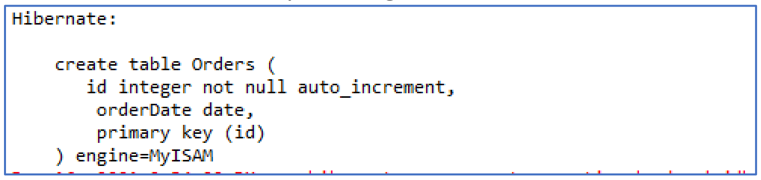
So let's see the example. Let's switch over to **App.java** and here I'm going to setup two orders and products associated with each order.



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

        try {
            entityManager.getTransaction().begin();
            Product productOne = new Product("iPhone 6s", 1);
            Product productTwo = new Product("Nike Sneakers", 2);

            List<Product> listOne = new ArrayList<>();
            listOne.add(productOne);
            listOne.add(productTwo);

            Order orderOne = new Order(listOne, new GregorianCalendar(2020, 1, 3).getTime());

            Product productThree = new Product("Samsung Galaxy", 1);
            Product productFour = new Product("Corcs", 1);
            Product productFive = new Product("BenQ Monitor", 4);

            List<Product> listTwo = new ArrayList<>();
            listTwo.add(productThree);
            listTwo.add(productFour);
            listTwo.add(productFive);

            Order orderTwo = new Order(listTwo, new GregorianCalendar(2020, 2, 5).getTime());

            entityManager.persist(orderOne);
            entityManager.persist(orderTwo);
            entityManager.persist(productOne);
            entityManager.persist(productTwo);
            entityManager.persist(productThree);
            entityManager.persist(productFour);
            entityManager.persist(productFive);


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

- Observe `orderOne` which takes in `listOne`. In `listOne` I've specified the “iPhone” first and then “Nike Sneaker”.
- So that is the order in which we've added products to the first order. Next for `orderTwo`, we've added `Products` in the following order, “Samsung” first, then “Crocs” and then the “BenQ Monitors”. This is the sort order in which `Products` will be persisted for each order and this will be tracked in the `order_persistence` column in the `Products` table.

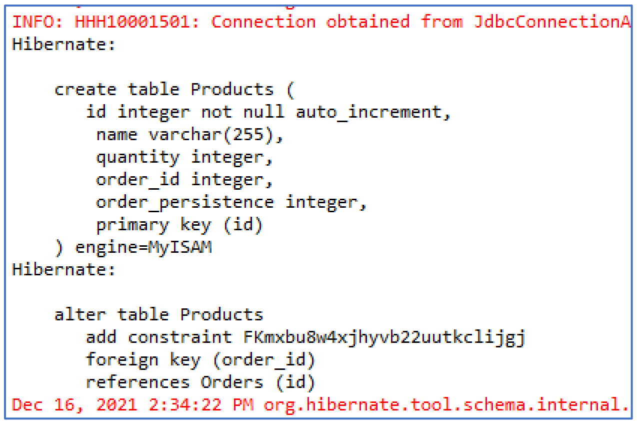
Let's see this in action by running our code.



```
Hibernate:

    create table Orders (
       id integer not null auto_increment,
        orderDate date,
        primary key (id)
    ) engine=MyISAM
```

Here within the console window you can see the `Orders` table has been created first, we have the order `id` and the `orderDate`.

If we scroll down further you'll see that we have the statement to create the Products table.

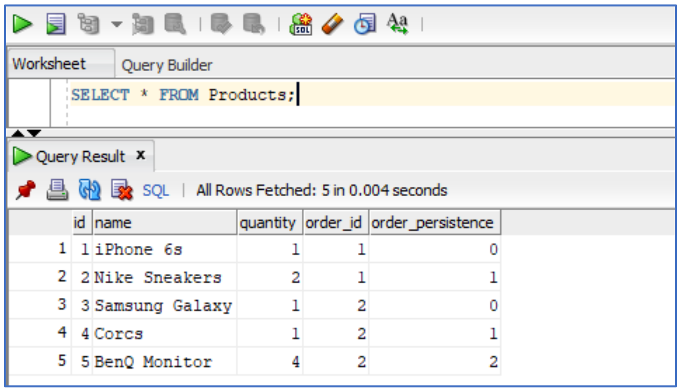


```
INFO: HHH10001501: Connection obtained from JdbcConnectionA
Hibernate:

    create table Products (
       id integer not null auto_increment,
        name varchar(255),
        quantity integer,
        order_id integer,
        order_persistence integer,
        primary key (id)
    ) engine=MyISAM
Hibernate:

    alter table Products
       add constraint FKmxbu8w4xjhyvb22uutkclijgj
       foreign key (order_id)
       references Orders (id)
Dec 16, 2021 2:34:22 PM org.hibernate.tool.schema.internal.
```

Observe that in addition to all of the attributes of individual Products, we have the `order_persistence` column which is of type **integer**.

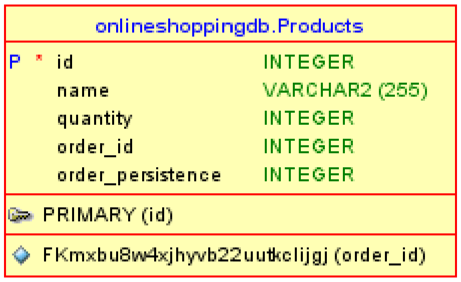
This is the column that'll track the order in which the Products associated with each order are persisted to the underlying table.

Let's now see how the information the `order_persistence` column is stored. I'm doing a *select ** on the *Products* table.



```
SELECT * FROM Products;
```

	id	name	quantity	order_id	order_persistence
1	1	iPhone 6s	1	1	0
2	2	Nike Sneakers	2	1	1
3	3	Samsung Galaxy	1	2	0
4	4	Crocs	1	2	1
5	5	BenQ Monitor	4	2	2

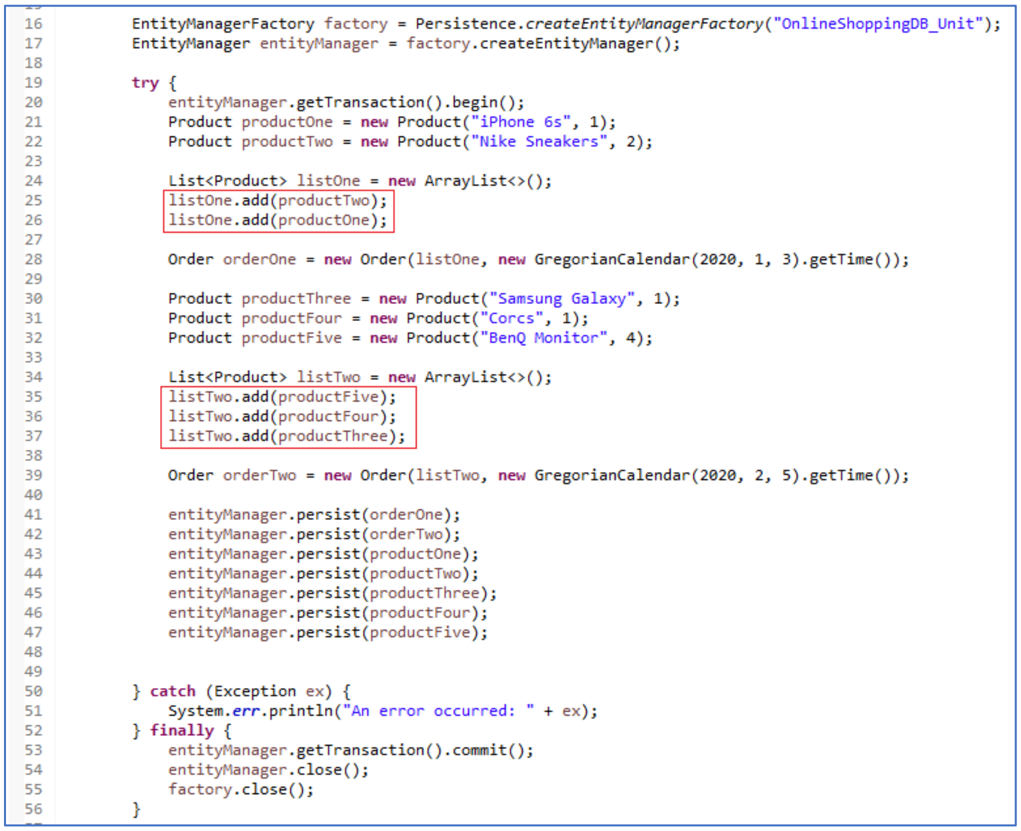


onlineshoppingdb.Products		
P *	id	INTEGER
	name	VARCHAR2 (255)
	quantity	INTEGER
	order_id	INTEGER
	order_persistence	INTEGER
	PRIMARY (id)	
	FKmxbu8w4xjhyvb22uutkclijgj (order_id)	

This is a table which has the `order_persistence` column.

- Notice that for `order_id` 1, the "iPhone" was persisted first and then "Nike Sneakers".
- For `order_id` 2, the "Samsung Galaxy" was persisted first, then "Crocs" and then "BenQ Monitors".

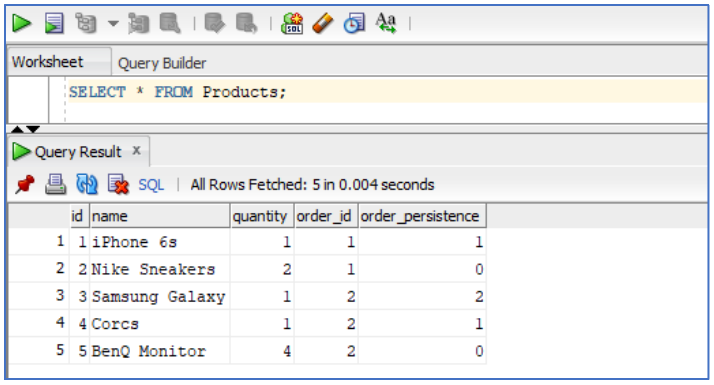
Here I am back within my Java application and this time I'm going to change the order in which I've added products to a particular order.



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

        try {
            entityManager.getTransaction().begin();
            Product productOne = new Product("iPhone 6s", 1);
            Product productTwo = new Product("Nike Sneakers", 2);

            List<Product> listOne = new ArrayList<>();
            listOne.add(productTwo);
            listOne.add(productOne);

            Order orderOne = new Order(listOne, new GregorianCalendar(2020, 1, 3).getTime());

            Product productThree = new Product("Samsung Galaxy", 1);
            Product productFour = new Product("Corcs", 1);
            Product productFive = new Product("BenQ Monitor", 4);

            List<Product> listTwo = new ArrayList<>();
            listTwo.add(productFive);
            listTwo.add(productFour);
            listTwo.add(productThree);

            Order orderTwo = new Order(listTwo, new GregorianCalendar(2020, 2, 5).getTime());

            entityManager.persist(orderOne);
            entityManager.persist(orderTwo);
            entityManager.persist(productOne);
            entityManager.persist(productTwo);
            entityManager.persist(productThree);
            entityManager.persist(productFour);
            entityManager.persist(productFive);


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

- So instead of adding the “iPhone” and then “Nike Sneakers”. I'm going to switch the order around so that I add the product “Nike” first and then product “iPhone”.
- I'll do the same thing for the second order here. I've changed the order in which I've added products, I've added the “BenQ Monitor” first, then “Crocs” and then the product “Samsung”.

Let's see how the values in the order_persistence column change. Go ahead and run this code. I'll now switch over to the MySQL Workbench and run a *select* * on the *Products* table.



```
SELECT * FROM Products;
```

All Rows Fetched: 5 in 0.004 seconds

	id	name	quantity	order_id	order_persistence
1	1	iPhone 6s	1	1	1
2	2	Nike Sneakers	2	1	0
3	3	Samsung Galaxy	1	2	2
4	4	Corcs	1	2	1
5	5	BenQ Monitor	4	2	0

Let's take a look at the `order_persistence` column here.

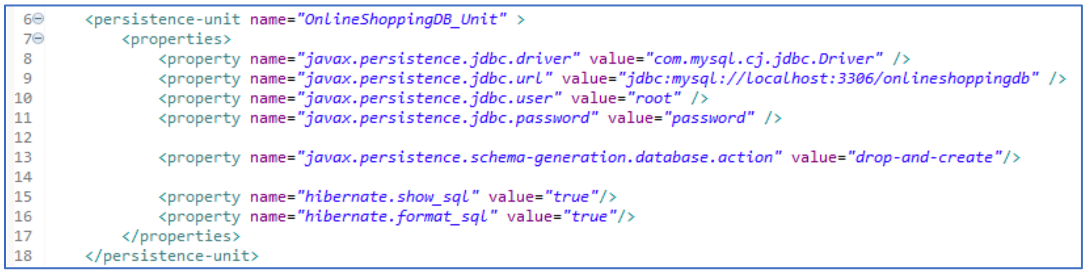
Notice that for `order_id` 1, the `order_persistence` entries are now different. “Nike Sneakers” was persisted first, that has a value 0. And then the “iPhone 6S” was persisted, that has a value 1.

The same thing is true for the products associated with `order_id` 2. “BenQ Monitors“ has a value of 0, then “Crocs” and then the “Samsung Galaxy”.

Many-to-one Unidirectional Mapping

It's possible that the real world relationship that you want to model using database tables is a many-to-one relationship. Now this is exactly the same as a one-to-many relationship, except that we're specifying the many side first.

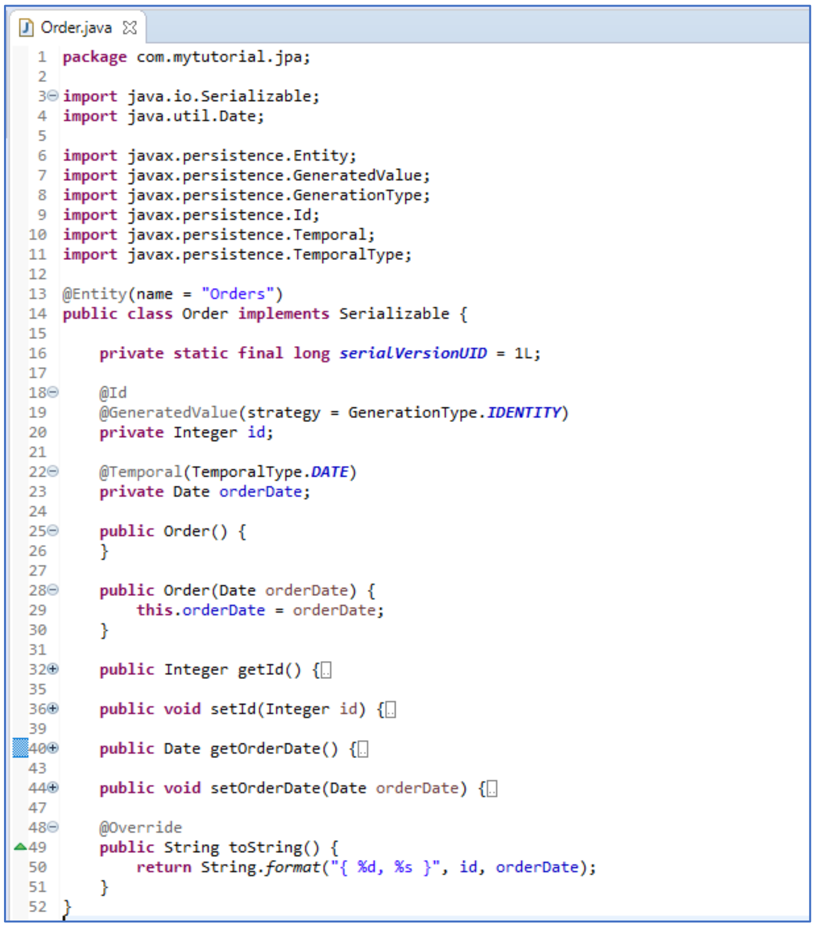
Now let's see how we can set up a many-to-one relationship using JPA and Hibernate. Here is a **persistence.xml**, the database action is set to drop-and-create.



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

We'll continue working with the same example that we used earlier in the one-to-many relationship mapping. This is a many-to-one mapping.

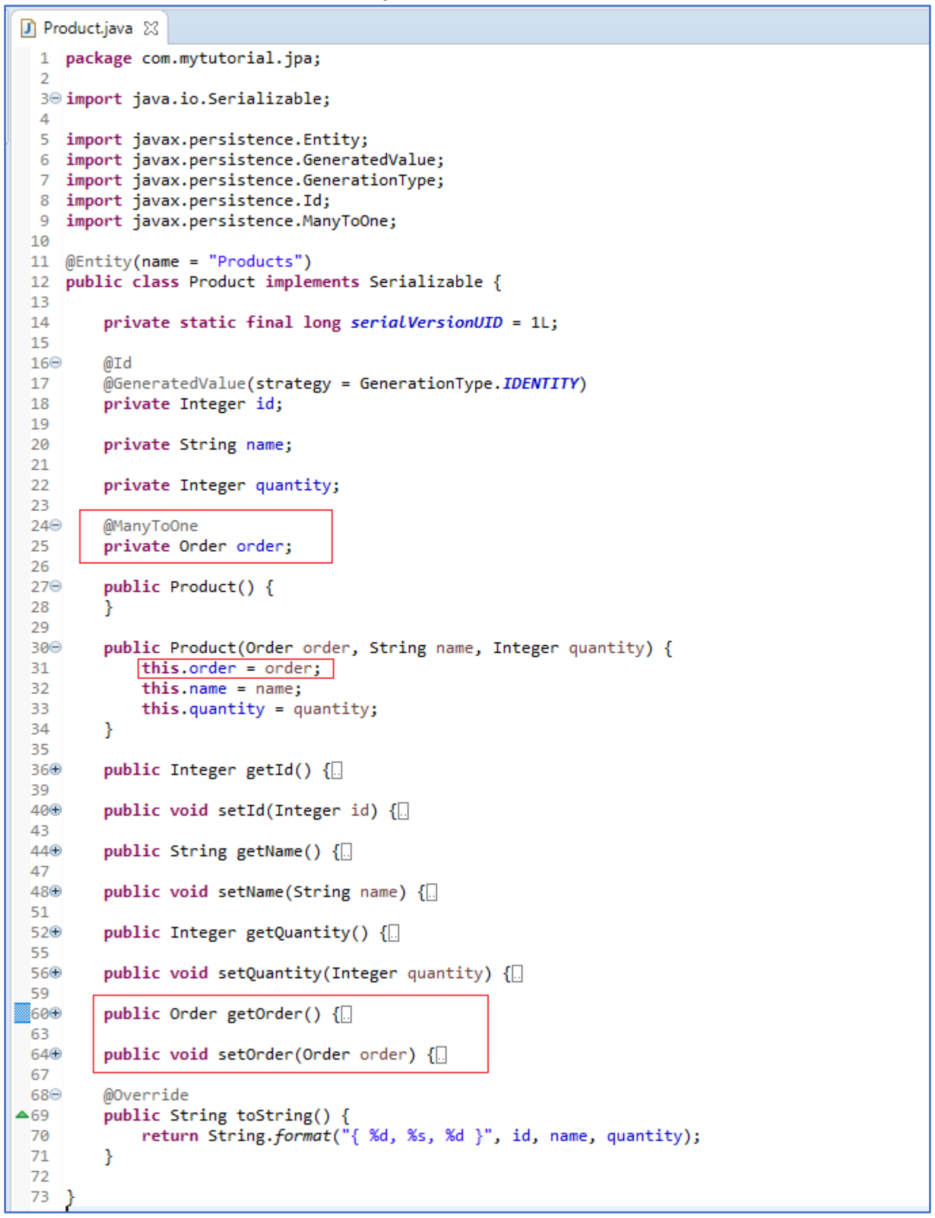
Here is our **Order.java** file. The file is more or less the same but there's an important difference. Observe the member variables of this order entity which maps to database columns.



```
 1  package com.mytutorial.jpa;
 2
 3  import java.io.Serializable;
 4  import java.util.Date;
 5
 6  import javax.persistence.Entity;
 7  import javax.persistence.GeneratedValue;
 8  import javax.persistence.GenerationType;
 9  import javax.persistence.Id;
10  import javax.persistence.Temporal;
11  import javax.persistence.TemporalType;
12
13  @Entity(name = "Orders")
14  public class Order implements Serializable {
15
16      private static final long serialVersionUID = 1L;
17
18      @Id
19      @GeneratedValue(strategy = GenerationType.IDENTITY)
20      private Integer id;
21
22      @Temporal(TemporalType.DATE)
23      private Date orderDate;
24
25      public Order() {
26      }
27
28      public Order(Date orderDate) {
29          this.orderDate = orderDate;
30      }
31
32      public Integer getId() {
35
36      public void setId(Integer id) {
39
40      public Date getOrderDate() {
43
44      public void setOrderDate(Date orderDate) {
47
48      @Override
49      public String toString() {
50          return String.format("{ %d, %s }", id, orderDate);
51      }
52  }
```

- We have the `id` and the `orderDate`, but we do not have a reference to the `products` that are associated with an order.
  So the mapping is not from `Order` to `Products`, instead it's in the reverse direction, from `Products` to `Order`.
- Many products make up an order that is a many-to-one mapping. If you look at our order constructor on line 28, observe that it only takes in an `orderDate`, it *does not* take in a reference to the `products` associated with this `Order`.
- If you look at the **getters** and **setters**, we have getters and setters only for the member variables that we'll find.

Let's now head over to **Product.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.ManyToOne;

@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 Integer quantity;

    @ManyToOne
    private Order order;

    public Product() {
    }

    public Product(Order order, String name, Integer quantity) {
        this.order = order;
        this.name = name;
        this.quantity = quantity;
    }

    public Integer getId() {

    public void setId(Integer id) {

    public String getName() {

    public void setName(String name) {

    public Integer getQuantity() {

    public void setQuantity(Integer quantity) {

    public Order getOrder() {

    public void setOrder(Order order) {

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

}
```

- This is the entity which represents a product. Many products are associated with an order which means we'll set up a many-to-one mapping here.
- First, set up the import statements for all of the annotations that you'll use.
- Next, we'll set up the member variables of the product class. `id` is the primary key. We have the `name` and `quantity` of the product, and then we have a reference to the `Order` associated with each product. This is the member variable called `order`.
- And to this, we have applied the at **@ManyToOne** mapping, indicating that there are many product entities that can be associated with a single order entity.

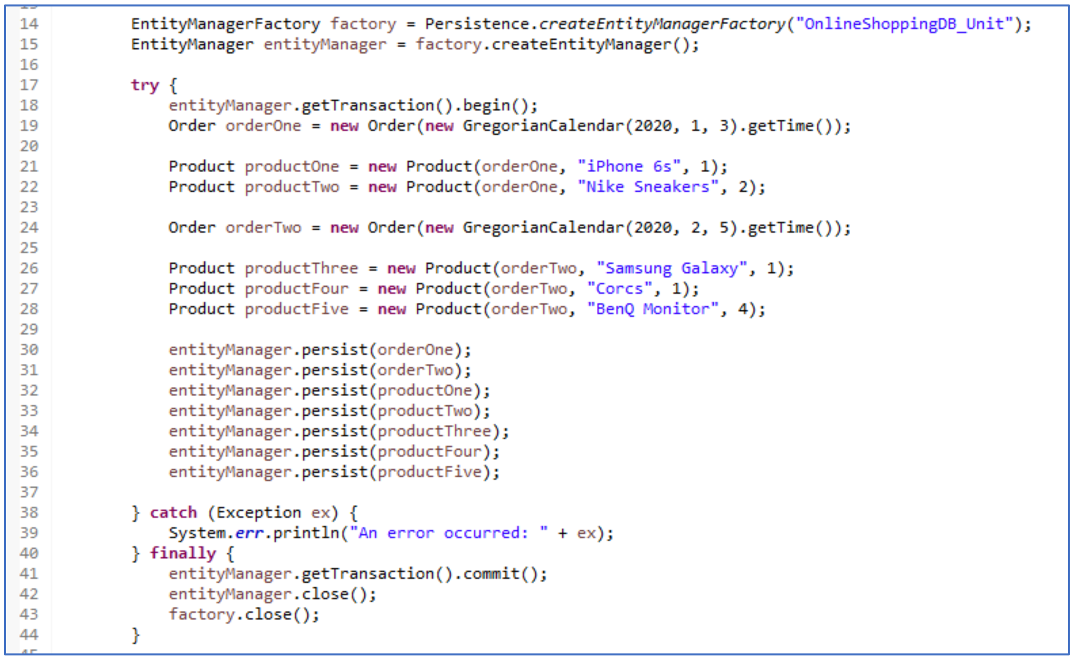
  We haven't specified any additional configuration for this many-to-one mapping. We'll use the default specifications that JPA provides.

At this point, we only have a reference from the `Product` entity to the `Order` entity which is a **@ManyToOne** mapping.

The `Product` entity is the owning entity in this relationship, which means we have to have a reference to the `Order` from the `Product` entity when we persist entities in our database.

- If you look at the `Product` **constructor** on line 30, observe that we pass in the `order` with which this `Product` is associated in the **constructor**.
- The remaining code here in this product class should be familiar. We have **getters** and **setters** for the `id` and the `name` variables as well as the `quantity` variables.
- And we have a **getter** and **setter** for the `order` associated with the `Product` as well.

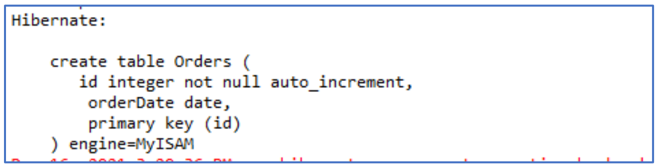
Now it's time for us to go to **App.java** where we'll create `Order` and `Product` entities that we want persisted within our `Orders` and `Products` table. Now, the way we persist entities remains the same, but we set up the mapping differently.



```java
14          EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
15          EntityManager entityManager = factory.createEntityManager();
16  
17          try {
18              entityManager.getTransaction().begin();
19              Order orderOne = new Order(new GregorianCalendar(2020, 1, 3).getTime());
20  
21              Product productOne = new Product(orderOne, "iPhone 6s", 1);
22              Product productTwo = new Product(orderOne, "Nike Sneakers", 2);
23  
24              Order orderTwo = new Order(new GregorianCalendar(2020, 2, 5).getTime());
25  
26              Product productThree = new Product(orderTwo, "Samsung Galaxy", 1);
27              Product productFour = new Product(orderTwo, "Corcs", 1);
28              Product productFive = new Product(orderTwo, "BenQ Monitor", 4);
29  
30              entityManager.persist(orderOne);
31              entityManager.persist(orderTwo);
32              entityManager.persist(productOne);
33              entityManager.persist(productTwo);
34              entityManager.persist(productThree);
35              entityManager.persist(productFour);
36              entityManager.persist(productFive);
37  
38          } catch (Exception ex) {
39              System.err.println("An error occurred: " + ex);
40          } finally {
41              entityManager.getTransaction().commit();
42              entityManager.close();
43              factory.close();
44          }
```

- Observe that I have `orderOne` that is a new order, then I have the product "iPhone" and the product "Nike".
  Both of these are associated with `orderOne` and I pass in `orderOne` in the constructor for these products, this is the code on lines 21 and 22.
- I've instantiated a new order `orderTwo` on line 24, and the products "Samsung", "Crocs" and "BenQ Monitors" are associated with `orderTwo`. And I set up this mapping with each product.
- I then persist the two orders and the five product entities that we've created.

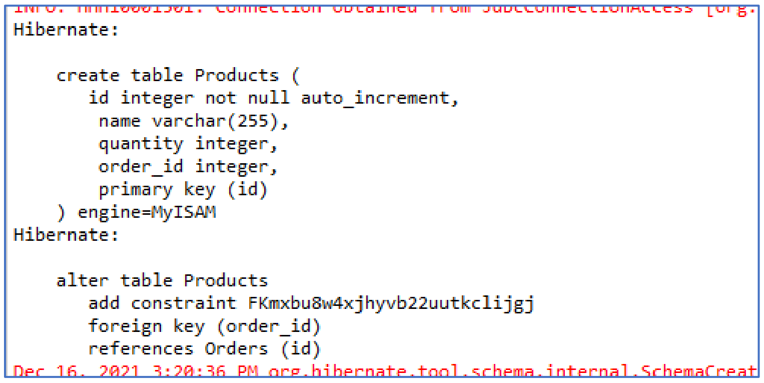
Let's run this code and take a look at how Hibernate models this many-to-one mapping using database tables.



```
Hibernate: 

    create table Orders (
       id integer not null auto_increment,
        orderDate date,
        primary key (id)
    ) engine=MyISAM
```

Here is the `Orders` table. This is fairly straightforward. We have the `id` that is the **primary key** and the `orderDate`.

Let's scroll down below and take a look at the Products table.



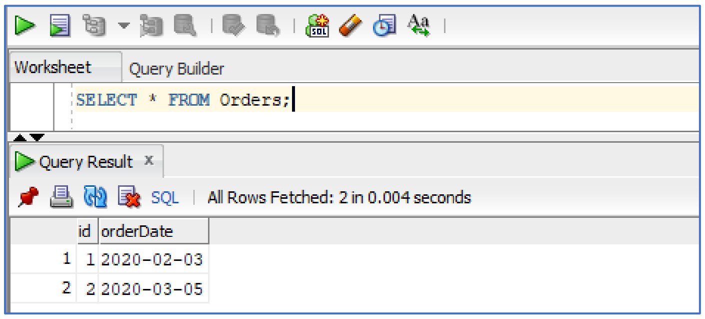
```
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.
Hibernate: 
    
    create table Products (
       id integer not null auto_increment,
        name varchar(255),
        quantity integer,
        order_id integer,
        primary key (id)
    ) engine=MyISAM
Hibernate: 
    
    alter table Products 
       add constraint FKmxbu8w4xjhyvb22uutkclijgj 
       foreign key (order_id) 
       references Orders (id)
Dec 16, 2021 3:20:36 PM org.hibernate.tool.schema.internal.SchemaCreat
```

- This Products table is the owning entity in this many-to-one unidirectional mapping.
- And this Products table has a **foreign key reference** to the Orders id associated with each product.
- Notice there is a column here called order_id that is of type **integer**.

When you specify the JPA many-to-one annotation to model a many-to-one relationship, the persistence provider that is Hibernate chooses the foreign key constraint method to model this relationship. This is the default method.

Notice that there is an alter table Products where we add a foreign key constraint where the order_id column in the Products table references the primary key of the Orders table.

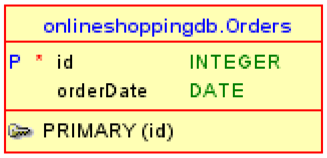
Finally, let's head over to MySQL Workbench and take a look at the entries that we've added to first the *Orders* table.



```
SELECT * FROM Orders;
```

Query Result — All Rows Fetched: 2 in 0.004 seconds

	id	orderDate
1	1	2020-02-03
2	2	2020-03-05



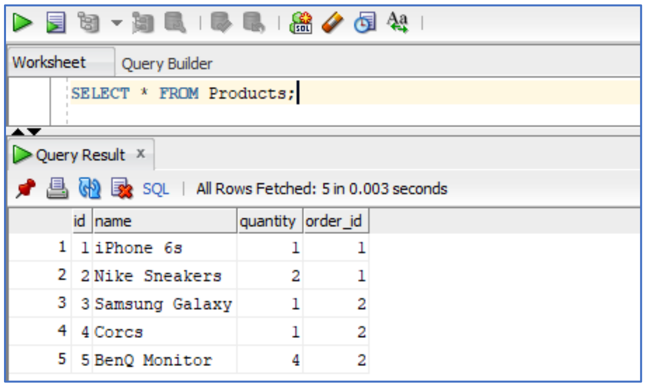
onlineshoppingdb.Orders

P *	id	INTEGER
	orderDate	DATE

PRIMARY (id)

Here are the two orders that we've persisted.

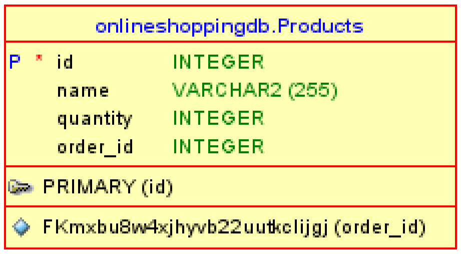
Let's run a *select ** on the *Products* table and take a look at the five products associated with these two orders.



```
SELECT * FROM Products;
```

Query Result — All Rows Fetched: 5 in 0.003 seconds

	id	name	quantity	order_id
1	1	iPhone 6s	1	1
2	2	Nike Sneakers	2	1
3	3	Samsung Galaxy	1	2
4	4	Corcs	1	2
5	5	BenQ Monitor	4	2



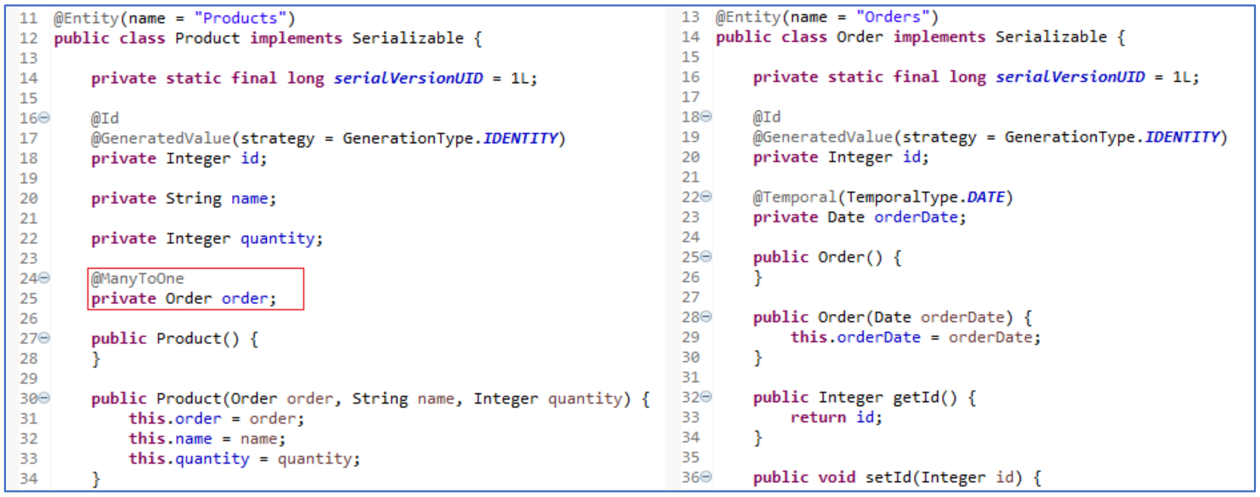
onlineshoppingdb.Products

P *	id	INTEGER
	name	VARCHAR2 (255)
	quantity	INTEGER
	order_id	INTEGER

PRIMARY (id)

FKmxbu8w4xjhyvb22uutkclijgj (order_id)

- Notice that we have an order_id column, the "iPhone 6S" and "Nike Sneakers" belong to order one.
- And the "Samsung Galaxy", "Crocs", and "BenQ Monitors" belong to order two.



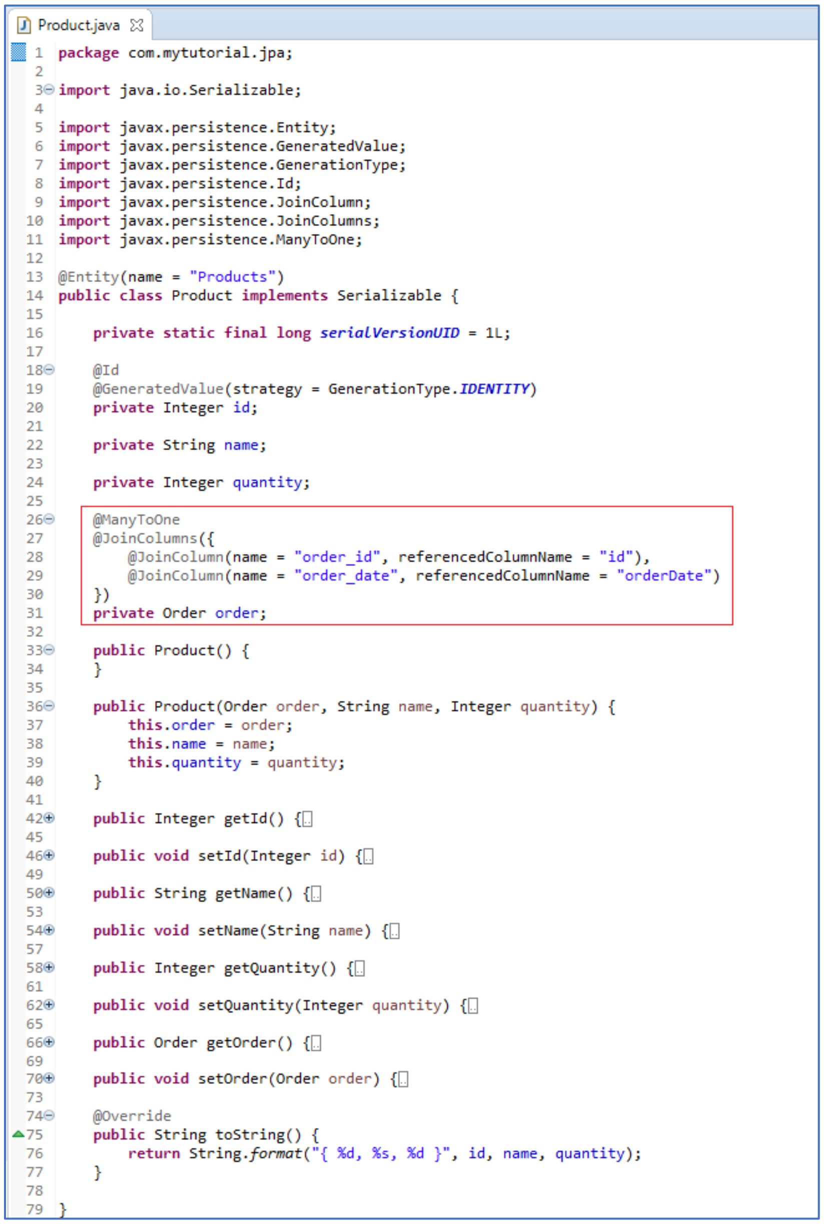
```java
@Entity(name = "Products")
public class Product implements Serializable {

    private static final long serialVersionUID = 1L;

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

    private String name;

    private Integer quantity;

    @ManyToOne
    private Order order;

    public Product() {
    }

    public Product(Order order, String name, Integer quantity) {
        this.order = order;
        this.name = name;
        this.quantity = quantity;
    }
```

```java
@Entity(name = "Orders")
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;

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

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    public Order() {
    }

    public Order(Date orderDate) {
        this.orderDate = orderDate;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
```

Many-to-one Mapping Multiple Join Columns

We've seen that when we specify a many to one mapping relationship, by default the relationship is modeled using a foreign key constraint.

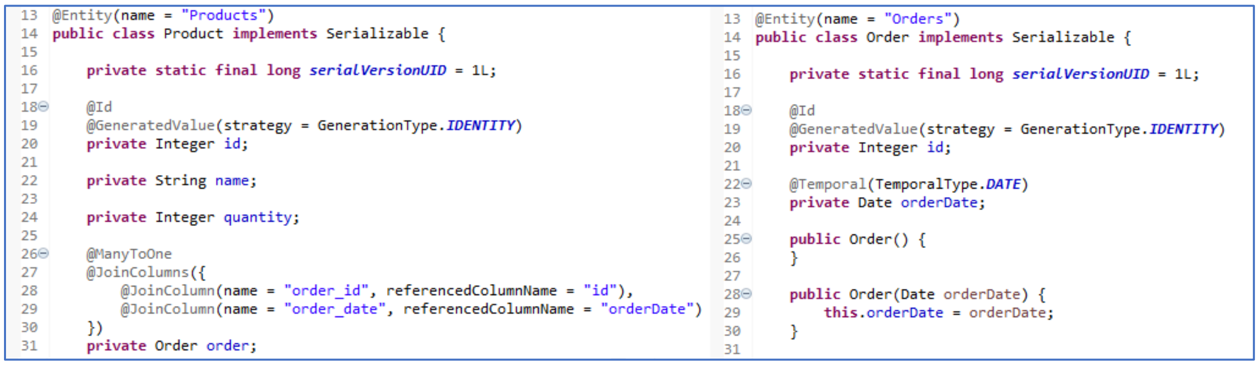
In our example here, many products map to a single order, and we set up a foreign key relationship from the `Products` table to the `Orders` table. Where every `Product` references the primary key of the `Order` associated with that `Product`.


Product.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.JoinColumns;
import javax.persistence.ManyToOne;

@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 Integer quantity;

    @ManyToOne
    @JoinColumns({
        @JoinColumn(name = "order_id", referencedColumnName = "id"),
        @JoinColumn(name = "order_date", referencedColumnName = "orderDate")
    })
    private Order order;

    public Product() {
    }

    public Product(Order order, String name, Integer quantity) {
        this.order = order;
        this.name = name;
        this.quantity = quantity;
    }

    public Integer getId() {

    public void setId(Integer id) {

    public String getName() {

    public void setName(String name) {

    public Integer getQuantity() {

    public void setQuantity(Integer quantity) {

    public Order getOrder() {

    public void setOrder(Order order) {

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

}
```

- I first set up the import statements for both of these annotations.
  `import javax.persistence.JoinColumn` and `import javax.persistence.JoinColumns`

- Instead of having every Product reference, just the Order id, which is the primary key of the Order. We'll have every Product reference, both the Order id as well as the orderDate. This is what I've specified using my **@JoinColumns** annotation.



```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 Integer quantity;

    @ManyToOne
    @JoinColumns({
        @JoinColumn(name = "order_id", referencedColumnName = "id"),
        @JoinColumn(name = "order_date", referencedColumnName = "orderDate")
    })
    private Order order;
```

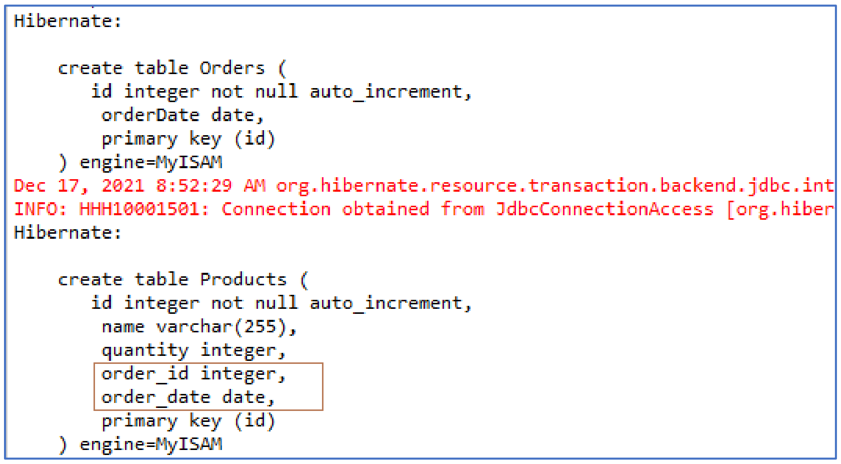
```java
@Entity(name = "Orders")
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;

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

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    public Order() {
    }

    public Order(Date orderDate) {
        this.orderDate = orderDate;
    }

```

Observe the plural here, we use **@JoinColumns**, because we have multiple columns on which we want the join operation to be performed.

- Within **@JoinColumns**, I specify a set of individual column names using the **@JoinColumn** annotation. In our underlying database table we want every record for a Product to reference the id of the Order as well as the orderDate.
- Observe the **@JoinColumn** annotations. The column with the name order_id will reference the id of the order.
- The column with name order_date in the Products table will reference the orderDate column in the Orders table.

And this is the only change that we need to make to have **multiple join columns**. That is multiple columns that **reference** the orders table **using foreign keys**.

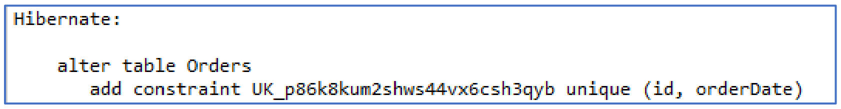
Let's take a look at the create table statement here.



```
Hibernate: 

    create table Orders (
       id integer not null auto_increment,
        orderDate date,
        primary key (id)
    ) engine=MyISAM
Dec 17, 2021 8:52:29 AM org.hibernate.resource.transaction.backend.jdbc.int
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hiber
Hibernate: 

    create table Products (
       id integer not null auto_increment,
        name varchar(255),
        quantity integer,
        order_id integer,
        order_date date,
        primary key (id)
    ) engine=MyISAM
```

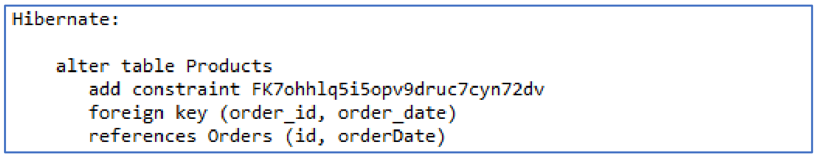
Notice that the Products table has two additional columns.
We have order_id, that is of type **integer** which references the **primary key** of the orders table. And order_date of type **date**, which here references the order date for the corresponding order.

These columns together make a composite foreign key reference to the Orders table. So Hibernate has added a uniqueness constraint on the Orders table so that the order id and the orderDate combination is unique.



```
Hibernate: 

    alter table Orders 
       add constraint UK_p86k8kum2shws44vx6csh3qyb unique (id, orderDate)
```

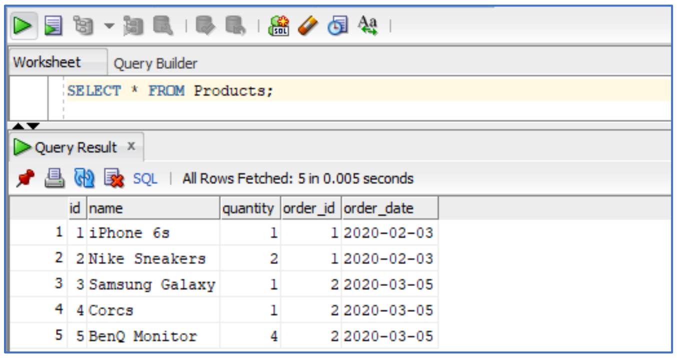
If you scroll down a bit, you'll see that Hibernate has also added a constraint for the foreign key reference.



```
Hibernate: 

    alter table Products 
       add constraint FK7ohhlq5i5opv9druc7cyn72dv 
       foreign key (order_id, order_date) 
       references Orders (id, orderDate)
```

The order_id and order_date columns in the Products table references the id and orderDate column in the Orders table.

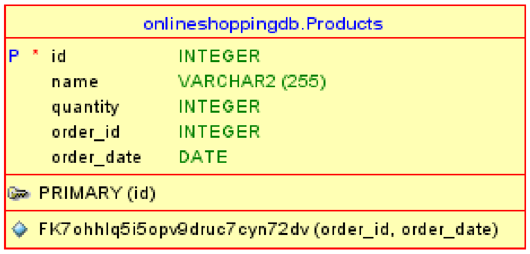
Let's head over to MySQL Workbench and take a look at the Products table here.



```
SELECT * FROM Products;
```

All Rows Fetched: 5 in 0.005 seconds

	id	name	quantity	order_id	order_date
1	1	iPhone 6s	1	1	2020-02-03
2	2	Nike Sneakers	2	1	2020-02-03
3	3	Samsung Galaxy	1	2	2020-03-05
4	4	Corcs	1	2	2020-03-05
5	5	BenQ Monitor	4	2	2020-03-05

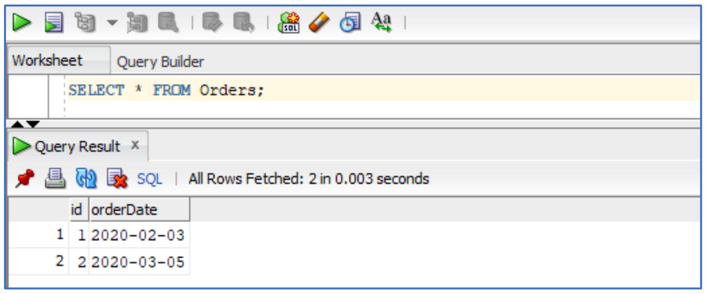


```
onlineshoppingdb.Products
P * id          INTEGER
    name        VARCHAR2 (255)
    quantity    INTEGER
    order_id    INTEGER
    order_date  DATE
PRIMARY (id)
FK7ohhlq5i5opv9druc7cyn72dv (order_id, order_date)
```

Notice the two additional columns order_id and order_date, which references the order corresponding to each product.

The “iPhone 6S” and “Nike Sneakers” belong to the same order with order_id 1.

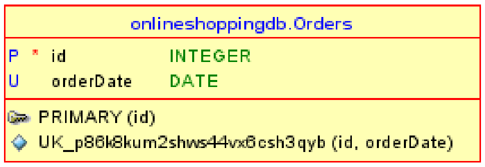
And “Samsung Galaxy”, “Crocs” and “BenQ Monitors” belong to the same order with order_id 2.



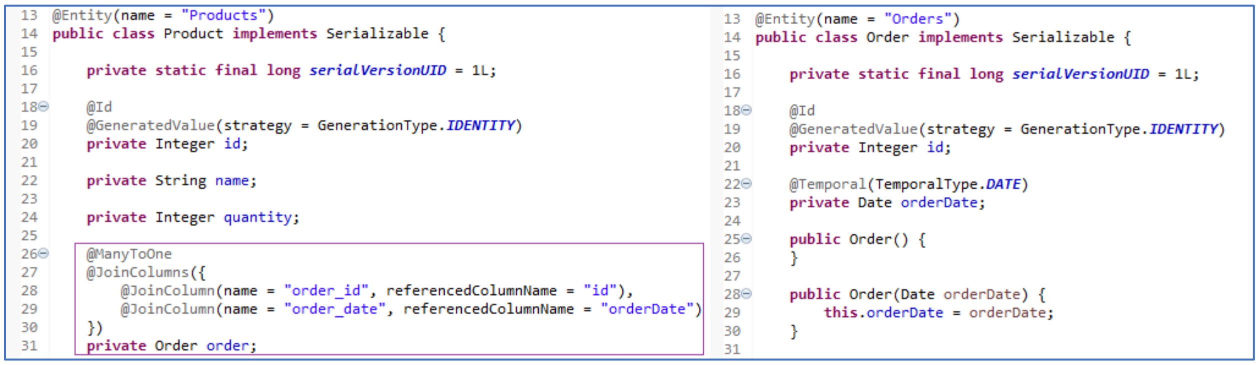
```
SELECT * FROM Orders;
```

All Rows Fetched: 2 in 0.003 seconds

	id	orderDate
1	1	2020-02-03
2	2	2020-03-05



```
onlineshoppingdb.Orders
P * id          INTEGER
U   orderDate   DATE
PRIMARY (id)
UK_p86k8kum2shws44vx6csh3qyb (id, orderDate)
```

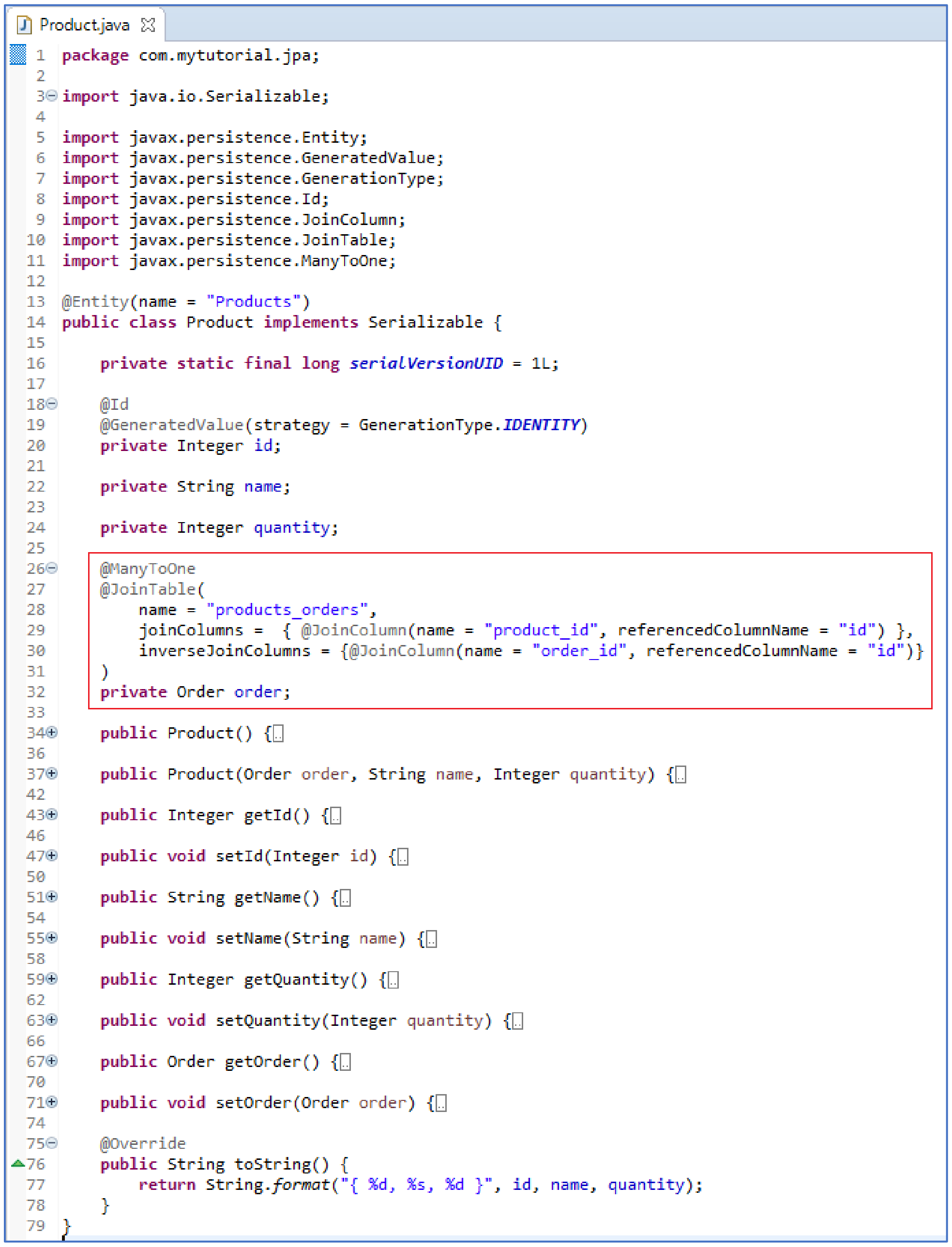


```java
@Entity(name = "Products")
public class Product implements Serializable {

    private static final long serialVersionUID = 1L;

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

    private String name;

    private Integer quantity;

    @ManyToOne
    @JoinColumns({
        @JoinColumn(name = "order_id", referencedColumnName = "id"),
        @JoinColumn(name = "order_date", referencedColumnName = "orderDate")
    })
    private Order order;
```

```java
@Entity(name = "Orders")
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;

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

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    public Order() {
    }

    public Order(Date orderDate) {
        this.orderDate = orderDate;
    }

```

Many-to-one Mapping Using Join Tables

It's also possible to model your many to one relationship using a separate join table, rather than using foreign key constraints. You need to configure this many to one relationship for the join table using specific JPA annotations.



Product.java

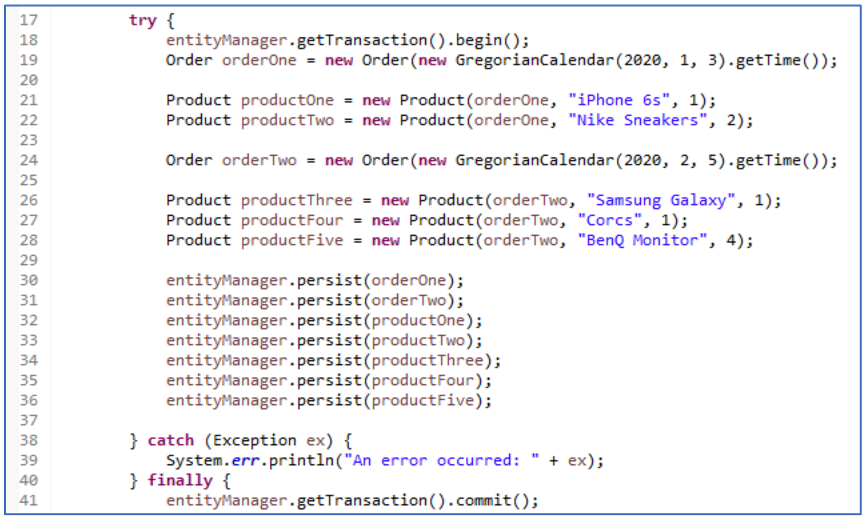
```java
package com.mytutorial.jpa;

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.JoinTable;
import javax.persistence.ManyToOne;

@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 Integer quantity;

    @ManyToOne
    @JoinTable(
        name = "products_orders",
        joinColumns =  { @JoinColumn(name = "product_id", referencedColumnName = "id") },
        inverseJoinColumns = {@JoinColumn(name = "order_id", referencedColumnName = "id")}
    )
    private Order order;

    public Product() {

    public Product(Order order, String name, Integer quantity) {

    public Integer getId() {

    public void setId(Integer id) {

    public String getName() {

    public void setName(String name) {

    public Integer getQuantity() {

    public void setQuantity(Integer quantity) {

    public Order getOrder() {

    public void setOrder(Order order) {

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

- Go ahead and set up the import statement for the **@JoinTable** annotation.
  `import javax.persistence.JoinTable`
- Then, for our reference to the order entity. From the `Product` entity, instead of using the **@JoinColumns** annotation, I'm going to change it to the **@JoinTable** annotation.

- This will set up a separate table. The **name** of that table I've specified as `"products_orders"`. That will hold the mapping relationship, the many to one relationship from many `Products` to a single `Order` entity. The name of the join table holding the mapping relationship is `products_orders`.
- And in the **`joinColumns`** property we've specified that the `product_id` column in the join table will reference the `id` column in the `Product` entity. That is the owning entity.
- In the **`inverseJoinColumns`** property, we've specified that the `order_id` column in the join table will reference the `id` of the `Order` entity. Which is the other side of this mapping relationship.

Notice that `product_id` and `order_id` are references to the **primary keys** of the `Products` and `Orders` table respectively.

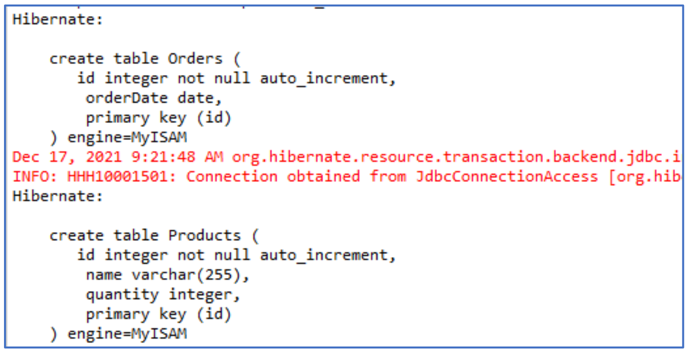
Let's go ahead and run this code after switching over to the **App.java** file.



```java
        try {
            entityManager.getTransaction().begin();
            Order orderOne = new Order(new GregorianCalendar(2020, 1, 3).getTime());

            Product productOne = new Product(orderOne, "iPhone 6s", 1);
            Product productTwo = new Product(orderOne, "Nike Sneakers", 2);

            Order orderTwo = new Order(new GregorianCalendar(2020, 2, 5).getTime());

            Product productThree = new Product(orderTwo, "Samsung Galaxy", 1);
            Product productFour = new Product(orderTwo, "Corcs", 1);
            Product productFive = new Product(orderTwo, "BenQ Monitor", 4);

            entityManager.persist(orderOne);
            entityManager.persist(orderTwo);
            entityManager.persist(productOne);
            entityManager.persist(productTwo);
            entityManager.persist(productThree);
            entityManager.persist(productFour);
            entityManager.persist(productFive);

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

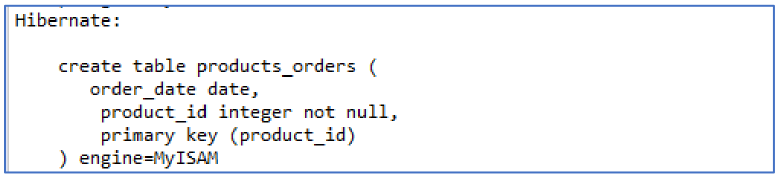
We've persisted two orders and five products to the underlying tables.

Here is the `Products` table with `id`, `name`, `quantity`, `id` is the **primary key**.



```
Hibernate:

    create table Orders (
       id integer not null auto_increment,
        orderDate date,
        primary key (id)
    ) engine=MyISAM
Dec 17, 2021 9:21:48 AM org.hibernate.resource.transaction.backend.jdbc.i
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hib
Hibernate:

    create table Products (
       id integer not null auto_increment,
        name varchar(255),
        quantity integer,
        primary key (id)
    ) engine=MyISAM
```

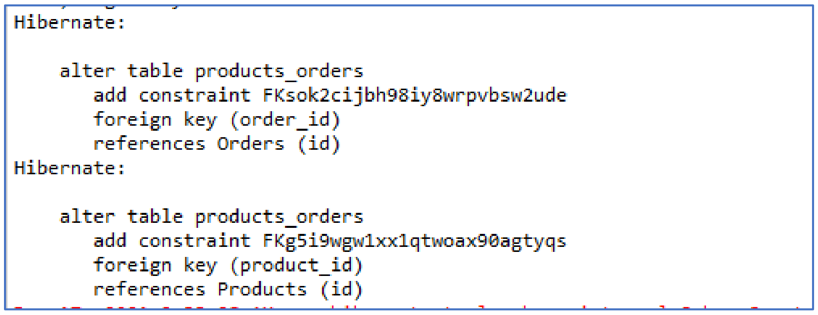
Here is the products_orders table, which is the join table holding this many to one mapping relationship.



```
Hibernate: 

    create table products_orders (
       order_date date,
        product_id integer not null,
        primary key (product_id)
    ) engine=MyISAM
```

The two columns in this join table are order_id and product_id, and product_id is the primary key. Remember, many products map to a single Order.

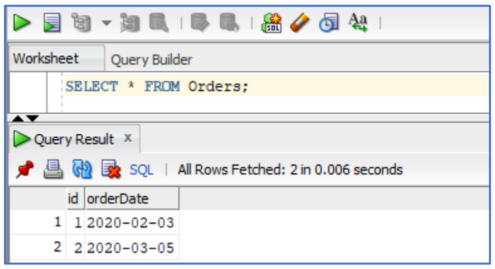
And here below are the foreign key references to the corresponding entity tables.



```
Hibernate: 

    alter table products_orders 
       add constraint FKsok2cijbh98iy8wrpvbsw2ude 
       foreign key (order_id) 
       references Orders (id)
Hibernate: 

    alter table products_orders 
       add constraint FKg5i9wgw1xx1qtwoax90agtyqs 
       foreign key (product_id) 
       references Products (id)
```

The order_id references the primary key of the Orders table. This is the first foreign key reference. And the product_id references the id of the Products table, the second foreign key reference.

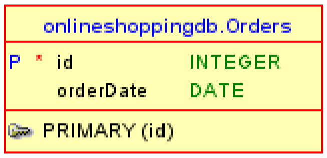
Back to MySQL Workbench to take a look at how this join table holds the mapping relationship. Let's do a _select *_ on the *Orders* table.



```
SELECT * FROM Orders;
```

All Rows Fetched: 2 in 0.006 seconds

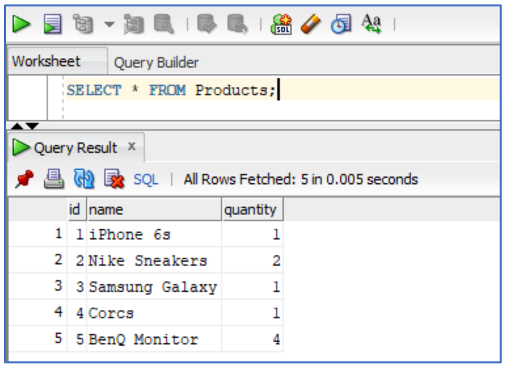
	id	orderDate
1	1	2020-02-03
2	2	2020-03-05



```
onlineshoppingdb.Orders
P * id          INTEGER
    orderDate   DATE
PRIMARY (id)
```

Here are the two orders which have entries in this table.

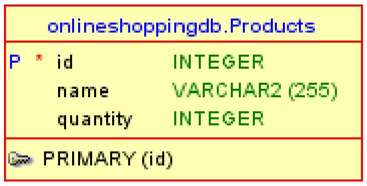
Let's do a _select *_ on the *Products* table, there are a total of five products.



```
SELECT * FROM Products;
```

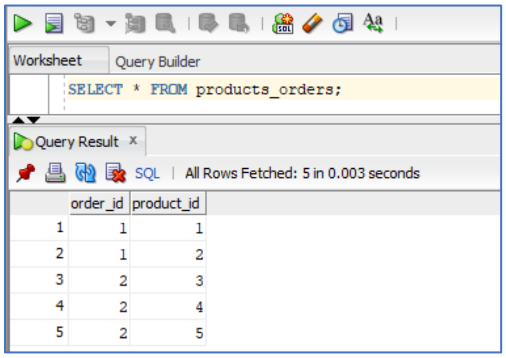
All Rows Fetched: 5 in 0.005 seconds

	id	name	quantity
1	1	iPhone 6s	1
2	2	Nike Sneakers	2
3	3	Samsung Galaxy	1
4	4	Corcs	1
5	5	BenQ Monitor	4



```
onlineshoppingdb.Products
P * id          INTEGER
    name        VARCHAR2 (255)
    quantity    INTEGER
PRIMARY (id)
```

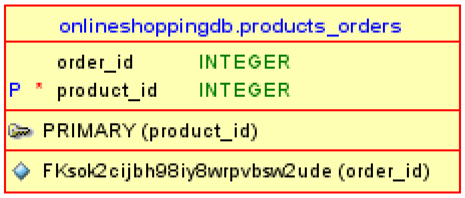
The mapping relationship that is many to one from products to orders is held in the `products_orders` table. Let's run a select * here.



```
SELECT * FROM products_orders;
```

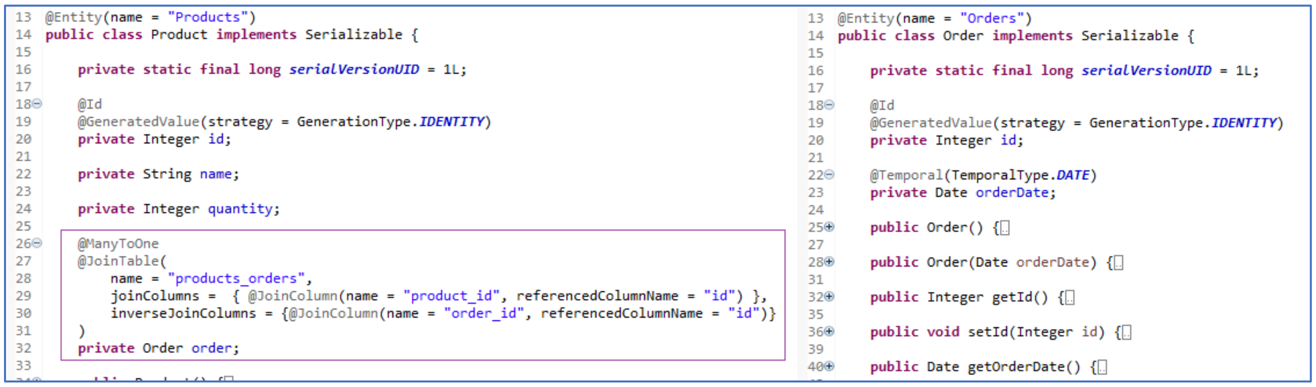
All Rows Fetched: 5 in 0.003 seconds

	order_id	product_id
1	1	1
2	1	2
3	2	3
4	2	4
5	2	5



```
onlineshoppingdb.products_orders
    order_id      INTEGER
P * product_id    INTEGER
PRIMARY (product_id)
FKsok2cijbh98iy8wrpvbsw2ude (order_id)
```

And you can see that `order_id` 1 references `Products` with `id` 1 and 2, and `order_id` 2 maps to `Products` with `id`s 3, 4, and 5.

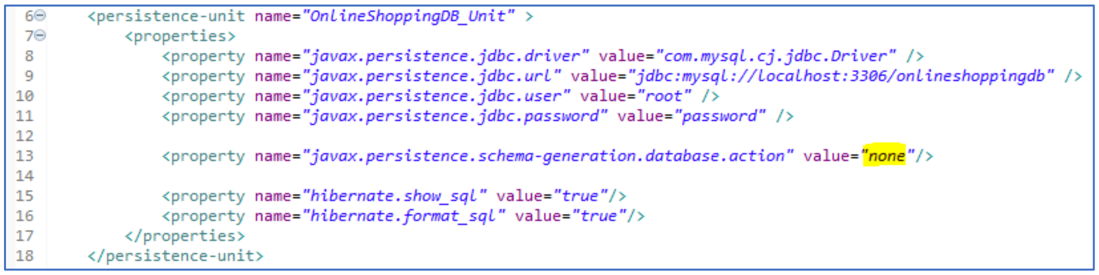


```java
@Entity(name = "Products")
public class Product implements Serializable {

    private static final long serialVersionUID = 1L;

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

    private String name;

    private Integer quantity;

    @ManyToOne
    @JoinTable(
        name = "products_orders",
        joinColumns =  { @JoinColumn(name = "product_id", referencedColumnName = "id") },
        inverseJoinColumns = {@JoinColumn(name = "order_id", referencedColumnName = "id")}
    )
    private Order order;
```

```java
@Entity(name = "Orders")
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;

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

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    public Order() {□

    public Order(Date orderDate) {□

    public Integer getId() {□

    public void setId(Integer id) {□

    public Date getOrderDate() {□
```

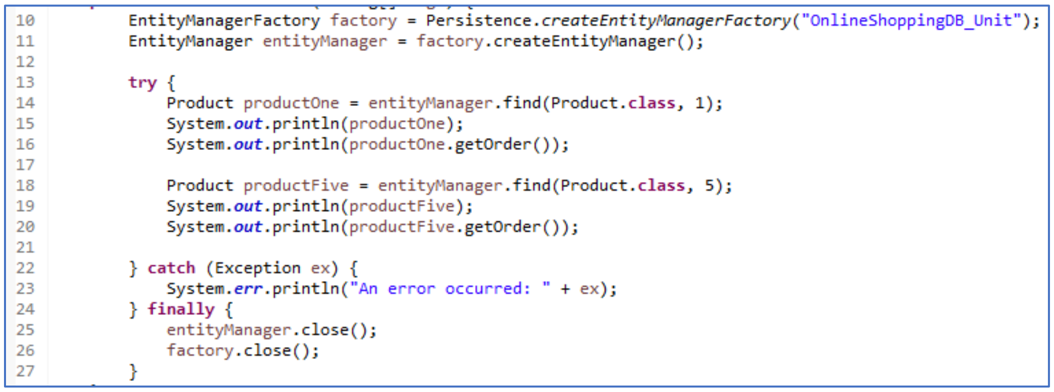
In our many to one relationship that we've set up here, the many side, that is the Product side, is the owning entity. Let's see how retrieval from the underlying database table works. When you retrieve a product, the corresponding order you'll see is fetched eagerly.

Let's change the database action here from drop-and-create to none. We'll work with the database that has already been created, we won't create new tables.



```
    <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="none"/>

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

Switching over to **App.java**, instead of creating and persisting entities, I'm going to retrieve entities using the find method. I'm going to retrieve Products, we'll see that the corresponding Order is retrieved automatically.

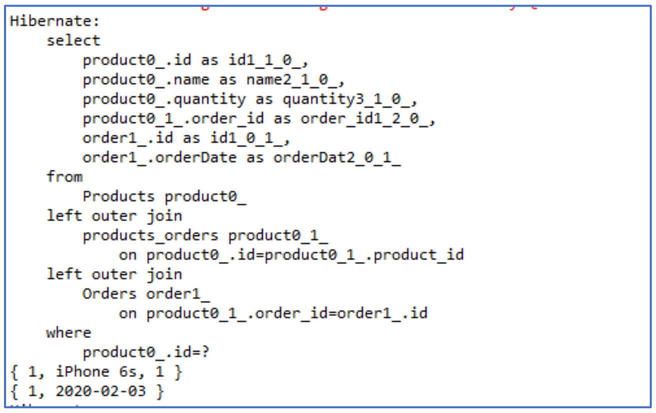


```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {
            Product productOne = entityManager.find(Product.class, 1);
            System.out.println(productOne);
            System.out.println(productOne.getOrder());

            Product productFive = entityManager.find(Product.class, 5);
            System.out.println(productFive);
            System.out.println(productFive.getOrder());

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

When it's a **Many To One** relationship map, the entity on the one side of the relationship is **eagerly loaded**. When you retrieve an entity on the many side of the relationship that is, when you retrieve a product, the order is loaded automatically.

- Our first find method invocation is for the Product with primary key 1.
- We then print out productOne and the associated Order.
- We then invoke find once again, to load the Product with id 5,
- we print out the product and the associated Order.

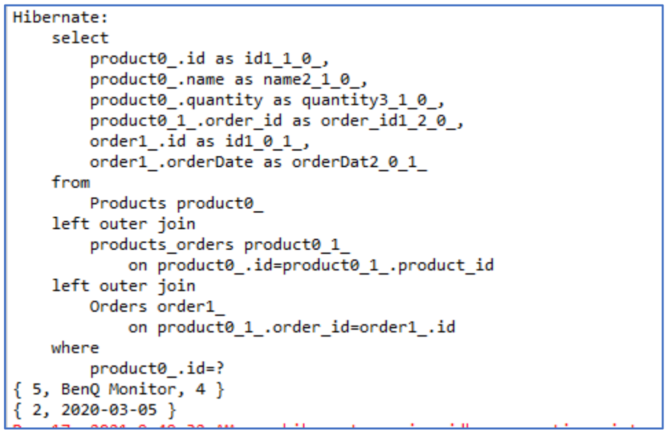
Let's go ahead and run this code. If you scroll below to the select statement, which is run under the hood in order to retrieve products.



```
Hibernate:
    select
        product0_.id as id1_1_0_,
        product0_.name as name2_1_0_,
        product0_.quantity as quantity3_1_0_,
        product0_1_.order_id as order_id1_2_0_,
        order1_.id as id1_0_1_,
        order1_.orderDate as orderDat2_0_1_
    from
        Products product0_
    left outer join
        products_orders product0_1_
            on product0_.id=product0_1_.product_id
    left outer join
        Orders order1_
            on product0_1_.order_id=order1_.id
    where
        product0_.id=?
{ 1, iPhone 6s, 1 }
{ 1, 2020-02-03 }
```

- Notice that Order fields are retrieved at the same time.
- We perform a single select statement with **join operations** to retrieve a Product and the corresponding order.
- Here is the first Product, the “iPhone 6S” printed out to screen, and its Order details have also been printed out to screen.

If you scroll down further,



```
Hibernate:
    select
        product0_.id as id1_1_0_,
        product0_.name as name2_1_0_,
        product0_.quantity as quantity3_1_0_,
        product0_1_.order_id as order_id1_2_0_,
        order1_.id as id1_0_1_,
        order1_.orderDate as orderDat2_0_1_
    from
        Products product0_
    left outer join
        products_orders product0_1_
            on product0_.id=product0_1_.product_id
    left outer join
        Orders order1_
            on product0_1_.order_id=order1_.id
    where
        product0_.id=?
{ 5, BenQ Monitor, 4 }
{ 2, 2020-03-05 }
```

- Here is the second select statement to retrieve the Product with primary key 5.
- Product information as well as the corresponding Order information is retrieved in a single select statement, which performs a **join operation**.
- And here at the bottom, we've printed out to screen the Product with id 5, “BenQ Monitors”, and the corresponding Order information as well.

Retrieving a Product entity eagerly retrieves the corresponding order. The Product is on the many side of the relationship, the Order on the one side.

One-to-many, Many-to-one Bidirectional Mapping

So far, we've seen examples of unidirectional mapping from one-to-many and many-to-one. In this demo, we'll see how we can set up a bidirectional relationship. Where we map from one entity to many entities, and from many entities to a single entity.

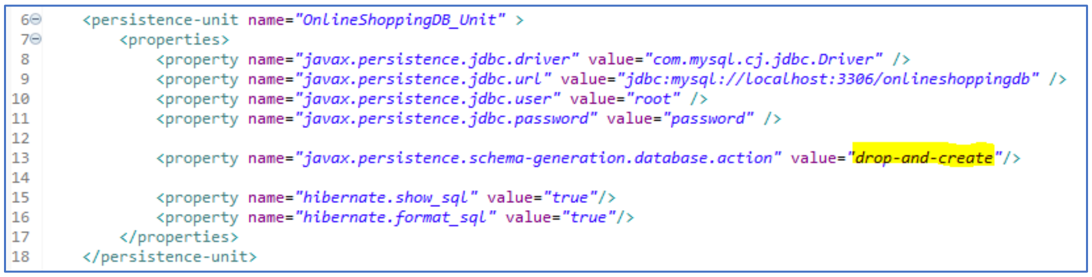
In a bidirectional mapping, entities on both sides of the relationship reference one another. So there is an **owning side** and an **inverse side**. We've seen this in the bidirectional mapping that we set up for one is to one mapping.

In the case of one-to-many and many-to-one relationship mapping.
"JPA best practice says that the **many side of the relationship should be the owning side**."

This is just a best practice as suggested by JPA. You might find it more intuitive to have the one side of the relationship be the owning side, and that's fine.

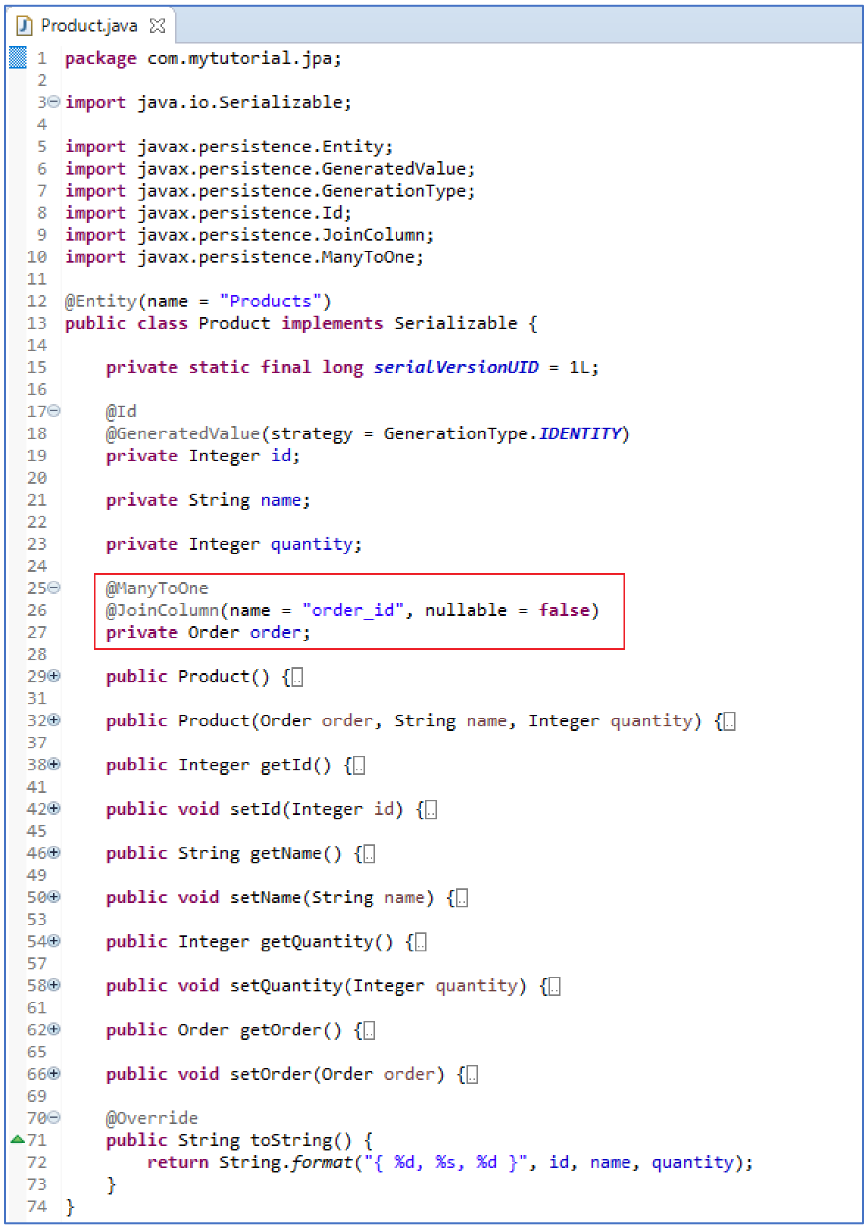
But here in our first demo here for bidirectional mapping of one-to-many and many-to-one relationship, we'll make the many side the owning side.

I'm going to update the **persistence.xml** file here so that the database action is drop-and-create. We'll recreate tables when the application runs.



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

I'll now setup the one-to-many and many-to-one bidirectional mapping starting from the many side, that is **Product.java**. Remember, many `Products` map to a single `Order`.


Product.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 = "Products")
public class Product implements Serializable {

    private static final long serialVersionUID = 1L;

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

    private String name;

    private Integer quantity;

    @ManyToOne
    @JoinColumn(name = "order_id", nullable = false)
    private Order order;

    public Product() {

    public Product(Order order, String name, Integer quantity) {

    public Integer getId() {

    public void setId(Integer id) {

    public String getName() {

    public void setName(String name) {

    public Integer getQuantity() {

    public void setQuantity(Integer quantity) {

    public Order getOrder() {

    public void setOrder(Order order) {

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

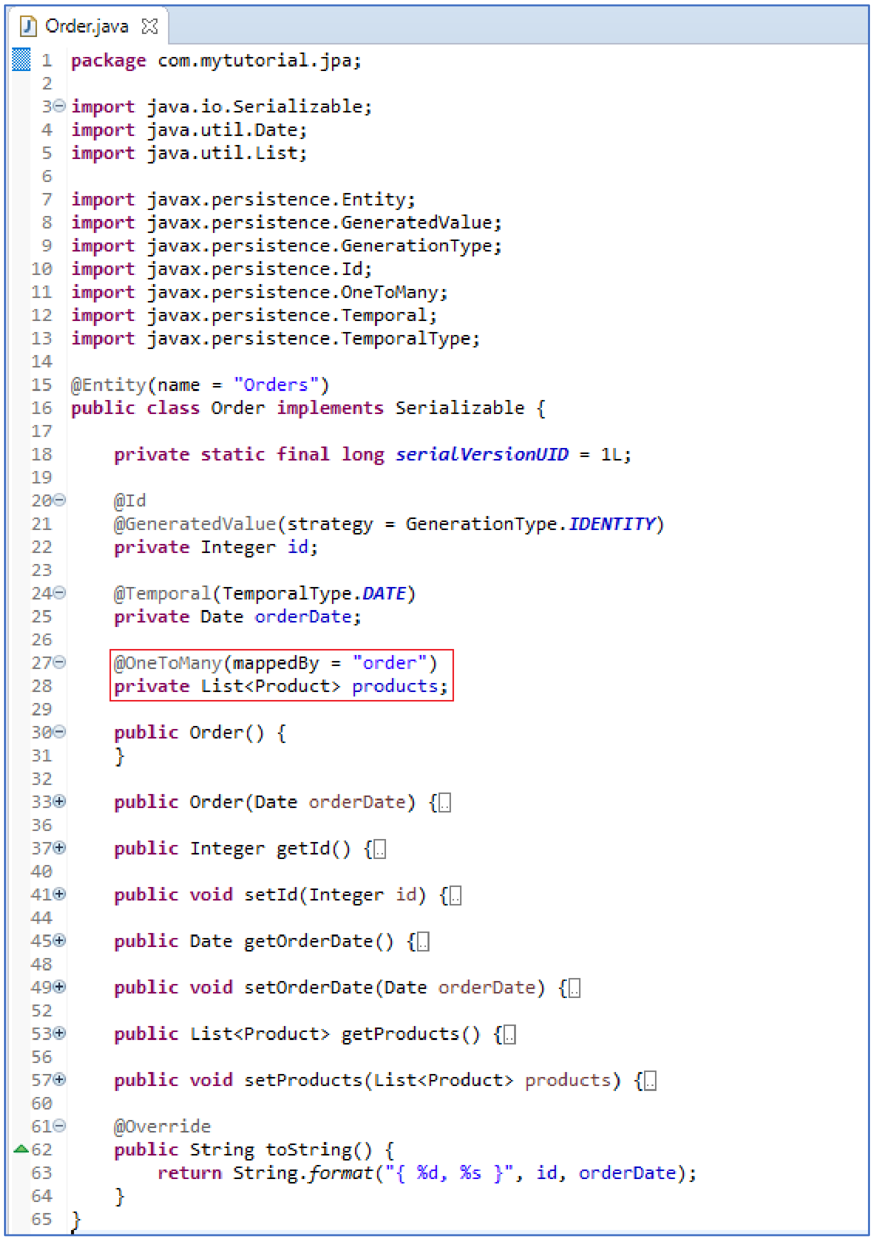
- Set up the import statements for the annotations. Remember, the `Product` is the many side of the relationship.
- The member variables that make up this class map to columns in the underlying `Products` table, `id`, `name`, and `quantity`.
- The `Product` also references the `Order` associated with each `Product`, and this is a **@ManyToOne** mapping, many `Products` exist within the same `order`.
- I have annotated a reference to the `order` entity using **@ManyToOne**, and I have also specified an **@JoinColumn** annotation.
  This allows us to configure the column within the `Products` table that holds a **foreign key reference** to the `Orders` table. The name of the column is `order_id`, and I have said that it's **not nullable**, nullable is equal to `false`.

- The rest of the code here in the `Product` or Java file is familiar to us. The **default no argument constructor**, the **constructor** which takes in a reference to the `order`, `name`, and `quantity` of the `Product`.

Because the product side of the relationship is the owning side, we'll set a reference to the `Order` from the corresponding `Product`. And before we persist the objects to the database.

- Set up **getters** and **setters** for each of the individual member variable, `order`, `quantity`, `id` and so on.

We're now ready to head over to the **Order.java** file. This is the inverse side of the relationship, product is the owning side, order is the inverse side.



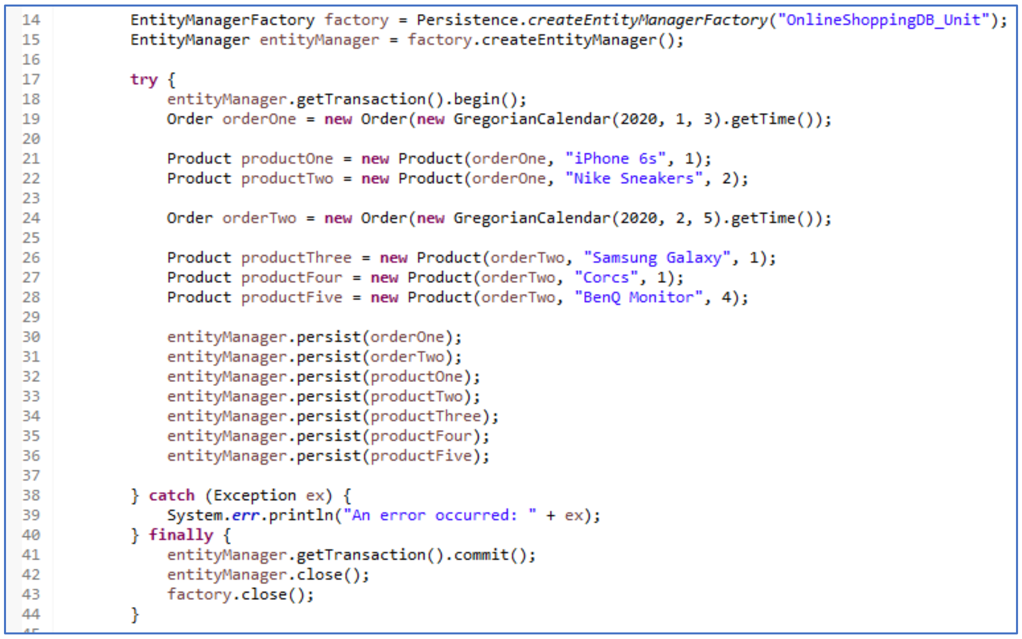
```java
package com.mytutorial.jpa;

import java.io.Serializable;
import java.util.Date;
import java.util.List;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity(name = "Orders")
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;

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

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    @OneToMany(mappedBy = "order")
    private List<Product> products;

    public Order() {
    }

    public Order(Date orderDate) {

    public Integer getId() {

    public void setId(Integer id) {

    public Date getOrderDate() {

    public void setOrderDate(Date orderDate) {

    public List<Product> getProducts() {

    public void setProducts(List<Product> products) {

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

- Now, because this is a bidirectional mapping. The reference that we have from `Order` to the `List` of `products` associated with the `Order` will be annotated using **@OneToMany**.
- The remaining attributes of the order remain the same. We have the `id` that is the **primary key** and the `orderDate` with the **@Temporal** and `TemporalType.DATE` annotation.
- Notice the `List` of `products` that we have referenced from within an `Order`, we use the **@OneToMany** annotation to annotate this `List` of `products`.
- We know that this is the inverse side of the relationship, not the owning side. Because we've specified the **mappedBy** property in this **@OneToMany** annotation.
- The value specified by the **mappedBy** property is the name of the member variable within the broader class that references the `Order`. Which in our case is just `order`.

  In a bidirectional mapping, it's possible that you make a mistake and configure your mapping with inconsistencies. The way you've defined the one-to-many side and the many-to-one side might be inconsistent. In such situations, JPA simply considers the configuration that you have set up in the owning entity. In our case, the owning entity is `Product`.

- The rest of the code in *Order.java* remains the same. We have getters and setters for the individual member variables, including the `List` of `products` associated with each order.

Now, let's switch over to **App.java** and set up the entities that we want to persist to the underlying table.

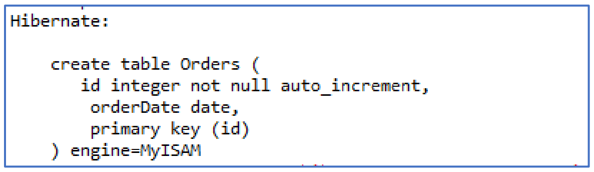


```
14          EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
15          EntityManager entityManager = factory.createEntityManager();
16
17          try {
18              entityManager.getTransaction().begin();
19              Order orderOne = new Order(new GregorianCalendar(2020, 1, 3).getTime());
20
21              Product productOne = new Product(orderOne, "iPhone 6s", 1);
22              Product productTwo = new Product(orderOne, "Nike Sneakers", 2);
23
24              Order orderTwo = new Order(new GregorianCalendar(2020, 2, 5).getTime());
25
26              Product productThree = new Product(orderTwo, "Samsung Galaxy", 1);
27              Product productFour = new Product(orderTwo, "Corcs", 1);
28              Product productFive = new Product(orderTwo, "BenQ Monitor", 4);
29
30              entityManager.persist(orderOne);
31              entityManager.persist(orderTwo);
32              entityManager.persist(productOne);
33              entityManager.persist(productTwo);
34              entityManager.persist(productThree);
35              entityManager.persist(productFour);
36              entityManager.persist(productFive);
37
38          } catch (Exception ex) {
39              System.err.println("An error occurred: " + ex);
40          } finally {
41              entityManager.getTransaction().commit();
42              entityManager.close();
43              factory.close();
44          }
```

- As we've always done, we'll use the `EntityManagerFactory` with our persistence unit and access the `EntityManager`.
- Within a transaction, I'll first instantiate an `Order` that is `orderOne` and then associate two `Products` with the same `Order`, product “iPhone” and product “Nike”.
- Make sure you set up the reference from the `Product` to the `Order`.
- Remember the `Product` side is the owning entity. If you notice on lines 21 and 22, I pass in a reference to `orderOne` while instantiating the “iPhone” and “Nike” products.

- I then instantiate `orderOne`. And while creating the "Samsung", "Crocs" and "BenQ" products, I pass in a reference to `orderOne`. This is on lines 26 to 28.
- In order to persist all of these entities to the underlying table called `entityManager.persist()` on our two `Order` and five `Product`.

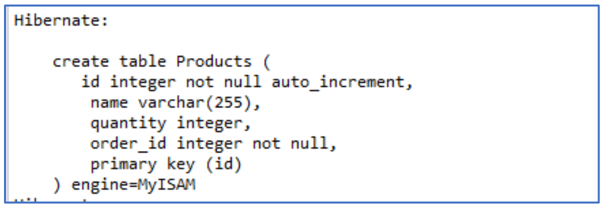
This is all the code that we need to write, not much different here. Let's go ahead and run our App.java and take a look at how the tables have been created. And orders and products have been inserted into tables. Here is the create statement for the `Orders` table,



```
Hibernate: 

    create table Orders (
       id integer not null auto_increment,
        orderDate date,
        primary key (id)
    ) engine=MyISAM
```

we have `id` and `orderDate`, the primary key is `id`.

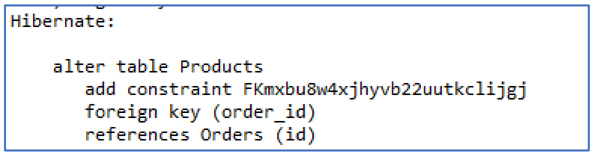
Here below is the create statement for the `Products` table.



```
Hibernate: 

    create table Products (
       id integer not null auto_increment,
        name varchar(255),
        quantity integer,
        order_id integer not null,
        primary key (id)
    ) engine=MyISAM
```

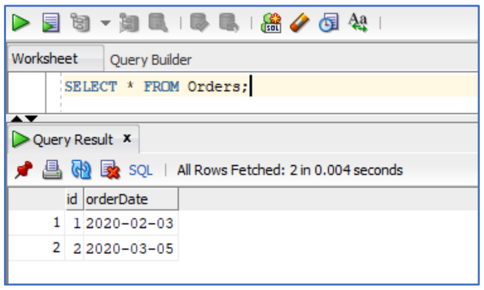
The `id`, `name`, `quantity` are the columns in a `Products`. You also have the `order_id` column. Which is a **foreign key reference** from the `Products` table to the corresponding `Order` that is associated with each product.

If you scroll down below, you can see that Hibernate has used an alter table command to add this foreign key constraint.



```
Hibernate: 

    alter table Products 
       add constraint FKmxbu8w4xjhyvb22uutkclijgj 
       foreign key (order_id) 
       references Orders (id)
```

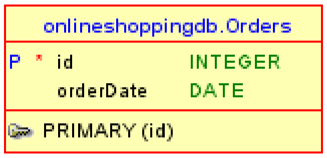
Let's quickly verify the entries in our database tables using MySQL Workbench. Do a *select ** from the *Orders* table.



```
SELECT * FROM Orders;
```

All Rows Fetched: 2 in 0.004 seconds

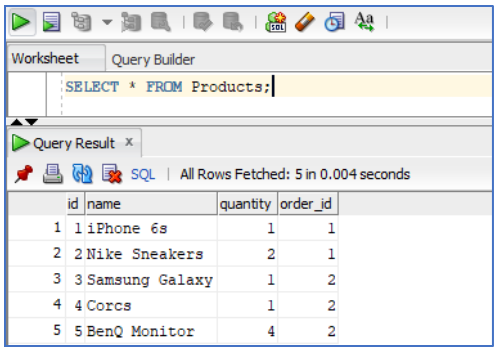
	id	orderDate
1	1	2020-02-03
2	2	2020-03-05



```
onlineshoppingdb.Orders
P * id          INTEGER
    orderDate   DATE
PRIMARY (id)
```

Here are the two order entries correctly represented.

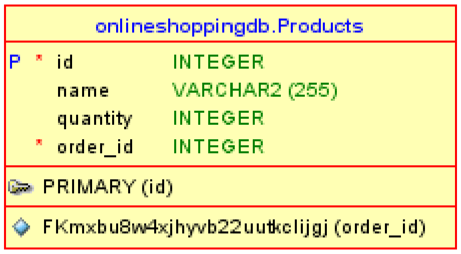
Let's do a *select ** on the *Products* table.



```
SELECT * FROM Products;
```

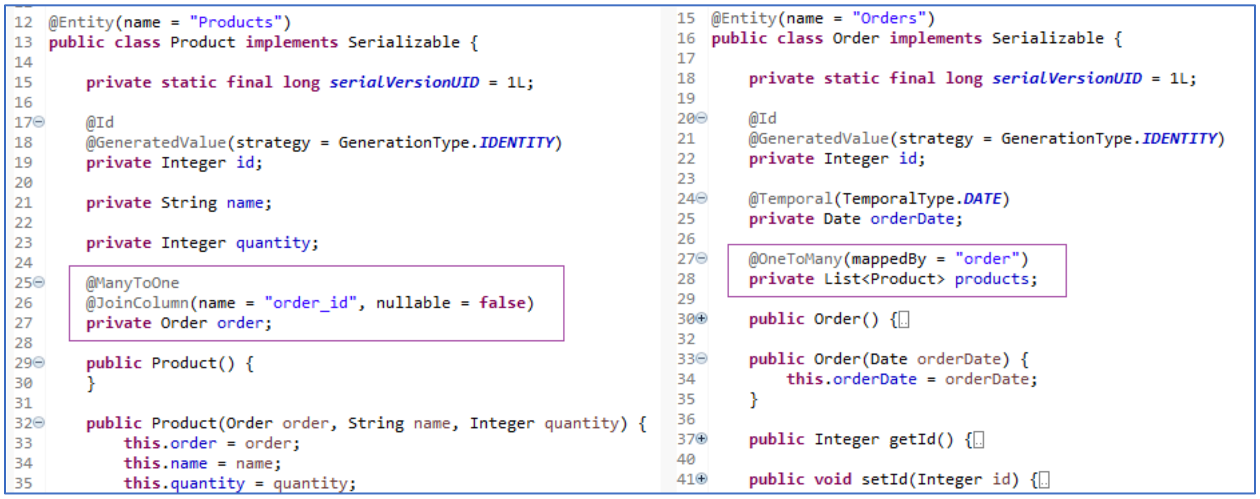
All Rows Fetched: 5 in 0.004 seconds

	id	name	quantity	order_id
1	1	iPhone 6s	1	1
2	2	Nike Sneakers	2	1
3	3	Samsung Galaxy	1	2
4	4	Corcs	1	2
5	5	BenQ Monitor	4	2



```
onlineshoppingdb.Products
P * id          INTEGER
    name        VARCHAR2 (255)
    quantity    INTEGER
  * order_id    INTEGER
PRIMARY (id)
FKmxbu8w4xjhyvb22uutkclijgj (order_id)
```

And you can see that we have a total of five entries, and every Product is associated with an order, mapped using the order_id column.
Our bidirectional one-to-many, many-to-one mapping has been set up correctly.



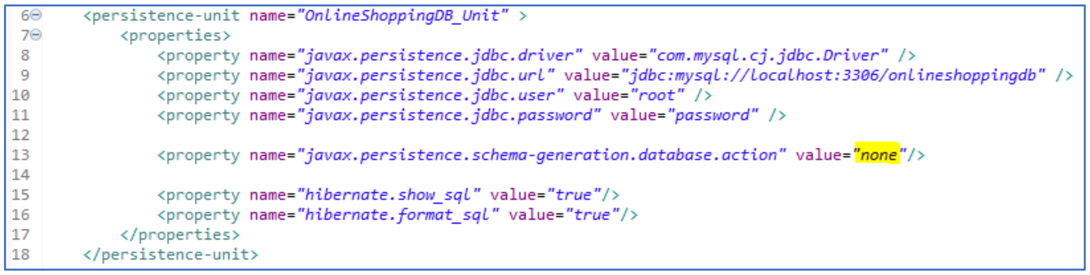
```java
@Entity(name = "Products")
public class Product implements Serializable {

    private static final long serialVersionUID = 1L;

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

    private String name;

    private Integer quantity;

    @ManyToOne
    @JoinColumn(name = "order_id", nullable = false)
    private Order order;

    public Product() {
    }

    public Product(Order order, String name, Integer quantity) {
        this.order = order;
        this.name = name;
        this.quantity = quantity;
```

```java
@Entity(name = "Orders")
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;

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

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    @OneToMany(mappedBy = "order")
    private List<Product> products;

    public Order() {□}

    public Order(Date orderDate) {
        this.orderDate = orderDate;
    }

    public Integer getId() {□}

    public void setId(Integer id) {□}
```

Retrieving Entities With Bidirectional Mapping

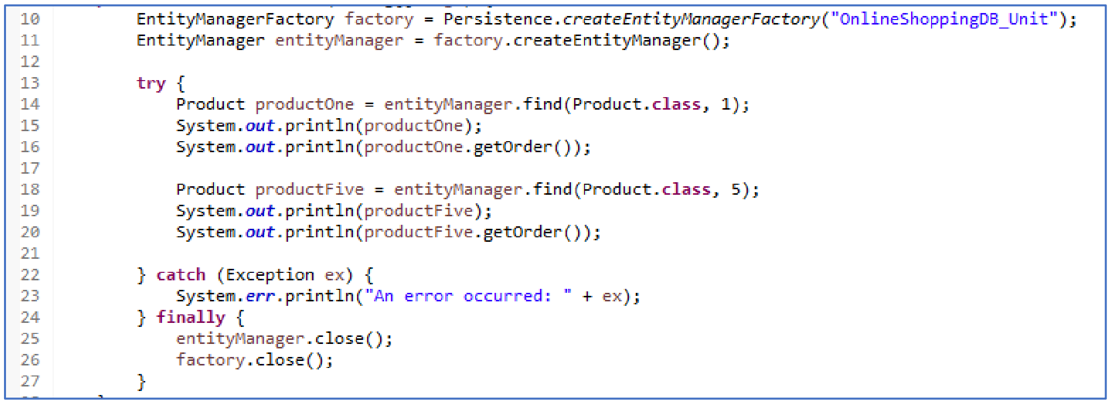
In this demo, we'll see how retrieving entities from the underlying database table works. Using our one-to-many, many-to-one bidirectional mapping.

Here I am on the **persistence.xml** file. I'll change the database action to be equal to none instead of drop-and-create. We'll work with the tables that we had set up in the previous demo.



```
<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="none"/>

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

Now switch over to **App.java** and let's write some code to retrieve database entities. If you remember in our bidirectional mapping, the `Product` side which is the many side of the relationship *is the owning side*.

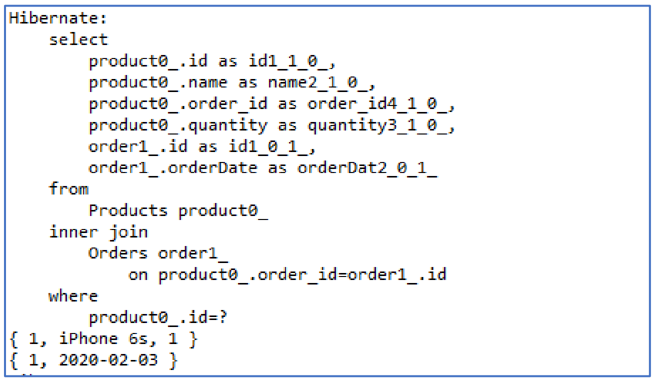


```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {
            Product productOne = entityManager.find(Product.class, 1);
            System.out.println(productOne);
            System.out.println(productOne.getOrder());

            Product productFive = entityManager.find(Product.class, 5);
            System.out.println(productFive);
            System.out.println(productFive.getOrder());

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

- I'm first going to use the `entityManager` to retrieve `Products`. I call `entityManager.find()` for `productOne` and `productFive`.
- And I'll print out the contents of a product to screen as well as the order corresponding to each `Product`.

Once again to emphasize, the entity that we retrieve here is on the many side of the relationship, it also happens to be the owning side.

Go ahead and run this code and let's take a look at the select statements that Hibernate has run under the hood.



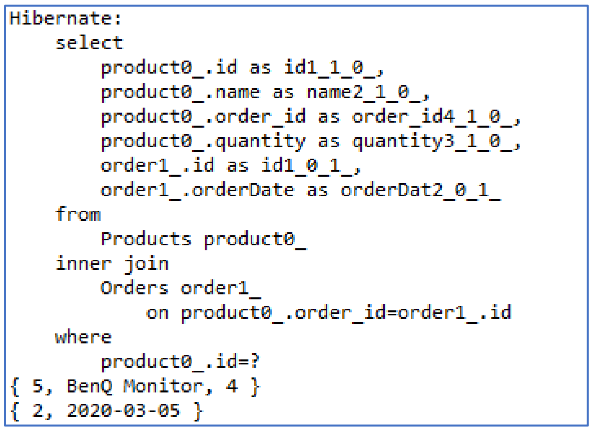
```
Hibernate:
    select
        product0_.id as id1_1_0_,
        product0_.name as name2_1_0_,
        product0_.order_id as order_id4_1_0_,
        product0_.quantity as quantity3_1_0_,
        order1_.id as id1_0_1_,
        order1_.orderDate as orderDat2_0_1_
    from
        Products product0_
    inner join
        Orders order1_
            on product0_.order_id=order1_.id
    where
        product0_.id=?
{ 1, iPhone 6s, 1 }
{ 1, 2020-02-03 }
```

Notice that when the information corresponding to a `Product` is selected. The same select statement retrieves information about the corresponding order as well.

This is because *when you **retrieve from the many side** of a relationship, the entity on the one side is **eagerly loaded***. So when we retrieve products, the order will be **eagerly loaded**.

If you look at the from clause, you'll see that there is a **join performed** to retrieve the `Order` corresponding to a `Product`. And here are the details of the product with `id` 1 and the corresponding `Order`, the "iPhone 6S" bought on "2020-02-03".

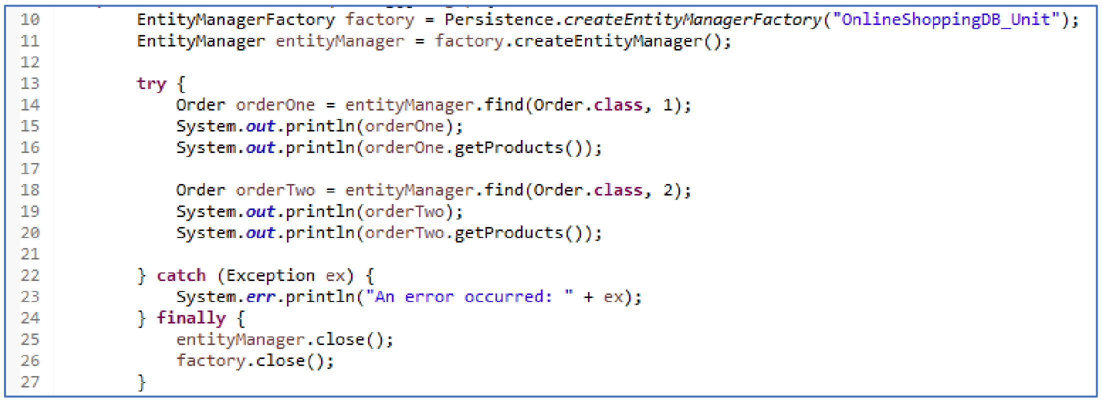
If you scroll further down you'll see the select statement for the retrieval of the second `Product` with `id` 5.



```
Hibernate:
    select
        product0_.id as id1_1_0_,
        product0_.name as name2_1_0_,
        product0_.order_id as order_id4_1_0_,
        product0_.quantity as quantity3_1_0_,
        order1_.id as id1_0_1_,
        order1_.orderDate as orderDat2_0_1_
    from
        Products product0_
    inner join
        Orders order1_
            on product0_.order_id=order1_.id
    where
        product0_.id=?
{ 5, BenQ Monitor, 4 }
{ 2, 2020-03-05 }
```

Along with the `Product`, the corresponding `Order` information is also retrieved using a **join operation**. And here is the `Product` with `id` 5, "BenQ Monitors" corresponding to `orderTwo`.

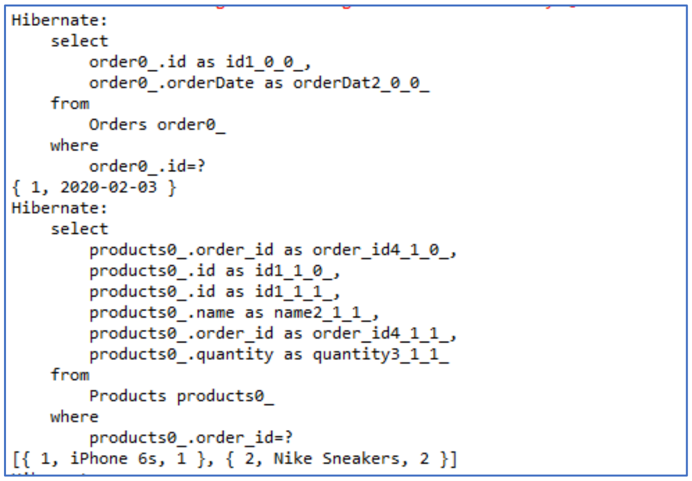
I'm now going to tweak and change my retrieval operation just a little bit. I am going to retrieve `Orders` instead of `Products`.



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

        try {
            Order orderOne = entityManager.find(Order.class, 1);
            System.out.println(orderOne);
            System.out.println(orderOne.getProducts());

            Order orderTwo = entityManager.find(Order.class, 2);
            System.out.println(orderTwo);
            System.out.println(orderTwo.getProducts());

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

Using `entityManager.find()`, I'm going to retrieve the `Order` with `id` : 1 and `id` : 2. After having retrieved each `Order` I'll print out the contents of the `Order`.
I'll then invoke `getProducts()` and print out the `Products` associated with each `Order`.

Let's go ahead and run this code and you'll see that the SQL statements are a little different. Here is the select statement where we retrieve an `Order` with a particular `id`.



```
Hibernate:
    select
        order0_.id as id1_0_0_,
        order0_.orderDate as orderDat2_0_0_
    from
        Orders order0_
    where
        order0_.id=?
{ 1, 2020-02-03 }
Hibernate:
    select
        products0_.order_id as order_id4_1_0_,
        products0_.id as id1_1_0_,
        products0_.id as id1_1_1_,
        products0_.name as name2_1_1_,
        products0_.order_id as order_id4_1_1_,
        products0_.quantity as quantity3_1_1_
    from
        Products products0_
    where
        products0_.order_id=?
[{ 1, iPhone 6s, 1 }, { 2, Nike Sneakers, 2 }]
```

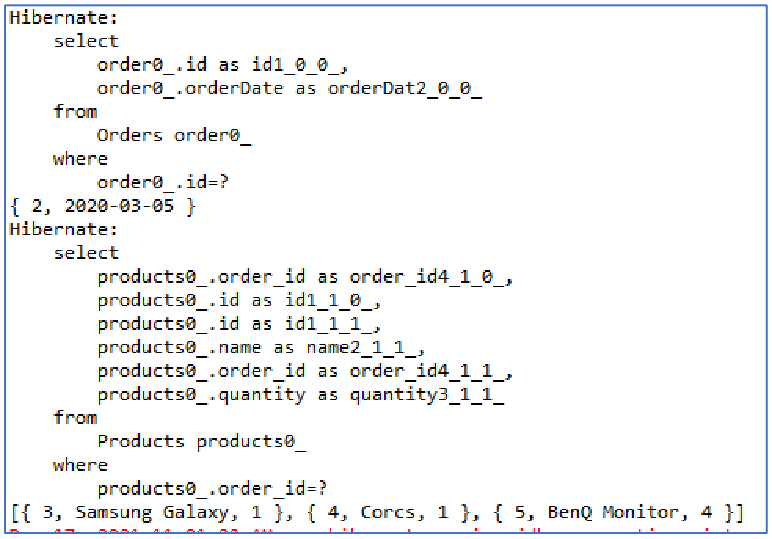
Notice that we don't immediately retrieve the `Products` information. `Products` is the many side of the relationship.

This is typically **lazily loaded**, not eagerly loaded.

Because we are explicitly accessing the products. Hibernate performs a second operation here to select the `Products` associated with my `Order`.

And if you scroll down further, “iPhone” and “Nike sneakers” are the two `Products` in `orderOne`.

And here below is the second select corresponding to `Order` with `id` : 2.



```
Hibernate:
    select
        order0_.id as id1_0_0_,
        order0_.orderDate as orderDat2_0_0_
    from
        Orders order0_
    where
        order0_.id=?
{ 2, 2020-03-05 }
Hibernate:
    select
        products0_.order_id as order_id4_1_0_,
        products0_.id as id1_1_0_,
        products0_.id as id1_1_1_,
        products0_.name as name2_1_1_,
        products0_.order_id as order_id4_1_1_,
        products0_.quantity as quantity3_1_1_
    from
        Products products0_
    where
        products0_.order_id=?
[{ 3, Samsung Galaxy, 1 }, { 4, Corcs, 1 }, { 5, BenQ Monitor, 4 }]
```

And with a further scroll you'll see the `Product` information associated with this `Order` is also retrieved using a select statement.

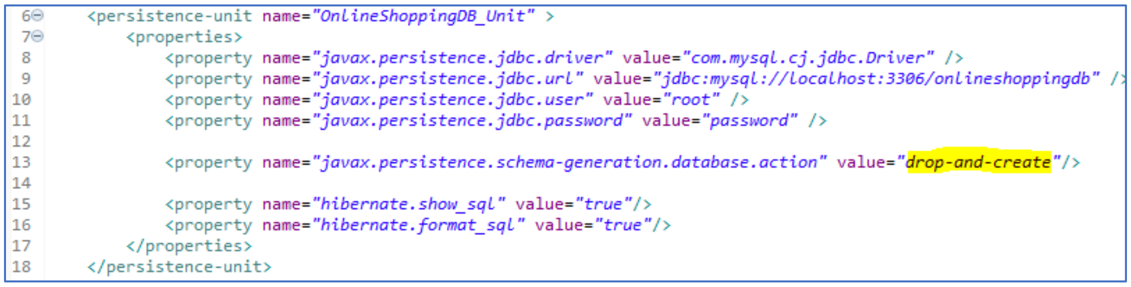
And the three products associated with order number 2 is displayed onto screen.

The entities on **the many side** of a relationship **are by default, lazily loaded** unless you specify otherwise.

Configuring the Owning Side and Owned Side

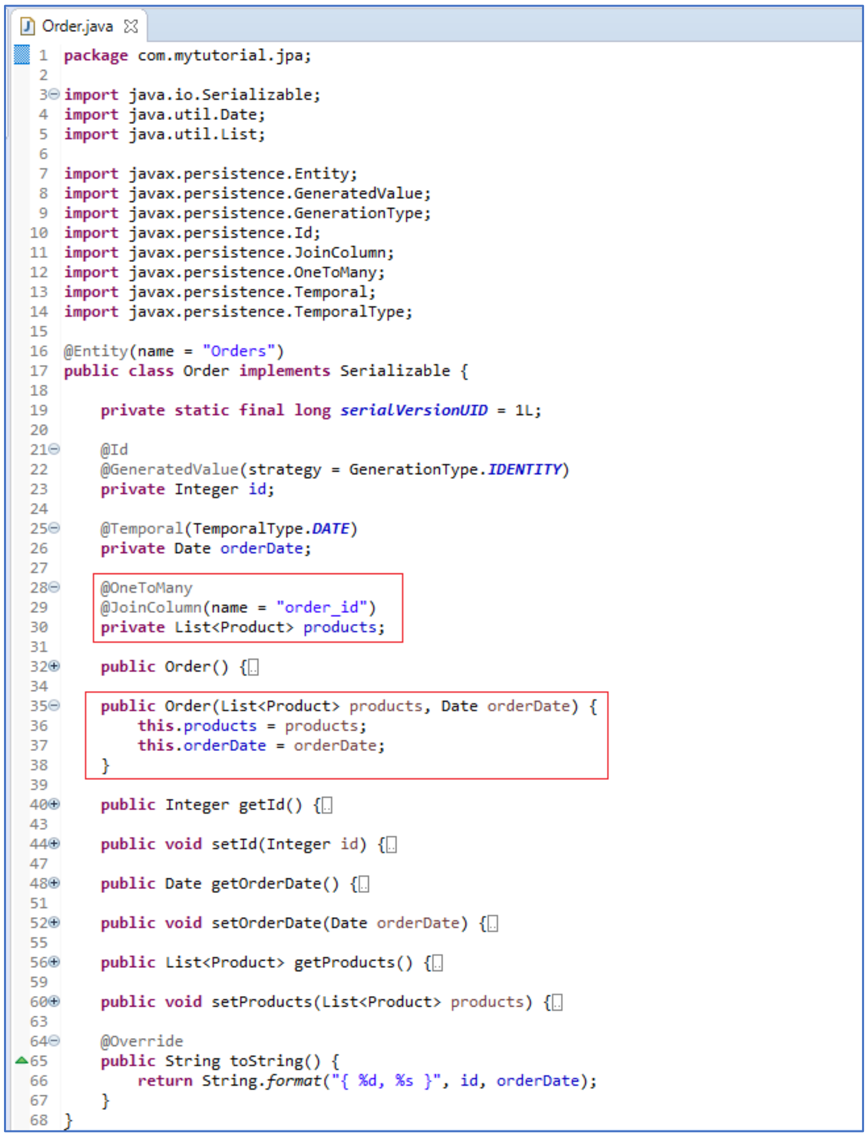
In this demo, we'll once again set up a bidirectional mapping, one-to-many, many-to-one. But this time we'll have the one side of the relationship be the owning entity.

Update the database action in your **persistence.xml** to drop-and-create. We want our tables to be recreated when the application runs.



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

In order to set up this updated bidirectional mapping, I'm going to first head over to the **Order.java** file. Order is the one side of this mapping relationship. One `Order` maps to multiple `Products`.

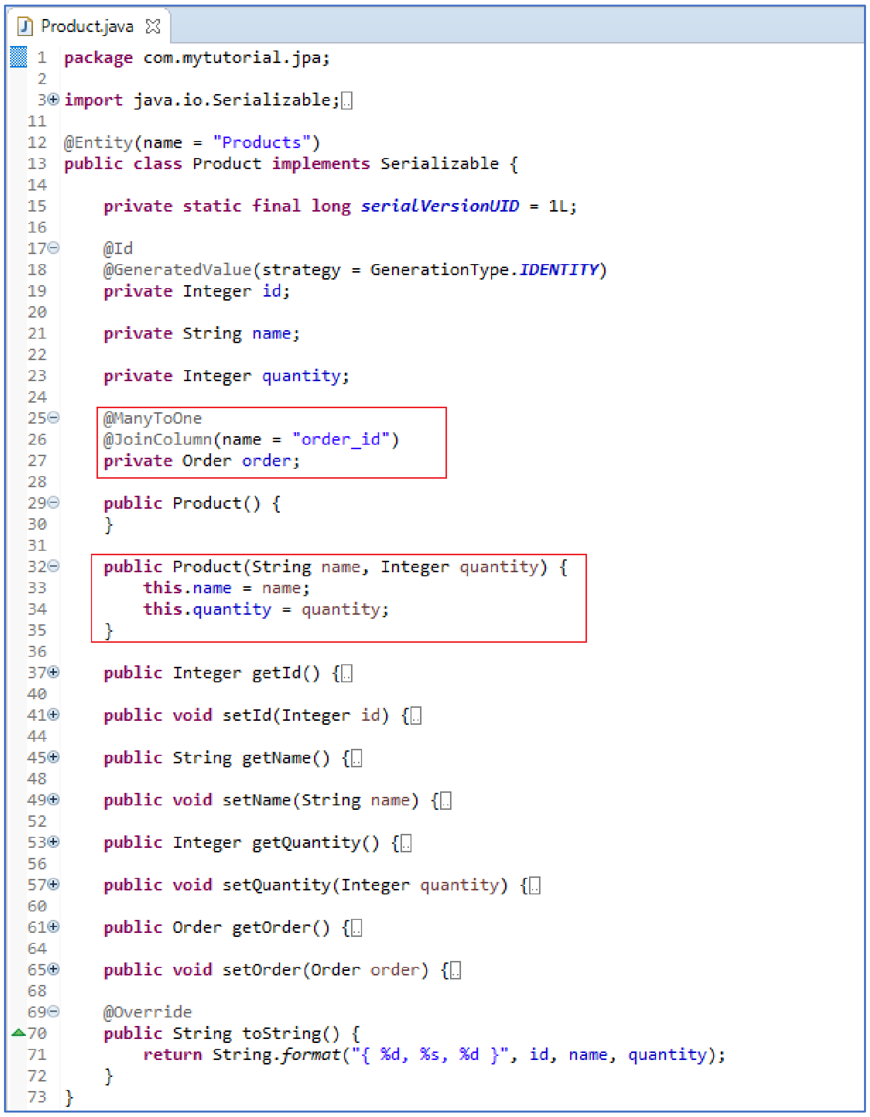


Order.java

```
package com.mytutorial.jpa;

import java.io.Serializable;
import java.util.Date;
import java.util.List;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToMany;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity(name = "Orders")
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;

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

    @Temporal(TemporalType.DATE)
    private Date orderDate;

    @OneToMany
    @JoinColumn(name = "order_id")
    private List<Product> products;

    public Order() {

    public Order(List<Product> products, Date orderDate) {
        this.products = products;
        this.orderDate = orderDate;
    }

    public Integer getId() {

    public void setId(Integer id) {

    public Date getOrderDate() {

    public void setOrderDate(Date orderDate) {

    public List<Product> getProducts() {

    public void setProducts(List<Product> products) {

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

- Set up the import for the join column here.
  `import javax.persistence.JoinColumn;`
- And update the annotation that we've applied to the list of products associated with an order. So we now have a **@OneToMany** annotation as well as an **@JoinColumn** annotation.
- Because we've removed the **mappedBy** property, this is no longer the inverse side of the relationship. It becomes the owning side, **@OneToMany @JoinColumn** and the *name* of the join column is `"order_id"`. This is the name of the **foreign key** column in the `Products` table that will reference the `Order` associated with each `Product`. `Order` now is the owning entity.
- Now in order to persist entities where `Order` is the owning entity, I need to make one more change here. And that is to the **constructor** for an Order object.
  I'm going to pass in the `List` of `products` associated with the `Order` into this **constructor**.
  When we persist entities, we want to ensure that an order has a reference to all the products associated with that order.

This completes the changes that we need to make to the Order.java file. Let's head over to **Product.java**.

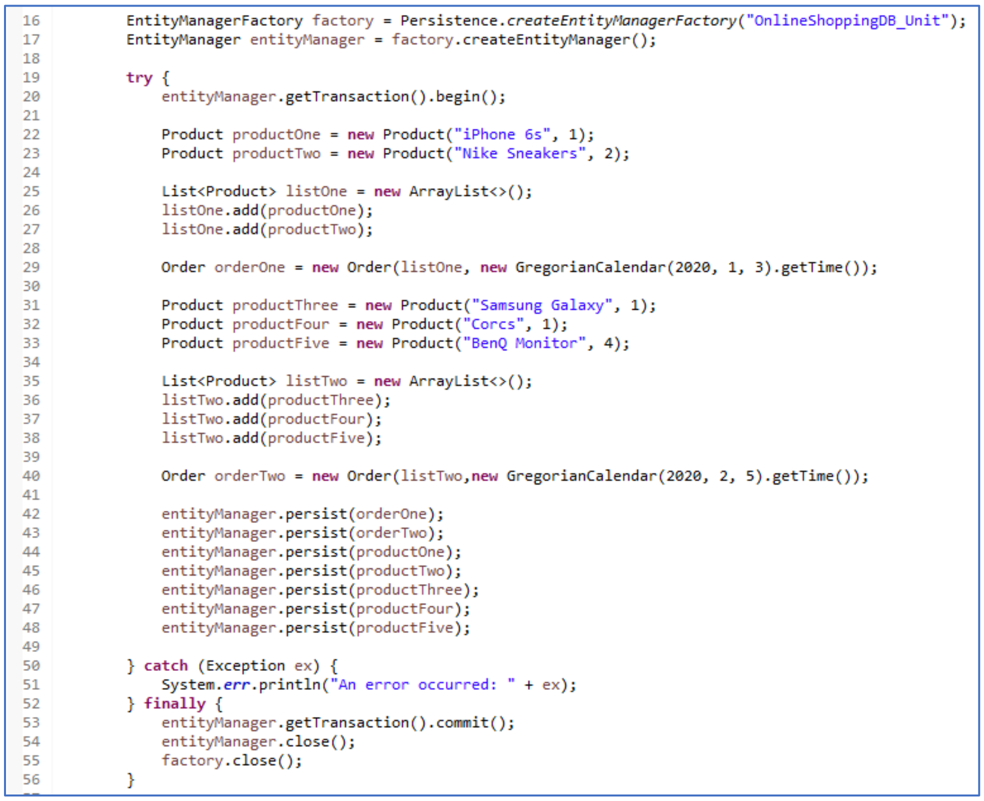


Product.java

```
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 Integer quantity;

    @ManyToOne
    @JoinColumn(name = "order_id")
    private Order order;

    public Product() {
    }

    public Product(String name, Integer quantity) {
        this.name = name;
        this.quantity = quantity;
    }

    public Integer getId() {

    public void setId(Integer id) {

    public String getName() {

    public void setName(String name) {

    public Integer getQuantity() {

    public void setQuantity(Integer quantity) {

    public Order getOrder() {

    public void setOrder(Order order) {

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

- I'm going to update the annotation applied to the order reference within a `Product`. `Product` is the many side of the relationship mapping.
- So we have the **@ManyToOne** annotation as we did before.
- We also have the **@JoinColumn** annotation where we specify the name of the JoinColumn within the `Products` table. `order_id` is a **foreign key reference** to the primary key of the `Order` associated with each `Product`.
- One last change that we'll make here. Let's update the **constructor** for the `Product` so that a `Product` need not have a reference to the corresponding `Order` when we save out entities.

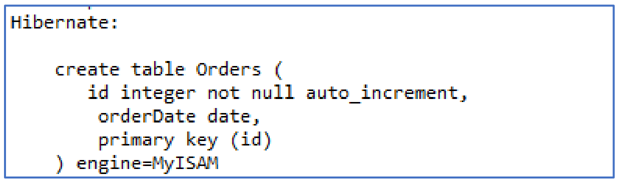
Having made these changes to our entity files, let's head over to **App.java**. And set up a transaction to persist `Orders` and corresponding `Products`.



```
16          EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
17          EntityManager entityManager = factory.createEntityManager();
18
19          try {
20              entityManager.getTransaction().begin();
21
22              Product productOne = new Product("iPhone 6s", 1);
23              Product productTwo = new Product("Nike Sneakers", 2);
24
25              List<Product> listOne = new ArrayList<>();
26              listOne.add(productOne);
27              listOne.add(productTwo);
28
29              Order orderOne = new Order(listOne, new GregorianCalendar(2020, 1, 3).getTime());
30
31              Product productThree = new Product("Samsung Galaxy", 1);
32              Product productFour = new Product("Corcs", 1);
33              Product productFive = new Product("BenQ Monitor", 4);
34
35              List<Product> listTwo = new ArrayList<>();
36              listTwo.add(productThree);
37              listTwo.add(productFour);
38              listTwo.add(productFive);
39
40              Order orderTwo = new Order(listTwo,new GregorianCalendar(2020, 2, 5).getTime());
41
42              entityManager.persist(orderOne);
43              entityManager.persist(orderTwo);
44              entityManager.persist(productOne);
45              entityManager.persist(productTwo);
46              entityManager.persist(productThree);
47              entityManager.persist(productFour);
48              entityManager.persist(productFive);
49
50          } catch (Exception ex) {
51              System.err.println("An error occurred: " + ex);
52          } finally {
53              entityManager.getTransaction().commit();
54              entityManager.close();
55              factory.close();
56          }
```

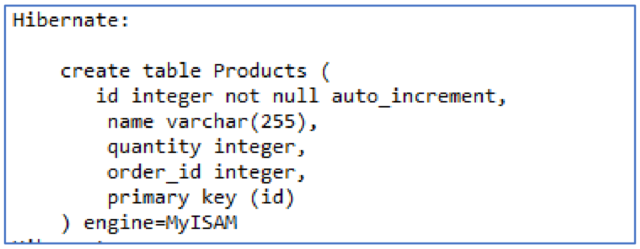
- We have `orderOne` that is associated with the product iPhone and Nike.
- Since order is the owning entity, observe on line 29 when we instantiate an order object, we pass in the list of `Products` associated with the `Order`.
- We then set up three `Products` associated with `orderTwo`.
- On line 40, when we create the `orderTwo` entity, we pass in `listTwo`, setting up a reference from the `Order` to the three `Products` associated with that `Order`. This is because order is the owning entity.
- We then call `entityManager.persist()` to persist two `Orders` and five `Products` to our underlying database tables.

We are now ready to run this code and see how this bidirectional mapping works. Where, the one side, that is order is the owning entity. Here is the `Orders` table as created by Hibernate.



```
Hibernate: 

    create table Orders (
       id integer not null auto_increment,
        orderDate date,
        primary key (id)
    ) engine=MyISAM
```

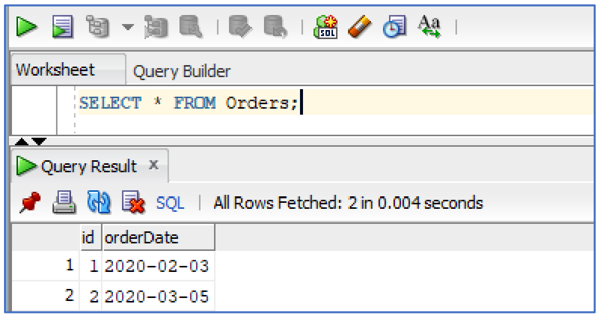
And here is the Products table as created by Hibernate.



```
Hibernate: 

    create table Products (
       id integer not null auto_increment,
        name varchar(255),
        quantity integer,
        order_id integer,
        primary key (id)
    ) engine=MyISAM
```

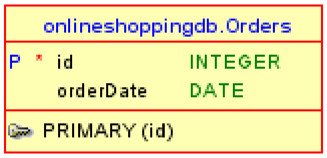
Notice that there is an order_id column here, which has a **foreign key reference** to the **primary key** in the Orders table.

We can always confirm that things have been set up correctly using MySQL Workbench.
Let's do a *select ** on the *Orders* table.



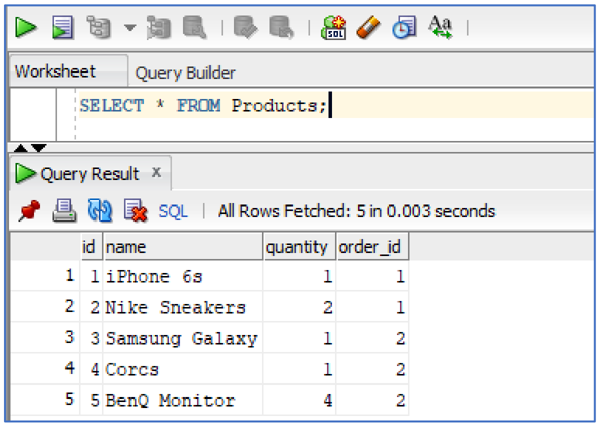
```
SELECT * FROM Orders;
```

	id	orderDate
1	1	2020-02-03
2	2	2020-03-05



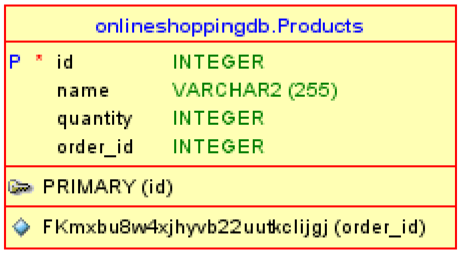
Here are the two orders that we had set up and persisted.

Let's do a *select ** on the *Products* table.



```
SELECT * FROM Products;
```

	id	name	quantity	order_id
1	1	iPhone 6s	1	1
2	2	Nike Sneakers	2	1
3	3	Samsung Galaxy	1	2
4	4	Corcs	1	2
5	5	BenQ Monitor	4	2

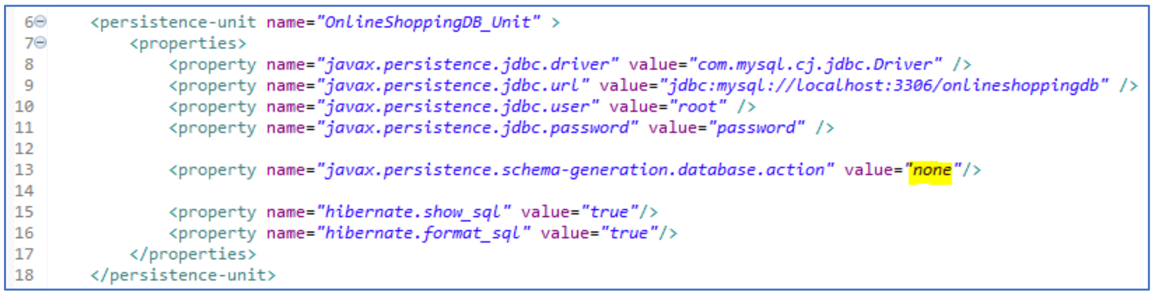


Here are the five Products where every Product is associated with an Order.

Thanks to the order_id column here with the **foreign key constraint**, we've set up a bidirectional mapping with order as the owning entity.

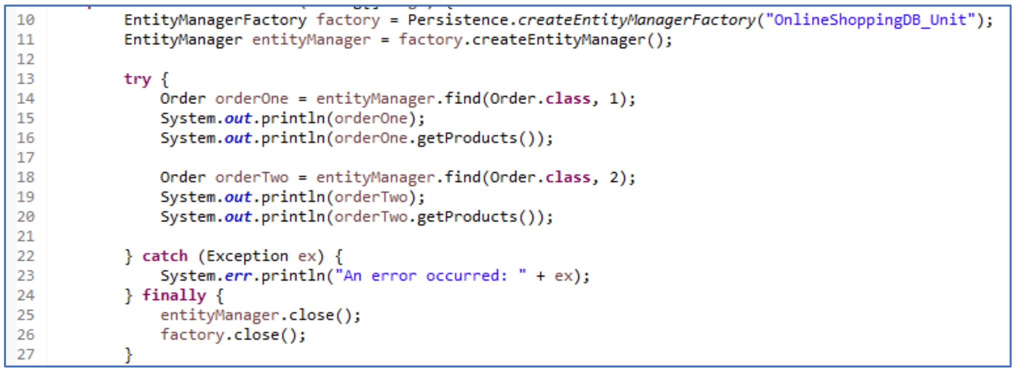
Before we complete this demo on bidirectional mapping, let's quickly see how we can retrieve entities now that we have set up the owning side differently. Order, that is the one side of the mapping is the owning side.

Let's update the database action to be equal to none, so that we don't recreate the underlying tables. We'll work with the tables that we've already created.



```
    <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="none"/>

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

I'll now head over to **App.java** and write code to retrieve entities using the find method.

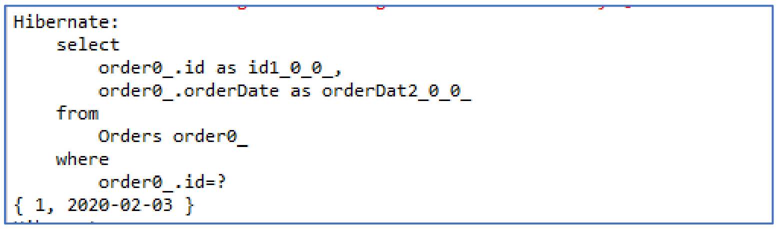


```
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {
            Order orderOne = entityManager.find(Order.class, 1);
            System.out.println(orderOne);
            System.out.println(orderOne.getProducts());

            Order orderTwo = entityManager.find(Order.class, 2);
            System.out.println(orderTwo);
            System.out.println(orderTwo.getProducts());

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

- I'll first retrieve the order entities. Remember, this is the owning side of the relationship.
- We'll retrieve the `Order` with id 1, and the order with id 2.
- For each order retrieved, we'll print out the `Order` details and the `Products`.

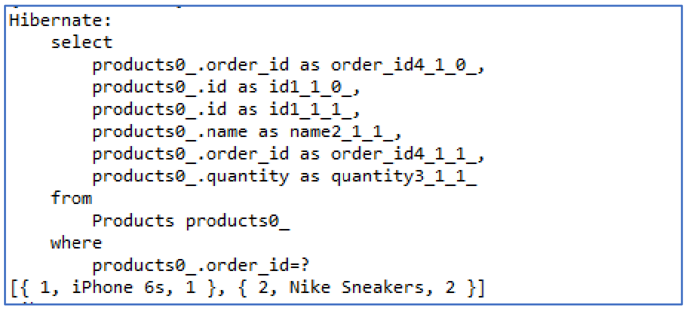
Because this is one-to-many mapping, remember that the many side of the relationship is always retrieved lazily by default.

Let's run this code and I'll show you using the select statements. Here is the select statement to retrieve a particular `Order`.



```
Hibernate:
    select
        order0_.id as id1_0_0_,
        order0_.orderDate as orderDat2_0_0_
    from
        Orders order0_
    where
        order0_.id=?
{ 1, 2020-02-03 }
```

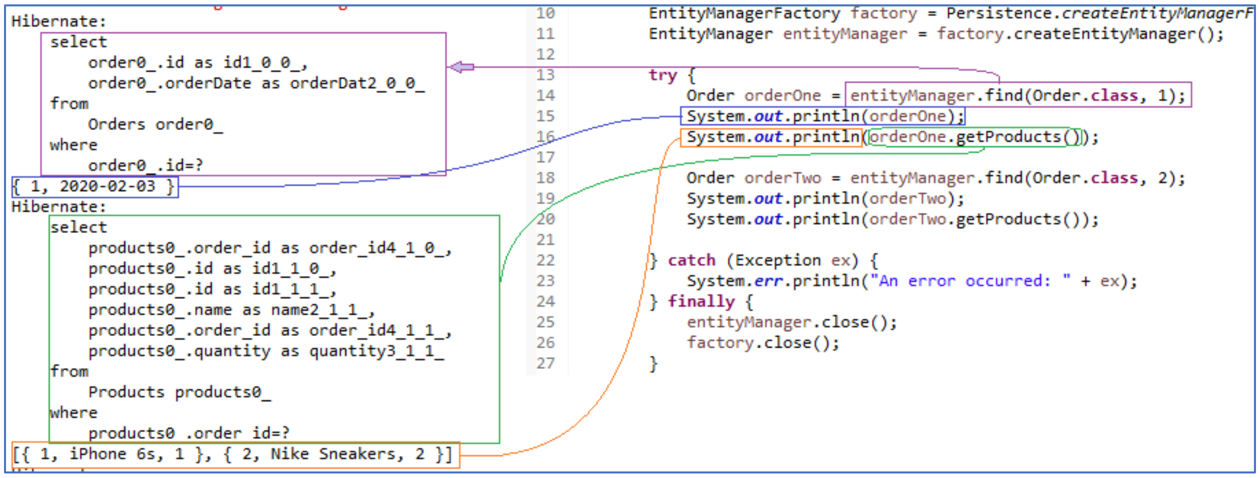
We only retrieve order details when we use the find method. Later, when we call `getProducts()`,



```
Hibernate:
    select
        products0_.order_id as order_id4_1_0_,
        products0_.id as id1_1_0_,
        products0_.id as id1_1_1_,
        products0_.name as name2_1_1_,
        products0_.order_id as order_id4_1_1_,
        products0_.quantity as quantity3_1_1_
    from
        Products products0_
    where
        products0_.order_id=?
[{ 1, iPhone 6s, 1 }, { 2, Nike Sneakers, 2 }]
```

we retrieve the `Products` associated with that `Order` as well.

And that is a separate select statement as you can see here. And here we are the `Products` associated with `orderOne`, the iPhone and Nike Sneakers.

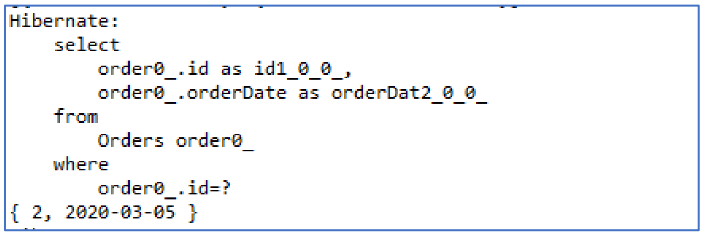


```
Hibernate:
    select
        order0_.id as id1_0_0_,
        order0_.orderDate as orderDat2_0_0_
    from
        Orders order0_
    where
        order0_.id=?
{ 1, 2020-02-03 }
Hibernate:
    select
        products0_.order_id as order_id4_1_0_,
        products0_.id as id1_1_0_,
        products0_.id as id1_1_1_,
        products0_.name as name2_1_1_,
        products0_.order_id as order_id4_1_1_,
        products0_.quantity as quantity3_1_1_
    from
        Products products0_
    where
        products0_.order_id=?
[{ 1, iPhone 6s, 1 }, { 2, Nike Sneakers, 2 }]
```

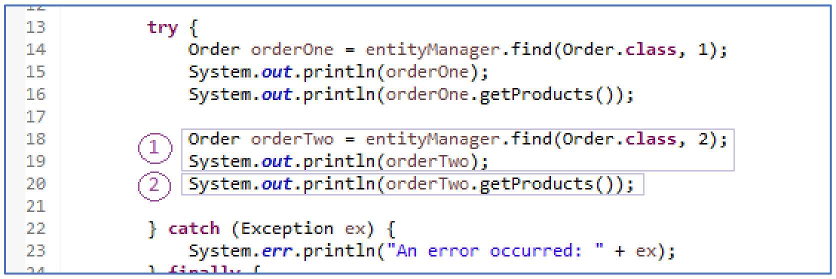
```java
        EntityManagerFactory factory = Persistence.createEntityManagerF
        EntityManager entityManager = factory.createEntityManager();

        try {
            Order orderOne = entityManager.find(Order.class, 1);
            System.out.println(orderOne);
            System.out.println(orderOne.getProducts());

            Order orderTwo = entityManager.find(Order.class, 2);
            System.out.println(orderTwo);
            System.out.println(orderTwo.getProducts());

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

Here is a select statement to retrieve the details for `orderTwo`.

1) We only retrieve `Order` information and `Product` information is retrieved lazily.

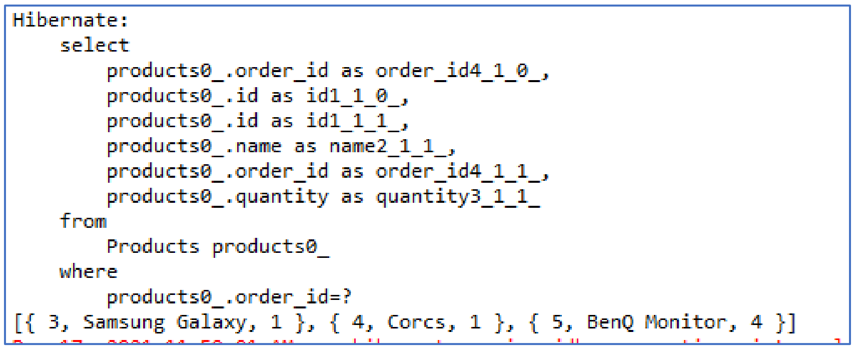


```
Hibernate:
    select
        order0_.id as id1_0_0_,
        order0_.orderDate as orderDat2_0_0_
    from
        Orders order0_
    where
        order0_.id=?
{ 2, 2020-03-05 }
```



```
        try {
            Order orderOne = entityManager.find(Order.class, 1);
            System.out.println(orderOne);
            System.out.println(orderOne.getProducts());

            Order orderTwo = entityManager.find(Order.class, 2);
            System.out.println(orderTwo);
            System.out.println(orderTwo.getProducts());

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

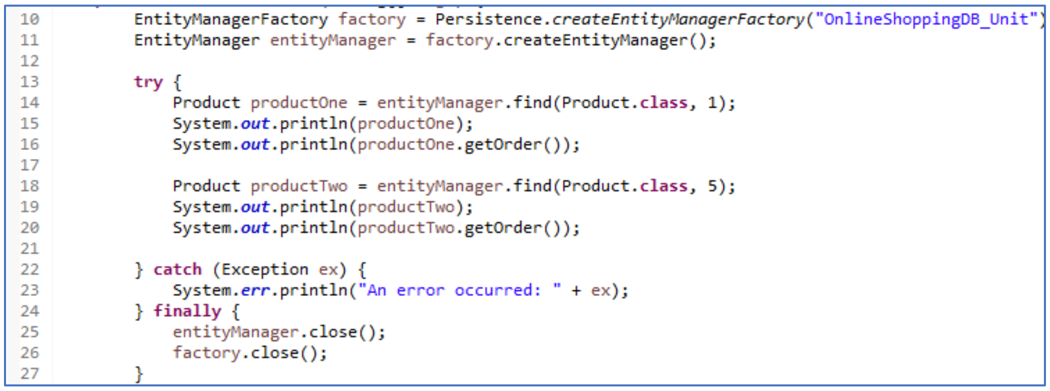
2) When we call `getProducts()`, that's when the product information is retrieved using the separate SQL statement.



```
Hibernate:
    select
        products0_.order_id as order_id4_1_0_,
        products0_.id as id1_1_0_,
        products0_.id as id1_1_1_,
        products0_.name as name2_1_1_,
        products0_.order_id as order_id4_1_1_,
        products0_.quantity as quantity3_1_1_
    from
        Products products0_
    where
        products0_.order_id=?
[{ 3, Samsung Galaxy, 1 }, { 4, Corcs, 1 }, { 5, BenQ Monitor, 4 }]
```

And finally, here are the three `Products` associated with `orderTwo`, the “Samsung Galaxy”, “Crocs”, and “BenQ monitors”.

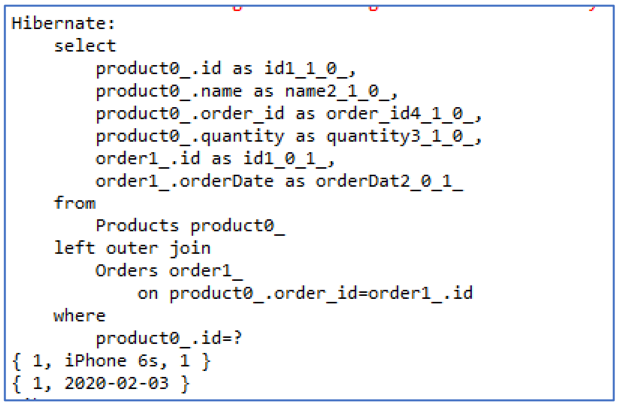
I'm now going to update my retrieval code so I retrieve products.



```
10          EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit")
11          EntityManager entityManager = factory.createEntityManager();
12
13          try {
14              Product productOne = entityManager.find(Product.class, 1);
15              System.out.println(productOne);
16              System.out.println(productOne.getOrder());
17
18              Product productTwo = entityManager.find(Product.class, 5);
19              System.out.println(productTwo);
20              System.out.println(productTwo.getOrder());
21
22          } catch (Exception ex) {
23              System.err.println("An error occurred: " + ex);
24          } finally {
25              entityManager.close();
26              factory.close();
27          }
```

- This will give us the referenced order because order is the one side of this mapping relationship that is fetched **eagerly**.
- I use `entityManager.find()` to find the `Product` with `id` 1, print out the product and the associated order. This is on lines 15 and 16.
- We then find the `Product` with `id` 5. And print out the product and its associated order as well.

Time to execute this code.



```
Hibernate:
    select
        product0_.id as id1_1_0_,
        product0_.name as name2_1_0_,
        product0_.order_id as order_id4_1_0_,
        product0_.quantity as quantity3_1_0_,
        order1_.id as id1_0_1_,
        order1_.orderDate as orderDat2_0_1_
    from
        Products product0_
    left outer join
        Orders order1_
            on product0_.order_id=order1_.id
    where
        product0_.id=?
{ 1, iPhone 6s, 1 }
{ 1, 2020-02-03 }
```

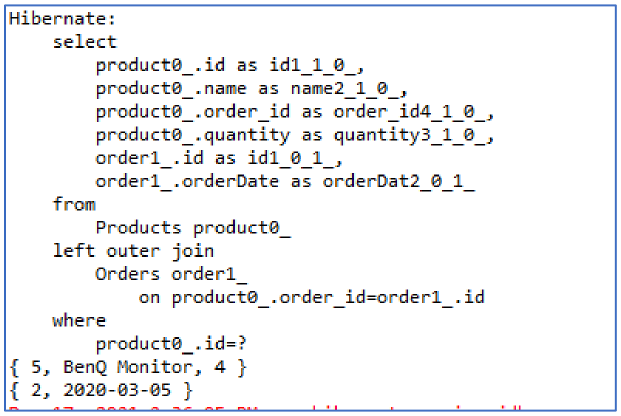
Here is the select statement for product information.

Notice the same select retrieves order information as well.

In the from clause you can see the **join operation** that we perform to retrieve `Order` information along with the product.

And here below are the `Product` and `Order` details for the "iPhone 6S" printed out to screen.

The same is true when we retrieve our second `Product`, the one with id 5.



```
Hibernate:
    select
        product0_.id as id1_1_0_,
        product0_.name as name2_1_0_,
        product0_.order_id as order_id4_1_0_,
        product0_.quantity as quantity3_1_0_,
        order1_.id as id1_0_1_,
        order1_.orderDate as orderDat2_0_1_
    from
        Products product0_
    left outer join
        Orders order1_
            on product0_.order_id=order1_.id
    where
        product0_.id=?
{ 5, BenQ Monitor, 4 }
{ 2, 2020-03-05 }
```

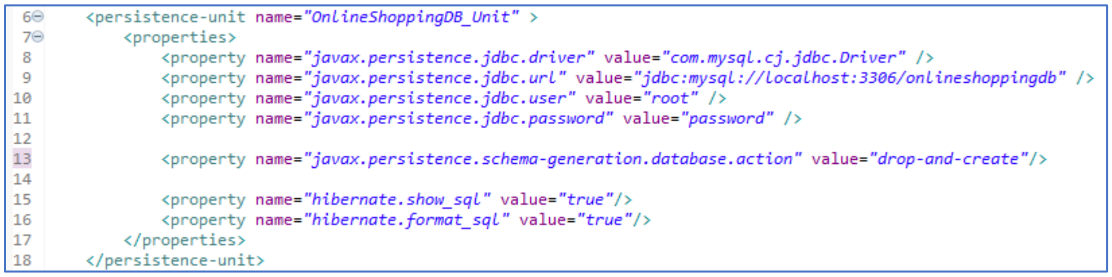
BenQ monitors, `Product` details, and `Order` details are retrieved together.

`Order` details are retrieved eagerly.

Many-to-many Bidirectional Mapping

We have one last kind of relationship mapping to explore before we are done with mapping. This is the many-to-many relationship mapping and that's exactly what we'll see here in this demo.

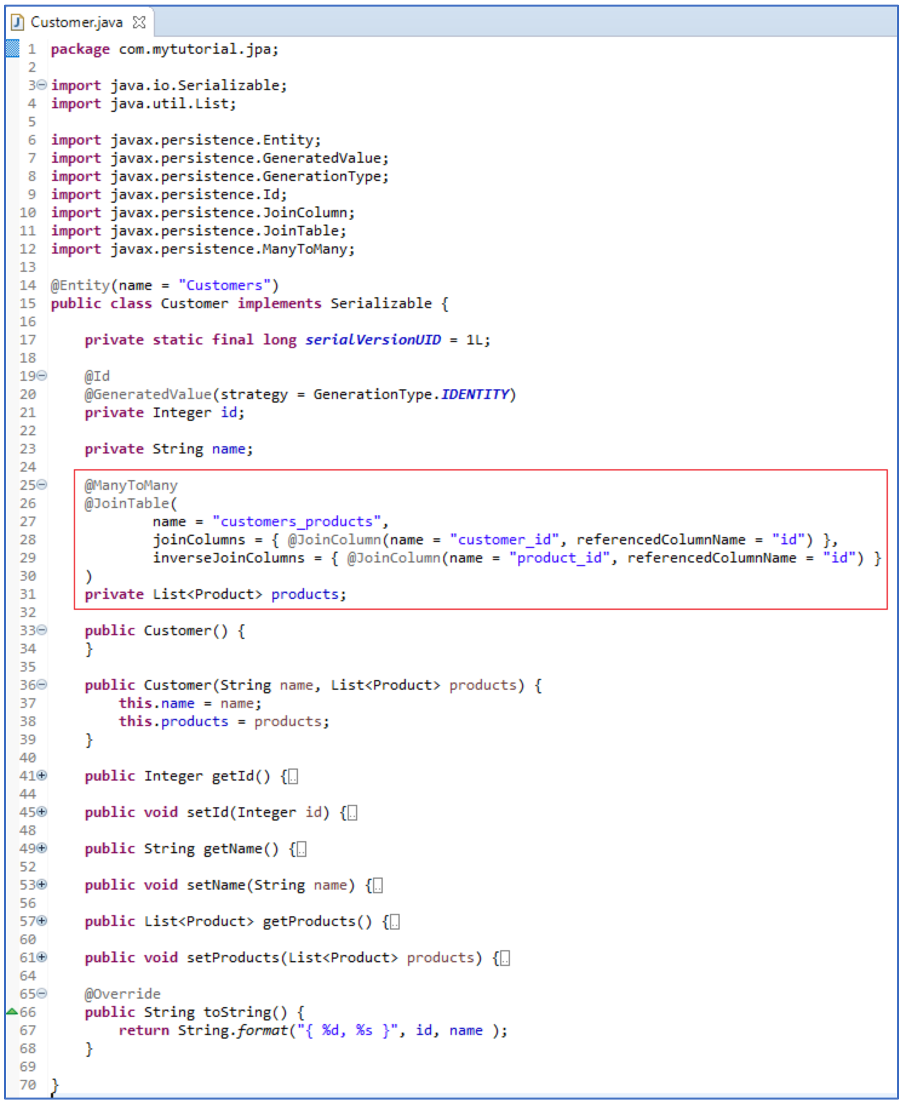
Make sure that within your **persistence.xml**, your database.action is set to drop-and-create. We're still working within our online shopping database.



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

In order to study many-to-many relationship maps, I'm going to create a new class called **Customer**. That's because a customer in our e-commerce site can buy many products, and the same product can be bought by many customers. That is a many-to-many relationship.

Here is our Customer class.



```java
package com.mytutorial.jpa;

import java.io.Serializable;
import java.util.List;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.ManyToMany;

@Entity(name = "Customers")
public class Customer implements Serializable {

    private static final long serialVersionUID = 1L;

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

    private String name;

    @ManyToMany
    @JoinTable(
            name = "customers_products",
            joinColumns = { @JoinColumn(name = "customer_id", referencedColumnName = "id") },
            inverseJoinColumns = { @JoinColumn(name = "product_id", referencedColumnName = "id") }
    )
    private List<Product> products;

    public Customer() {
    }

    public Customer(String name, List<Product> products) {
        this.name = name;
        this.products = products;
    }

    public Integer getId() {

    public void setId(Integer id) {

    public String getName() {

    public void setName(String name) {

    public List<Product> getProducts() {

    public void setProducts(List<Product> products) {

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

}
```

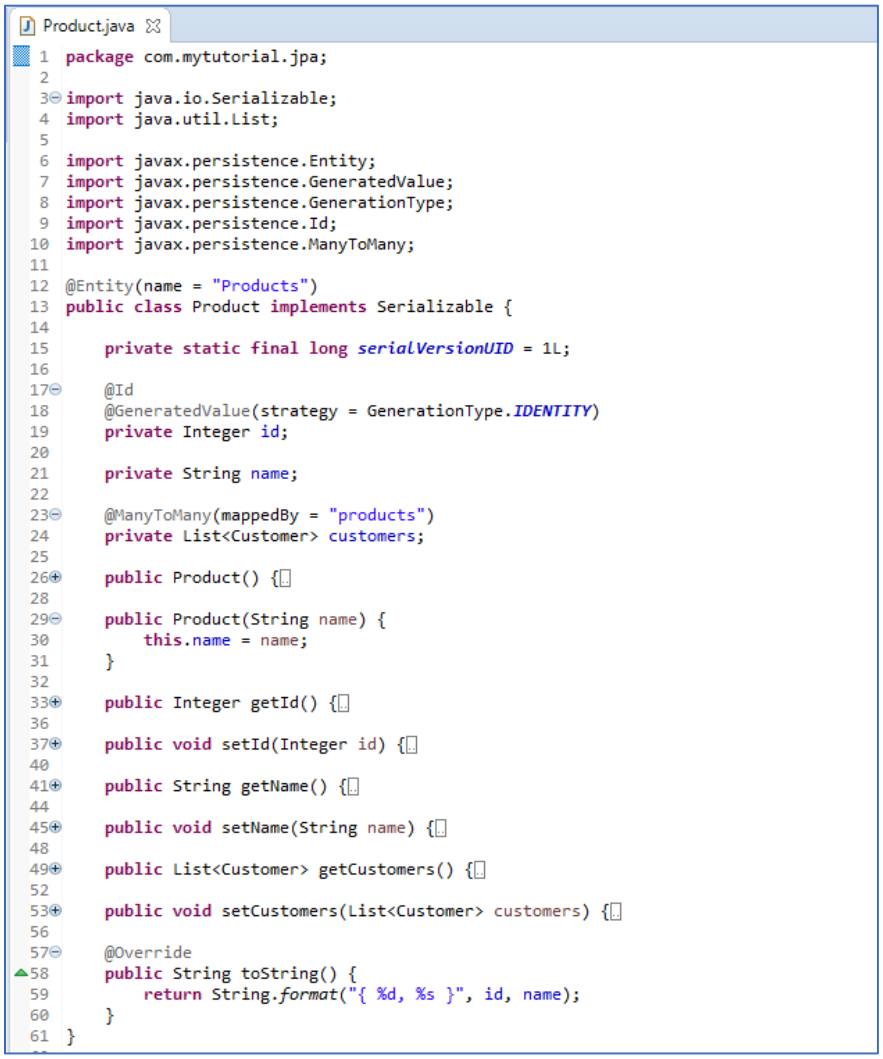
- Set up the import statements for the various annotations in this class.
- Make sure that `Customer` implements **Serializable**.
- `Customer` is an entity, that is, we'll insert `Customer` records in the underlying table.
  So I annotate this with the **@Entity** annotation and specify the name of the underlying table as `"Customers"`.
- Next, I'll set up the member variables of this `Customer` class that will map to the columns in the `Customers` table. `id` is the **primary key**, and I use the generation strategy ***`IDENTITY`***.
- We also have the `name` of the customer.
- Now, any `Customer` in our e-commerce site may have purchased any number of `products`. So we have a `List` of `products` here which define all of the products associated with a particular customer entity.
  Because single customer may have purchased multiple products and any product could have been bought by any number of customers, this is a many-to-many relationship.

- This `List` of `products` have annotated with the **@ManyToMany** annotation.
- Now, `Customer` is the owning side of this many-to-many relationship. And here is where I specify the configuration of the join table that will hold this many to many mapping.
- Many-to-many mapping relationships are always modeled using the join table. I have the **@JoinTable** annotation here. The name of the join table is `"customers_products"`.
- Within the **@JoinTable** annotation, I specify the **joinColumns**.
  The **joinColumns** are what will set up the references between `Customers` and `Products`.
- Within the **joinColumns** property as specified in **@JoinColumn**, The name of the column in the join table is `"customer_id"`. And this column in the join table references the `id` of the owning entity that is the `Customer` entity.
- I've also specified the **inverseJoinColumns** property using the **@JoinColumn** annotation once again. The name of the column in the join table is `"product_id"`. And this product ID column references the primary key `id` of the `Products` table.

  This many-to-many relationship will specify as a bidirectional relationship with the `Customer` as the owning entity.

- The remaining code here within Customer.java is the same, **getters** and **setters** for the various member variables.

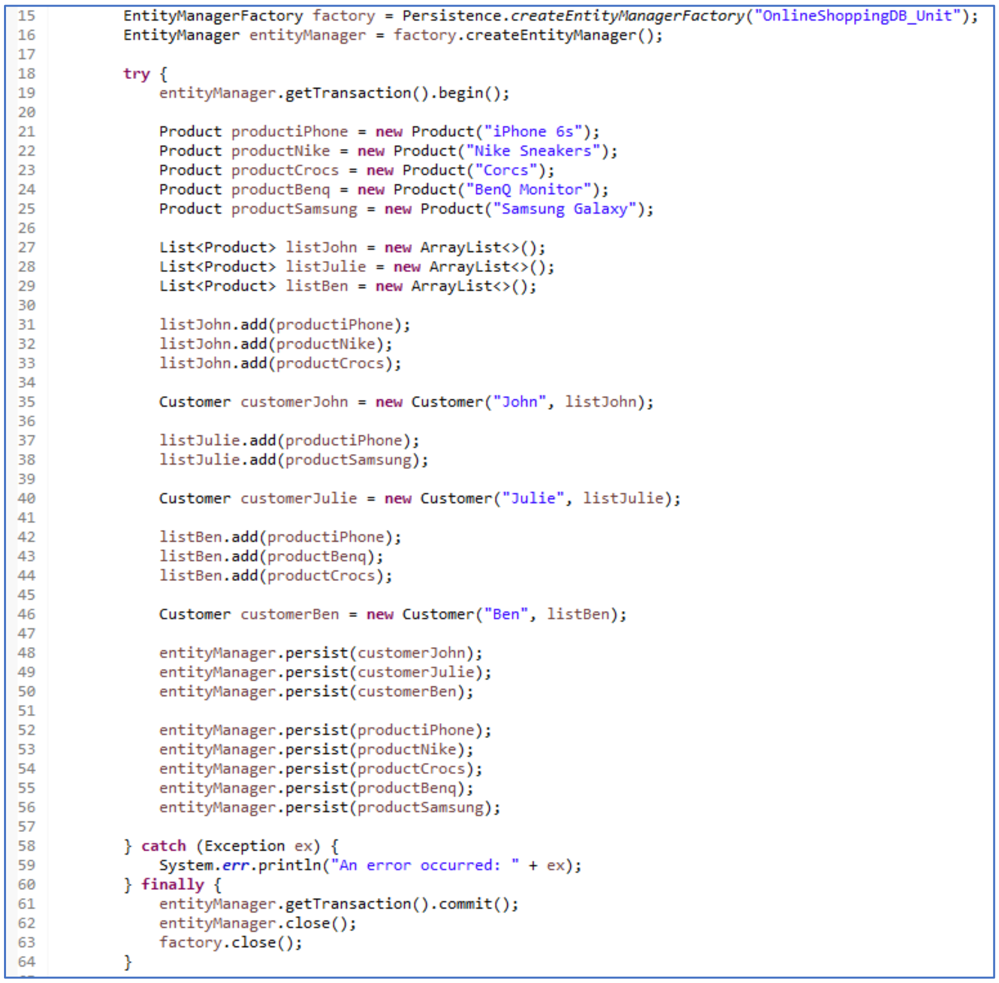
Let's now head over to **Product.java**.



```java
package com.mytutorial.jpa;

import java.io.Serializable;
import java.util.List;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToMany;

@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;

    @ManyToMany(mappedBy = "products")
    private List<Customer> customers;

    public Product() {

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

    public Integer getId() {

    public void setId(Integer id) {

    public String getName() {

    public void setName(String name) {

    public List<Customer> getCustomers() {

    public void setCustomers(List<Customer> customers) {

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

- Set up the import statements for the various annotations and decorate the `Product` class using the **@Entity** annotation.
- Make sure it's **Serializable**. The name of the underlying table is `"Products"`.
- We'll now specify the member variables which are the attributes of the `Product` and map to columns in the products table. We have the `id` and the `name` of every product, and we have a `List` of `customers`. This references all of the `Customer` entities that have bought this particular `Product`. This reference to customers from the `Product` objects sets up the bidirectional mapping.
- I use the **@ManyToMany** annotation on this `List` of `customers` and I specify the **mappedBy** property. The presence of this **mappedBy** property indicates that this is the **inverse side** of the relationship, not the owning side.
- The value here is `"products"` because `products` is the name of the member variable within the `Customers` object which references the `List` of `products`.
- The rest of the code here is familiar to us, **getters** and **setters** for all of the member variables here within this `Products` class.

We are now ready to head over to **App.java**, and create `Customer` and `Product` entities to persist in our underlying database tables.

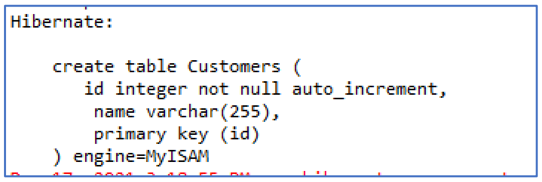


```
15          EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
16          EntityManager entityManager = factory.createEntityManager();
17
18          try {
19              entityManager.getTransaction().begin();
20
21              Product productiPhone = new Product("iPhone 6s");
22              Product productNike = new Product("Nike Sneakers");
23              Product productCrocs = new Product("Corcs");
24              Product productBenq = new Product("BenQ Monitor");
25              Product productSamsung = new Product("Samsung Galaxy");
26
27              List<Product> listJohn = new ArrayList<>();
28              List<Product> listJulie = new ArrayList<>();
29              List<Product> listBen = new ArrayList<>();
30
31              listJohn.add(productiPhone);
32              listJohn.add(productNike);
33              listJohn.add(productCrocs);
34
35              Customer customerJohn = new Customer("John", listJohn);
36
37              listJulie.add(productiPhone);
38              listJulie.add(productSamsung);
39
40              Customer customerJulie = new Customer("Julie", listJulie);
41
42              listBen.add(productiPhone);
43              listBen.add(productBenq);
44              listBen.add(productCrocs);
45
46              Customer customerBen = new Customer("Ben", listBen);
47
48              entityManager.persist(customerJohn);
49              entityManager.persist(customerJulie);
50              entityManager.persist(customerBen);
51
52              entityManager.persist(productiPhone);
53              entityManager.persist(productNike);
54              entityManager.persist(productCrocs);
55              entityManager.persist(productBenq);
56              entityManager.persist(productSamsung);
57
58          } catch (Exception ex) {
59              System.err.println("An error occurred: " + ex);
60          } finally {
61              entityManager.getTransaction().commit();
62              entityManager.close();
63              factory.close();
64          }
```

- We'll use the `EntityManager` as we've done so far. I'll first create my `Product` entities. I have five products here, the “IPhone”, “Nike Sneakers”, “Crocs”, “BenQ monitors” and the “Samsung Galaxy”.
- And I have five entities to correspond to each of these products. The `Customers` of our e-commerce site could have bought any of these products. The `List` of `Products` purchased by each customer I'm going to set up using a `List`, a `List` for “John”, “Julie” and “Ben”, these are three `ArrayLists()`.
- Now, John has bought the IPhone, Nike Sneakers and Crocs.
- I've added these three product entities to `listJohn` and I've instantiated a new `customerJohn` entity object and passed in this list as a reference.
- Notice that I'm referencing the `List` of `Products` from the `Customer` entity, because `Customer` is the owning side of this relationship.
- Next, on line 37, I populate the list of `Products` for “Julie”, the “IPhone” and “Samsung Galaxy”.
- I instantiate the `customerJulie` and pass in this `List`.

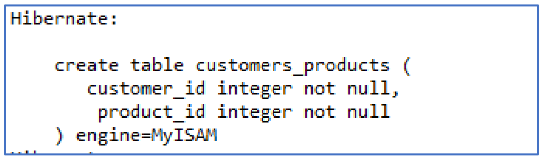
- Next, I populate what Ben has bought, the “IPhone”, “BenQ monitors” and “Crocs” and instantiate an entity for the `customerBen` on line 46.
- With our entities created and the references set up, we can call `entityManager.persist()` on all of our entity objects, three `Customers` and five `Products`.

Let's run this code and take a look at how this many-to-many bidirectional mapping is set up from `Customers` to `Products`.



```
Hibernate: 

    create table Customers (
       id integer not null auto_increment,
        name varchar(255),
        primary key (id)
    ) engine=MyISAM
```

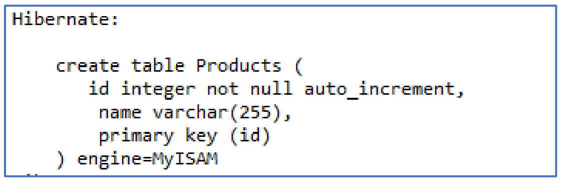
Here is the `Customers` table with `id` and `name`,
And here we have the `customers_products` table.



```
Hibernate: 

    create table customers_products (
       customer_id integer not null,
        product_id integer not null
    ) engine=MyISAM
```

- This is the join table that holds the many-to-many mapping from `Customers` to `Products`.
- The `customer_id` column holds `customer ids`,
- the `product_id` column holds `product  ids`, both of them are **not null**.

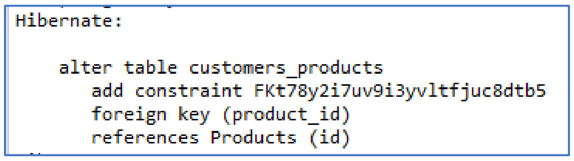
We have the `Products` table here that holds product information,



```
Hibernate: 

    create table Products (
       id integer not null auto_increment,
        name varchar(255),
        primary key (id)
    ) engine=MyISAM
```

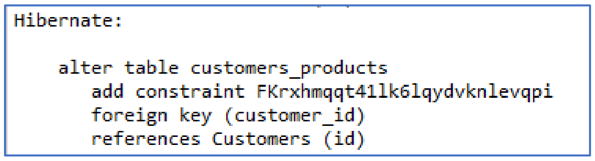
`id` is the primary key.

Notice that we set up a **foreign key constraint** where the `Product_id` from the `customers_products` table references the primary key of the `Products` table.



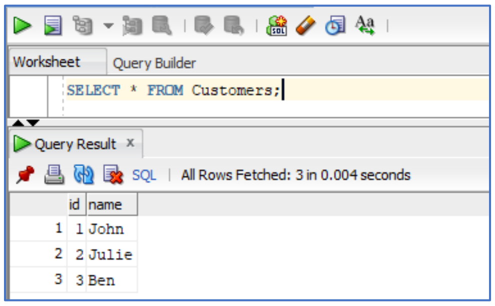
```
Hibernate: 

    alter table customers_products 
       add constraint FKt78y2i7uv9i3yvltfjuc8dtb5 
       foreign key (product_id) 
       references Products (id)
```

Here is another **foreign key constraint** where the `customer_id` in the join table that is `customers_products` references the primary key of the `Customers` table.



```
Hibernate: 

    alter table customers_products 
       add constraint FKrxhmqqt41lk6lqydvknlevqpi 
       foreign key (customer_id) 
       references Customers (id)
```

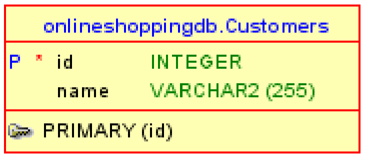
Now to verify our mapping entries in MySQL Workbench, let's run the *select ** on the *Customers* table.



```
SELECT * FROM Customers;
```

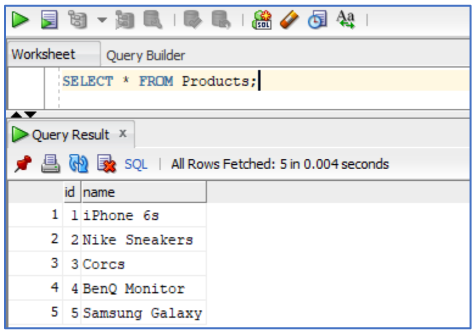
All Rows Fetched: 3 in 0.004 seconds

	id	name
1	1	John
2	2	Julie
3	3	Ben



```
onlineshoppingdb.Customers
P * id      INTEGER
    name    VARCHAR2 (255)
PRIMARY (id)
```

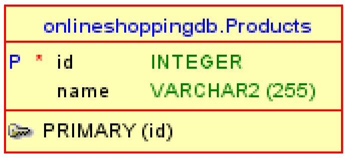
Here are the three `Customers`, "John", "Julie" and "Ben".
Let's run a *select ** on the *Products* table.



```
SELECT * FROM Products;
```

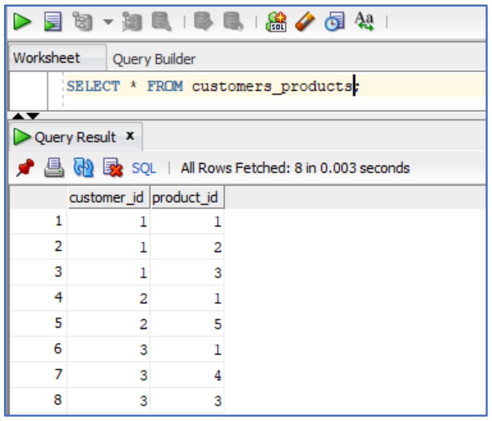
All Rows Fetched: 5 in 0.004 seconds

	id	name
1	1	iPhone 6s
2	2	Nike Sneakers
3	3	Corcs
4	4	BenQ Monitor
5	5	Samsung Galaxy



```
onlineshoppingdb.Products
P * id      INTEGER
    name    VARCHAR2 (255)
PRIMARY (id)
```

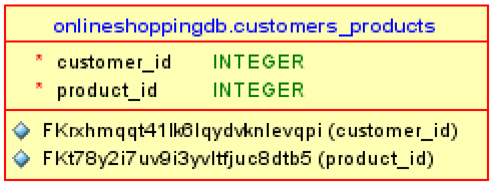
And this will give us the five `Products` that our `Customers` have bought on our e-commerce site. The mapping is set up in yet another table, *`customers_products`*. Running a *select ** will give us a mapping of which `customer` has bought which `product`.



```
SELECT * FROM customers_products;
```

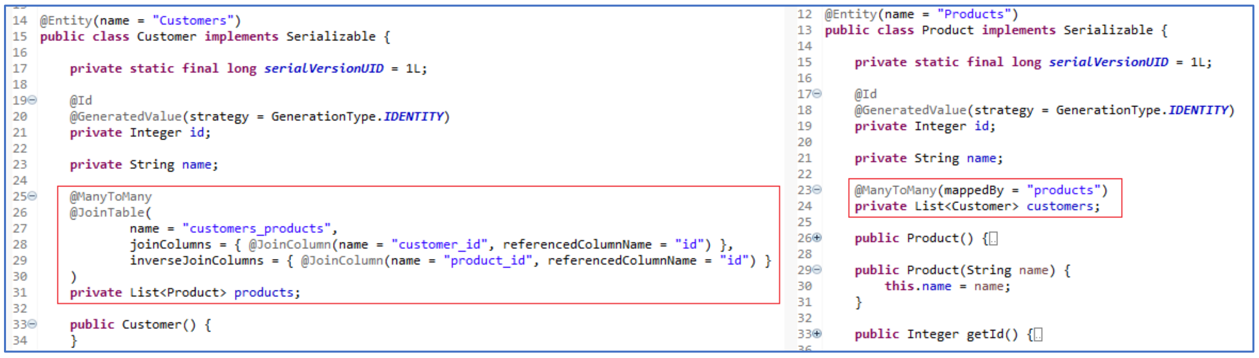
All Rows Fetched: 8 in 0.003 seconds

	customer_id	product_id
1	1	1
2	1	2
3	1	3
4	2	1
5	2	5
6	3	1
7	3	4
8	3	3



```
onlineshoppingdb.customers_products
* customer_id   INTEGER
* product_id    INTEGER
FKrxhmqqt41lk6lqydvknlevqpi (customer_id)
FKt78y2i7uv9i3yvltfjuc8dtb5 (product_id)
```

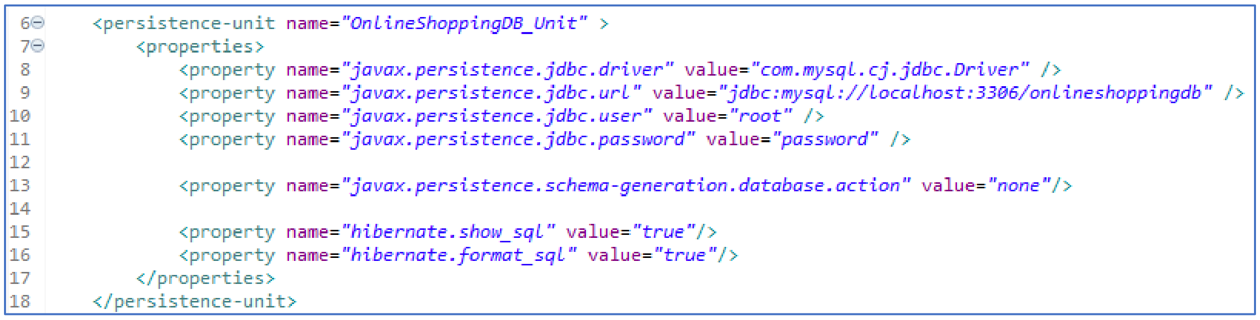
The `customer` with `id` 1 has bought `products` 1, 2 and 4. This refers to "John" who has bought the "IPhone", "Nike Sneakers" and "Crocs". The `customer` with id 2 has bought `products` with id 1 and 3.



```java
@Entity(name = "Customers")
public class Customer implements Serializable {

    private static final long serialVersionUID = 1L;

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

    private String name;

    @ManyToMany
    @JoinTable(
            name = "customers_products",
            joinColumns = { @JoinColumn(name = "customer_id", referencedColumnName = "id") },
            inverseJoinColumns = { @JoinColumn(name = "product_id", referencedColumnName = "id") }
    )
    private List<Product> products;

    public Customer() {
    }
```

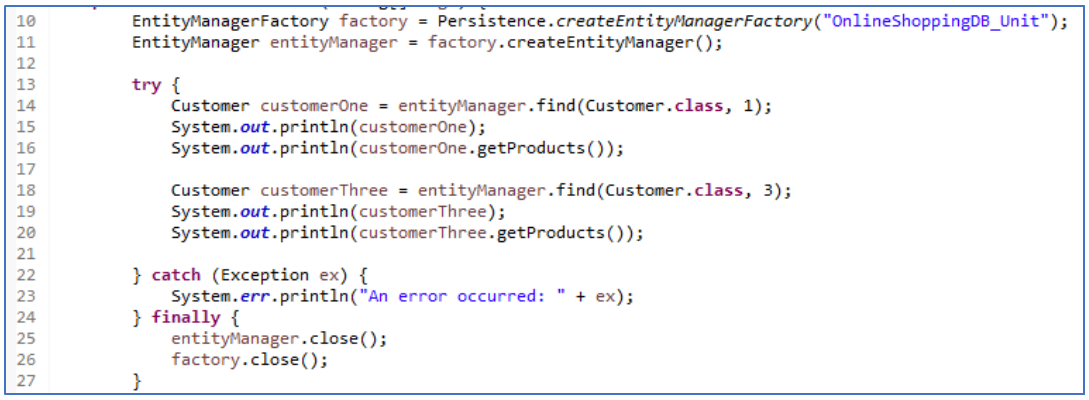
```java
@Entity(name = "Products")
public class Product implements Serializable {

    private static final long serialVersionUID = 1L;

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

    private String name;

    @ManyToMany(mappedBy = "products")
    private List<Customer> customers;

    public Product() {

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

    public Integer getId() {
```

Let's see how entity retrieval from the underlying database tables works for this bidirectional many-to-many mapping. I'm going to update the database action here to be equal to none so that we work with the tables that we have already created and don't create new ones.



```xml
    <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="none"/>

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

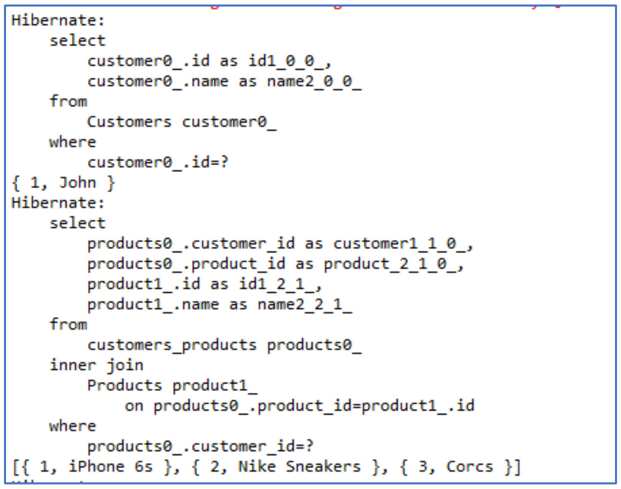
Let's head over to **App.java**.



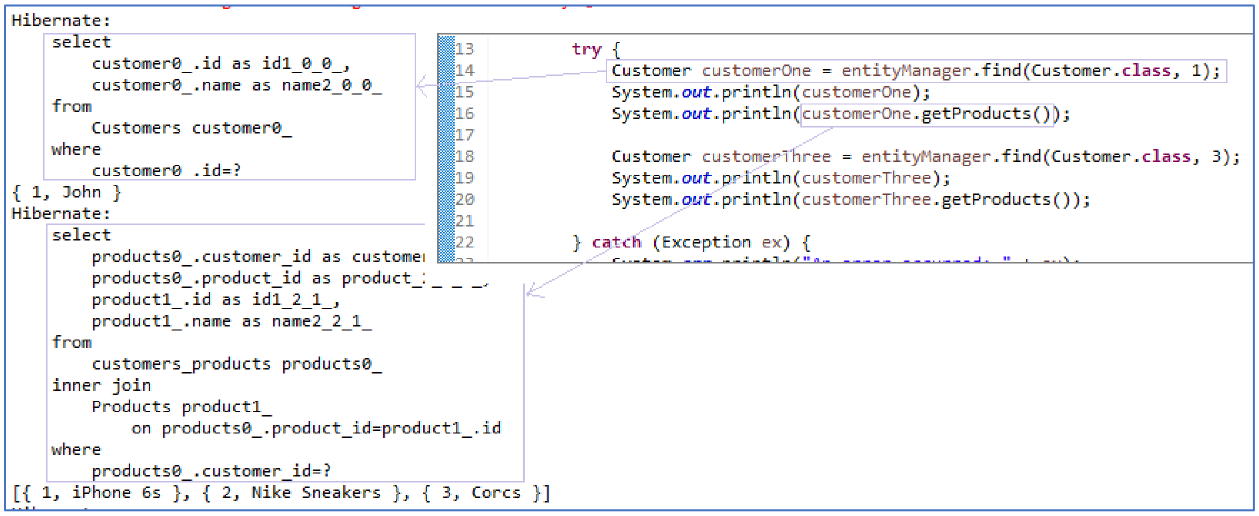
```java
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("OnlineShoppingDB_Unit");
        EntityManager entityManager = factory.createEntityManager();

        try {
            Customer customerOne = entityManager.find(Customer.class, 1);
            System.out.println(customerOne);
            System.out.println(customerOne.getProducts());

            Customer customerThree = entityManager.find(Customer.class, 3);
            System.out.println(customerThree);
            System.out.println(customerThree.getProducts());

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

- And here I'll use the find method to retrieve customer entities, `entityManager.find()` `Customer` with `id` 1,
- I'll print out the details of the customer,
- and then I'll invoke `getProducts()` to print out the products that a customer has bought.
- I'll similarly retrieve the customer with id 3,
- print out the details of the customer and the products bought by this customer.

Run this code and let's take a look at the SQL statements executed under the hood.



```
Hibernate: 
    select
        customer0_.id as id1_0_0_,
        customer0_.name as name2_0_0_ 
    from
        Customers customer0_ 
    where
        customer0_.id=?
{ 1, John }
Hibernate: 
    select
        products0_.customer_id as customer1_1_0_,
        products0_.product_id as product_2_1_0_,
        product1_.id as id1_2_1_,
        product1_.name as name2_2_1_ 
    from
        customers_products products0_ 
    inner join
        Products product1_ 
            on products0_.product_id=product1_.id 
    where
        products0_.customer_id=?
[{ 1, iPhone 6s }, { 2, Nike Sneakers }, { 3, Corcs }]
```



```
Hibernate:
    select
        customer0_.id as id1_0_0_,
        customer0_.name as name2_0_0_
    from
        Customers customer0_
    where
        customer0_.id=?
{ 1, John }
Hibernate:
    select
        products0_.customer_id as custome
        products0_.product_id as product_
        product1_.id as id1_2_1_,
        product1_.name as name2_2_1_
    from
        customers_products products0_
    inner join
        Products product1_
            on products0_.product_id=product1_.id
    where
        products0_.customer_id=?
[{ 1, iPhone 6s }, { 2, Nike Sneakers }, { 3, Corcs }]
```

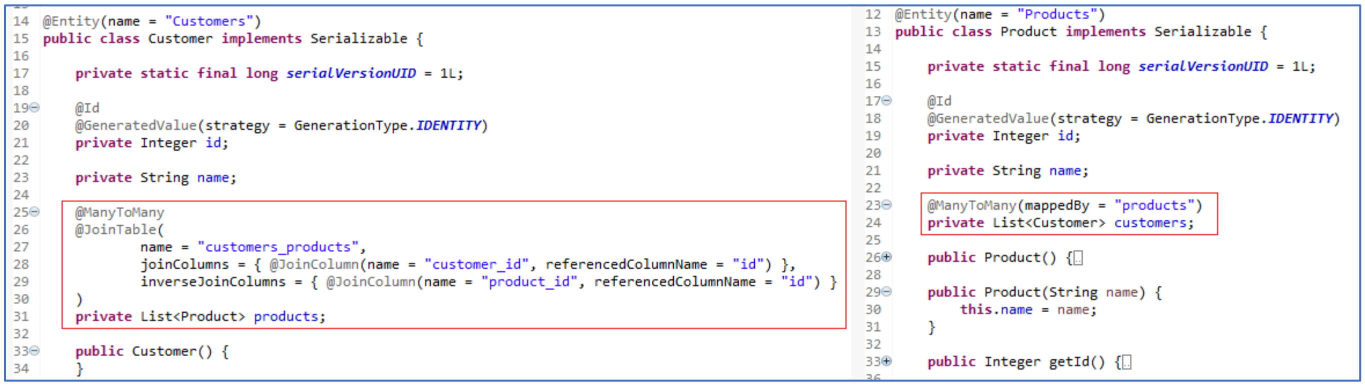
```
        try {
            Customer customerOne = entityManager.find(Customer.class, 1);
            System.out.println(customerOne);
            System.out.println(customerOne.getProducts());

            Customer customerThree = entityManager.find(Customer.class, 3);
            System.out.println(customerThree);
            System.out.println(customerThree.getProducts());

        } catch (Exception ex) {
```

1) Observe that in this many-to-many mapping, when we retrieve the `customer` entity, we perform only the select statement to retrieve `customer` details.

   In **many-to-many** mapping, by default, **the fetch type is always lazy**, we lazily load the products information when we retrieve customers.

2) After retrieving the `customer` John, we print out his details to screen and then Hibernate runs another select query to retrieve all `products` that John has bought.
   Here is the select query, which is run separately.
   This is lazy loading. The `products` that John has bought are retrieved only when we explicitly request these `products`.

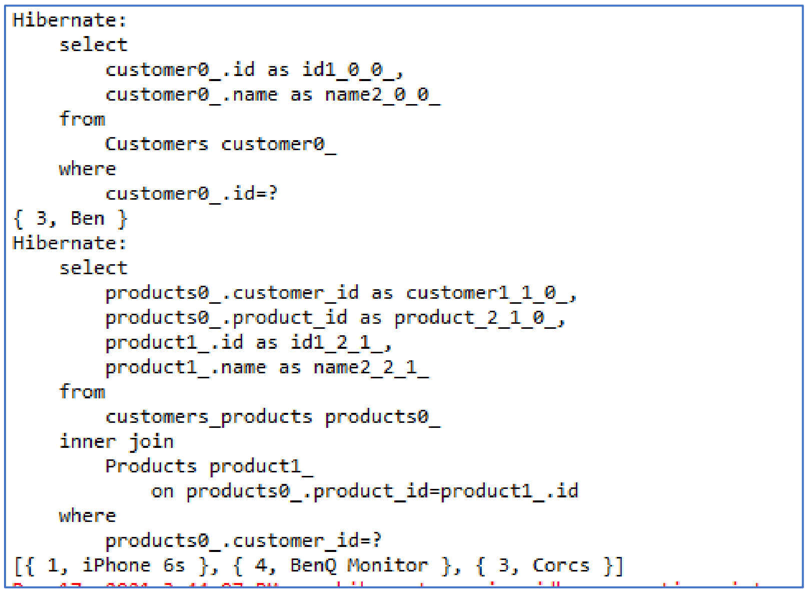
3) And here are the three products purchased by John, IPhone, Nike Sneakers and Crocs.



```
@Entity(name = "Customers")
public class Customer implements Serializable {

    private static final long serialVersionUID = 1L;

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

    private String name;

    @ManyToMany
    @JoinTable(
            name = "customers_products",
            joinColumns = { @JoinColumn(name = "customer_id", referencedColumnName = "id") },
            inverseJoinColumns = { @JoinColumn(name = "product_id", referencedColumnName = "id") }
    )
    private List<Product> products;

    public Customer() {
    }
```

```
@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;

    @ManyToMany(mappedBy = "products")
    private List<Customer> customers;

    public Product() {

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

    public Integer getId() {
```

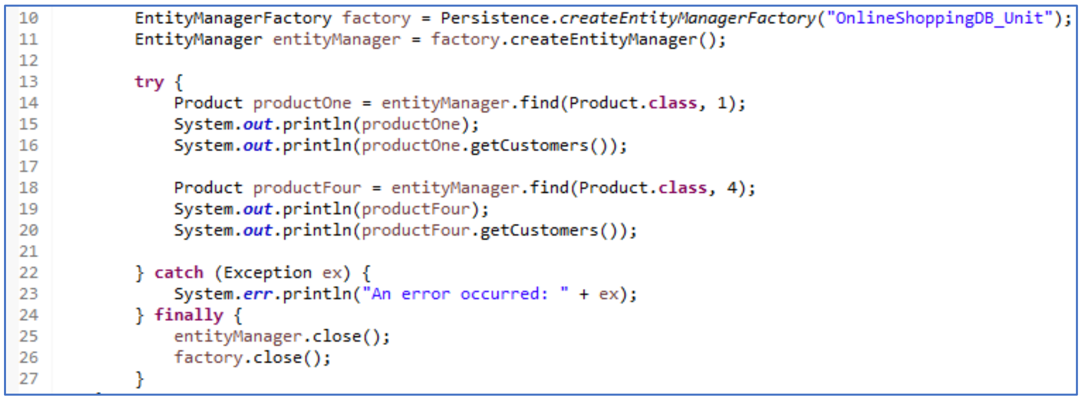
Let's take a look at the second select query here in order to retrieve the customer with id 3.



```
Hibernate:
    select
        customer0_.id as id1_0_0_,
        customer0_.name as name2_0_0_
    from
        Customers customer0_
    where
        customer0_.id=?
{ 3, Ben }
Hibernate:
    select
        products0_.customer_id as customer1_1_0_,
        products0_.product_id as product_2_1_0_,
        product1_.id as id1_2_1_,
        product1_.name as name2_2_1_
    from
        customers_products products0_
    inner join
        Products product1_
            on products0_.product_id=product1_.id
    where
        products0_.customer_id=?
[{ 1, iPhone 6s }, { 4, BenQ Monitor }, { 3, Corcs }]
```

1) Notice that only the customer details are retrieved first, and then we print out the details corresponding to Ben.
2) And then when we invoke the `getProducts()` method, Hibernate runs a separate query to retrieve all of the products purchased by Ben. Ben has purchased a total of three products here, the “IPhone”, “BenQ monitors” and “Crocs”.

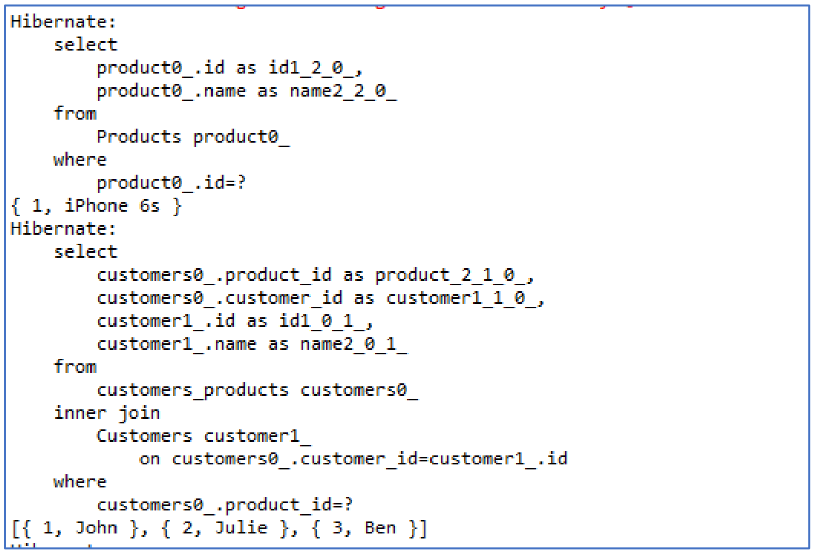
We can also retrieve products and see which customers have bought a particular product.



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

        try {
            Product productOne = entityManager.find(Product.class, 1);
            System.out.println(productOne);
            System.out.println(productOne.getCustomers());

            Product productFour = entityManager.find(Product.class, 4);
            System.out.println(productFour);
            System.out.println(productFour.getCustomers());

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

- Here is our retrieval code for the product with id 1 and the product with id 5.
- We print out the details of the product and we invoke `getCustomers()` to see which customers have bought each product.

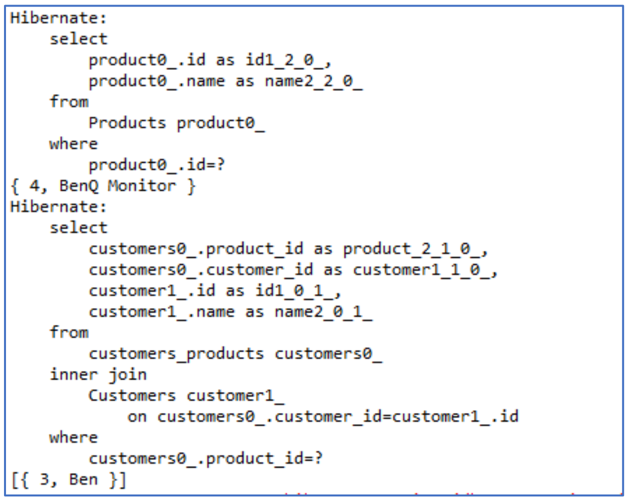
Now, once again with this many-to-many mapping, retrieving customers associated with the product is a lazy operation.



```
Hibernate: 
    select
        product0_.id as id1_2_0_,
        product0_.name as name2_2_0_ 
    from
        Products product0_ 
    where
        product0_.id=?
{ 1, iPhone 6s }
Hibernate: 
    select
        customers0_.product_id as product_2_1_0_,
        customers0_.customer_id as customer1_1_0_,
        customer1_.id as id1_0_1_,
        customer1_.name as name2_0_1_ 
    from
        customers_products customers0_ 
    inner join
        Customers customer1_ 
            on customers0_.customer_id=customer1_.id 
    where
        customers0_.product_id=?
[{ 1, John }, { 2, Julie }, { 3, Ben }]
```

1) Notice the Select query to first retrieve just the product information for product with id 1, that is the "IPhone 6S" printed out to screen.
2) Next, when we invoke get customers, that's when another select query is run to retrieve the customers who have bought the IPhone. All three customers have bought the IPhone, "John", "Julie" and "Ben".

I'll leave it to you to figure out the details for the second product here, the one with id 4, this is the BenQ monitors.

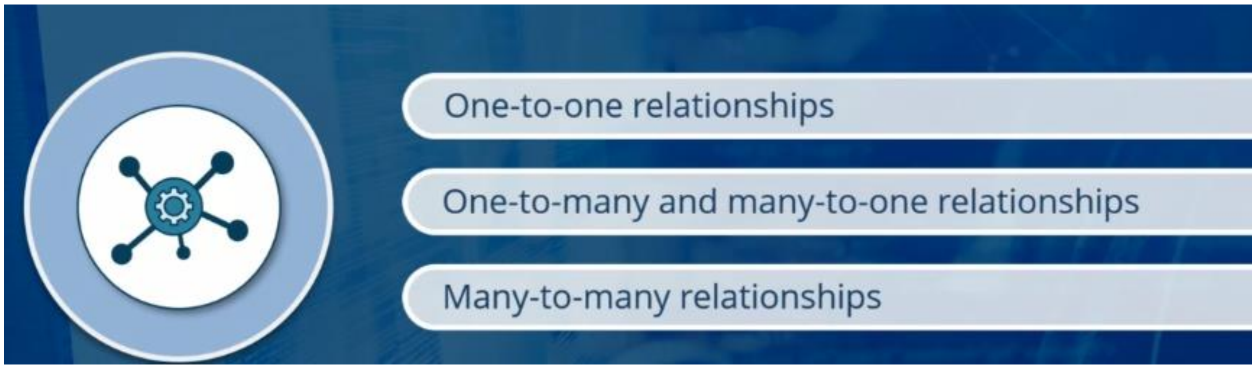


```
Hibernate: 
    select
        product0_.id as id1_2_0_,
        product0_.name as name2_2_0_ 
    from
        Products product0_ 
    where
        product0_.id=?
{ 4, BenQ Monitor }
Hibernate: 
    select
        customers0_.product_id as product_2_1_0_,
        customers0_.customer_id as customer1_1_0_,
        customer1_.id as id1_0_1_,
        customer1_.name as name2_0_1_ 
    from
        customers_products customers0_ 
    inner join
        Customers customer1_ 
            on customers0_.customer_id=customer1_.id 
    where
        customers0_.product_id=?
[{ 3, Ben }]
```

Here is the product information, and if you scroll down below, you'll see that only Ben has bought these monitors.

Course Summary

Mapping and Configuring Relationships


One-to-one relationships
One-to-many and many-to-one relationships
Many-to-many relationships

In this course, we explored the various kinds of relationships that you might want to express using JPA annotations. And how these relationships map to relational table design. We specifically looked at unidirectional, as well as bidirectional relationships. And we looked at one-to-one, one-to-many, many-to-one, and many-to-many relationships in each of these categories.

We started off by mapping one-to-one relationships using unidirectional mapping. Where the relationship is set up in just one direction from the owning entity to the owned entity.

We then saw how the same one-to-one relationship can be expressed in both directions. We implemented different techniques to express the same relationship such as having a join column in one of the tables. Having both tables use a shared primary key. And using a separate join table.

We then moved on to the one-to-many unidirectional relationship. And configured this using a join table as well as a join column, along with foreign key constraints.

We also saw how the entities on the many side of this relationship can be both eagerly as well as lazily loaded into our application.

We then setup this relationship in a bidirectional manner using a many-to-one mapping in addition to our one-to-many mapping.

We rounded off our understanding of entity relationships by exploring and implementing many-to-many relationships. And saw in some detail how we can configure the owning side and the owned side of this relationship.